- BlockPtr specifies the starting address for Mask, in units of 16 blocks. For example, BlockPtr = 00_h indicates block 0, BlockPtr = 01_h indicates block 16, BlockPtr = 02_h indicates block 32. BlockPtr uses EBV formatting (see Annex A).
- BlockRange specifies the range of Mask, starting at BlockPtr and ending (16×BlockRange)–1 blocks later. If BlockRange = 00_h then a Tag shall ignore the *BlockPermalock* command and instead backscatter an error code (see Annex I), remaining in the **secured** state.
- Mask specifies which memory blocks a Tag permalocks. Mask depends on the Read/Lock bit as follows:
  - Read/Lock = 0: The Interrogator shall omit Mask from the *BlockPermalock* command.
  - Read/Lock = 1: The Interrogator shall include a Mask of length 16×BlockRange bits in the *BlockPermalock* command. The Mask bits shall be ordered from lower-order block to higher (i.e. if BlockPtr = 00_h then the leading Mask bit refers to block 0). The Tag shall interpret each bit of Mask as follows:
    - Mask bit = 0: Retain the current permalock setting for the corresponding memory block.
    - Mask bit = 1: Permalock the corresponding memory block. If a block is already permalocked then the Tag shall retain the current permalock setting. A memory block, once permalocked, cannot be un-permalocked except by recommissioning the Tag (see 6.3.2.10).

The following example illustrates the usage of Read/Lock, BlockPtr, BlockRange, and Mask:

- If Read/Lock=1, BlockPtr=01_h, and BlockRange=01_h the Tag operates on sixteen blocks starting at block 16 and ending at block 31, permalocking those blocks whose corresponding bits are asserted in Mask.

The *BlockPermalock* command contains 8 RFU bits. Interrogators shall set these bits to 00_h when communicating with Class-1 Tags. If a Class-1 Tag receives a *BlockPermalock* command containing nonzero RFU bits it shall ignore the command and instead backscatter an error code (see Annex I), remaining in the **secured** state. Higher-class Tags may use these bits to expand the functionality of the *BlockPermalock* command.

The *BlockPermalock* command also includes the Tag's handle and a CRC-16. The CRC-16 is calculated over the first command-code bit to the last handle bit. If a Tag receives a *BlockPermalock* with a valid CRC-16 but an invalid handle it shall ignore the *BlockPermalock* and remain in the **secured** state.

If a Tag receives a *BlockPermalock* command that it cannot execute because User memory does not exist, or in which the LSB and/or the 2SB of a Tag's XPC_W1 is/are asserted (see Table 6.15), or in which one of the asserted Mask bits references a non-existent block, then the Tag shall ignore the *BlockPermalock* command and instead backscatter an error code (see Annex I), remaining in the **secured** state. A Tag shall treat as invalid a *BlockPermalock* command in which Read/Lock=0 but Mask is not omitted, or a *BlockPermalock* command in which Read/Lock=1 but Mask has a length that is not equal to 16×BlockRange bits (see 6.3.2.11 for the definition of an "invalid" command).

Certain Tags, depending on the Tag manufacturer's implementation, may be unable to execute a *BlockPermalock* command with certain BlockPtr and BlockRange values, in which case the Tag shall ignore the *BlockPermalock* command and instead backscatter an error code (see Annex I), remaining in the **secured** state. Because a Tag contains information in its TID memory that an Interrogator can use to uniquely identify the optional features that the Tag supports (see 6.3.2.1.3), this specification recommends that Interrogators read a Tag's TID memory prior to issuing a *BlockPermalock* command.

If an Interrogator issues a *BlockPermalock* command in which BlockPtr and BlockRange specify one or more nonexistent blocks, but Mask only asserts permalocking on existent blocks, then the Tag shall execute the command.

A *BlockPermalock* shall be prepended with a frame-sync (see 6.3.1.2.8).

After issuing a *BlockPermalock* command an Interrogator shall transmit CW for the lesser of T_{REPLY} or 20ms, where T_{REPLY} is the time between the Interrogator's *BlockPermalock* and the Tag's backscattered reply. An Interrogator may observe several possible outcomes from a *BlockPermalock* command, depending on the value of the Read/Lock bit in the command and, if Read/Lock = 1, the success or failure of the Tag's memory-lock operation:

- **Read/Lock = 0 and the Tag is able to execute the command:** The Tag shall backscatter the reply shown in Table 6.52, within time T_1 in Table 6.13, comprising a header (a 0-bit), the requested permalock bits, the Tag's handle, and a CRC-16 calculated over the 0-bit, permalock bits, and handle. The Tag's reply shall use the preamble specified by the TRext value in the *Query* that initiated the round.

Table 6.51 – ***BlockPermalock*** command

	Command	RFU	Read/Lock	MemBank	BlockPtr	BlockRange	Mask	RN	CRC-16
# of bits	8	8	1	2	EBV	8	Variable	16	16
description	11001001	00_h	0: Read 1: Permalock	00: RFU 01: EPC 10: TID 11: User	Mask starting address, specified in units of 16 blocks	Mask range, specified in units of 16 blocks	0: Retain current permalock setting 1: Assert permalock	handle	

Table 6.52 – Tag reply to a successful ***BlockPermalock*** command with Read/Lock = 0

	Header	Data	RN	CRC-16
# of bits	1	Variable	16	16
description	0	Permalock bits	handle	

Table 6.53 – Tag reply to a successful ***BlockPermalock*** command with Read/Lock = 1

	Header	RN	CRC-16
# of bits	1	16	16
description	0	handle	

- **Read/Lock = 0 and the Tag is unable to execute the command:** the Tag shall backscatter an error code, within time T_1 in Table 6.13, rather than the reply shown in Table 6.52 (see Annex I for error-code definitions and for the reply format). The Tag's reply shall use the preamble specified by the TRext value in the *Query* that initiated the round. An example of an unexecutable command is a Class-1 Tag receiving a *BlockPermalock* with MemBank<>11.
- **Read/Lock = 1 and The *BlockPermalock* succeeds:** After completing the *BlockPermalock* the Tag shall backscatter the reply shown in Table 6.53 comprising a header (a 0-bit), the Tag's handle, and a CRC-16 calculated over the 0-bit and handle. If the Interrogator observes this reply within 20 ms then the *BlockPermalock* completed successfully. The Tag's reply shall use the extended preamble shown in Figure 6.11 or Figure 6.15, as appropriate (i.e. the Tag shall reply as if TRext=1 regardless of the TRext value in the *Query* that initiated the round).
- **Read/Lock = 1 and the Tag encounters an error:** The Tag shall backscatter an error code during the CW period rather than the reply shown in Table 6.53 (see Annex I for error-code definitions and for the reply format). The Tag's reply shall use the extended preamble shown in Figure 6.11 or Figure 6.15, as appropriate (i.e. the Tag shall reply as if TRext=1 regardless of the TRext value in the *Query* that initiated the round).
- **Read/Lock = 1 and the *BlockPermalock* does not succeed:** If the Interrogator does not observe a reply within 20ms then the *BlockPermalock* did not complete successfully. The Interrogator may issue a *Req_RN* command (containing the Tag's handle) to verify that the Tag is still in the Interrogator's field, and may reissue the *BlockPermalock*.

Upon receiving a valid and executable *BlockPermalock* command a Tag shall perform the commanded operation, unless the Tag does not support block permalocking, in which case it shall ignore the command.

7. Intellectual property rights intrinsic to this specification

Attention is drawn to the possibility that some of the elements of this specification may be the subject of patent and/or other intellectual-property rights. EPCglobal shall not be held responsible for identifying any or all such patent or intellectual-property rights.

Annex A
(normative)
Extensible bit vectors (EBV)

An *extensible bit vector* (EBV) is a data structure with an extensible data range.

An EBV is an array of *blocks*. Each block contains a single extension bit followed by a specific number of data bits. If B represents the total number of bits in one block, then a block contains B – 1 data bits. Although a general EBV may contain blocks of varying lengths, Tags and Interrogators manufactured according to this specification shall use blocks of length 8 bits (EBV-8).

The data value represented by an EBV is simply the bit string formed by the data bits as read from left-to-right, ignoring the extension bits.

Tags and Interrogators shall use the EBV-8 word format specified in Table A.1.

Table A.1 – EBV-8 word format

	0	0	0000000				
	1	0	0000001				
$2^7 - 1$	127	0	1111111				
2^7	128	1	0000001	0	0000000		
$2^{14} - 1$	16383	1	1111111	0	1111111		
2^{14}	16384	1	0000001	1	0000000	0	0000000

Because each block has 7 data bits available, the EBV-8 can represent numeric values between 0 and 127 with a single block. To represent the value 128, the extension bit is set to 1 in the first block, and a second block is appended to the EBV-8. In this manner, an EBV-8 can represent arbitrarily large values.

This specification uses EBV-8 values to represent memory addresses and mask lengths.

Annex B
(normative)
State-transition tables

State-transition tables B.1 to B.7 shall define a Tag's response to Interrogator commands. The term "handle" used in the state-transition tables is defined in 6.3.2.4.5; error codes are defined in Table I.2; "slot" is the slot-counter output shown in Figure 6.19 and detailed in Annex J; "–" in the "Action" column means that a Tag neither modifies its **SL** or **inventoried** flags nor backscatters a reply.

B.1 Present state: Ready

Table B.1 – Ready state-transition table

Command	Condition	Action	Next State
Query[1]	slot=0; matching **inventoried** & **SL** flags	backscatter new RN16	**reply**
	slot<>0; matching **inventoried** & **SL** flags	–	**arbitrate**
	otherwise	–	**ready**
QueryRep	all	–	**ready**
QueryAdjust	all	–	**ready**
ACK	all	–	**ready**
NAK	all	–	**ready**
Req_RN	all	–	**ready**
Select	all	assert or deassert **SL**, or set **inventoried** to *A* or *B*	**ready**
Read	all	–	**ready**
Write	all	–	**ready**
Kill	all	–	**ready**
Lock	all	–	**ready**
Access	all	–	**ready**
BlockWrite	all	–	**ready**
BlockErase	all	–	**ready**
BlockPermalock	all	–	**ready**
Invalid[2]	all	–	**ready**

1: *Query* starts a new round and may change the session. *Query* also instructs a Tag to load a new random value into its slot counter.

2: "Invalid" shall mean an erroneous command, an unsupported command, a command with invalid parameters, a command with a CRC error, or any other command either not recognized or not executable by the Tag.

B.2 Present state: Arbitrate

Table B.2 – Arbitrate state-transition table

Command	Condition	Action	Next State
Query [1,2]	slot=0; matching **inventoried** & **SL** flags	backscatter new RN16	**reply**
	slot<>0; matching **inventoried** & **SL** flags	–	**arbitrate**
	otherwise	–	**ready**
QueryRep	slot=0 after decrementing slot counter	backscatter new RN16	**reply**
	slot<>0 after decrementing slot counter	–	**arbitrate**
QueryAdjust [2]	slot=0	backscatter new RN16	**reply**
	slot<>0	–	**arbitrate**
ACK	all	–	**arbitrate**
NAK	all	–	**arbitrate**
Req_RN	all	–	**arbitrate**
Select	all	assert or deassert **SL**, or set **inventoried** to *A* or *B*	**ready**
Read	all	–	**arbitrate**
Write	all	–	**arbitrate**
Kill	all	–	**arbitrate**
Lock	all	–	**arbitrate**
Access	all	–	**arbitrate**
Erase	all	–	**arbitrate**
BlockWrite	all	–	**arbitrate**
BlockErase	all	–	**arbitrate**
BlockPermalock	all	–	**arbitrate**
Invalid [3]	all	–	**arbitrate**

1: *Query* starts a new round and may change the session.

2: *Query* and *QueryAdjust* instruct a Tag to load a new random value into its slot counter.

3: "Invalid" shall mean an erroneous command, an unsupported command, a command with invalid parameters, a command with a CRC error, a command (other than a *Query*) with a session parameter not matching that of the inventory round currently in progress, or any other command either not recognized or not executable by the Tag.

B.3 Present state: Reply

Table B.3 – Reply state-transition table

Command	Condition	Action	Next State
Query [1,2]	slot=0; matching **inventoried** & **SL** flags	backscatter new RN16	**reply**
	slot<>0; matching **inventoried** & **SL** flags	–	**arbitrate**
	otherwise	–	**ready**
QueryRep	all	–	**arbitrate**
QueryAdjust [2]	slot=0	backscatter new RN16	**reply**
	slot<>0	–	**arbitrate**
ACK	valid RN16	see Table 6.14	**acknowledged**
	invalid RN16	–	**arbitrate**
NAK	all	–	**arbitrate**
Req_RN	all	–	**arbitrate**
Select	all	assert or deassert **SL**, or set **inventoried** to *A* or *B*	**ready**
Read	all	–	**arbitrate**
Write	all	–	**arbitrate**
Kill	all	–	**arbitrate**
Lock	all	–	**arbitrate**
Access	all	–	**arbitrate**
BlockWrite	all	–	**arbitrate**
BlockErase	all	–	**arbitrate**
BlockPermalock	all	–	**arbitrate**
T_2 timeout	See Figure 6.16 and Table 6.13	–	**arbitrate**
Invalid [3]	all	–	**reply**

1: *Query* starts a new round and may change the session.

2: *Query* and *QueryAdjust* instruct a Tag to load a new random value into its slot counter.

3: "Invalid" shall mean an erroneous command, an unsupported command, a command with invalid parameters, a command with a CRC error, a command (other than a *Query*) with a session parameter not matching that of the inventory round currently in progress, or any other command either not recognized or not executable by the Tag.

B.4 Present state: Acknowledged

Table B.4 – Acknowledged state-transition table

Command	Condition	Action	Next State
Query[1]	slot=0; matching **inventoried**[2] & **SL** flags	backscatter new RN16; transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**reply**
	slot<>0; matching **inventoried**[2] & **SL** flags	transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**arbitrate**
	otherwise	transition **inventoried** from *A*→*B* or *B*→*A* if and only if new session matches prior session	**ready**
QueryRep	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
QueryAdjust	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
ACK	valid RN16	see Table 6.14	**acknowledged**
	invalid RN16	–	**arbitrate**
NAK	all	–	**arbitrate**
Req_RN	valid RN16 & access password<>0	backscatter handle	**open**
	valid RN16 & access password=0	backscatter handle	**secured**
	invalid RN16	–	**acknowledged**
Select	all	assert or deassert **SL**, or set **inventoried** to *A* or *B*	**ready**
Read	all	–	**arbitrate**
Write	all	–	**arbitrate**
Kill	all	–	**arbitrate**
Lock	all	–	**arbitrate**
Access	all	–	**arbitrate**
BlockWrite	all	–	**arbitrate**
BlockErase	all	–	**arbitrate**
Block-Permalock	all	–	**arbitrate**
T_2 timeout	See Figure 6.16 and Table 6.13	–	**arbitrate**
Invalid[3]	all	–	**acknowledged**

1: *Query* starts a new round and may change the session. *Query* also instructs a Tag to load a new random value into its slot counter.

2: As described in 6.3.2.8, a Tag transitions its **inventoried** flag prior to evaluating the condition.

3: "Invalid" shall mean an erroneous command, an unsupported command, a command with invalid parameters, a command with a CRC error, a command (other than a *Query*) with a session parameter not matching that of the inventory round currently in progress, or any other command either not recognized or not executable by the Tag.

B.5 Present state: Open

Table B.5 – Open state-transition table

Command	Condition	Action	Next State
Query[1]	slot=0; matching **inventoried**[2] & **SL** flags	backscatter new RN16; transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**reply**
	slot<>0; matching **inventoried**[2] & **SL** flags	transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**arbitrate**
	otherwise	transition **inventoried** from *A*→*B* or *B*→*A* if and only if new session matches prior session	**ready**
QueryRep	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
QueryAdjust	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
ACK	valid handle	see Table 6.14	**open**
	invalid handle	–	**arbitrate**
NAK	all	–	**arbitrate**
Req_RN	valid handle	backscatter new RN16	**open**
	invalid handle	–	**open**
Select	all	assert or deassert **SL**, or set **inventoried** to *A* or *B*	**ready**
Read	valid handle & valid memory access	backscatter data and handle	**open**
	valid handle & invalid memory access	backscatter error code	**open**
	invalid handle	–	**open**
Write	valid handle & valid memory access	backscatter handle when done	**open**
	valid handle & invalid memory access	backscatter error code	**open**
	invalid handle	–	**open**
Kill[3] (see also Figure 6.23)	valid handle & valid nonzero kill password & Recom = 0	backscatter handle when done	**killed**
	valid handle & valid nonzero kill password & Recom <> 0	backscatter handle when done	**open**
	valid handle & invalid nonzero kill password	–	**arbitrate**
	valid handle & kill password=0	backscatter error code	**open**
	invalid handle	–	**open**
Lock	all	–	**open**
Access (see also Figure 6.25)	valid handle & valid access password	backscatter handle	**secured**
	valid handle & invalid access password	–	**arbitrate**
	invalid handle	–	**open**
BlockWrite	valid handle & valid memory access	backscatter handle when done	**open**
	valid handle & invalid memory access	backscatter error code	**open**
	invalid handle	–	**open**
BlockErase	valid handle & valid memory access	backscatter handle when done	**open**
	valid handle & invalid memory access	backscatter error code	**open**
	invalid handle	–	**open**
Block-Permalock	all	–	**open**
Invalid[4]	all, excluding valid commands interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25)	–	**open**
	otherwise valid commands, except *Req_RN* or *Query*, interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25).	–	**arbitrate**

1: *Query* starts a new round and may change the session. *Query* also instructs a Tag to load a new random value into its slot counter.

2: As described in 6.3.2.8, a Tag transitions its **inventoried** flag prior to evaluating the condition.

3. As described in 6.3.2.11.3.4, if a Tag does not implement recommissioning then the Tag treats nonzero Recom bits as though Recom = 0.

4: "Invalid" shall mean an erroneous command; an unsupported command; a command with invalid parameters; a command with a CRC error; a command (other than a *Query*) with a session parameter not matching that of the inventory round currently in progress; or any other command either not recognized or not executable by the Tag.

B.6 Present state: Secured

Table B.6 – Secured state-transition table

Command	Condition	Action	Next State
Query[1]	slot=0; matching **inventoried**[2] & **SL** flags	backscatter new RN16; transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**reply**
	slot<>0; matching **inventoried**[2] & **SL** flags	transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**arbitrate**
	otherwise	transition **inventoried** from *A*→*B* or *B*→*A* if and only if new session matches prior session	**ready**
QueryRep	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
QueryAdjust	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
ACK	valid handle	see Table 6.14	**secured**
	invalid handle	–	**arbitrate**
NAK	all	–	**arbitrate**
Req_RN	valid handle	backscatter new RN16	**secured**
	invalid handle	–	**secured**
Select	all	assert or deassert **SL**, or set **inventoried** to *A* or *B*	**ready**
Read	valid handle & valid memory access	backscatter data and handle	**secured**
	valid handle & invalid memory access	backscatter error code	**secured**
	invalid handle	–	**secured**
Write	valid handle & valid memory access	backscatter handle when done	**secured**
	valid handle & invalid memory access	backscatter error code	**secured**
	invalid handle	–	**secured**
Kill[3] (see also Figure 6.23)	valid handle & valid nonzero kill password & Recom = 0	backscatter handle when done	**killed**
	valid handle & valid nonzero kill password & Recom <> 0	backscatter handle when done	**secured**
	valid handle & invalid nonzero kill password	–	**arbitrate**
	valid handle & kill password=0	backscatter error code	**secured**
	invalid handle	–	**secured**
Lock	valid handle & valid lock payload	backscatter handle when done	**secured**
	valid handle & invalid lock payload	backscatter error code	**secured**
	invalid handle	–	**secured**
Access (see also Figure 6.25)	valid handle & valid access password	backscatter handle	**secured**
	valid handle & invalid access password	–	**arbitrate**
	invalid handle	–	**secured**
BlockWrite	valid handle & valid memory access	backscatter handle when done	**secured**
	valid handle & invalid memory access	backscatter error code	**secured**
	invalid handle	–	**secured**
BlockErase	valid handle & valid memory access	backscatter handle when done	**secured**
	valid handle & invalid memory access	backscatter error code	**secured**
	invalid handle	–	**secured**
Block-Permalock	valid handle, valid payload, & Read/Lock = 0	backscatter permalock bits and handle	**secured**
	valid handle, invalid payload, & Read/Lock = 0	backscatter error code	**secured**
	valid handle, valid payload, & Read/Lock = 1	backscatter handle when done	**secured**
	valid handle, invalid payload, & Read/Lock = 1	backscatter error code	**secured**
	invalid handle	–	**secured**
Invalid[4]	all, excluding valid commands interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25).	–	**secured**
	otherwise valid commands, except *Req_RN* or *Query*, interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25).	–	**arbitrate**

1: *Query* starts a new round and may change the session. *Query* also instructs a Tag to load a new random value into its slot counter.

2: As described in 6.3.2.8, a Tag transitions its **inventoried** flag prior to evaluating the condition.

3. As described in 6.3.2.11.3.4, if a Tag does not implement recommissioning then the Tag treats nonzero Recom bits as though Recom = 0.

4: "Invalid" shall mean an erroneous command; an unsupported command; a command with invalid parameters; a command with a CRC error; a command (other than a *Query*) with a session parameter not matching that of the inventory round currently in progress; or any other command either not recognized or not executable by the Tag.

B.7 Present state: Killed

Table B.7 – Killed state-transition table

Command	Condition	Action	Next State
Query	all	–	**killed**
QueryRep	all	–	**killed**
QueryAdjust	all	–	**killed**
ACK	all	–	**killed**
NAK	all	–	**killed**
Req_RN	all	–	**killed**
Select	all	–	**killed**
Read	all	–	**killed**
Write	all	–	**killed**
Kill	all	–	**killed**
Lock	all	–	**killed**
Access	all	–	**killed**
BlockWrite	all	–	**killed**
BlockErase	all	–	**killed**
BlockPermalock	all	–	**killed**
Invalid [1]	all	–	**killed**

1: "Invalid" shall mean an erroneous command, an unsupported command, a command with invalid parameters, a command with a CRC error, or any other command either not recognized or not executable by the Tag.

Annex C
(normative)
Command-Response Tables

Command-response tables C.1 to C.18 shall define a Tag's response to Interrogator commands. The term "handle" used in the state-transition tables is defined in 6.3.2.4.5; error codes are defined in Table I.2; "slot" is the slot-counter output shown in Figure 6.19 and detailed in Annex J; "–" in the "Response" column means that a Tag neither modifies its **SL** or **inventoried** flags nor backscatters a reply.

C.1 Command response: Power-up

Table C.1 – Power-up command-response table

Starting State	Condition	Response	Next State
ready, arbitrate, reply, acknowledged, open, secured	power-up	–	**ready**
killed	all	–	**killed**

C.2 Command response: *Query*

Table C.2 – *Query*[1] command-response table

Starting State	Condition	Response	Next State
ready, arbitrate, reply	slot=0; matching **inventoried** & **SL** flags	backscatter new RN16	**reply**
	slot<>0; matching **inventoried** & **SL** flags	–	**arbitrate**
	otherwise	–	**ready**
acknowledged, open, secured	slot=0; matching **inventoried**[2] & **SL** flags	backscatter new RN16; transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**reply**
	slot<>0; matching **inventoried**[2] & **SL** flags	transition **inventoried**[2] from *A*→*B* or *B*→*A* if and only if new session matches prior session	**arbitrate**
	otherwise	transition **inventoried** from *A*→*B* or *B*→*A* if and only if new session matches prior session	**ready**
killed	all	–	**killed**

1: *Query* (in any state other than **killed**) starts a new round and may change the session; *Query* also instructs a Tag to load a new random value into its slot counter.

2: As described in 6.3.2.8, a Tag transitions its **inventoried** flag prior to evaluating the condition.

C.3 Command response: *QueryRep*

Table C.3 – ***QueryRep*** command-response table[1]

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate	slot<>0 after decrementing slot counter	–	**arbitrate**
	slot=0 after decrementing slot counter	backscatter new RN16	**reply**
reply	all	–	**arbitrate**
acknowledged, open, secured	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
killed	all	–	**killed**

1: See Table C.18 for the Tag response to a *QueryRep* whose session parameter does not match that of the current inventory round.

C.4 Command response: *QueryAdjust*

Table C.4 – ***QueryAdjust***[1] command-response table[2]

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply	slot<>0	–	**arbitrate**
	slot=0	backscatter new RN16	**reply**
acknowledged, open, secured	all	transition **inventoried** from *A*→*B* or *B*→*A*	**ready**
killed	all	–	**killed**

1: *QueryAdjust*, in the **arbitrate** or **reply** states, instructs a Tag to load a new random value into its slot counter.

2: See Table C.18 for the Tag response to a *QueryAdjust* whose session parameter does not match that of the current inventory round.

C.5 Command response: *ACK*

Table C.5 – *ACK* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate	all	–	**arbitrate**
reply	valid RN16	see Table 6.14	**acknowledged**
	invalid RN16	–	**arbitrate**
acknowledged	valid RN16	see Table 6.14	**acknowledged**
	invalid RN16	–	**arbitrate**
open	valid handle	see Table 6.14	**open**
	invalid handle	–	**arbitrate**
secured	valid handle	see Table 6.14	**secured**
	invalid handle	–	**arbitrate**
killed	all	–	**killed**

C.6 Command response: *NAK*

Table C.6 – *NAK* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged, open, secured	all	–	**arbitrate**
killed	all	–	**killed**

C.7 Command response: *Req_RN*

Table C.7 – *Req_RN* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply	all	–	**arbitrate**
acknowledged	valid RN16 & access password<>0	backscatter handle	**open**
	valid RN16 & access password=0	backscatter handle	**secured**
	invalid RN16	–	**acknowledged**
open	valid handle	backscatter new RN16	**open**
	invalid handle	–	**open**
secured	valid handle	backscatter new RN16	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

C.8 Command response: *Select*

Table C.9 – *Select* command-response table

Starting State	Condition	Response	Next State
ready, arbitrate, reply, acknowledged, open, secured	all	assert or deassert **SL**, or set **inventoried** to *A* or *B*	**ready**
killed	all	–	**killed**

C.9 Command response: *Read*

Table C.8 – *Read* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open	valid handle & invalid memory access	backscatter error code	**open**
	valid handle & valid memory access	backscatter data and handle	**open**
	invalid handle	–	**open**
secured	valid handle & invalid memory access	backscatter error code	**secured**
	valid handle & valid memory access	backscatter data and handle	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

C.10 Command response: *Write*

Table C.10 – *Write* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open	valid handle & invalid memory access	backscatter error code	**open**
	valid handle & valid memory access	backscatter handle when done	**open**
	invalid handle	–	**open**
secured	valid handle & invalid memory access	backscatter error code	**secured**
	valid handle & valid memory access	backscatter handle when done	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

C.11 Command response: *Kill*

Table C.11 – *Kill*[1] command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open[2]	valid handle & kill password=0	backscatter error code	**open**
	valid handle & invalid nonzero kill password	–	**arbitrate**
	valid handle & valid nonzero kill password & Recom=0	backscatter handle when done	**killed**
	valid handle & valid nonzero kill password & Recom<>0	backscatter handle when done	**open**
	invalid handle	–	**open**
secured[2]	valid handle & kill password=0	backscatter error code	**secured**
	valid handle & invalid nonzero kill password	–	**arbitrate**
	valid handle & valid nonzero kill password & Recom=0	backscatter handle when done	**killed**
	valid handle & valid nonzero kill password & Recom<>0	backscatter handle when done	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

1: See also Figure 6.23.

2: As described in 6.3.2.11.3.4, if a Tag does not implement recommissioning then the Tag treats nonzero Recom bits as though Recom=0.

C.12 Command response: *Lock*

Table C.12 – *Lock* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open	all	–	**open**
secured	valid handle & invalid lock payload	backscatter error code	**secured**
	valid handle & valid lock payload	backscatter handle when done	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

C.13 Command response: *Access*

Table C.13 – *Access*[1] command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open	valid handle & invalid access password	–	**arbitrate**
	valid handle & valid access password	backscatter handle	**secured**
	invalid handle	–	**open**
secured	valid handle & invalid access password	–	**arbitrate**
	valid handle & valid access password	backscatter handle	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

1: See also Figure 6.25.

C.14 Command response: *BlockWrite*

Table C.14 – *BlockWrite* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open	valid handle & invalid memory access	backscatter error code	**open**
	valid handle & valid memory access	backscatter handle when done	**open**
	invalid handle	–	**open**
secured	valid handle & invalid memory access	backscatter error code	**secured**
	valid handle & valid memory access	backscatter handle when done	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

C.15 Command response: *BlockErase*

Table C.15 – *BlockErase* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open	valid handle & invalid memory access	backscatter error code	**open**
	valid handle & valid memory access	backscatter handle when done	**open**
	invalid handle	–	**open**
secured	valid handle & invalid memory access	backscatter error code	**secured**
	valid handle & valid memory access	backscatter handle when done	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

C.16 Command response: *BlockPermalock*

Table C.16 – *BlockPermalock* command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate, reply, acknowledged	all	–	**arbitrate**
open	all	–	**open**
secured	valid handle, valid payload, & Read/Lock = 0	backscatter permalock bits and handle	**secured**
	valid handle, invalid payload, & Read/Lock = 0	backscatter error code	**secured**
	valid handle, valid payload, & Read/Lock = 1	backscatter handle when done	**secured**
	valid handle, invalid payload, & Read/Lock = 1	backscatter error code	**secured**
	invalid handle	–	**secured**
killed	all	–	**killed**

C.17 Command response: T_2 timeout

Table C.17 – T_2 timeout command-response table

Starting State	Condition	Response	Next State
ready	all	–	**ready**
arbitrate	all	–	**arbitrate**
reply, acknowledged	See Figure 6.16 and Table 6.13	–	**arbitrate**
open	all	–	**open**
secured	all	–	**secured**
killed	all	–	**killed**

C.18 Command response: Invalid command

Table C.18 – Invalid command-response table

Starting State	Condition	Response	Next State
ready[1]	all	–	**ready**
arbitrate[2]	all	–	**arbitrate**
reply[2]	all	–	**reply**
acknowledged[2]	all	–	**acknowledged**
open[2]	all, excluding valid commands interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25).	–	**open**
	otherwise valid commands, except *Req_RN* or *Query*, interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25).	–	**arbitrate**
secured[2]	all, excluding valid commands interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25).	–	**secured**
	otherwise valid commands, except *Req_RN* or *Query*, interspersed between successive *Kill* or *Access* commands in a kill or access sequence, respectively (see Figure 6.23 and Figure 6.25).	–	**arbitrate**
killed[1]	all	–	**killed**

1: "Invalid" shall mean an erroneous command, an unsupported command, a command with invalid parameters, a command with a CRC error, or any other command either not recognized or not executable by the Tag.

2: "Invalid" shall mean an erroneous command; an unsupported command; a command with invalid parameters; a command with a CRC error; a command (other than a *Query*) with a session parameter not matching that of the inventory round currently in progress; or any other command either not recognized or not executable by the Tag.

Annex D
(informative)
Example slot-count (*Q*) selection algorithm

D.1 Example algorithm an Interrogator might use to choose *Q*

Figure D.1 shows an algorithm an Interrogator might use for setting the slot-count parameter *Q* in a *Query* command. Q_{fp} is a floating-point representation of *Q*; an Interrogator rounds Q_{fp} to an integer value and substitutes this integer value for *Q* in the *Query*. Typical values for *C* are $0.1 < C < 0.5$. An Interrogator typically uses small values of *C* when *Q* is large, and larger values of *C* when *Q* is small.

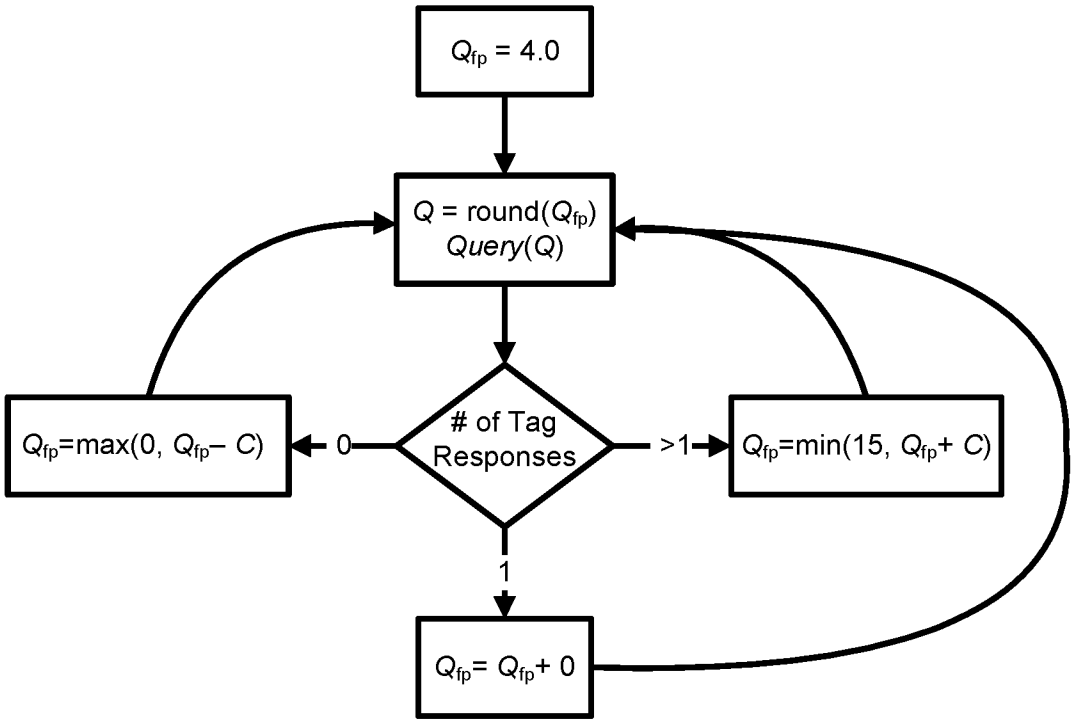


Figure D.1 – Example algorithm for choosing the slot-count parameter *Q*

Annex E
(informative)
Example of Tag inventory and access

E.1 Example inventory and access of a single Tag

Figure E.1 shows the steps by which an Interrogator inventories and accesses a single Tag.

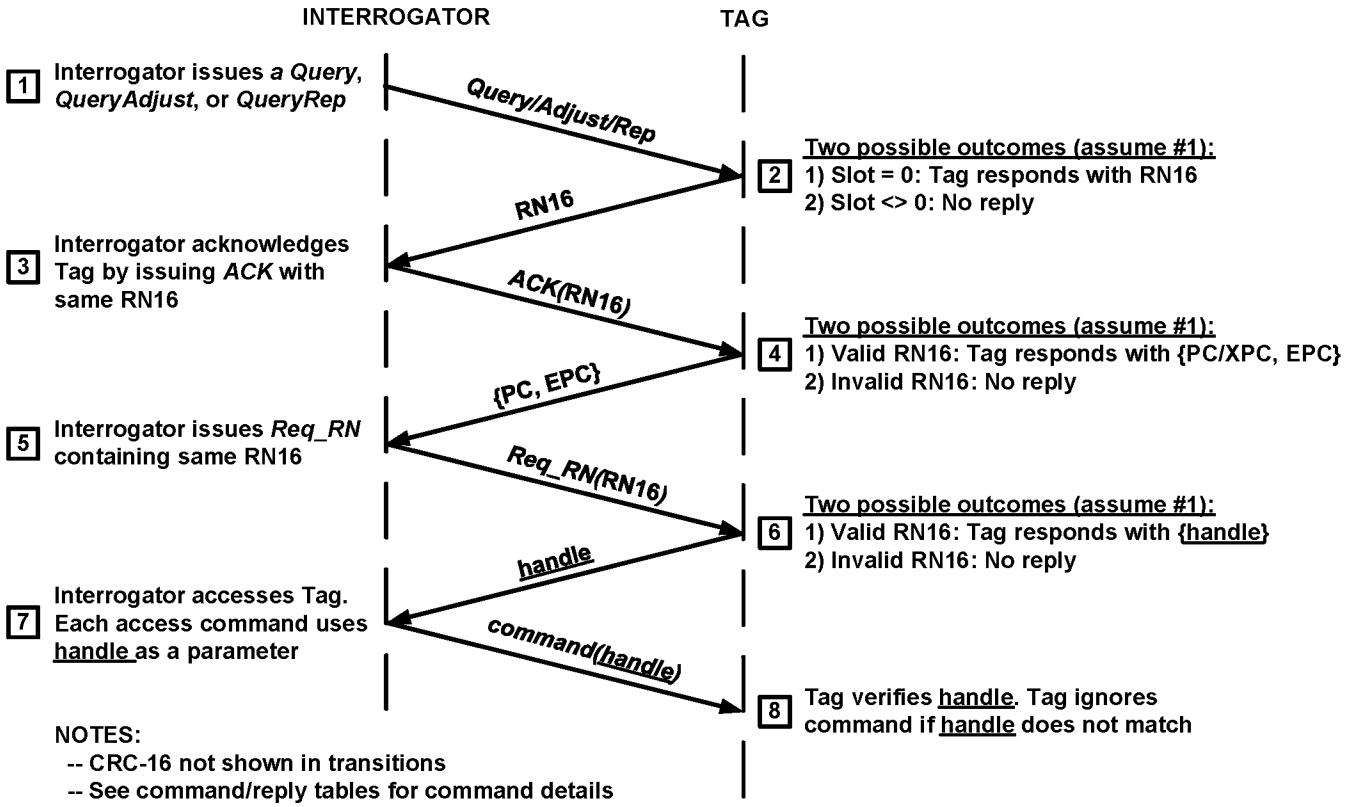


Figure E.1 – Example of Tag inventory and access

Annex F
(informative)
Calculation of 5-bit and 16-bit cyclic redundancy checks

F.1 Example CRC-5 encoder/decoder

An exemplary schematic diagram for a CRC-5 encoder/decoder is shown in Figure F.1, using the polynomial and preset defined in Table 6.12.

To calculate a CRC-5, first preload the entire CRC register (i.e. Q[4:0], Q4 being the MSB and Q0 the LSB) with the value 01001_2 (see Table F.1), then clock the data bits to be encoded into the input labeled DATA, MSB first. After clocking in all the data bits, Q[4:0] holds the CRC-5 value.

To check a CRC-5, first preload the entire CRC register (Q[4:0]) with the value 01001_2, then clock the received data and CRC-5 {data, CRC-5} bits into the input labeled DATA, MSB first. The CRC-5 check passes if the value in Q[4:0] = 00000_2.

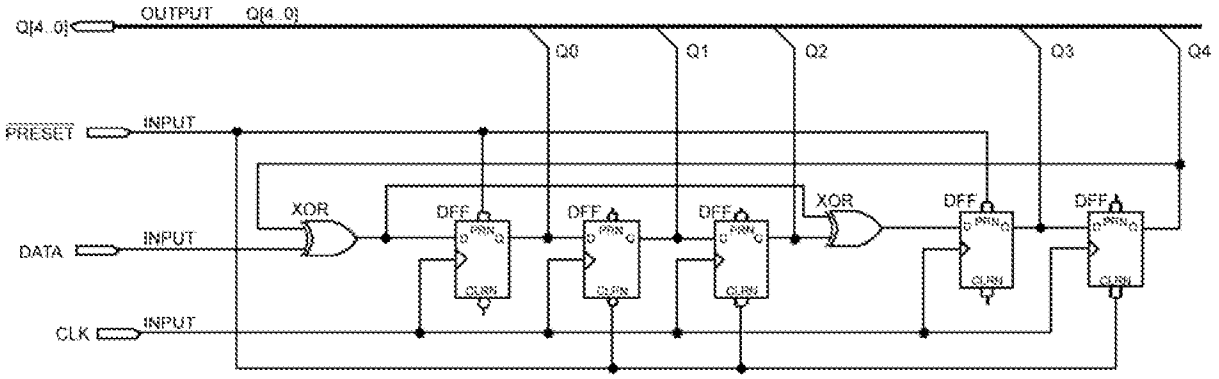


Figure F.1 – Example CRC-5 circuit

Table F.1 – CRC-5 register preload values

Register	Preload value
Q0	1
Q1	0
Q2	0
Q3	1
Q4	0

F.2 Example CRC-16 encoder/decoder

An exemplary schematic diagram for a CRC-16 encoder/decoder is shown in Figure F.2, using the polynomial and preset defined in Table 6.11 (the polynomial used to calculate the CRC-16, $x^{16} + x^{12} + x^5 + 1$, is the CRC-CCITT International Standard, ITU Recommendation X.25).

To calculate a CRC-16, first preload the entire CRC register (i.e. Q[15:0], Q15 being the MSB and Q0 the LSB) with the value $FFFF_h$. Second, clock the data bits to be encoded into the input labeled DATA, MSB first. After clocking in all the data bits, Q[15:0] holds the ones-complement of the CRC-16. Third, invert all the bits of Q[15:0] to produce the CRC-16.

There are two methods to check a CRC-16:

Method 1: First preload the entire CRC register (Q[15:0]) with the value $FFFF_h$, then clock the received data and CRC-16 {data, CRC-16} bits into the input labeled DATA, MSB first. The CRC-16 check passes if the value in Q[15:0]=$1D0F_h$.

Method 2: First preload the entire CRC register (Q[15:0]) with the value $FFFF_h$. Second, clock the received data bits into the input labeled DATA, MSB first. Third, invert all bits of the received CRC-16, and clock the inverted CRC-16 bits into the input labeled DATA, MSB first. The CRC-16 check passes if the value in Q[15:0]=0000_h.

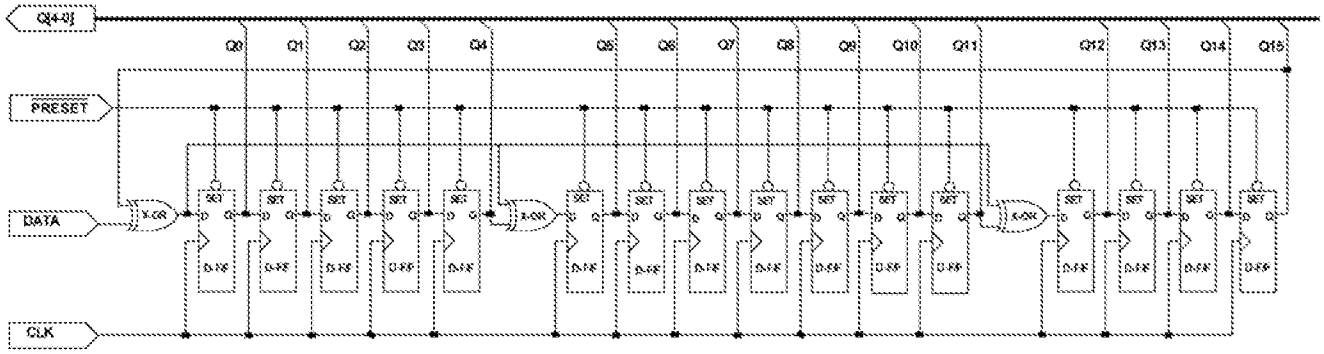


Figure F.2 – Example CRC-16 circuit

F.3 Example CRC-16 calculations

This example shows the StoredCRC (a CRC-16) that a Tag would calculate at power-up.

As shown in Figure 6.17, EPC memory contains a StoredCRC starting at address 00_h, a StoredPC starting at address 10_h, zero or more EPC words starting at address 20_h, an optional XPC_W1 starting at address 210_h, and an optional XPC_W2 starting at address 220_h. As described in 6.3.2.1.2.1, a Tag calculates its StoredCRC over its StoredPC and EPC, but omits the XPC_W1 and XPC_W2 from the calculation. Table F.2 shows the StoredCRC that a Tag would calculate and logically map into EPC memory at power-up, for the indicated example StoredPC and EPC word values. In each successive column, one more word of EPC memory is written, with the entire EPC memory written in the rightmost column. The indicated StoredPC values correspond to the number of EPC words written, with StoredPC bits 15_h–$1F_h$ set to zero. Entries marked N/A mean that that word of EPC memory is not included as part of the CRC calculation.

Table F.2 – EPC memory contents for an example Tag

EPC word starting address	EPC word contents	EPC word values						
00_h	StoredCRC	$E2F0_h$	$CCAE_h$	$968F_h$	$78F6_h$	$C241_h$	$2A91_h$	1835_h
10_h	StoredPC	0000_h	0800_h	1000_h	1800_h	2000_h	2800_h	3000_h
20_h	EPC word 1	N/A	1111_h	1111	1111_h	1111_h	1111_h	1111_h
30_h	EPC word 2	N/A	N/A	2222_h	2222_h	2222_h	2222_h	2222_h
20_h	EPC word 3	N/A	N/A	N/A	3333_h	3333_h	3333_h	3333_h
40_h	EPC word 4	N/A	N/A	N/A	N/A	4444_h	4444_h	4444_h
50_h	EPC word 5	N/A	N/A	N/A	N/A	N/A	5555_h	5555_h
60_h	EPC word 6	N/A	N/A	N/A	N/A	N/A	N/A	6666_h

Annex G
(Normative)
Multiple- and dense-Interrogator channelized signaling

This Annex describes channelized signaling in the optional multiple- and dense-Interrogator operating modes. It provides methods that Interrogators may use, as permitted by local authorities, to maximize the spectral efficiency and performance of RFID systems while minimizing the interference to non-RFID systems.

Because regulatory requirements vary worldwide, and even within a given regulatory region are prone to ongoing reinterpretation and revision, this Annex does not specify multiple- or dense-Interrogator operating requirements for any given regulatory region. Instead, this Annex merely outlines the goals of channelized signaling, and defers specification of the Interrogator operating requirements for each individual regulatory region to the channel plans located at www.epcglobalinc.org/regulatorychannelplans.

When an Interrogator in a multiple- or dense-Interrogator environment instructs Tags to use subcarrier backscatter, the Interrogator shall adopt the channel plan found at the above-referenced link for the regulatory region in which it is operating. When an Interrogator in a multiple- and dense-Interrogator environment instructs Tags to use FM0 backscatter, the Interrogator shall adopt a channel plan in accordance with local regulations.

Regardless of the regulatory region and the choice of Tag backscatter data encoding,

- Interrogator signaling (both modulated and CW) shall be centered in a channel with the frequency accuracy specified in 6.3.1.2.1, unless local regulations specify tighter frequency accuracy, in which case the Interrogator shall meet the local regulations, and
- Interrogator transmissions shall satisfy the multiple- or dense-Interrogator transmit mask in 6.3.1.2.11 (as appropriate), unless local regulations specify a tighter mask, in which case the Interrogator shall meet the local regulations.

If an Interrogator uses SSB-ASK modulation, the transmit spectrum shall be centered in the channel during R=>T signaling, and the CW shall be centered in the channel during Tag backscatter.

G.1 Overview of dense-interrogator channelized signaling (informative)

In environments containing two or more Interrogators, the range and rate at which Interrogators singulate Tags can be improved by preventing Interrogator transmissions from colliding spectrally with Tag responses. This section describes three frequency-division multiplexing (FDM) methods that minimize such Interrogator-on-Tag collisions. In each of these methods, Interrogator transmissions and Tag responses are separated spectrally.

1. *Channel-boundary backscatter*: Interrogator transmissions are constrained to occupy only a small portion of the center of each channel, and Tag backscatter is situated at the channel boundaries.
2. *Alternative-channel backscatter*: Interrogator transmissions are located in a subset of the channels, and Tag backscatter is located in a different subset of the channels.
3. *In-channel backscatter*: Interrogator transmissions are constrained to occupy only a small portion of the center of each channel, and Tag backscatter is situated near but within the channel boundaries.

Figure G.1, shows examples of these FDM dense-Interrogator methods. For optimum performance, the operating requirements located at www.epcglobalinc.org/regulatorychannelplans suggest (but do not require) choosing values for BLF and M that allow a guardband between Interrogator signaling and Tag responses.

Example 1: Channel-boundary backscatter

FCC 15.247, dated October 2000, authorizes frequency-hopping operation in the ISM band from 902–928 MHz with 500 kHz maximum channel width, and does not prohibit channel-boundary backscatter. In such an environment Interrogators will use 500 kHz channels with channel-boundary backscatter. Example 1 of Figure G.1 shows Interrogator transmissions using PR-ASK modulation with Tari = 25 µs, and 62.5 kbps Tag data backscatter on a 250 kHz subcarrier (BLF = 250 kHz; M = 4). Interrogators center their R=>T signaling in the channels, with transmissions unsynchronized in time, hopping among channels.

Example 2: Alternative-channel backscatter

ETSI Technical Group 34 has proposed an amendment to ERC REC 70-03E Annex 11 allocating four high-power 200 kHz channels, each spaced 600 kHz apart, in the 865–868 MHz frequency range. This amendment allows adjacent-channel Tag backscatter. In such an environment Interrogators will use alternative-channel

backscatter. Example 2 of Figure G.1 shows Interrogator transmissions using SSB-ASK modulation with Tari = 25 µs, and 75 kbps Tag data backscatter on a 300 kHz subcarrier (BLF = 300 kHz, M = 4).

Example 3: FDM in-channel backscatter

A hypothetical regulatory region allocates four 500 kHz channels and disallows adjacent-channel and channel-boundary backscatter. In such an environment Interrogators will use in-channel backscatter. Example 3 of Figure G.1 shows Interrogator transmissions using PR-ASK modulation with Tari = 25 µs, and 25 kbps Tag data backscatter on a 200 kHz subcarrier (BLF = 200 kHz, M = 8).

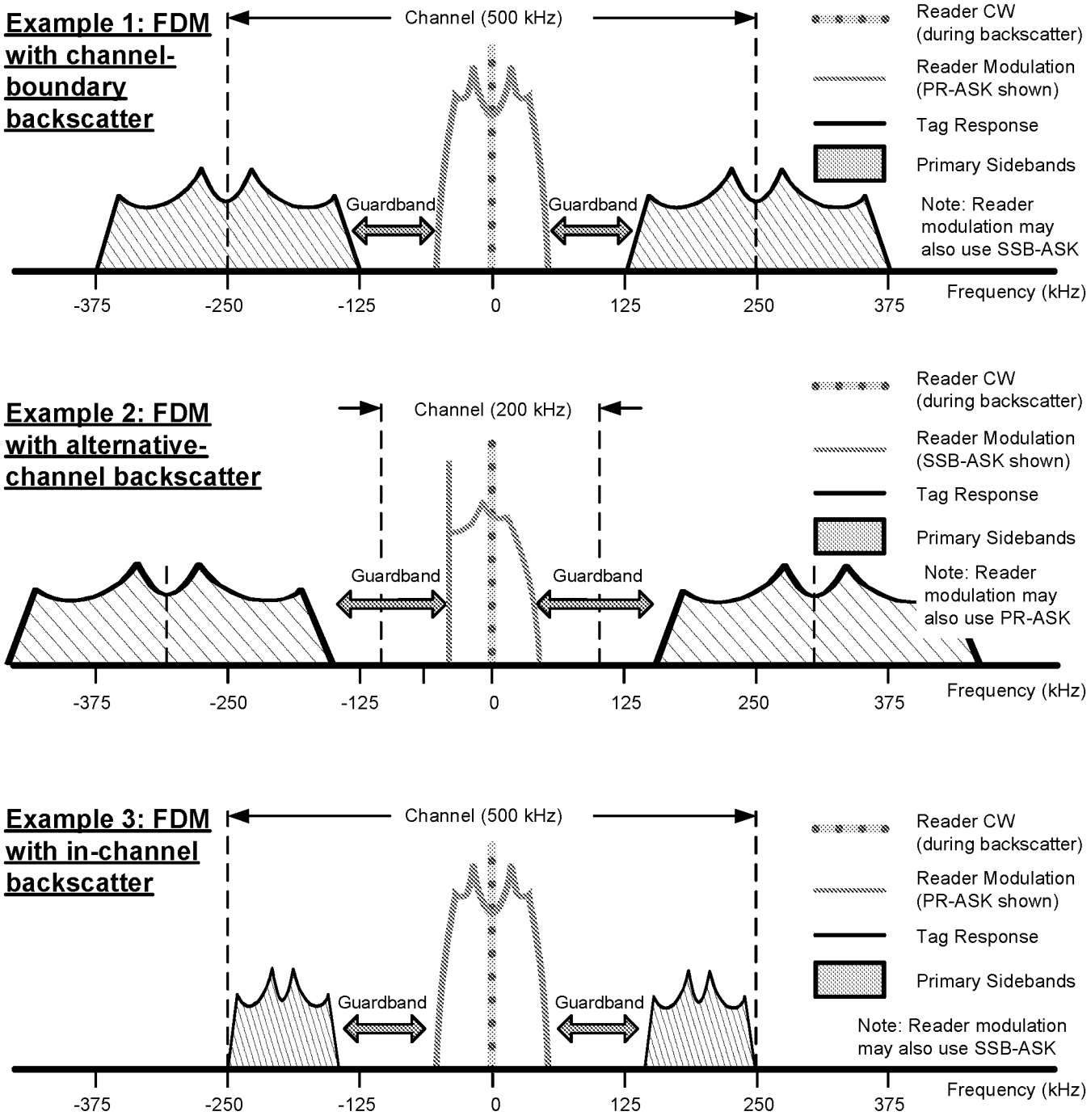


Figure G.1 – Examples of dense-Interrogator-mode operation

Annex H
(informative)
Interrogator-to-Tag link modulation

H.1 Baseband waveforms, modulated RF, and detected waveforms

Figure H.1 shows R=>T baseband and modulated waveforms as generated by an Interrogator, and the corresponding waveforms envelope-detected by a Tag, for DSB- or SSB-ASK modulation, and for PR-ASK modulation.

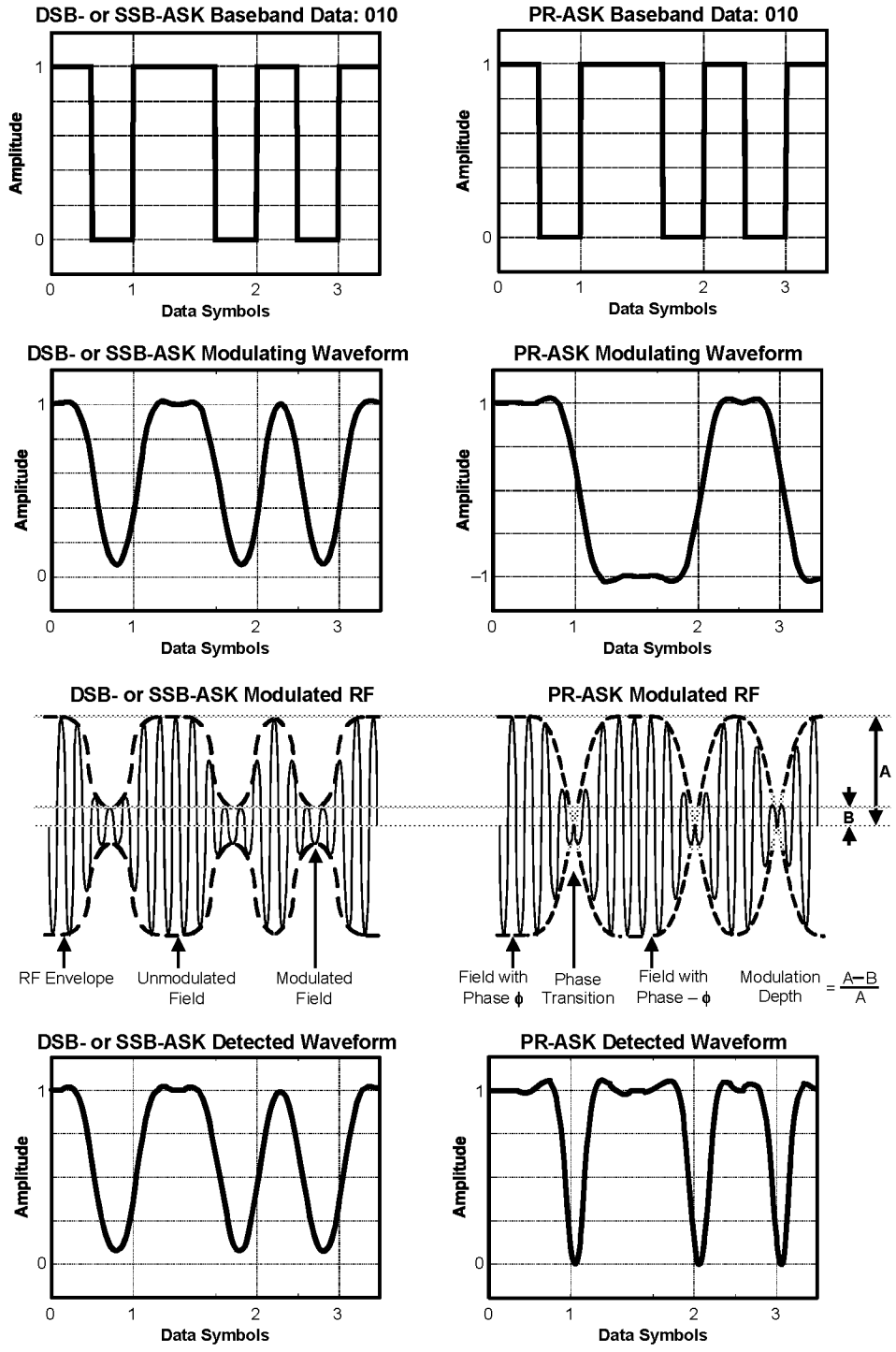


Figure H.1 – Interrogator-to-Tag modulation

Annex I
(Normative)
Error codes

I.1 Tag error codes and their usage

If a Tag encounters an error when executing an access command that reads from or writes to memory, and if the command is a handle-based command (i.e. *Read*, *Write*, *Kill*, *Lock*, *BlockWrite*, *BlockErase*, or *BlockPermalock*), then the Tag shall backscatter an error code as shown in Table I.1 instead of its normal reply.

- If the Tag supports error-specific codes, it shall use the error-specific codes shown in Table I.2.
- If the Tag does not support error-specific codes, it shall backscatter error code 00001111_2 (indicating a non-specific error) as shown in Table I.2.
- Tags shall backscatter error codes only from the **open** or **secured** states.
- A Tag shall not backscatter an error code if it receives an invalid access command; instead, it shall ignore the command.
- If an error is described by more than one error code, the more specific error code shall take precedence and shall be the code that the Tag backscatters.
- The header for an error code is a 1-bit, unlike the header for a normal Tag response, which is a 0-bit.

Table I.1 – Tag-error reply format

	Header	Error Code	RN	CRC-16
# of bits	1	8	16	16
description	1	Error code	handle	

Table I.2 – Tag error codes

Error-Code Support	Error Code	Error-Code Name	Error Description
Error-specific	00000000_2	Other error	Catch-all for errors not covered by other codes
	00000011_2	Memory overrun	The specified memory location does not exist or the EPC length field is not supported by the Tag
	00000100_2	Memory locked	The specified memory location is locked and/or permalocked and is either not writeable or not readable
	00001011_2	Insufficient power	The Tag has insufficient power to perform the memory-write operation
Non-specific	00001111_2	Non-specific error	The Tag does not support error-specific codes

Annex J
(normative)
Slot counter

J.1 Slot-counter operation

As described in 6.3.2.4.8, Tags implement a 15-bit slot counter. As described in 6.3.2.8, Interrogators use the slot counter to regulate the probability of a Tag responding to a *Query*, *QueryAdjust*, or *QueryRep* command. Upon receiving a *Query* or *QueryAdjust* a Tag preloads a *Q*-bit value, drawn from the Tag's RNG (see 6.3.2.5), into its slot counter. *Q* is an integer in the range (0, 15). A *Query* specifies *Q*; a *QueryAdjust* may modify *Q* from the prior *Query*.

A Tag in the **arbitrate** state shall decrement its slot counter every time it receives a *QueryRep* command, transitioning to the **reply** state and backscattering an RN16 when its slot-counter value reaches 0000_h. A Tag whose slot-counter value reached 0000_h, who replied, and who was not acknowledged (including a Tag that responded to the original *Query* and was not acknowledged) returns to **arbitrate** with a slot-counter value of 0000_h.

A Tag that returns to **arbitrate** with a slot-counter value of 0000_h shall decrement its slot-counter from 0000_h to $7FFF_h$ (i.e. the slot counter rolls over) at the next *QueryRep* with matching session. Because the slot-counter value is now nonzero, the Tag remains in **arbitrate**. Slot counters implements continuous counting, meaning that, after a slot counter rolls over it begins counting down again from $7FFF_h$, effectively preventing subsequent Tag replies until the Tag receives either a *Query* or a *QueryAdjust* and loads a new random value into its slot counter.

Annex B and Annex C contain tables describing a Tag's response to Interrogator commands; "slot" is a parameter in these tables.

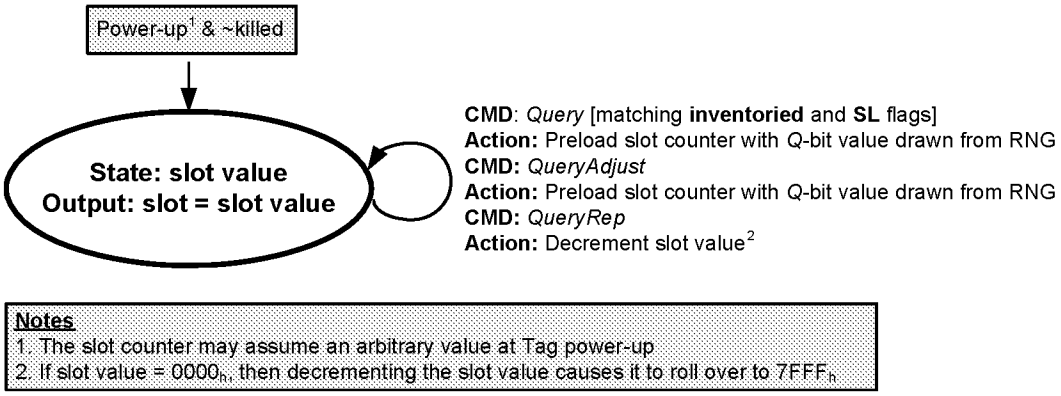


Figure J.1 – Slot-counter state diagram

Annex K
(informative)
Example data-flow exchange

K.1 Overview of the data-flow exchange

The following example describes a data exchange, between an Interrogator and a single Tag, during which the Interrogator reads the kill password stored in the Tag's Reserved memory. This example assumes that:

- The Tag has been singulated and is in the **acknowledged** state.
- The Tag's Reserved memory is locked but not permalocked, meaning that the Interrogator must issue the access password and transition the Tag to the **secured** state before performing the read operation.
- The random numbers the Tag generates (listed in sequence, and not random for reasons of clarity) are:
  - RN16_0 1600_h (the RN16 the Tag backscattered prior to entering **acknowledged**)
  - RN16_1 1601_h (will become the handle for the entire access sequence)
  - RN16_2 1602_h
  - RN16_3 1603_h
- The Tag's EPC is 64 bits in length.
- The Tag's access password is $ACCEC0DE_h$.
- The Tag's kill password is $DEADC0DE_h$.
- The 1^{st} half of the access password EXORed with RN16_2 = $ACCE_h \otimes 1602_h = BACC_h$.
- The 2^{nd} half of the access password EXORed with RN16_3 = $C0DE_h \otimes 1603_h = D6DD_h$.

K.2 Tag memory contents and lock-field values

Table K.1 and Table K.2 show the example Tag memory contents and lock-field values, respectively.

Table K.1 – Tag memory contents

Memory Bank	Memory Contents	Memory Addresses	Memory Values
TID	TID[15:0]	10_h–$1F_h$	$54E2_h$
	TID[31:16]	00_h–$0F_h$	$A986_h$
EPC	EPC[15:0]	50_h–$5F_h$	3210_h
	EPC[31:16]	40_h–$4F_h$	7654_h
	EPC[47:32]	30_h–$3F_h$	$BA98_h$
	EPC[63:48]	20_h–$2F_h$	$FEDC_h$
	StoredPC[15:0]	10_h–$1F_h$	2000_h
	StoredCRC[15:0]	00_h–$0F_h$	as calculated (see Annex F)
Reserved	access password[15:0]	30_h–$3F_h$	$C0DE_h$
	access password[31:16]	20_h–$2F_h$	$ACCE_h$
	kill password[15:0]	10_h–$1F_h$	$C0DE_h$
	kill password[31:16]	00_h–$0F_h$	$DEAD_h$

Table K.2 – Lock-field values

Kill Password		Access Password		EPC Memory		TID Memory		User Memory	
1	0	1	0	0	0	0	0	N/A	N/A

K.3 Data-flow exchange and command sequence

The data-flow exchange follows the *Access* procedure outlined in Figure 6.25 with a *Read* command added at the end. The sequence of Interrogator commands and Tag replies is:

Step 1: *Req_RN*[RN16_0, CRC-16]
Tag backscatters RN16_1, which becomes the handle for the entire access sequence

Step 2: *Req_RN*[handle, CRC-16]
Tag backscatters RN16_2

Step 3: *Access*[access password[31:16] EXORed with RN16_2, handle, CRC-16]
Tag backscatters handle

Step 4: *Req_RN*[handle, CRC-16]
Tag backscatters RN16_3

Step 5: *Access*[access password[15:0] EXORed with RN16_3, handle, CRC-16]
Tag backscatters handle

Step 6: *Read*[MemBank=Reserved, WordPtr=00_h, WordCount=2, handle, CRC-16]
Tag backscatters kill password

Table K.3 shows the detailed Interrogator commands and Tag replies. For reasons of clarity, the CRC-16 has been omitted from all commands and replies.

Table K.3 – Interrogator commands and Tag replies

Step	Data Flow	Command	Parameter and/or Data	Tag State
1a: *Req_RN* command	R => T	11000001	0001 0110 0000 0000 (RN16_0=1600_h)	**acknowledged → open**
1b: Tag response	T => R		0001 0110 0000 0001 (handle=1601_h)	
2a: *Req_RN* command	R => T	11000001	0001 0110 0000 0001 (handle=1601_h)	**open → open**
2b: Tag response	T => R		0001 0110 0000 0010 (RN16_2=1602_h)	
3a: *Access* command	R => T	11000110	1011 1010 1100 1100 ($BACC_h$) 0001 0110 0000 0001 (handle=1601_h)	**open → open**
3b: Tag response	T => R		0001 0110 0000 0001 (handle=1601_h)	
4a: *Req_RN* command	R => T	11000001	0001 0110 0000 0001 (handle=1601_h)	**open → open**
4b: Tag response	T => R		0001 0110 0000 0011 (RN16_2=1603_h)	
5a: *Access* command	R => T	11000110	1101 0110 1101 1101 ($D6DD_h$) 0001 0110 0000 0001 (handle=1601_h)	**open → secured**
5b: Tag response	T => R		0001 0110 0000 0001 (handle=1601_h)	
6a: *Read* command	R => T	11000010	00 (MemBank=Reserved) 00000000 (WordPtr=kill password) 00000010 (WordCount=2) 0001 0110 0000 0001 (handle=1601_h)	**secured → secured**
6b: Tag response	T => R		0 (header) 1101 1110 1010 1101 ($DEAD_h$) 1100 0000 1101 1110 ($C0DE_h$)	

Annex L
(informative)
Optional Tag Features

The following options are available to Tags certified to this protocol.

L.1 Optional Tag passwords

Kill password: A Tag may optionally implement a kill password. A Tag that does not implement a kill password operates as if it has a zero-valued kill password that is permanently read/write locked. See 6.3.2.1.1.1.

Access password: A Tag may optionally implement an access password. A Tag that does not implement an access password operates as if it has a zero-valued access password that is permanently read/write locked. See 6.3.2.1.1.2.

L.2 Optional Tag memory banks and memory-bank sizes

Reserved memory: Reserved memory is optional. If a Tag does not implement either a kill password or an access password then the Tag need not physically implement Reserved memory. Because a Tag with non-implemented passwords operates as if it has zero-valued password(s) that are permanently read/write locked, these passwords must still be logically addressable in Reserved memory at the memory locations specified in 6.3.2.1.1.1 and 6.3.2.1.1.2.

EPC memory: EPC memory is required, but its size is vendor-defined. The minimum size is 32 bits, to contain a 16-bit StoredCRC and a 16-bit StoredPC. EPC memory may be larger than 32 bits, to contain an EPC whose vendor-specified length may be 16 to 496 bits (if a Tag does not support XPC functionality) or to 464 bits (if a Tag supports XPC functionality) in 16-bit increments, as well as an optional XPC word or words. See 6.3.2.1.2.

TID memory: TID memory is required, but its size is vendor-defined. The minimum-size TID memory contains an 8-bit ISO/IEC 15963 allocation class identifier, as well as sufficient identifying information for an Interrogator to uniquely identify the custom commands and/or optional features that a Tag supports. TID memory may optionally contain vendor-specific data. See 6.3.2.1.3.

User memory: User memory is optional. See 6.3.2.1.4, 6.3.2.1.4.1, and 6.3.2.1.4.2.

L.3 Optional Tag commands

***Proprietary*:** A Tag may support proprietary commands. See 2.3.3.

***Custom*:** A Tag may support custom commands. See 2.3.4.

***Access*:** A Tag may support the *Access* command. See 6.3.2.11.3.6.

***BlockWrite*:** A Tag may support the *BlockWrite* command. See 6.3.2.11.3.7.

***BlockErase*:** A Tag may support the *BlockErase* command. See 6.3.2.11.3.8.

***BlockPermalock*:** A Tag may support the *BlockPermalock* command. See 6.3.2.11.3.9.

L.4 Optional Tag error-code reporting format

A Tag may support error-specific or non-error-specific error-code reporting. See Annex I.

L.5 Optional Tag backscatter modulation format

A Tag may support ASK and/or PSK backscatter modulation. See 6.3.1.3.1.

L.6 Optional Tag functionality

A Tag may implement the UMI by one of two methods. See 6.3.2.1.2.2.

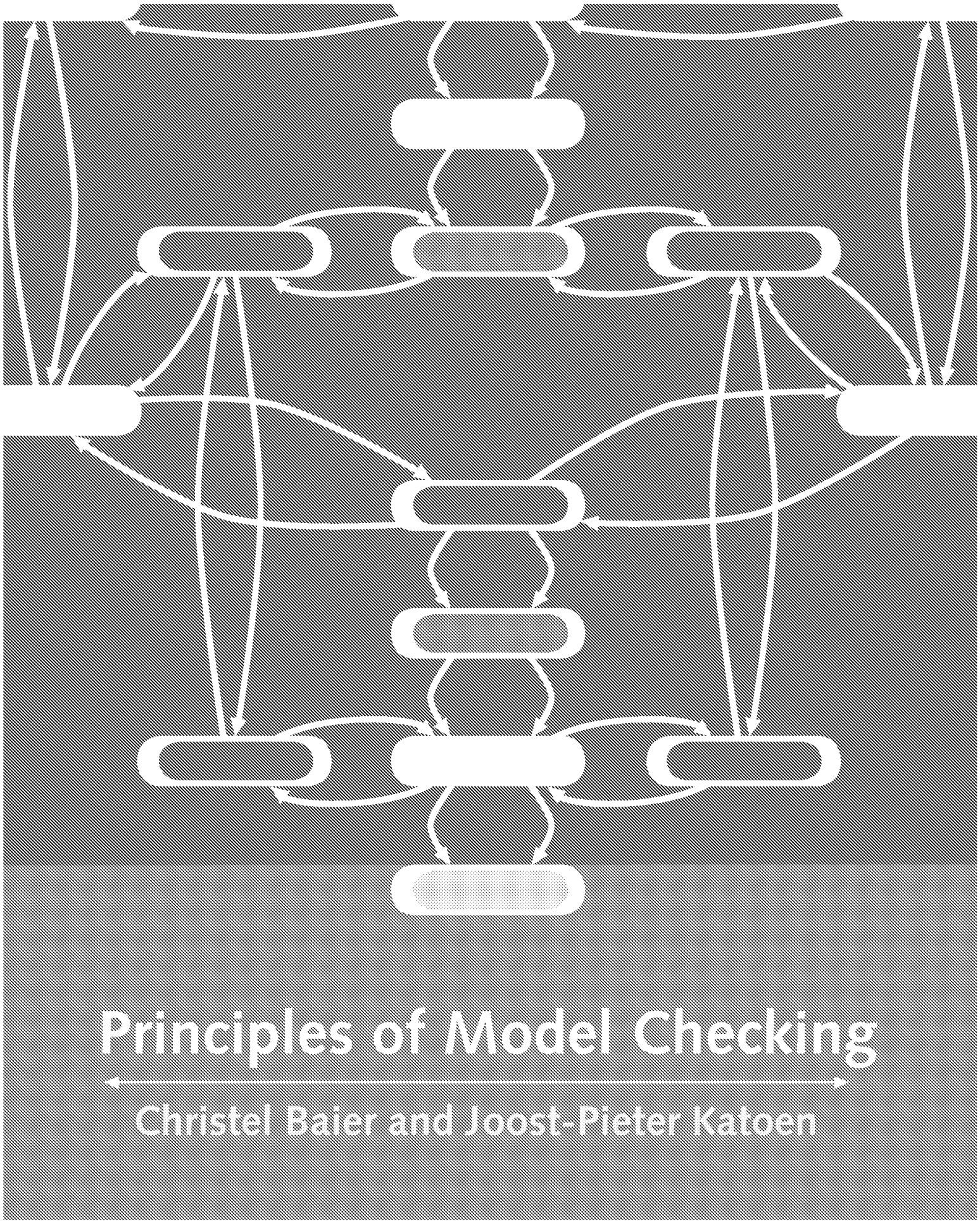
A Tag may implement an XPC_W1, XPC_W2, XI, and XEB. See 6.3.2.1.2.2 and 6.3.2.1.2.5.

A Tag may implement recommissioning. See 6.3.2.1.2.5, 6.3.2.10, and 6.3.2.11.3.4.

Annex M
(informative)
Revision History

Table M.1 – Revision history

Date & Version Number	Section(s)	Change	Approved by
Sept 8, 2004 Version 1.0.4	All	Modified Chicago protocol V1.0.3 as per August 17, 2004 "combo" CRC change template.	
Sept 14, 2004 Version 1.0.5	All	Modified Gen2 protocol V1.0.4 as per September 10, 2004 CRC review.	
Sept 17, 2004 Version 1.0.6	All	Modified Gen2 protocol V1.0.5 as per September 17, 2004 HAG review.	
Sept 24, 2004 Version 1.0.7	All	Modified Gen2 protocol V1.0.6 as per September 21, 2004 CRC review to fix errata. Changed OID to EPC.	
Dec 11, 2004 Version 1.0.8	Multiple	Modified Gen2 protocol V1.0.7 as per the V1.0.7 errata.	
Jan 26, 2005 Version 1.0.9	Multiple	Modified Gen2 protocol V1.0.8 as per the V1.0.8 errata and AFI enhancement requests	
Dec 1, 2005 Version 1.1.0	Multiple	Harmonized Gen2 protocol V1.0.9 with the ISO 18000-6 Type C amendment.	
May 11, 2008 Version 1.2.0	Multiple	Modified Gen2 protocol V1.1.0 to satisfy the ILT JRG requirements V1.2.3.	



Principles of Model Checking

Christel Baier and Joost-Pieter Katoen

Principles of Model Checking

Principles of Model Checking

Christel Baier
Joost-Pieter Katoen

The MIT Press
Cambridge, Massachusetts
London, England

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email special_sales@mitpress.mit.edu or write to Special Sales Department, The MIT Press, 55 Hayward Street, Cambridge, MA 02142.

This book was set in Aachen and Dresden by Christel Baier and Joost-Pieter Katoen. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Baier, Christel.
Principles of model checking / Christel Baier and Joost-Pieter Katoen ; foreword by Kim Guldstrand Larsen.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-262-02649-9 (hardcover : alk. paper) 1. Computer systems–Verification. 2. Computer software–Verification. I.
Katoen, Joost-Pieter. II. Title.
QA76.76.V47B35 2008
004.2'4–dc22

2007037603

10 9 8 7 6 5 4 3 2 1

To Michael, Gerda, Inge, and Karl

To Erna, Fons, Joost, and Tom

Contents

Foreword

Society is increasingly dependent on dedicated computer and software systems to assist us in almost every aspect of daily life. Often we are not even aware that computers and software are involved. Several control functions in modern cars are based on embedded software solutions, e.g., braking, airbags, cruise control, and fuel injection. Mobile phones, communication systems, medical devices, audio and video systems, and consumer electronics in general are containing vast amounts of software. Also transport, production, and control systems are increasingly applying embedded software solutions to gain flexibility and cost-efficiency.

A common pattern is the constantly increasing complexity of systems, a trend which is accelerated by the adaptation of wired and wireless networked solutions: in a modern car the control functions are distributed over several processing units communicating over dedicated networks and buses. Yet computer- and software-based solutions are becoming ubiquitous and are to be found in several safety-critical systems. Therefore a main challenge for the field of computer science is to provide formalisms, techniques, and tools that will enable the efficient design of correct and well-functioning systems despite their complexity.

Over the last two decades or so a very attractive approach toward the correctness of computer-based control systems is that of model checking. Model checking is a formal verification technique which allows for desired behavioral properties of a given system to be verified on the basis of a suitable model of the system through systematic inspection of all states of the model. The attractiveness of model checking comes from the fact that it is completely automatic – i.e., the learning curve for a user is very gentle – and that it offers counterexamples in case a model fails to satisfy a property serving as indispensable debugging information. On top of this, the performance of model-checking tools has long since proved mature as witnessed by a large number of successful industrial applications.

It is my pleasure to recommend the excellent book *Principles of Model Checking* by Christel Baier and Joost-Pieter Katoen as the definitive textbook on model checking, providing both a comprehensive and a comprehensible account of this important topic. The book contains detailed and complete descriptions of first principles of classical Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) model checking. Also, state-of-the art methods for coping with state-space explosion, including symbolic model checking, abstraction and minimization techniques, and partial order reduction, are fully accounted for. The book also covers model checking of real-time and probabilistic systems, important new directions for model checking in which the authors, being two of the most industrious and creative researchers of today, are playing a central role.

The exceptional pedagogical style of the authors provides careful explanations of constructions and proofs, plus numerous examples and exercises of a theoretical, practical and tool-oriented nature. The book will therefore be the ideal choice as a textbook for both graduate and advanced undergraduate students, as well as for self-study, and should definitely be on the bookshelf of any researcher interested in the topic.

Kim Guldstrand Larsen
Professor in Computer Science
Aalborg University, Denmark
May 2007

Preface

It is fair to state, that in this digital era
correct systems for information processing
are more valuable than gold.

(H. Barendregt. The quest for correctness.
In: Images of SMC Research 1996, pages 39–58, 1996.)

This book is on model checking, a prominent formal verification technique for assessing functional properties of information and communication systems. Model checking requires a model of the system under consideration and a desired property and systematically checks whether or not the given model satisfies this property. Typical properties that can be checked are deadlock freedom, invariants, and request-response properties. Model checking is an automated technique to check the absence of errors (i.e., property violations) and alternatively can be considered as an intelligent and effective debugging technique. It is a general approach and is applied in areas like hardware verification and software engineering. Due to unremitting improvements of underlying algorithms and data structures together with hardware technology improvements, model-checking techniques that two decades ago only worked for simple examples are nowadays applicable to more realistic designs. It is fair to say that in the last two decades model checking has developed as a mature and heavily used verification and debugging technique.

Aims and Scope

This book attempts to introduce model checking from first principles, so to speak, and is intended as a textbook for bachelor and master students, as well as an introductory book for researchers working in other areas of computer science or related fields. The reader is introduced to the material by means of an extensive set of examples, most of which are examples running throughout several chapters. The book provides a complete set of basic results together with all detailed proofs. Each chapter is concluded by a summary,

bibliographic notes, and a series of exercises, of both a theoretical and of a practical nature (i.e., experimenting with actual model checkers).

Prerequisites

The concepts of model checking have their roots in mathematical foundations such as propositional logic, automata theory and formal languages, data structures, and graph algorithms. It is expected that readers are familiar with the basics of these topics when starting with our book, although an appendix is provided that summarizes the essentials. Knowledge on complexity theory is required for the theoretical complexity considerations of the various model-checking algorithms.

Content

This book is divided into ten chapters. Chapter 1 motivates and introduces model checking. Chapter 2 presents transition systems as a model for software and hardware systems. Chapter 3 introduces a classification of linear-time properties into safety and liveness, and presents the notion of fairness. Automata-based algorithms for checking (regular) safety and ω-regular properties are presented in Chapter 4. Chapter 5 deals with Linear Temporal Logic (LTL) and shows how the algorithms of Chapter 4 can be used for LTL model checking. Chapter 6 introduces the branching-time temporal logic Computation Tree Logic (CTL), compares this to LTL, and shows how to perform CTL model checking, both explicitly and symbolically. Chapter 7 deals with abstraction mechanisms that are based on trace, bisimulation, and simulation relations. Chapter 8 treats partial-order reduction for LTL and CTL. Chapter 9 is focused on real-time properties and timed automata, and the monograph is concluded with a chapter on the verification of probabilistic models. The appendix summarizes basic results on propositional logic, graphs, language, and complexity theory.

How to Use This Book

A natural plan for an introductory course into model checking that lasts one semester (two lectures a week) comprises Chapters 1 through 6. A follow-up course of about a semester could cover Chapters 7 through 10, after a short refresher on LTL and CTL model checking.

Acknowledgments

This monograph has been developed and extended during the last five years. The following colleagues supported us by using (sometimes very) preliminary versions of this monograph: Luca Aceto (Aalborg, Denmark and Reykjavik, Iceland), Henrik Reif Andersen (Copenhagen, Denmark), Dragan Boshnacki (Eindhoven, The Netherlands), Franck van Breughel (Ottawa, Canada), Josée Desharnais (Quebec, Canada), Susanna Donatelli (Turin, Italy), Stefania Gnesi (Pisa, Italy), Michael R. Hansen (Lyngby, Denmark), Holger Hermanns (Saarbrücken, Germany), Yakov Kesselman (Chicago, USA), Martin Lange (Aarhus, Denmark), Kim G. Larsen (Aalborg, Denmark), Mieke Massink (Pisa, Italy), Mogens Nielsen (Aarhus, Denmark), Albert Nymeyer (Sydney, Australia), Andreas Podelski (Freiburg, Germany), Theo C. Ruys (Twente, The Netherlands), Thomas Schwentick (Dortmund, Germany), Wolfgang Thomas (Aachen, Germany), Julie Vachon (Montreal, Canada), and Glynn Winskel (Cambridge, UK). Many of you provided us with very helpful feedback that helped us to improve the lecture notes.

Henrik Bohnenkamp, Tobias Blechmann, Frank Ciesinski, Marcus Grösser, Tingting Han, Joachim Klein, Sascha Klüppelholz, Miriam Nasfi, Martin Neuhäusser, and Ivan S. Zapreev provided us with many detailed comments, and provided several exercises. Yen Cao is kindly thanked for drawing a part of the figures and Ulrich Schmidt-Görtz for his assistance with the bibliography.

Many people have suggested improvements and pointed out mistakes. We thank everyone for providing us with helpful comments.

Finally, we thank all our students in Aachen, Bonn, Dresden, and Enschede for their feedback and comments.

Christel Baier

Joost-Pieter Katoen

Chapter 1

System Verification

Our reliance on the functioning of ICT systems (Information and Communication Technology) is growing rapidly. These systems are becoming more and more complex and are massively encroaching on daily life via the Internet and all kinds of embedded systems such as smart cards, hand-held computers, mobile phones, and high-end television sets. In 1995 it was estimated that we are confronted with about 25 ICT devices on a daily basis. Services like electronic banking and teleshopping have become reality. The daily cash flow via the Internet is about 10^{12} million US dollars. Roughly 20% of the product development costs of modern transportation devices such as cars, high-speed trains, and airplanes is devoted to information processing systems. ICT systems are universal and omnipresent. They control the stock exchange market, form the heart of telephone switches, are crucial to Internet technology, and are vital for several kinds of medical systems. Our reliance on embedded systems makes their reliable operation of large social importance. Besides offering a good performance in terms like response times and processing capacity, the absence of annoying errors is one of the major quality indications.

It is all about money. We are annoyed when our mobile phone malfunctions, or when our video recorder reacts unexpectedly and wrongly to our issued commands. These software and hardware errors do not threaten our lives, but may have substantial financial consequences for the manufacturer. Correct ICT systems are essential for the survival of a company. Dramatic examples are known. The bug in Intel's Pentium II floating-point division unit in the early nineties caused a loss of about 475 million US dollars to replace faulty processors, and severely damaged Intel's reputation as a reliable chip manufacturer. The software error in a baggage handling system postponed the opening of Denver's airport for 9 months, at a loss of 1.1 million US dollar per day. Twenty-four hours of failure of

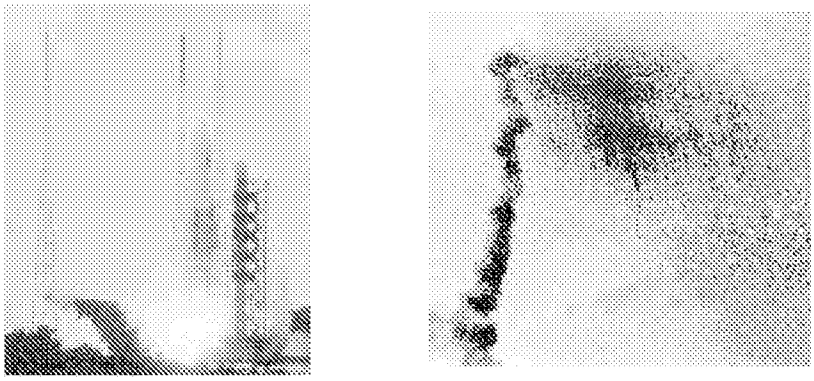


Figure 1.1: The Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value.

the worldwide online ticket reservation system of a large airplane company will cause its bankruptcy because of missed orders.

It is all about safety: errors can be catastrophic too. The fatal defects in the control software of the Ariane-5 missile (Figure 1.1), the Mars Pathfinder, and the airplanes of the Airbus family led to headlines in newspapers all over the world and are notorious by now. Similar software is used for the process control of safety-critical systems such as chemical plants, nuclear power plants, traffic control and alert systems, and storm surge barriers. Clearly, bugs in such software can have disastrous consequences. For example, a software flaw in the control part of the radiation therapy machine Therac-25 caused the death of six cancer patients between 1985 and 1987 as they were exposed to an overdose of radiation.

The increasing reliance of critical applications on information processing leads us to state:

The reliability of ICT systems is a key issue in the system design process.

The magnitude of ICT systems, as well as their complexity, grows apace. ICT systems are no longer standalone, but are typically embedded in a larger context, connecting and interacting with several other components and systems. They thus become much more vulnerable to errors – the number of defects grows exponentially with the number of interacting system components. In particular, phenomena such as concurrency and nondeterminism that are central to modeling interacting systems turn out to be very hard to handle with standard techniques. Their growing complexity, together with the pressure to drastically reduce system development time ("time-to-market"), makes the delivery of low-defect ICT systems an enormously challenging and complex activity.

Hard- and Software Verification

System verification techniques are being applied to the design of ICT systems in a more reliable way. Briefly, system verification is used to establish that the design or product under consideration possesses certain properties. The properties to be validated can be quite elementary, e.g., a system should never be able to reach a situation in which no progress can be made (a deadlock scenario), and are mostly obtained from the *system's specification*. This specification prescribes what the system has to do and what not, and thus constitutes the basis for any verification activity. A defect is found once the system does not fulfill one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system. A schematic view of verification is depicted in Figure 1.2.

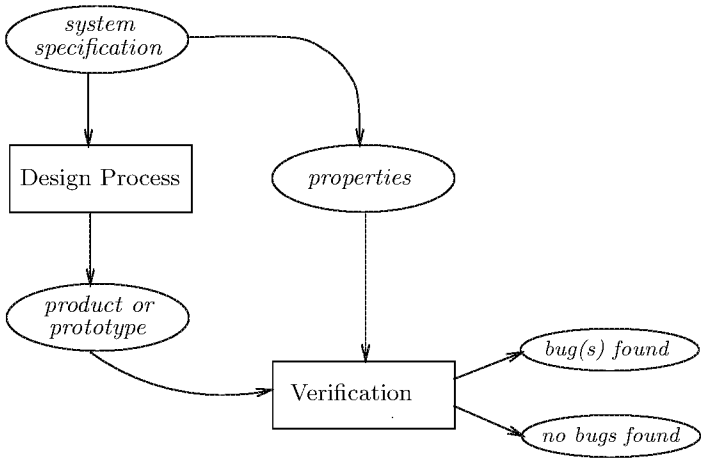


Figure 1.2: Schematic view of an a posteriori system verification.

This book deals with a verification technique called model checking that starts from a formal system specification. Before introducing this technique and discussing the role of formal specifications, we briefly review alternative software and hardware verification techniques.

Software Verification Peer reviewing and testing are the major software verification techniques used in practice.

A *peer review* amounts to a software inspection carried out by a team of software engineers that preferably has not been involved in the development of the software under review. The

uncompiled code is not executed, but analyzed completely statically. Empirical studies indicate that peer review provides an effective technique that catches between 31 % and 93 % of the defects with a median around 60%. While mostly applied in a rather ad hoc manner, more dedicated types of peer review procedures, e.g., those that are focused at specific error-detection goals, are even more effective. Despite its almost complete manual nature, peer review is thus a rather useful technique. It is therefore not surprising that some form of peer review is used in almost 80% of all software engineering projects. Due to its static nature, experience has shown that subtle errors such as concurrency and algorithm defects are hard to catch using peer review.

Software testing constitutes a significant part of any software engineering project. Between 30% and 50% of the total software project costs are devoted to testing. As opposed to peer review, which analyzes code statically without executing it, testing is a dynamic technique that actually runs the software. Testing takes the piece of software under consideration and provides its compiled code with inputs, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. Based on the observations during test execution, the actual output of the software is compared to the output as documented in the system specification. Although test generation and test execution can partly be automated, the comparison is usually performed by human beings. The main advantage of testing is that it can be applied to all sorts of software, ranging from application software (e.g., e-business software) to compilers and operating systems. As exhaustive testing of all execution paths is practically infeasible; in practice only a small subset of these paths is treated. Testing can thus never be complete. That is to say, testing can only show the presence of errors, not their absence. Another problem with testing is to determine when to stop. Practically, it is hard, and mostly impossible, to indicate the intensity of testing to reach a certain defect density – the fraction of defects per number of uncommented code lines.

Studies have provided evidence that peer review and testing catch different classes of defects at different stages in the development cycle. They are therefore often used together. To increase the reliability of software, these software verification approaches are complemented with software process improvement techniques, structured design and specification methods (such as the Unified Modeling Language), and the use of version and configuration management control systems. Formal techniques are used, in one form or another, in about 10 % to 15% of all software projects. These techniques are discussed later in this chapter.

Catching software errors: the sooner the better. It is of great importance to locate software bugs. The slogan is: the sooner the better. The costs of repairing a software flaw during maintenance are roughly 500 times higher than a fix in an early design phase (see Figure 1.3). System verification should thus take place early stage in the design process.

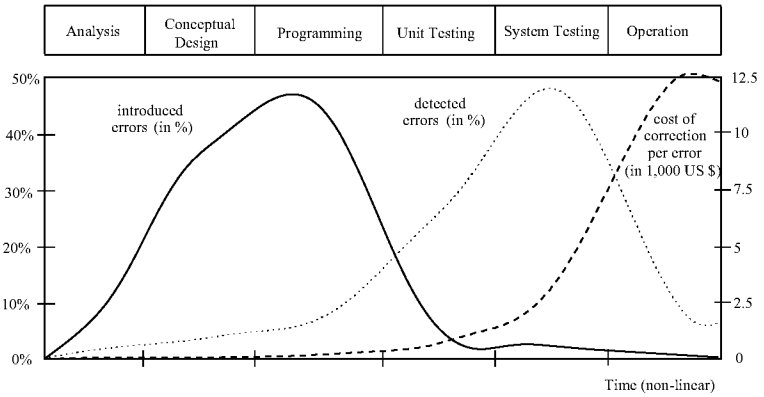


Figure 1.3: Software lifecycle and error introduction, detection, and repair costs [275].

About 50% of all defects are introduced during programming, the phase in which actual coding takes place. Whereas just 15% of all errors are detected in the initial design stages, most errors are found during testing. At the start of unit testing, which is oriented to discovering defects in the individual software modules that make up the system, a defect density of about 20 defects per 1000 lines of (uncommented) code is typical. This has been reduced to about 6 defects per 1000 code lines at the start of system testing, where a collection of such modules that constitutes a real product is tested. On launching a new software release, the typical accepted software defect density is about one defect per 1000 lines of code lines[1].

Errors are typically concentrated in a few software modules – about half of the modules are defect free, and about 80% of the defects arise in a small fraction (about 20%) of the modules – and often occur when interfacing modules. The repair of errors that are detected prior to testing can be done rather economically. The repair cost significantly increases from about $ 1000 (per error repair) in unit testing to a maximum of about $ 12,500 when the defect is demonstrated during system operation only. It is of vital importance to seek techniques that find defects as early as possible in the software design process: the costs to repair them are substantially lower, and their influence on the rest of the design is less substantial.

Hardware Verification Preventing errors in hardware design is vital. Hardware is subject to high fabrication costs; fixing defects after delivery to customers is difficult, and quality expectations are high. Whereas software defects can be repaired by providing

[1] For some products this is much higher, though. Microsoft has acknowledged that Windows 95 contained at least 5000 defects. Despite the fact that users were daily confronted with anomalous behavior, Windows 95 was very successful.

users with patches or updates – nowadays users even tend to anticipate and accept this – hardware bug fixes after delivery to customers are very difficult and mostly require refabrication and redistribution. This has immense economic consequences. The replacement of the faulty Pentium II processors caused Intel a loss of about $ 475 million. Moore's law – the number of logical gates in a circuit doubles every 18 months – has proven to be true in practice and is a major obstacle to producing correct hardware. Empirical studies have indicated that more than 50% of all ASICs (Application-Specific Integrated Circuits) do not work properly after initial design and fabrication. It is not surprising that chip manufacturers invest a lot in getting their designs right. Hardware verification is a well-established part of the design process. The design effort in a typical hardware design amounts to only 27% of the total time spent on the chip; the rest is devoted to error detection and prevention.

Hardware verification techniques. Emulation, simulation, and structural analysis are the major techniques used in hardware verification.

Structural analysis comprises several specific techniques such as synthesis, timing analysis, and equivalence checking that are not described in further detail here.

Emulation is a kind of testing. A reconfigurable generic hardware system (the emulator) is configured such that it behaves like the circuit under consideration and is then extensively tested. As with software testing, emulation amounts to providing a set of stimuli to the circuit and comparing the generated output with the expected output as laid down in the chip specification. To fully test the circuit, all possible input combinations in every possible system state should be examined. This is impractical and the number of tests needs to be reduced significantly, yielding potential undiscovered errors.

With *simulation*, a model of the circuit at hand is constructed and simulated. Models are typically provided using hardware description languages such as Verilog or VHDL that are both standardized by IEEE. Based on stimuli, execution paths of the chip model are examined using a simulator. These stimuli may be provided by a user, or by automated means such as a random generator. A mismatch between the simulator's output and the output described in the specification determines the presence of errors. Simulation is like testing, but is applied to models. It suffers from the same limitations, though: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice.

Simulation is the most popular hardware verification technique and is used in various design stages, e.g., at register-transfer level, gate and transistor level. Besides these error detection techniques, *hardware testing* is needed to find fabrication faults resulting from layout defects in the fabrication process.

1.1 Model Checking

In software and hardware design of complex systems, more time and effort are spent on verification than on construction. Techniques are sought to reduce and ease the verification efforts while increasing their coverage. Formal methods offer a large potential to obtain an early integration of verification in the design process, to provide more effective verification techniques, and to reduce the verification time.

Let us first briefly discuss the role of formal methods. To put it in a nutshell, formal methods can be considered as "the applied mathematics for modeling and analyzing ICT systems". Their aim is to establish system correctness with mathematical rigor. Their great potential has led to an increasing use by engineers of formal methods for the verification of complex software and hardware systems. Besides, formal methods are one of the "highly recommended" verification techniques for software development of safety-critical systems according to, e.g., the best practices standard of the IEC (International Electrotechnical Commission) and standards of the ESA (European Space Agency). The resulting report of an investigation by the FAA (Federal Aviation Authority) and NASA (National Aeronautics and Space Administration) about the use of formal methods concludes that

> *Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers.*

During the last two decades, research in formal methods has led to the development of some very promising verification techniques that facilitate the early detection of defects. These techniques are accompanied by powerful software tools that can be used to automate various verification steps. Investigations have shown that formal verification procedures would have revealed the exposed defects in, e.g., the Ariane-5 missile, Mars Pathfinder, Intel's Pentium II processor, and the Therac-25 therapy radiation machine.

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. It turns out that – prior to any form of verification – the accurate modeling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications. Such problems are usually only discovered at a much later stage of the design. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing). Due to unremitting improvements of un-

derlying algorithms and data structures, together with the availability of faster computers and larger computer memories, model-based techniques that a decade ago only worked for very simple examples are nowadays applicable to realistic designs. As the startingpoint of these techniques is a model of the system under consideration, we have as a given fact that

> *Any verification using model-based techniques is only as good as the model of the system.*

Model checking is a verification technique that explores all possible system states in a brute-force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. It is a real challenge to examine the largest possible state spaces that can be treated with current means, i.e., processors and memories. State-of-the-art model checkers can handle state spaces of about 10^8 to 10^9 states with explicit state-space enumeration. Using clever algorithms and tailored data structures, larger state spaces (10^{20} up to even 10^{476} states) can be handled for specific problems. Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

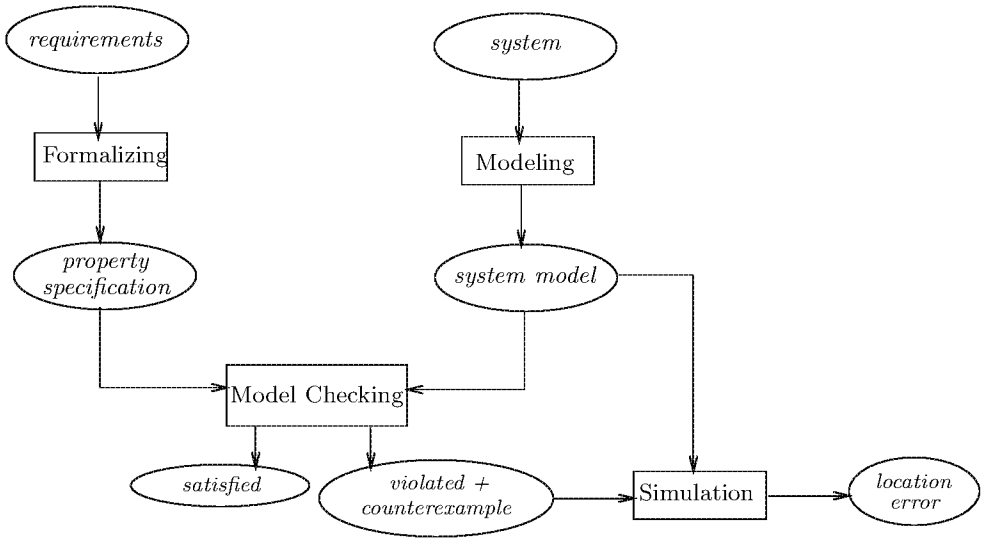


Figure 1.4: Schematic view of the model-checking approach.

Typical properties that can be checked using model checking are of a qualitative nature: Is the generated result OK?, Can the system reach a deadlock situation, e.g., when two

concurrent programs are waiting for each other and thus halting the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset?, or, Is a response always received within 8 minutes? Model checking requires a precise and unambiguous statement of the properties to be examined. As with making an accurate system model, this step often leads to the discovery of several ambiguities and inconsistencies in the informal documentation. For instance, the formalization of all system properties for a subset of the ISDN user part protocol revealed that 55% (!) of the original, informal system requirements were inconsistent.

The system model is usually automatically generated from a model description that is specified in some appropriate dialect of programming languages like C or Java or hardware description languages such as Verilog or VHDL. Note that the property specification prescribes *what* the system should do, and what it should not do, whereas the model description addresses *how* the system behaves. The model checker examines all relevant system states to check whether they satisfy the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information, and adapt the model (or the property) accordingly (see Figure 1.4).

Model checking has been successfully applied to several ICT systems and their applications. For instance, deadlocks have been detected in online airline reservation systems, modern e-commerce protocols have been verified, and several studies of international IEEE standards for in-house communication of domestic appliances have led to significant improvements of the system specifications. Five previously undiscovered errors were identified in an execution module of the Deep Space 1 spacecraft controller (see Figure 1.5), in one case identifying a major design flaw. A bug identical to one discovered by model checking escaped testing and caused a deadlock during a flight experiment 96 million km from earth. In the Netherlands, model checking has revealed several serious design flaws in the control software of a storm surge barrier that protects the main port of Rotterdam against flooding.

Example 1.1. Concurrency and Atomicity

Most errors, such as the ones exposed in the Deep Space-1 spacecraft, are concerned with classical concurrency errors. Unforeseen interleavings between processes may cause undesired events to happen. This is exemplified by analysing the following concurrent program, in which three processes, Inc, Dec, and Reset, cooperate. They operate on the shared integer variable x with arbitrary initial value that can be accessed (i.e., read), and

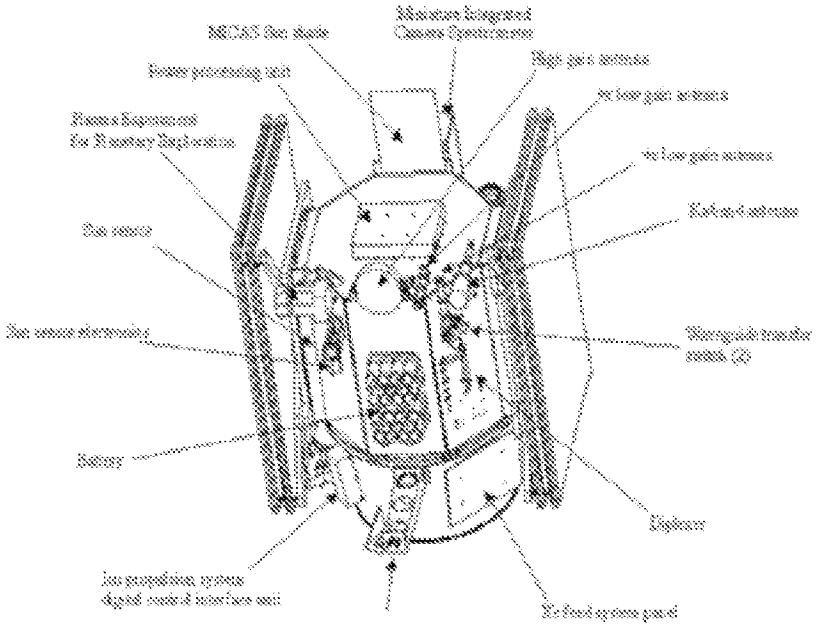


Figure 1.5: Modules of NASA's Deep Space-1 space-craft (launched in October 1998) have been thoroughly examined using model checking.

modified (i.e., written) by each of the individual processes. The processes are

$$\begin{aligned}
\textbf{proc}\ \mathsf{Inc} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x < 200\ \textbf{then}\ x := x + 1\ \textbf{fi}\ \textbf{od} \\
\textbf{proc}\ \mathsf{Dec} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x > 0\ \textbf{then}\ x := x - 1\ \textbf{fi}\ \textbf{od} \\
\textbf{proc}\ \mathsf{Reset} &= \textbf{while}\ \text{true}\ \textbf{do}\ \textbf{if}\ x = 200\ \textbf{then}\ x := 0\ \textbf{fi}\ \textbf{od}
\end{aligned}$$

Process Inc increments x if its value is smaller than 200, Dec decrements x if its value is at least 1, and Reset resets x once it has reached the value 200. They all do so repetitively.

Is the value of x always between (and including) 0 and 200? At first sight this seems to be true. A more thorough inspection, though, reveals that this is not the case. Suppose x equals 200. Process Dec tests the value of x, and passes the test, as x exceeds 0. Then, control is taken over by process Reset. It tests the value of x, passes its test, and immediately resets x to zero. Then, control is returned to process Dec and this process decrements x by one, resulting in a negative value for x (viz. -1). Intuitively, we tend to interpret the tests on x and the assignments to x as being executed atomically, i.e., as a single step, whereas in reality this is (mostly) not the case. ■

1.2 Characteristics of Model Checking

This book is devoted to the principles of model checking:

> *Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.*

The next chapters treat the elementary technical details of model checking. This section describes the process of model checking (how to use it), presents its main advantages and drawbacks, and discusses its role in the system development cycle.

1.2.1 The Model-Checking Process

In applying model checking to a design the following different phases can be distinguished:

- *Modeling* phase:
  - model the system under consideration using the model description language of the model checker at hand;
  - as a first sanity check and quick assessment of the model perform some simulations;
  - formalize the property to be checked using the property specification language.
- *Running* phase: run the model checker to check the validity of the property in the system model.
- *Analysis* phase:
  - property satisfied? → check next property (if any);
  - property violated? →
    1. analyze generated counterexample by simulation;
    2. refine the model, design, or property;
    3. repeat the entire procedure.
  - out of memory? → try to reduce the model and try again.

In addition to these steps, the entire verification should be planned, administered, and organized. This is called *verification organization*. We discuss these phases of model checking in somewhat more detail below.

Modeling The prerequisite inputs to model checking are a model of the system under consideration and a formal characterization of the property to be checked.

Models of systems describe the behavior of systems in an accurate and unambiguous way. They are mostly expressed using *finite-state automata*, consisting of a finite set of states and a set of transitions. States comprise information about the current values of variables, the previously executed statement (e.g., a program counter), and the like. Transitions describe how the system evolves from one state into another. For realistic systems, finite-state automata are described using a model description language such as an appropriate dialect/extension of C, Java, VHDL, or the like. Modeling systems, in particular concurrent ones, at the right abstraction level is rather intricate and is really an art; it is treated in more detail in Chapter 2.

In order to improve the quality of the model, a simulation prior to the model checking can take place. Simulation can be used effectively to get rid of the simpler category of modeling errors. Eliminating these simpler errors before any form of thorough checking takes place may reduce the costly and time-consuming verification effort.

To make a rigorous verification possible, properties should be described in a precise and unambiguous manner. This is typically done using a property specification language. We focus in particular on the use of a *temporal logic* as a property specification language, a form of modal logic that is appropriate to specify relevant properties of ICT systems. In terms of mathematical logic, one checks that the system description is a model of a temporal logic formula. This explains the term "model checking". Temporal logic is basically an extension of traditional propositional logic with operators that refer to the behavior of systems over time. It allows for the specification of a broad range of relevant system properties such as functional correctness (does the system do what it is supposed to do?), reachability (is it possible to end up in a deadlock state?), safety ("something bad never happens"), liveness ("something good will eventually happen"), fairness (does, under certain conditions, an event occur repeatedly?), and real-time properties (is the system acting in time?).

Although the aforementioned steps are often well understood, in practice it may be a serious problem to judge whether the formalized problem statement (model + properties) is an adequate description of the actual verification problem. This is also known as the *validation* problem. The complexity of the involved system, as well as the lack of precision

of the informal specification of the system's functionality, may make it hard to answer this question satisfactorily. Verification and validation should not be confused. Verification amounts to check that the design satisfies the requirements that have been identified, i.e., verification is "check that we are building the thing right". In validation, it is checked whether the formal model is consistent with the informal conception of the design, i.e., validation is "check that we are verifying the right thing".

Running the Model Checker The model checker first has to be initialized by appropriately setting the various options and directives that may be used to carry out the exhaustive verification. Subsequently, the actual model checking takes place. This is basically a solely algorithmic approach in which the validity of the property under consideration is checked in all states of the system model.

Analyzing the Results There are basically three possible outcomes: the specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.

In case the property is valid, the following property can be checked, or, in case all properties have been checked, the model is concluded to possess all desired properties.

Whenever a property is falsified, the negative result may have different causes. There may be a *modeling error*, i.e., upon studying the error it is discovered that the model does not reflect the design of the system. This implies a correction of the model, and verification has to be restarted with the improved model. This reverification includes the verification of those properties that were checked before on the erroneous model and whose verification may be invalidated by the model correction! If the error analysis shows that there is no undue discrepancy between the design and its model, then either a *design error* has been exposed, or a *property error* has taken place. In case of a design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. It may be the case that upon studying the exposed error it is discovered that the property does not reflect the informal requirement that had to be validated. This implies a modification of the property, and a new verification of the model has to be carried out. As the model is not changed, no reverification of properties that were checked before has to take place. The design is verified if and only if all properties have been checked with respect to a valid model.

Whenever the model is too large to be handled – state spaces of real-life systems may be many orders of magnitude larger than what can be stored by currently available memories – there are various ways to proceed. A possibility is to apply techniques that try to exploit

implicit regularities in the structure of the model. Examples of these techniques are the representation of state spaces using symbolic techniques such as binary decision diagrams or partial order reduction. Alternatively, rigorous abstractions of the complete system model are used. These abstractions should preserve the (non-)validity of the properties that need to be checked. Often, abstractions can be obtained that are sufficiently small with respect to a single property. In that case, different abstractions need to be made for the model at hand. Another way of dealing with state spaces that are too large is to give up the precision of the verification result. The probabilistic verification approaches explore only part of the state space while making a (often negligible) sacrifice in the verification coverage. The most important state-space reduction strategies are discussed in Chapters 7 through 9 of this monograph.

Verification Organization The entire model-checking process should be well organized, well structured, and well planned. Industrial applications of model checking have provided evidence that the use of version and configuration management is of particular relevance. During the verification process, for instance, different model descriptions are made describing different parts of the system, various versions of the verification models are available (e.g., due to abstraction), and plenty of verification parameters (e.g., model-checking options) and results (diagnostic traces, statistics) are available. This information needs to be documented and maintained very carefully in order to manage a practical model-checking process and to allow the reproduction of the experiments that were carried out.

1.2.2 Strengths and Weaknesses

The strengths of model checking:

- It is a *general* verification approach that is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.
- It supports *partial* verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first. No complete requirement specification is needed.
- It is not vulnerable to the likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.
- It provides *diagnostic information* in case a property is invalidated; this is very useful for debugging purposes.

- It is a potential "push-button" technology; the use of model checking requires neither a high degree of user interaction nor a high degree of expertise.
- It enjoys a rapidly increasing *interest by industry*; several hardware companies have started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers have become available.
- It can be easily *integrated* in existing development cycles; its learning curve is not very steep, and empirical studies indicate that it may lead to shorter development times.
- It has a *sound and mathematical underpinning*; it is based on theory of graph algorithms, data structures, and logic.

The weaknesses of model checking:

- It is mainly appropriate to *control-intensive* applications and less suited for data-intensive applications as data typically ranges over infinite domains.
- Its applicability is subject to *decidability issues*; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable.
- It verifies a *system model*, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model. Complementary techniques, such as testing, are needed to find fabrication faults (for hardware) or coding errors (for software).
- It checks only *stated requirements*, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.
- It suffers from the *state-space explosion* problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem (see Chapters 7 and 8), models of realistic systems may still be too large to fit in memory.
- Its usage requires some *expertise* in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.
- It is not guaranteed to yield correct results: as with any tool, a model checker may contain *software defects*.[2]

[2]Parts of the more advanced model-checking procedures have been formally proven correct using theorem provers to circumvent this.

- It does not allow checking *generalizations*: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

We believe that one can never achieve absolute guaranteed correctness for systems of realistic size. Despite the above limitations we conclude that

> *Model checking is an effective technique to expose potential design errors.*

Thus, model checking can provide a significant increase in the level of confidence of a system design.

1.3 Bibliographic Notes

Model checking. Model checking originates from the independent work of two pairs in the early eighties: Clarke and Emerson [86] and Queille and Sifakis [347]. The term *model checking* was coined by Clarke and Emerson. The brute-force examination of the entire state space in model checking can be considered as an extension of automated protocol validation techniques by Hajek [182] and West [419, 420]. While these earlier techniques were restricted to checking the absence of deadlocks or livelocks, model checking allows for the examination of broader classes of properties. Introductory papers on model checking can be found in [94, 95, 96, 293, 426]. The limitations of model checking were discussed by Apt and Kozen [17]. More information on model checking is available in the earlier books by Holzmann [205], McMillan [288], and Kurshan [250] and the more recent works by Clarke, Grumberg, and Peled [92], Huth and Ryan [219], Schneider [365], and Bérard et al. [44]. The model-checking trajectory has recently been described by Ruys and Brinksma [360].

Software verification. Empirical data about software engineering is gathered by the Center for Empirically Based Software Engineering (`www.cebase.org`); their collected data about software defects has recently been summarized by Boehm and Basili [53]. The different characterizations of verification ("are we building the thing right?") and validation ("are we building the right thing?") originate from Boehm [52]. An overview of software testing is given by Whittaker [421]; books about software testing are by Myers [308] and Beizer [36]. Testing based on formal specifications has been studied extensively in the area of communication protocols. This has led to an international standard for conformance

testing [222]. The use of software verification techniques by the German software industry has been studied by Liggesmeyer et al. [275]. Books by Storey [381] and Leveson [269] describe techniques for developing safety-critical software and discuss the role of formal verification in this context. Rushby [359] addresses the role of formal methods for developing safety-critical software. The book of Peled [327] gives a detailed account of formal techniques for software reliability that includes testing, model checking, and deductive methods.

Model-checking software. Model-checking communication protocols has become popular through the pioneering work of Holzmann [205, 206]. An interesting project at Bell Labs in which a model-checking team and a traditional design team worked on the design of part of the ISDN user part protocol has been reported by Holzmann [207]. In this large case study, 112 serious design flaws were discovered while checking 145 formal properties in about 10,000 verification runs. Errors found by Clarke et al. [89] in the IEEE Futurebus+ standard (checking a model of more than 10^{30} states) has led to a substantial revision of the protocol by IEEE. Chan et al. [79] used model checking to verify the control software of a traffic control and alert system for airplanes. Recently, Staunstrup et al. [377] have reported the succesful model checking of a train model consisting of 1421 state machines comprising a state space of 10^{476} states. Lowe [278], using model checking, discovered a flaw in the well-known Needham-Schroeder authentication algorithm that remained undetected for over 17 years. The usage of formal methods (that includes model checking) in the software development process of a safety-critical system within a Dutch software house is presented by Tretmans, Wijbrans, and Chaudron [393]. The formal analysis of NASA's Mars Pathfinder and the Deep Space-1 spacecraft are addressed by Havelund, Lowry, and Penix [194], and Holzmann, Najm, and Serhrouchini [210], respectively. The automated generation of abstract models amenable to model checking from programs written in programming languages such as `C`, `C++`, or Java has been pursued, for instance, by Godefroid [170], Dwyer, Hatcliff, and coworkers [193], at Microsoft Research by Ball, Podelski, and Rajamani [33] and at NASA Research by Havelund and Pressburger [195].

Model-checking hardware. Applying model checking to hardware originates from Browne et al. [66] analyzing some moderate-size self-timed sequential circuits. Successful applications of (symbolic) model checking to large hardware systems have been first reported by Burch et al. [75] in the early nineties. They analyzed a synchronous pipeline circuit of approximately 10^{20} states. Overviews of formal hardware verification techniques can be found in works by Gupta [179], and the books by Yoeli [428] and Kropf [246]. The need for formal verification techniques for hardware verification has been advocated by, among others, Sangiovanni-Vincentelli, McGeer, and Saldanha [362]. The integration of model-checking techniques for error finding in the hardware development process at IBM has been recently described by Schlipf et al. [364] and Abarbanel-Vinov et al. [2]. They conclude that model checking is a powerful extension of the traditional verification pro-

cess, and consider it as complementary to simulation/emulation. The design of a memory bus adapter at IBM showed, e.g., that 24% of all defects were found with model checking, while 40% of these errors would most likely not have been found by simulation.

Chapter 2

Modelling Concurrent Systems

A prerequisite for model checking is a model of the system under consideration. This chapter introduces transition systems, a (by now) standard class of models to represent hardware and software systems. Different aspects for modeling concurrent systems are treated, ranging from the simple case in which processes run completely autonomously to the more realistic setting where processes communicate in some way. The chapter is concluded by considering the problem of state-space explosion.

2.1 Transition Systems

Transition systems are often used in computer science as models to describe the behavior of systems. They are basically directed graphs where nodes represent *states*, and edges model *transitions*, i.e., state changes. A state describes some information about a system at a certain moment of its behavior. For instance, a state of a traffic light indicates the current color of the light. Similarly, a state of a sequential computer program indicates the current values of all program variables together with the current value of the program counter that indicates the next program statement to be executed. In a synchronous hardware circuit, a state typically represents the current value of the registers together with the values of the input bits. Transitions specify how the system can evolve from one state to another. In the case of the traffic light a transition may indicate a switch from one color to another, whereas for the sequential program a transition typically corresponds to the execution of a statement and may involve the change of some variables and the program counter. In the case of the synchronous hardware circuit, a transition models the change of the registers and output bits on a new set of inputs.

In the literature, many different types of transition systems have been proposed. We use transition systems with *action names* for the transitions (state changes) and *atomic propositions* for the states. Action names will be used for describing communication mechanisms between processes. We use letters at the beginning of the Greek alphabet (such as α, β, and so on) to denote actions. Atomic propositions are used to formalize temporal characteristics. Atomic propositions intuitively express simple known facts about the states of the system under consideration. They are denoted by arabic letters from the beginning of the alphabet, such as a, b, c, and so on. Examples of atomic propositions are "x equals 0", or "x is smaller than 200" for some given integer variable x. Other examples are "there is more than a liter of fluid in the tank" or "there are no customers in the shop".

Definition 2.1. Transition System (TS)

A *transition system* TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

TS is called *finite* if S, Act, and AP are finite. ■

For convenience, we write $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \longrightarrow$. The intuitive behavior of a transition system can be described as follows. The transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\longrightarrow$. That is, if s is the current state, then a transition $s \xrightarrow{\alpha} s'$ originating from s is selected *nondeterministically* and taken, i.e., the action α is performed and the transition system evolves from state s into the state s'. This selection procedure is repeated in state s' and finishes once a state is encountered that has no outgoing transitions. (Note that I may be empty; in that case, the transition system has no behavior at all as no initial state can be selected.) It is important to realize that in case a state has more than one outgoing transition, the "next" transition is chosen in a purely nondeterministic fashion. That is, the outcome of this selection process is not known a priori, and, hence, no statement can be made about

the likelihood with which a certain transition is selected. Similarly, when the set of initial states consists of more than one state, the start state is selected nondeterministically.

The labeling function L relates a set $L(s) \in 2^{AP}$ of atomic propositions to any state s.[1] $L(s)$ intuitively stands for exactly those atomic propositions $a \in AP$ which are satisfied by state s. Given that Φ is a propositional logic formula, then s satisfies the formula Φ if the evaluation induced by $L(s)$ makes the formula Φ true; that is:

$$s \models \Phi \quad \text{iff} \quad L(s) \models \Phi.$$

(Basic principles of propositional logic are explained in Appendix A.3, see page 915 ff.)

Example 2.2. Beverage Vending Machine

We consider an (somewhat foolish) example, which has been established as standard in the field of process calculi. The transition system in Figure 2.1 models a preliminary design of a beverage vending machine. The machine can either deliver beer or soda. States are represented by ovals and transitions by labeled edges. State names are depicted inside the ovals. Initial states are indicated by having an incoming arrow without source.

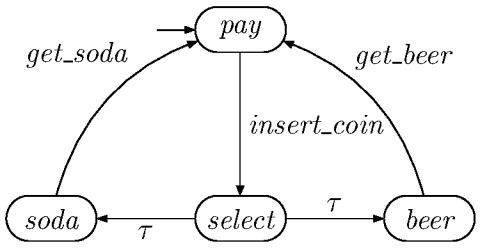


Figure 2.1: A transition system of a simple beverage vending machine.

The state space is $S = \{ pay, select, soda, beer \}$. The set of initial states consists of only one state, i.e., $I = \{ pay \}$. The (user) action *insert_coin* denotes the insertion of a coin, while the (machine) actions *get_soda* and *get_beer* denote the delivery of soda and beer, respectively. Transitions of which the action label is not of further interest here, e.g., as it denotes some internal activity of the beverage machine, are all denoted by the distinguished action symbol τ. We have:

$$Act = \{ insert\_coin, get\_soda, get\_beer, \tau \}.$$

Some example transitions are:

$$pay \xrightarrow{insert\_coin} select \quad \text{and} \quad beer \xrightarrow{get\_beer} pay.$$

[1] Recall that 2^{AP} denotes the power set of AP.

It is worthwhile to note that after the insertion of a coin, the vending machine nondeterministically can choose to provide either beer or soda.

The atomic propositions in the transition system depend on the properties under consideration. A simple choice is to let the state names act as atomic propositions, i.e., $L(s) = \{ s \}$ for any state s. If, however, the only relevant properties do not refer to the selected beverage, as in the property

"The vending machine only delivers a drink after providing a coin",

it suffices to use the two-element set of propositions $AP = \{ paid, drink \}$ with labeling function:

$$L(pay) = \varnothing, \quad L(soda) = L(beer) = \{ paid, drink \}, \quad L(select) = \{ paid \}.$$

Here, the proposition *paid* characterizes exactly those states in which the user has already paid but not yet obtained a beverage. ■

The previous example illustrates a certain arbitrariness concerning the choice of atomic propositions and action names. Even if the formal definition of a transition system requires determining the set of actions Act and the set of propositions AP, the components Act and AP are casually dealt with in the following. Actions are only necessary for modeling communication mechanisms as we will see later on. In cases where action names are irrelevant, e.g., because the transition stands for an internal process activity, we use a special symbol τ or, in cases where action names are not relevant, even omit the action label. The set of propositions AP is always chosen depending on the characteristics of interest. In depicting transition systems, the set of propositions AP often is not explicitly indicated and it is assumed that $AP \subseteq S$ with labeling function $L(s) = \{ s \} \cap AP$.

Crucial for modeling hard- or software systems by transition systems is the nondeterminism, which in this context is by far more than a theoretical concept. Later in this chapter (Section 2.2), we will explain in detail how transition systems can serve as a formal model for parallel systems. We mention here only that nondeterministic choices serve to model the parallel execution of independent activities by *interleaving* and to model the *conflict situations* that arise, e.g., if two processes aim to access a shared resource. Essentially, interleaving means the nondeterministic choice of the order in which order the actions of the processes that run in parallel are executed. Besides parallelism, the nondeterminism is also important for *abstraction* purposes, for *underspecification*, and to model the interface with an unknown or unpredictable *environment* (e.g., a human user). An example of the last is provided by the beverage vending machine where the user resolves the nondeterministic choice between the two τ-transitions in state "select" by choosing one of the two available drinks. The notion "underspecification" refers to early design phases where

a coarse model for a system is provided that represents several options for the possible behaviors by nondeterminism. The rough idea is that in further refinement steps the designer realizes one of the nondeterministic alternatives, but skips the others. In this sense, nondeterminism in a transition system can represent implementation freedom.

Definition 2.3. Direct Predecessors and Successors

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. For $s \in S$ and $\alpha \in Act$, the set of direct *α-successors* of s is defined as:

$$Post(s,\alpha) \;=\; \Big\{\, s' \in S \;\mid\; s \xrightarrow{\alpha} s' \,\Big\}, \qquad Post(s) \;=\; \bigcup_{\alpha \in Act} Post(s,\alpha).$$

The set of *α-predecessors* of s is defined by:

$$Pre(s,\alpha) \;=\; \Big\{\, s' \in S \;\mid\; s' \xrightarrow{\alpha} s \,\Big\}, \qquad Pre(s) \;=\; \bigcup_{\alpha \in Act} Pre(s,\alpha).$$

■

Each state $s' \in Post(s,\alpha)$ is called a direct α-successor of s. Accordingly, each state $s' \in Post(s)$ is called a direct successor of s. The notations for the sets of direct successors are expanded to subsets of S in the obvious way (i.e., pointwise extension): for $C \subseteq S$, let

$$Post(C,\alpha) \;=\; \bigcup_{s \in C} Post(s,\alpha), \quad Post(C) \;=\; \bigcup_{s \in C} Post(s).$$

The notations $Pre(C,\alpha)$ and $Pre(C)$ are defined in an analogous way:

$$Pre(C,\alpha) \;=\; \bigcup_{s \in C} Pre(s,\alpha), \quad Pre(C) \;=\; \bigcup_{s \in C} Pre(s).$$

Terminal states of a transition system TS are states without any outgoing transitions. Once the system described by TS reaches a terminal state, the complete system comes to a halt.

Definition 2.4. Terminal State

State s in transition system TS is called *terminal* if and only if $Post(s) = \varnothing$. ■

For a transition system modeling a sequential computer program, terminal states occur as a natural phenomenon representing the termination of the program. Later on, we will

see that for transition systems modeling parallel systems, such terminal states are usually considered to be undesired (see Section 3.1, page 89 ff.).

We mentioned above that nondeterminism is crucial for modeling computer systems. However, it is often useful to consider transition systems where the "observable" behavior is deterministic, according to some notion of observables. There are two general approaches to formalize the visible behavior of a transition system: one relies on the actions, the other on the labels of the states. While the action-based approach assumes that only the executed actions are observable from outside, the state-based approach ignores the actions and relies on the atomic propositions that hold in the current state to be visible. Transition systems that are deterministic in the action-based view have at most one outgoing transition labeled with action α per state, while determinism from the view of state labels means that for any state label $A \in 2^{AP}$ and any state there is at most one outgoing transition leading to a state with label A. In both cases, it is required that there be at most one initial state.

Definition 2.5. Deterministic Transition System

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system.

1. TS is called *action-deterministic* if $|\, I \,| \leqslant 1$ and $|\, Post(s, \alpha) \,| \leqslant 1$ for all states s and actions α.
2. TS is called *AP-deterministic* if $|\, I \,| \leqslant 1$ and $|Post(s) \cap \{\, s' \in S \mid L(s') = A \,\}| \leqslant 1$ for all states s and $A \in 2^{AP}$.

■

2.1.1 Executions

So far, the behavior of a transition system has been described at an intuitive level. This will now be formalized using the notion of *executions* (also called *runs*). An execution of a transition system results from the resolution of the possible nondeterminism in the system. An execution thus describes a possible behavior of the transition system. Formally:

Definition 2.6. Execution Fragment

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *finite* execution fragment ϱ of TS is an alternating sequence of states and actions ending with a state

$$\varrho \;=\; s_0\, \alpha_1\, s_1\, \alpha_2 \ldots \alpha_n\, s_n \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leqslant i < n,$$

where $n \geqslant 0$. We refer to n as the length of the execution fragment ϱ. An *infinite* execution fragment ρ of TS is an infinite, alternating sequence of states and actions:

$$\rho \;=\; s_0\,\alpha_1\,s_1\,\alpha_2\,s_2\,\alpha_3\ldots \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leqslant i.$$

■

Note that the sequence s with $s \in S$ is a legal finite execution fragment of length n=0. Each prefix of odd length of an infinite execution fragment is a finite execution fragment. From now on, the term *execution fragment* will be used to denote either a finite or an infinite execution fragment. Execution fragments $\varrho = s_0\,\alpha_1\,\ldots\,\alpha_n\,s_n$ and $\rho = s_0\,\alpha_1\,s_1\,\alpha_2\,\ldots$ will be written respectively as

$$\varrho \;=\; s_0 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} s_n \qquad \text{and} \qquad \rho \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots\ .$$

An execution fragment is called maximal when it cannot be prolonged:

Definition 2.7. Maximal and Initial Execution Fragment

A *maximal* execution fragment is either a finite execution fragment that ends in a terminal state, or an infinite execution fragment. An execution fragment is called *initial* if it starts in an initial state, i.e., if $s_0 \in I$. ■

Example 2.8. Executions of the Beverage Vending Machine

Some examples of execution fragments of the beverage vending machine described in Example 2.2 (page 21) are as follows. For simplicity, the action names are abbreviated, e.g., *sget* is a shorthand for *get_soda* and *coin* for *insert_coin*.

$$\rho_1 \;=\; \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \ldots$$

$$\rho_2 \;=\; \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{beer} \xrightarrow{\mathit{bget}} \ldots$$

$$\varrho \;=\; \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{sget}} \mathit{pay} \xrightarrow{\mathit{coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda}\ .$$

Execution fragments ρ_1 and ϱ are initial, but ρ_2 is not. ϱ is not maximal as it does not end in a terminal state. Assuming that ρ_1 and ρ_2 are infinite, they are maximal. ■

Definition 2.9. Execution

An *execution* of transition system TS is an initial, maximal execution fragment. ■

In Example 2.8, ρ_1 is an execution, while ρ_2 and ϱ are not. Note that ρ_2 is maximal but not initial, while ϱ is initial but not maximal.

A state s is called reachable if there is some execution fragment that ends in s and that starts in some initial state.

Definition 2.10. Reachable States

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A state $s \in S$ is called *reachable* in TS if there exists an initial, finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n = s \, .$$

$Reach(TS)$ denotes the set of all reachable states in TS. ■

2.1.2 Modeling Hardware and Software Systems

This section illustrates the use of transition systems by elaborating on the modeling of (synchronous) hardware circuits and sequential data-dependent systems – a kind of simple sequential computer programs. For both cases, the basic concept is that states represent possible storage configurations (i.e., evaluations of relevant "variables"), and state changes (i.e., transitions) represent changes of "variables". Here, the term "variable" has to be understood in the broadest sense. For computer programs a variable can be a control variable (like a program counter) or a program variable. For circuits a variable can, e.g, stand for either a register or an input bit.

Sequential Hardware Circuits

Before presenting a general recipe for modeling sequential hardware circuits as transition systems we consider a simple example to clarify the basic concepts.

Example 2.11. A Simple Sequential Hardware Circuit

Consider the circuit diagram of the sequential circuit with input variable x, output variable y, and register r (see left part of Figure 2.2). The control function for output variable y is given by

$$\lambda_y = \neg(x \oplus r)$$

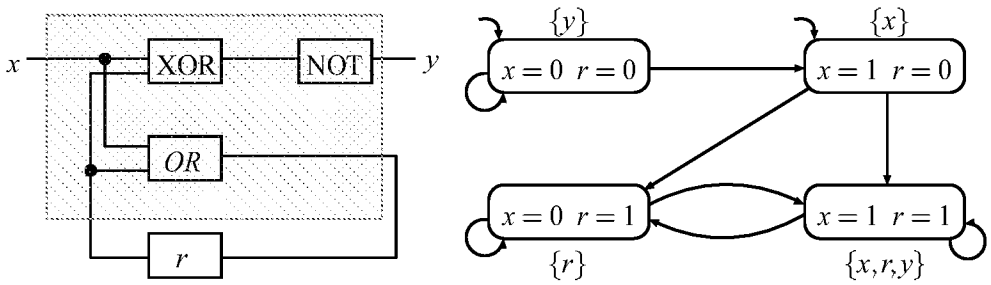


Figure 2.2: Transition system representation of a simple hardware circuit.

where $\oplus$ stands for exclusive or (XOR, or parity function). The register evaluation changes according to the circuit function

$$\delta_r \;=\; x \vee r \;.$$

Note that once the register evaluation is $[r = 1]$, r keeps that value. Under the initial register evaluation $[r = 0]$, the circuit behavior is modeled by the transition system TS with state space

$$S \;=\; \mathit{Eval}(x, r)$$

where $\mathit{Eval}(x, r)$ stands for the set of evaluations of input variable x and register variable r. The initial states of TS are $I = \{\, \langle x = 0, r = 0\rangle, \langle x = 1, r = 0\rangle \,\}$. Note that there are two initial states as we do not make any assumption about the initial value of the input bit x.

The set of actions is irrelevant and omitted here. The transitions result directly from the functions λ_y and δ_r. For instance, $\langle x = 0, r = 1\rangle \rightarrow \langle x = 0, r = 1\rangle$ if the next input bit equals 0, and $\langle x = 0, r = 1\rangle \rightarrow \langle x = 1, r = 1\rangle$ if the next input bit is 1.

It remains to consider the labeling L. Using the set of atomic propositions $AP = \{\, x, y, r \,\}$, then, e.g., the state $\langle x = 0, r = 1\rangle$ is labeled with $\{\, r \,\}$. It is not labeled with y since the circuit function $\neg(x \oplus r)$ results in the value 0 for this state. For state $\langle x = 1, r = 1\rangle$ we obtain $L(\langle x = 1, r = 1\rangle) = \{\, x, r, y \,\}$, as λ_y yields the value 1. Accordingly, we obtain: $L(\langle x = 0, r = 0\rangle) = \{\, y \,\}$, and $L(\langle x = 1, r = 0\rangle) = \{\, x \,\}$. The resulting transition system (with this labeling) is depicted in the right part of Figure 2.2.

Alternatively, using the set of propositions $AP' = \{\, x, y \,\}$ – the register evaluations are assumed to be "invisible" – one obtains:

$$\begin{array}{lcllcl} L'(\langle x = 0, r = 0\rangle) & = & \{\, y \,\} & L'(\langle x = 0, r = 1\rangle) & = & \varnothing \\ L'(\langle x = 1, r = 0\rangle) & = & \{\, x \,\} & L'(\langle x = 1, r = 1\rangle) & = & \{\, x, y \,\} \end{array}$$

The propositions in AP' suffice to formalize, e.g., the property "the output bit y is set infinitely often". Properties that refer to the register r are not expressible. ■

The approach taken in this example can be generalized toward arbitrary sequential hardware circuits (without "don't cares") with n input bits $x_1, \ldots, x_n$, m output bits $y_1, \ldots, y_m$, and k registers $r_1, \ldots, r_k$ as follows. The states of the transition system represent the evaluations of the $n{+}k$ input and register bits $x_1, \ldots, x_n, r_1, \ldots, r_k$. The evaluation of output bits depends on the evaluations of input bits and registers and can be derived from the states. Transitions represent the behavior, whereas it is assumed that the values of input bits are *nondeterministically* provided (by the circuit environment). Furthermore, we assume a given initial register evaluation

$$[r_1 = c_{0,1}, \ldots, r_k = c_{0,k}]$$

where $c_{0,i}$ denotes the initial value of register i for $0 < i \leqslant k$. Alternatively, a set of possible initial register evaluations may be given.

The transition system $TS = (S, Act, \rightarrow, I, AP, L)$ modeling this sequential hardware circuit has the following components. The state space S is determined by

$$S \;=\; \mathit{Eval}(x_1, \ldots, x_n, r_1, \ldots, r_k).$$

Here, $\mathit{Eval}(x_1, \ldots, x_n, r_1, \ldots, r_k)$ stands for the set of evaluations of input variables x_i and registers r_j and can be identified with the set $\{0,1\}^{n+k}$.[2] Initial states are of the form $(\ldots, c_{0,1}, \ldots, c_{0,k})$ where the k registers are evaluated with their initial value. The first n components prescribing the values of input bits are arbitrary. Thus, the set of initial states is

$$I \;=\; \Big\{(a_1, \ldots, a_n, c_{0,1}, \ldots, c_{0,k}) \mid a_1, \ldots, a_n \in \{0,1\}\Big\}.$$

The set Act of actions is irrelevant, and we choose $Act = \{\tau\}$. For simplicity, let the set of atomic propositions be

$$AP \;=\; \{x_1, \ldots, x_n, y_1, \ldots, y_m, r_1, \ldots, r_k\}\ .$$

(In practice, this could be defined as any subset of this AP). Thus, any register, any input bit, and any output bit can be used as an atomic proposition. The labeling function assigns to any state $s \in \mathit{Eval}(x_1, \ldots, x_n, r_1, \ldots, r_k)$ exactly those atomic propositions x_i, r_j which are evaluated to 1 under s. If for state s, output bit y_i is evaluated to 1, then (and only then) the atomic proposition y_i is part of $L(s)$. Thus,

$$\begin{aligned} L(a_1, \ldots, a_n, c_1, \ldots, c_k) \;=\; & \{x_i \mid a_i = 1\} \;\cup\; \{r_j \mid c_j = 1\} \\ & \cup\, \{y_i \mid s \models \lambda_{y_i}(a_1, \ldots, a_n, c_1, \ldots, c_k) = 1\} \end{aligned}$$

[2] An evaluation $s \in \mathit{Eval}(\cdot)$ is a mapping which assigns a value $s(x_i) \in \{0,1\}$ to any input bit x_i. Similarly, every register r_j is mapped onto a value $s(r_j) \in \{0,1\}$. To simplify matters, we assume every element $s \in S$ to be a bit-tuple of length $n{+}k$. The ith bit is set if and only if x_i is evaluated to 1 $(0 < i \leqslant n)$. Accordingly, the $n{+}j$th bit indicates the evaluation of r_j $(0 < j \leqslant k)$.

where $\lambda_{y_i} : S \to \{0,1\}$ is the switching function corresponding to output bit y_i that results from the gates of the circuit.

Transitions exactly represent the behavior. In the following, let δ_{r_j} denote the transition function for register r_j resulting from the circuit diagram. Then:

$$(\underbrace{a_1,\ldots,a_n}_{\text{input evaluation}}, \underbrace{c_1,\ldots,c_k}_{\text{register evaluation}}) \xrightarrow{\tau} (a'_1,\ldots,a'_n,c'_1,\ldots,c'_k)$$

if and only if $c'_j = \delta_{r_j}(a_1,\ldots,a_n,c_1,\ldots,c_k)$. Assuming that the evaluation of input bits changes nondeterministically, no restrictions on the bits $a'_1,\ldots,a'_n$ are imposed.

It is left to the reader to check that applying this recipe to the example circuit in the left part of Figure 2.2 indeed results in the transition system depicted in the right part of that figure.

Data-Dependent Systems

The executable actions of a data-dependent system typically result from conditional branching, as in

$$\textbf{if } x\%2 = 1 \textbf{ then } x := x+1 \textbf{ else } x := 2{\cdot}x \textbf{ fi}.$$

In principle, when modeling this program fragment as a transition system, the conditions of transitions could be omitted and conditional branchings could be replaced by nondeterminism; but, generally speaking, this results in a very abstract transition system for which only a few relevant properties can be verified. Alternatively, *conditional transitions* can be used and the resulting graph (labeled with conditions) can be unfolded into a transition system that subsequently can be subject to verification. This unfolding approach is detailed out below. We first illustrate this by means of an example.

Example 2.12. Beverage Vending Machine Revisited

Consider an extension of the beverage vending machine described earlier in Example 2.2 (page 21) which counts the number of soda and beer bottles and returns inserted coins if the vending machine is empty. For the sake of simplicity, the vending machine is represented by the two locations *start* and *select*. The following conditional transitions

$$\textit{start} \xrightarrow{\textit{true : coin}} \textit{select} \qquad \text{and} \qquad \textit{start} \xrightarrow{\textit{true : refill}} \textit{start}$$

model the insertion of a coin and refilling the vending machine. Labels of conditional transitions are of the form $g : \alpha$ where g is a Boolean condition (called guard), and α is an action that is possible once g holds. As the condition for both conditional transitions

above always holds, the action *coin* is always enabled in the starting location. To keep things simple, we assume that by *refill* both storages are entirely refilled. Conditional transitions

$$\textit{select} \xleftarrow{\textit{nsoda} > 0 \,:\, \textit{sget}} \textit{start} \qquad \text{and} \qquad \textit{select} \xleftarrow{\textit{nbeer} > 0 \,:\, \textit{bget}} \textit{start}$$

model that soda (or beer) can be obtained if there is some soda (or beer) left in the vending machine. The variables *nsoda* and *nbeer* record the number of soda and beer bottles in the machine, respectively. Finally, the vending machine automatically switches to the initial *start* location while returning the inserted coin once there are no bottles left:

$$\textit{select} \xleftarrow{\textit{nsoda} = 0 \,\wedge\, \textit{nbeer} = 0 \colon \textit{ret\_coin}} \textit{start}$$

Let the maximum capacity of both bottle repositories be *max*. The insertion of a coin (by action *coin*) leaves the number of bottles unchanged. The same applies when a coin is returned (by action *ret_coin*). The effect of the other actions is as follows:

Action	Effect
refill	$\textit{nsoda} := \textit{max};\ \textit{nbeer} := \textit{max}$
sget	$\textit{nsoda} := \textit{nsoda} - 1$
bget	$\textit{nbeer} := \textit{nbeer} - 1$

The graph consisting of locations as nodes and conditional transitions as edges is not a transition system, since the edges are provided with conditions. A transition system, however, can be obtained by "unfolding" this graph. For instance, Figure 2.3 on page 31 depicts this unfolded transition system when *max* equals 2. The states of the transition system keep track of the current location in the graph described above and of the number of soda- and beer bottles in the vending machine (as indicated by the gray and black dots, respectively, inside the nodes of the graph). ■

The ideas outlined in the previous example are formalized by using so-called *program graphs* over a set *Var* of typed variables such as *nsoda* and *nbeer* in the example. Essentially, this means that a standardized type (e.g., `boolean`, `integer`, or `char`) is associated with each variable. The type of variable x is called the domain $dom(x)$ of x. Let $Eval(Var)$ denote the set of (variable) evaluations that assign values to variables. $Cond(Var)$ is the set of Boolean conditions over *Var*, i.e., propositional logic formulae whose propositional symbols are of the form "$\overline{x} \in \overline{D}$" where $\overline{x} = (x_1, \ldots, x_n)$ is a tuple consisting of pairwise distinct variables in *Var* and $\overline{D}$ is a subset of $dom(x_1) \times \ldots \times dom(x_n)$. The proposition

$$(-3 < x - x' \leqslant 5) \ \wedge\ (x \leqslant 2{\cdot}x') \ \wedge\ (y = \textit{green}),$$

for instance, is a legal Boolean condition for integer variables x and x', and y a variable with, e.g., $dom(y) = \{\, \textit{red}, \textit{green} \,\}$. Here and in the sequel, we often use simplified notations

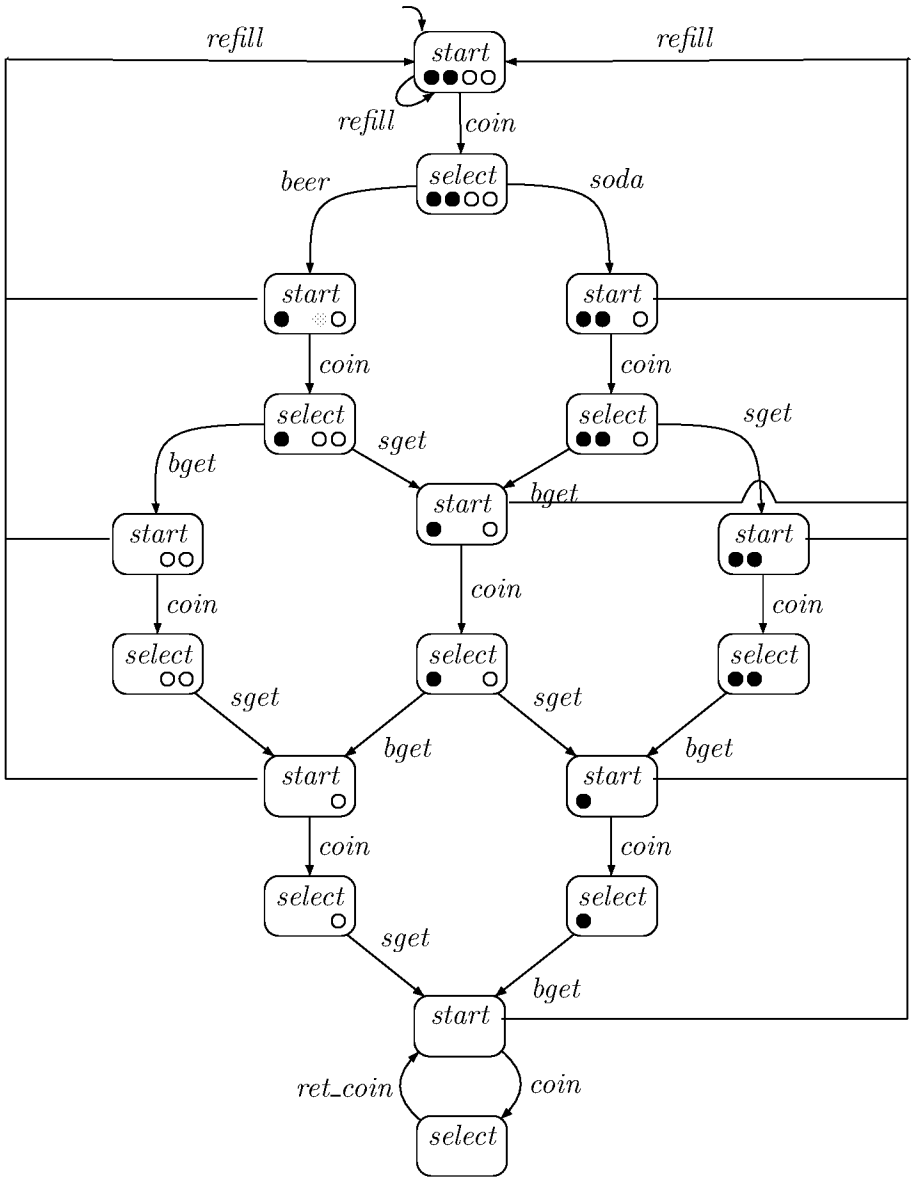


Figure 2.3: Transition system modeling the extended beverage vending machine.

for the propositional symbols such as "$3 < x - x' \leqslant 5$" instead of "$(x, x') \in \{\, (n, m) \in \mathbb{N}^2 \mid 3 < n - m \leqslant 5 \,\}$".

Initially, we do not restrict the domains. $dom(x)$ can be an arbitrary, possibly infinite, set. Even if in real computer systems all domains are finite (e.g., the type `integer` only includes integers n of a finite domain, like $-2^{16} < n < 2^{16}$), then the logical or algorithmic structure of a program is often based on infinite domains. The decision which restrictions on domains are useful for implementation, e.g., how many bits should be provided for representation of variables of type `integer` is delayed until a later design stage and is ignored here.

A *program graph* over a set of typed variables is a digraph whose edges are labeled with conditions on these variables and actions. The effect of the actions is formalized by means of a mapping

$$\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$$

which indicates how the evaluation η of variables is changed by performing an action. If, e.g., α denotes the action $x := y{+}5$, where x and y are integer variables, and η is the evaluation with $\eta(x) = 17$ and $\eta(y) = -2$, then

$$\mathit{Effect}(\alpha, \eta)(x) \;=\; \eta(y) + 5 \;=\; -2 + 5 = 3, \quad \text{and} \quad \mathit{Effect}(\alpha, \eta)(y) \;= \eta(y) \;=\; -2.$$

$\mathit{Effect}(\alpha, \eta)$ is thus the evaluation that assigns 3 to x and -2 to y. The nodes of a program graph are called *locations* and have a control function since they specify which of the conditional transitions are possible.

Definition 2.13. Program Graph (PG)

A *program graph* PG over set Var of typed variables is a tuple $(Loc, Act, \mathit{Effect}, \hookrightarrow, Loc_0, g_0)$ where

- Loc is a set of locations and Act is a set of actions,
- $\mathit{Effect} : Act \times \mathit{Eval}(Var) \to \mathit{Eval}(Var)$ is the effect function,
- $\hookrightarrow \subseteq\ Loc \times \mathit{Cond}(Var) \times Act \times Loc$ is the conditional transition relation,
- $Loc_0 \subseteq Loc$ is a set of initial locations,
- $g_0 \in \mathit{Cond}(Var)$ is the initial condition.

■

The notation $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ is used as shorthand for $(\ell, g, \alpha, \ell') \in \hookrightarrow$. The condition g is also called the *guard* of the conditional transition $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$. If the guard is a tautology (e.g., $g = \text{true}$ or $g = (x < 1) \vee (x \geqslant 1)$), then we simply write $\ell \overset{\alpha}{\hookrightarrow} \ell'$.

The behavior in location $\ell \in Loc$ depends on the current variable evaluation η. A nondeterministic choice is made between all transitions $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ which satisfy condition g in evaluation η (i.e., $\eta \models g$). The execution of action α changes the evaluation of variables according to $Effect(\alpha, \cdot)$. Subsequently, the system changes into location ℓ'. If no such transition is possible, the system stops.

Example 2.14. Beverage Vending Machine

The graph described in Example 2.12 (page 29) is a program graph. The set of variables is

$$Var = \{ nsoda, \quad nbeer \}$$

where both variables have the domain $\{ 0, 1, \ldots, max \}$. The set Loc of locations equals $\{ start, select \}$ with $Loc_0 = \{ start \}$, and

$$Act = \{ bget, sget, coin, ret\_coin, refill \} .$$

The effect of the actions is defined by:

$$\begin{array}{lcl} Effect(coin, \eta) & = & \eta \\ Effect(ret\_coin, \eta) & = & \eta \\ Effect(sget, \eta) & = & \eta[nsoda := nsoda{-}1] \\ Effect(bget, \eta) & = & \eta[nbeer := nbeer{-}1] \\ Effect(refill, \eta) & = & [nsoda := max, nbeer := max] \end{array}$$

Here, $\eta[nsoda := nsoda{-}1]$ is a shorthand for evaluation η' with $\eta'(nsoda) = \eta(nsoda){-}1$ and $\eta'(x) = \eta(x)$ for all variables different from *nsoda*. The initial condition g_0 states that initially both storages are entirely filled, i.e., $g_0 = (nsoda = max \wedge nbeer = max)$. ■

Each program graph can be interpreted as a transition system. The underlying transition system of a program graph results from *unfolding*. Its states consist of a control component, i.e., a location ℓ of the program graph, together with an evaluation η of the variables. States are thus pairs of the form $\langle \ell, \eta \rangle$. Initial states are initial locations that satisfy the initial condition g_0. To formulate properties of the system described by a program graph, the set AP of propositions is comprised of locations $\ell \in Loc$ (to be able to state at which control location the system currently is), and Boolean conditions for the variables. For example, a proposition like

$$(x \leqslant 5) \ \wedge \ (y \text{ is even}) \ \wedge \ (\ell \in \{ 1, 2 \})$$

can be formulated with x, y being integer variables and with locations being naturals. The labeling of states is such that $\langle \ell, v \rangle$ is labeled with ℓ and with all conditions (over *Var*) that hold in η. The transition relation is determined as follows. Whenever $\ell \xrightarrow{g:\alpha} \ell'$ is a conditional transition in the program graph, and the guard g holds in the current evaluation η, then there is a transition from state $\langle \ell, \eta \rangle$ to state $\langle \ell', \mathit{Effect}(\alpha, \eta) \rangle$. Note that the transition is not guarded. Formally:

Definition 2.15. Transition System Semantics of a Program Graph

The transition system $TS(PG)$ of program graph

$$PG = (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0)$$

over set *Var* of variables is the tuple $(S, \mathit{Act}, \longrightarrow, I, AP, L)$ where

- $S = \mathit{Loc} \times \mathit{Eval}(\mathit{Var})$
- $\longrightarrow \subseteq S \times \mathit{Act} \times S$ is defined by the following rule (see remark below):

$$\frac{\ell \overset{g:\alpha}{\hookrightarrow} \ell' \quad \wedge \quad \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \mathit{Effect}(\alpha, \eta) \rangle}$$

- $I = \{\langle \ell, \eta \rangle \mid \ell \in \mathit{Loc}_0, \eta \models g_0\}$
- $AP = \mathit{Loc} \cup \mathit{Cond}(\mathit{Var})$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in \mathit{Cond}(\mathit{Var}) \mid \eta \models g\}$.

■

The definition of $TS(PG)$ determines a very large set of propositions AP. But generally, only a small part of AP is necessary to formulate the relevant system properties. In the following, we exploit the degrees of freedom in choosing the set of propositions of $TS(PG)$ and only use the atomic propositions needed in the context at hand.

Remark 2.16. Structured Operational Semantics

In Definition 2.15, the transition relation is defined using the so-called SOS-notation (Structured Operational Semantics). This notation will be frequently used in the remainder of this monograph. The notation

$$\frac{\text{premise}}{\text{conclusion}}$$

should be read as follows. If the proposition above the "solid line" (i.e., the premise) holds, then the proposition under the fraction bar (i.e., the conclusion) holds as well. Such "if ..., then ..." propositions are also called *inference rules* or simply *rules*. If the premise is a tautology, it may be omitted (as well as the "solid line"). In the latter case, the rule is also called an *axiom*.

Phrases like "The relation $\rightarrow$ is defined by the following (axioms and) rules" have the meaning of an inductive definition where the relation $\rightarrow$ is defined as the *smallest* relation satisfying the indicated axioms and rules. ■

2.2 Parallelism and Communication

In the previous section, we have introduced the notion of transition systems and have shown how sequential hardware circuits and data-dependent systems (like simple sequential computer programs) can be effectively modeled as transition systems. In reality, however, most hard- and software systems are not sequential but parallel in nature. This section describes several mechanisms to provide operational models for *parallel* systems by means of transition systems. These mechanisms range from simple mechanisms where no communication between the participating transitions systems takes place, to more advanced (and realistic) schemes in which messages can be transferred, either synchronously (i.e., by means of "handshaking") or asynchronously (i.e., by buffers with a positive capacity). Let us assume that the operational (stepwise) behavior of the processes that run in parallel are given by transition systems $TS_1, \ldots, TS_n$. The goal is to define an operator $\|$, such that:

$$TS \;=\; TS_1 \parallel TS_2 \parallel \ldots \parallel TS_n$$

is a transition system that specifies the behavior of the parallel composition of transition systems TS_1 through TS_n. Here, it is assumed that $\|$ is a commutative and associative operator. The nature of the operator $\|$ will, of course, depend on the kind of communication that is supported. We will for instance see that some notions of parallel composition do not yield an associative operator. In the remainder of this section, several variants of $\|$ will be considered and illustrated by means of examples. Note that the above scheme may be repeated for TS_i, i.e., TS_i may again be a transition system that is composed of several transition systems:

$$TS_i \;=\; TS_{i,1} \parallel TS_{i,1} \parallel \ldots \parallel TS_{i,n_i} \;.$$

By using parallel composition in this hierarchical way, complex systems can be described in a rather structured way.

2.2.1 Concurrency and Interleaving

A widely adopted paradigm for parallel systems is that of *interleaving*. In this model, one abstracts from the fact that a system is actually composed of a set of (partly) independent components. That is to say, the global system state – composed of the current individual states of the components – plays a key role in interleaving. Actions of an independent component are merged (also called weaved), or "interleaved", with actions from other components. Thus, concurrency is represented by (pure) interleaving, that is, the nondeterministic choice between activities of the simultaneously acting processes (or components). This perspective is based on the view that only one processor is available on which the actions of the processes are interlocked. The "one-processor view" is only a modeling concept and also applies if the processes run on different processors. Thereby, (at first) no assumptions are made about the order in which the different processes are executed. If there are, e.g., two nonterminating processes P and Q, say, acting completely independent of each other, then

$$\begin{array}{cccccccccc} P & Q & P & Q & P & Q & Q & Q & P & \ldots \\ P & P & Q & P & P & Q & P & P & Q & \ldots \\ P & Q & P & P & Q & P & P & P & Q & \ldots \end{array}$$

are three possible sequences in which the steps (i.e., execution of actions) of P and Q can be interlocked. (In Chapter 3, certain restrictions will be discussed to ensure that each participating processor is treated in a somewhat "fair" manner. In particular, execution sequences like $P, P, P, \ldots$, where Q is completely ignored, are ruled out. Unless stated otherwise, we accept all possible interleavings, including the unfair ones.)

The interleaving representation of concurrency is subject to the idea that there is a scheduler which interlocks the steps of concurrently executing processes according to an a priori unknown strategy. This type of representation completely abstracts from the speed of the participating processes and thus models *any* possible realization by a single-processor machine or by several processors with arbitrary speeds.

Example 2.17. Two Independent Traffic Lights

Consider the transition systems of two traffic lights for nonintersecting (i.e., parallel) roads. It is assumed that the traffic lights switch completely independent of each other. For example, the traffic lights may be controlled by pedestrians who would like to cross the road. Each traffic light is modeled as a simple transition system with two states, one state

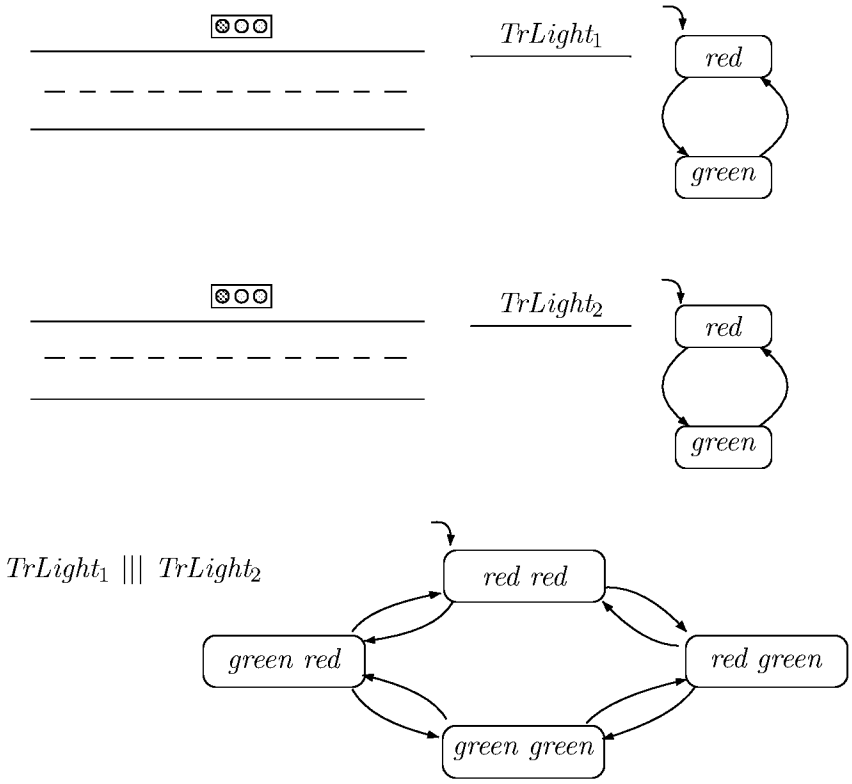


Figure 2.4: Example of interleaving operator for transition systems.

modeling a red light, the other one modeling a green light (see upper part of Figure 2.4). The transition system of the parallel composition of both traffic lights is sketched at the bottom of Figure 2.4 where ||| denotes the interleaving operator. In principle, any form of interlocking of the "actions" of the two traffic lights is possible. For instance, in the initial state where both traffic lights are red, there is a non-deterministic choice between which of the lights turns green. Note that this nondeterminism is descriptive, and does not model a scheduling problem between the traffic lights (although it may seem so). ■

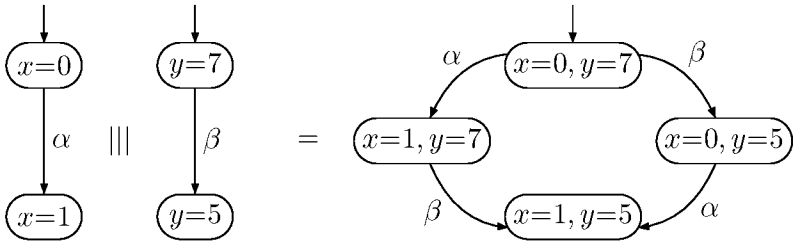
An important justification for interleaving is the fact that the effect of concurrently executed, independent actions α and β, say, is identical to the effect when α and β are successively executed in arbitrary order. This can symbolically be stated as

$$\mathit{Effect}(\alpha \,|||\, \beta, \eta) \;=\; \mathit{Effect}((\alpha\,;\,\beta) + (\beta\,;\,\alpha), \eta)$$

where the operator semicolon ; stands for sequential execution, + stands for nondeterministic choice, and ||| for the concurrent execution of independent activities. This fact can be easily understood when the effect is considered from two independent value assignments

$$\underbrace{x := x + 1}_{=\alpha} \;|||\; \underbrace{y := y - 2}_{=\beta}.$$

When initially $x = 0$ and $y = 7$, then x has the value 1 and y the value 5 after executing α and β, independent of whether the assignments occur concurrently (i.e., simultaneously) or in some arbitrary successive order. This is depicted in terms of transition systems as follows:



Note that the independence of actions is crucial. For dependent actions, the order of actions is typically essential: e.g., the final value of variable x in the parallel program $x := x{+}1 \,|||\, x := 2{\cdot}x$ (with initial value $x{=}0$, say) depends on the order in which the assignments $x := x{+}1$ and $x := 2{\cdot}x$ take place.

We are now in a position to formally define the interleaving (denoted $|||$) of transition systems. The transition system $TS_1 \,|||\, TS_2$ represents a parallel system resulting from the weaving (or merging) of the actions of the components as described by TS_1 and TS_2. It is assumed that no communication and no contentions (on shared variables) occur at all. The ("global") states of $TS_1 \,|||\, TS_2$ are pairs $\langle s_1, s_2 \rangle$ consisting of "local" states s_i of the components TS_i. The outgoing transitions of the global state $\langle s_1, s_2 \rangle$ consist of the outgoing transitions of s_1 together with those of s_2. Accordingly, whenever the composed system is in state $\langle s_1, s_2 \rangle$, a nondeterministic choice is made between all outgoing transitions of local state s_1 and those of local state s_2.

Definition 2.18. Interleaving of Transition Systems

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$ $i{=}1,2$, be two transition systems. The transition system $TS_1 \,|||\, TS_2$ is defined by:

$$TS_1 \,|||\, TS_2 \;=\; (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where the transition relation $\rightarrow$ is defined by the following rules:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1', s_2 \rangle} \quad \text{and} \quad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle s_1, s_2' \rangle}$$

and the labeling function is defined by $L(\langle s_1, s_2 \rangle) = L(s_1) \cup L(s_2)$. ■

Example 2.19.

Consider the two independent traffic lights described in Example 2.17 (page 36). The depicted transition system is actually the transition system

$$TS \;=\; TrLight_1 \;|||\; TrLight_2$$

that originates from interleaving. ■

For program graphs PG_1 (on Var_1) and PG_2 (on Var_2) without shared variables (i.e., $Var_1 \cap Var_2 = \varnothing$), the interleaving operator, which is applied to the appropriate transition systems, yields a transition system

$$TS(PG_1) \;|||\; TS(PG_2)$$

that describes the behavior of the simultaneous execution of PG_1 and PG_2.

2.2.2 Communication via Shared Variables

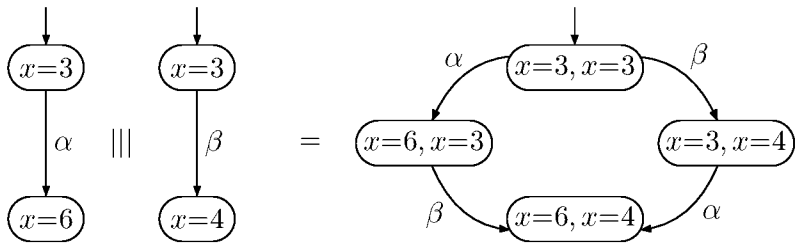
The interleaving operator $|||$ can be used to model *asynchronous* concurrency in which the subprocesses act completely independent of each other, i.e., without any form of message passing or contentions on shared variables. The interleaving operator for transition systems is, however, too simplistic for most parallel systems with concurrent or communicating components. An example of a system whose components have variables in common—shared variables so to speak—will make this clear.

Example 2.20. The Interleaving Operator for Concurrent Processes

Regard the program graph for the instructions α and β of the parallel program

$$\underbrace{x := 2{\cdot}x}_{\text{action } \alpha} \;|||\; \underbrace{x := x+1}_{\text{action } \beta}$$

where we assume that initially $x = 3$. (To simplify the picture, the locations have been skipped.) The transition system $TS(PG_1) \,|||\, TS(PG_2)$ contains, e.g., the inconsistent state $\langle x{=}6, x{=}4\rangle$ and, thus, does not reflect the intuitive behavior of the parallel execution of α and β:



The problem in this example is that the actions α and β access the shared variable x and therefore are competing. The interleaving operator for transition systems, however, "blindly" constructs the Cartesian product of the individual state spaces without considering these potential conflicts. Accordingly, it is not identified that the local states x=6 and x=4 describe exclusive events. ■

In order to deal with parallel programs with shared variables, an interleaving operator will be defined on the level of program graphs (instead of directly on transition systems). The interleaving of program graphs PG_1 and PG_2 is denoted $PG_1 \;|||\; PG_2$. The underlying transition system of the resulting program graph $PG_1 \;|||\; PG_2$, i.e., $TS(PG_1 \,|||\, PG_2)$ (see Definition 2.15, page 34) faithfully describes a parallel system whose components communicate via shared variables. Note that, in general, $TS(PG_1 \,|||\, PG_2) \neq TS(PG_1) \,|||\, TS(PG_2)$.

Definition 2.21. Interleaving of Program Graphs

Let $PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})$, for i=1, 2 be two program graphs over the variables Var_i. Program graph $PG_1 \,|||\, PG_2$ over $Var_1 \cup Var_2$ is defined by

$$PG_1 \,|||\, PG_2 \;=\; (Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

where $\hookrightarrow$ is defined by the rules:

$$\frac{\ell_1 \overset{g:\alpha}{\hookrightarrow_1} \ell_1'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1', \ell_2 \rangle} \quad \text{and} \quad \frac{\ell_2 \overset{g:\alpha}{\hookrightarrow_2} \ell_2'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1, \ell_2' \rangle}$$

and $Effect(\alpha, \eta) \;=\; Effect_i(\alpha, \eta)$ if $\alpha \in Act_i$. ■

The program graphs PG_1 and PG_2 have the variables $Var_1 \cap Var_2$ in common. These are the shared (sometimes also called "global") variables. The variables in $Var_1 \setminus Var_2$ are the local variables of PG_1, and similarly, those in $Var_2 \setminus Var_1$ are the local variables of PG_2.

Example 2.22. Interleaving of Program Graphs

Consider the program graphs PG_1 and PG_2 that correspond to the assignments $x := x{+}1$ and $x := 2{\cdot}x$, respectively. The program graph $PG_1 \,|||\, PG_2$ is depicted in the bottom left of Figure 2.5. Its underlying transition system $TS(PG_1 \,|||\, PG_2)$ is depicted in the bottom right of that figure where it is assumed that initially x equals 3. Note that the nondeterminism in the initial state of the transition system does not represent concurrency but just the possible resolution of the contention between the statements $x := 2{\cdot}x$ and $x := x{+}1$ that both modify the shared variable x. ■

The distinction between local and shared variables has also an impact on the actions of the composed program graph $PG_1 \,|||\, PG_2$. Actions that access (i.e., inspect or modify) shared

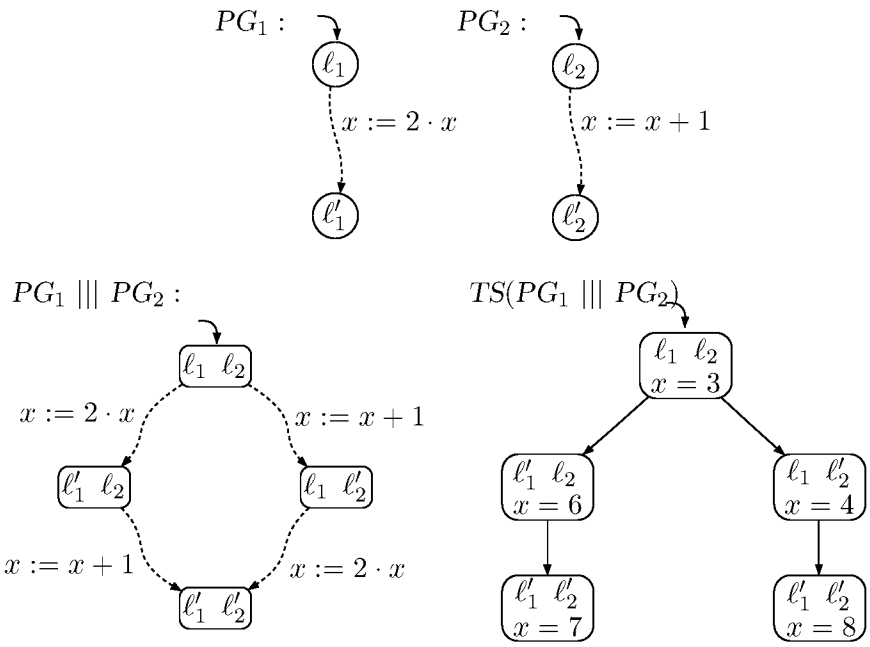


Figure 2.5: Interleaving of two example program graphs.

variables may be considered as "critical"; otherwise, they are viewed to be noncritical. (For the sake of simplicity, we are a bit conservative here and consider the inspection of shared variables as critical.) The difference between the critical and noncritical actions becomes clear when interpreting the (possible) nondeterminism in the transition system $TS(PG_1 \,|||\, PG_2)$. nondeterminism in a state of this transition system may stand either for

(i) an "internal" nondeterministic choice within program graph PG_1 or PG_2,

(ii) the interleaving of noncritical actions of PG_1 and PG_2, or

(iii) the resolution of a contention between critical actions of PG_1 and PG_2 (concurrency).

In particular, a noncritical action of PG_1 can be executed in parallel with critical or noncritical actions of PG_2 as it will only affect its local variables. By symmetry, the same applies to the noncritical actions of PG_2. Critical actions of PG_1 and PG_2, however, cannot be executed simultaneously as the value of the shared variables depends on the order of executing these actions (see Example 2.20). Instead, any global state where critical actions of PG_1 and PG_2 are enabled describe a concurrency situation that has to be resolved by an appropriate scheduling strategy. (Simultaneous reading of shared variables could be allowed, however.)

Remark 2.23. On Atomicity

For modeling a parallel system by means of the interleaving operator for program graphs it is decisive that the actions $\alpha \in Act$ are indivisible. The transition system representation only expresses the effect of the completely executed action α. If there is, e.g., an action α with its effect being described by the statement sequence

$$x := x + 1;\ y := 2x + 1;\ \textbf{if}\ x < 12\ \textbf{then}\ z := (x - z)^2 * y\ \textbf{fi},$$

then an implementation is assumed which does *not* interlock the basic substatements $x := x+1$, $y := 2x+1$, the comparison "$x < 12$", and, possibly, the assignment $z := (x-z)^2 * y$ with other concurrent processes. In this case,

$$\begin{array}{lcl} \mathit{Effect}(\alpha,\eta)(x) & = & \eta(x) + 1 \\ \mathit{Effect}(\alpha,\eta)(y) & = & 2(\eta(x) + 1) + 1 \\ \mathit{Effect}(\alpha,\eta)(z) & = & \begin{cases} (\eta(x) + 1 - \eta(z))^2 * 2(\eta(x) + 1) + 1 & \text{if } \eta(x) + 1 < 12 \\ \eta(z) & \text{otherwise} \end{cases} \end{array}$$

Hence, statement sequences of a process can be declared *atomic* by program graphs when put as a single label to an edge. In program texts such multiple assignments are surrounded by brackets $\langle \ldots \rangle$. ■

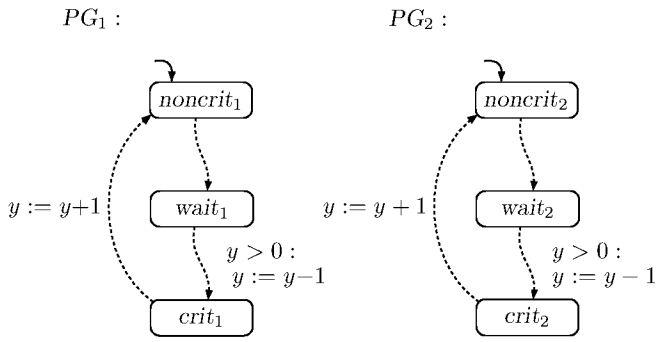


Figure 2.6: Individual program graphs for semaphore-based mutual exclusion.

Example 2.24. Mutual Exclusion with Semaphores

Consider two simplified processes $P_i, i{=}1, 2$ of the form:

```
P_i  loop forever
     ⋮                    (* noncritical actions *)
     request
     critical section
     release
     ⋮                    (* noncritical actions *)
     end loop
```

Processes P_1 and P_2 are represented by the program graphs PG_1 and PG_2, respectively, that share the binary semaphore y. $y{=}0$ indicates that the semaphore—the lock to get access to the critical section—is currently possessed by one of the processes. When $y{=}1$, the semaphore is free. The program graphs PG_1 and PG_2 are depicted in Figure 2.6.

For the sake of simplicity, local variables and shared variables different from y are not considered. Also, the activities inside and outside the critical sections are omitted. The locations of PG_i are $noncrit_i$ (representing the noncritical actions), $wait_i$ (modeling the situation in which P_i waits to enter its critical section), and $crit_i$ (modeling the critical section). The program graph $PG_1 \,|||\, PG_2$ consists of nine locations, including the (undesired) location $\langle crit_1, crit_2 \rangle$ that models the situation where both P_1 and P_2 are in their critical section, see Figure 2.7.

When unfolding $PG_1 \,|||\, PG_2$ into the transition system $TS_{Sem} = TS(PG_1 \,|||\, PG_2)$ (see Figure 2.8 on page 45), it can be easily checked that from the 18 global states in TS_{Sem}

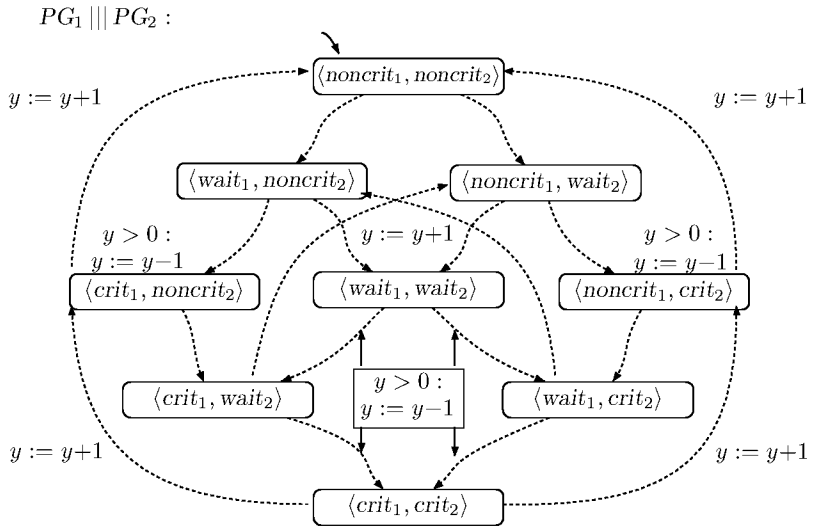


Figure 2.7: $PG_1 \,|||\, PG_2$ for semaphore-based mutual exclusion.

only the following eight states are reachable:

$$\begin{array}{ll} \langle noncrit_1, noncrit_2, y = 1 \rangle & \langle noncrit_1, wait_2, y = 1 \rangle \\ \langle wait_1, noncrit_2, y = 1 \rangle & \langle wait_1, wait_2, y = 1 \rangle \\ \langle noncrit_1, crit_2, y = 0 \rangle & \langle crit_1, noncrit_2, y = 0 \rangle \\ \langle wait_1, crit_2, y = 0 \rangle & \langle crit_1, wait_2, y = 0 \rangle \end{array}$$

States $\langle noncrit_1, noncrit_2, y = 1 \rangle$, and $\langle noncrit_1, crit_2, y = 0 \rangle$ stand for examples of situations where both P_1 and P_2 are able to concurrently execute actions. Note that in Figure 2.8 n stands for *noncrit*, w for *wait*, and c for *crit*. The nondeterminism in these states thus stand for interleaving of noncritical actions. State $\langle crit_1, wait_2, y = 0 \rangle$, e.g., represents a situation where only PG_1 is active, whereas PG_2 is waiting.

From the fact that the global state $\langle crit_1, crit_2, y = \ldots \rangle$ is unreachable in TS_{Sem}, it follows that processes P_1 and P_2 cannot be simultaneously in their critical section. The parallel system thus satisfies the so-called *mutual exclusion* property. ■

In the previous example, the nondeterministic choice in state $\langle wait_1, wait_2, y = 1 \rangle$ represents a contention between allowing either P_1 or P_2 to enter its critical section. The resolution of this scheduling problem—which process is allowed to enter its critical section next?—is left open, however. In fact, the parallel program of the previous example is "abstract" and does not provide any details on how to resolve this contention. At later design stages, for example, when implementing the semaphore y by means of a queue of waiting processes (or the like), a decision has to be made on how to schedule the processes that are enqueued for acquiring the semaphore. At that stage, a last-in first-out (LIFO),

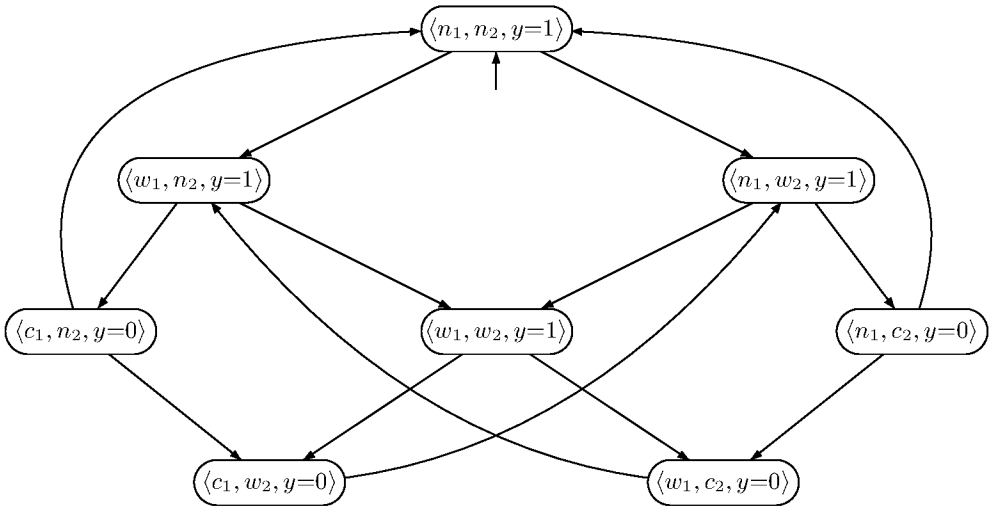


Figure 2.8: Mutual exclusion with semaphore (transition system representation).

first-in first-out (FIFO), or some other scheduling discipline can be chosen. Alternatively, another (more concrete) mutual exclusion algorithm could be selected that resolves this scheduling issue explicitly. A prominent example of such algorithm has been provided in 1981 by Peterson [332].

Example 2.25. Peterson's Mutual Exclusion Algorithm

Consider the processes P_1 and P_2 with the shared variables b_1, b_2, and x. b_1 and b_2 are Boolean variables, while x can take either the value 1 or 2, i.e., $dom(x) = \{1, 2\}$. The scheduling strategy is realized using x as follows. If both processes want to enter the critical section (i.e., they are in location $wait_i$), the value of variable x decides which of the two processes may enter its critical section: if $x = i$, then P_i may enter its critical section (for $i = 1, 2$). On entering location $wait_1$, process P_1 performs $x := 2$, thus giving privilege to process P_2 to enter the critical section. The value of x thus indicates which process has its turn to enter the critical section. Symmetrically, P_2 sets x to 1 when starting to wait. The variables b_i provide information about the current location of P_i. More precisely,

$$b_i \;=\; wait_i \vee crit_i \;.$$

b_i is set when P_i starts to wait. In pseudocode, P_1 performs as follows (the code for process P_2 is similar):

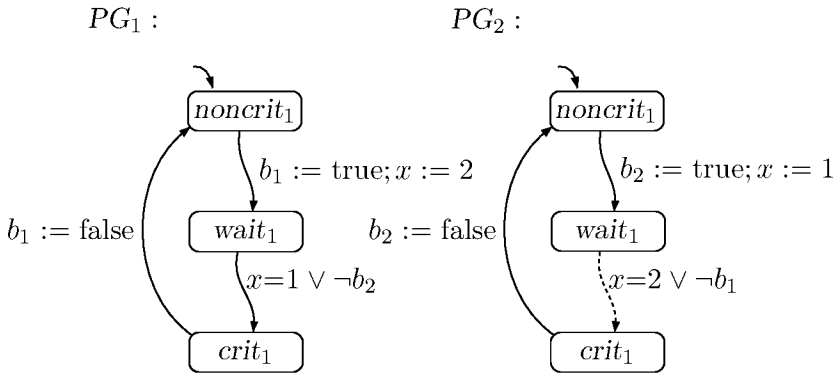


Figure 2.9: Program graphs for Peterson's mutual exclusion algorithm.

P_1 **loop forever**
⋮ (* noncritical actions *)
$\langle b_1 := \text{true};\ x := 2 \rangle$; (* request *)
wait until $(x = 1 \ \vee \ \neg b_2)$
do critical section **od**
$b_1 := \text{false}$ (* release *)
⋮ (* noncritical actions *)
end loop

Process P_i is represented by program graph PG_i over $Var = \{ x, b_1, b_2 \}$ with locations $noncrit_i$, $wait_i$, and $crit_i$, see Figure 2.9 above. The reachable part of the underlying transition system $TS_{Pet} = TS(PG_1 \,|||\, PG_2)$ has the form as indicated in Figure 2.10 (page 47), where for convenience n_i, w_i, c_i are used for $noncrit_i$, $wait_i$, and $crit_i$, respectively. The last digit of the depicted states indicates the evaluation of variable x. For convenience, the values for b_i are not indicated. Its evaluation can directly be deduced from the location of PG_i. Further, $b_1 = b_2 = \text{false}$ is assumed as the initial condition.

Each state in TS_{Pet} has the form $\langle loc_1, loc_2, x, b_1, b_2 \rangle$. As PG_i has three possible locations and b_i and x each can take two different values, the total number of states of TS_{Pet} is 72. Only ten of these states are reachable. Since there is no reachable state with P_1 and P_2 being in their critical section, it can be concluded that Peterson's algorithm satisfies the mutual exclusion property.

In the above program, the multiple assignments $b_1 := \text{true}; x := 2$ and $b_2 := \text{true}; x := 1$ are considered as indivisible (i.e., atomic) actions. This is indicated by the brackets $\langle$

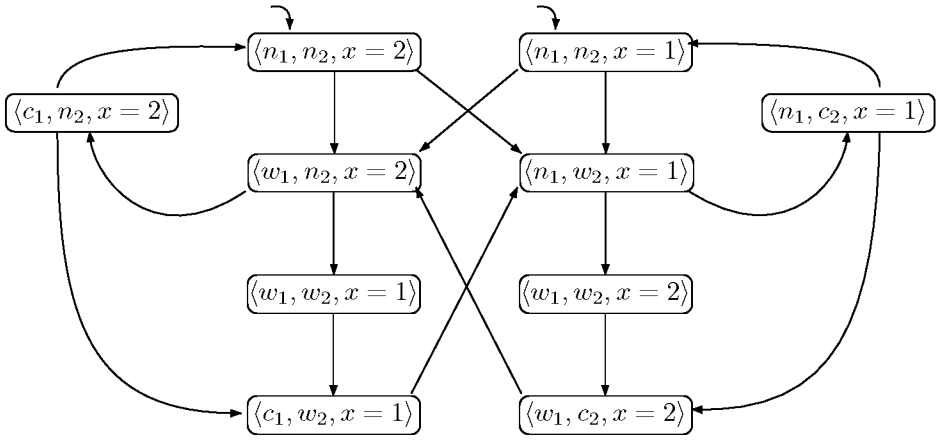


Figure 2.10: Transition system of Peterson's mutual exclusion algorithm.

and $\rangle$ in the program text, and is also indicated in the program graphs PG_1 and PG_2. We like to emphasize that this is not essential, and has only been done to simplify the transition system TS_{Pet}. Mutual exclusion is also ensured when both processes perform the assignments $b_i := \text{true}$ and $x := \ldots$ in this order but in a nonatomic way. Note that, for instance, the order "first $x := \ldots$, then $b_i := \text{true}$" does not guarantee mutual exclusion. This can be seen as follows. Assume that the location inbetween the assignments $x := \ldots$ and $b_i := \text{true}$ in program graph P_i is called req_i. The state sequence

$$\begin{array}{rrlll}
\langle noncrit_1, & noncrit_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle noncrit_1, & req_2, & x = 1, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle req_1, & req_2, & x = 2, & b_1 = \text{false}, & b_2 = \text{false}\rangle \\
\langle wait_1, & req_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle crit_1, & req_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{false}\rangle \\
\langle crit_1, & wait_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle \\
\langle crit_1, & crit_2, & x = 2, & b_1 = \text{true}, & b_2 = \text{true}\rangle
\end{array}$$

is an initial execution fragment where P_1 enters its critical section first (as $b_2 = \text{false}$) after which P_2 enters its critical section (as $x = 2$). As a result, both processes are simultaneously in their critical section and mutual exclusion is violated. ■

2.2.3 Handshaking

So far, two mechanisms for parallel processes have been considered: interleaving and shared-variable programs. In interleaving, processes evolve completely autonomously

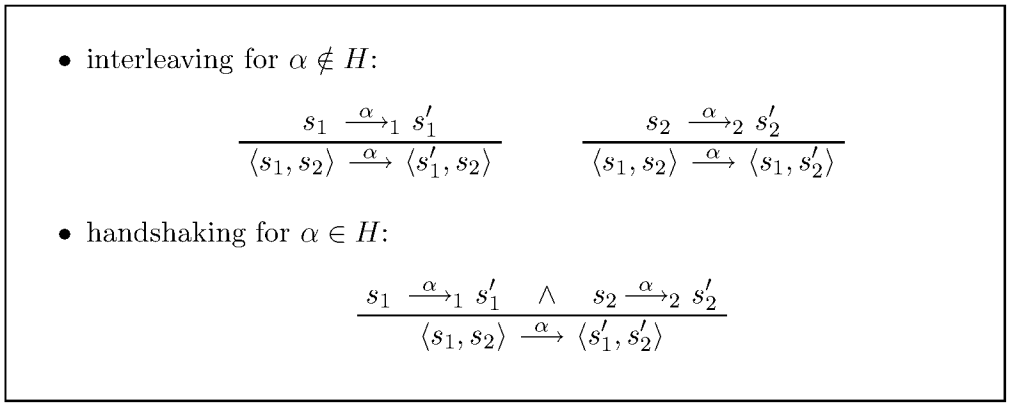


Figure 2.11: Rules for handshaking.

whereas according to the latter type processes "communicate" via shared variables. In this subsection, we consider a mechanism by which concurrent processes interact via handshaking. The term "handshaking" means that concurrent processes that want to interact have to do this in a *synchronous* fashion. That is to say, processes can interact only if they are both participating in this interaction at the same time—they "shake hands".

Information that is exchanged during handshaking can be of various nature, ranging from the value of a simple integer, to complex data structures such as arrays or records. In the sequel, we do not dwell upon the content of the exchanged messages. Instead, an abstract view is adopted and only communication (also called synchronization) actions are considered that represent the occurrence of a handshake and not the content.

To do so, a set H of *handshake actions* is distinguished with $\tau \notin H$. Only if both participating processes are ready to execute the same handshake action, can message passing take place. All actions outside H (i.e., actions in $Act \setminus H$) are independent and therefore can be executed autonomously in an interleaved fashion.

Definition 2.26. Handshaking (Synchronous Message Passing)

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, i=1,2 be transition systems and $H \subseteq Act_1 \cap Act_2$ with $\tau \notin H$. The transition system $TS_1 \parallel_H TS_2$ is defined as follows:

$$TS_1 \parallel_H TS_2 \;=\; (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where $L(\langle s_1, s_2\rangle) \;=\; L_1(s_1) \cup L_2(s_2)$, and where the transition relation $\rightarrow$ is defined by the rules shown in Figure 2.11. ■

Notation: $TS_1 \parallel TS_2$ abbreviates $TS_1 \parallel_H TS_2$ for $H = Act_1 \cap Act_2$.

Remark 2.27. Empty Set of Handshake Actions

When the set H of handshake actions is empty, all actions of the participating processes can take place autonomously, i.e., in this special case, handshaking reduces to interleaving

$$TS_1 \parallel_\varnothing TS_2 \quad = \quad TS_1 \mid\mid\mid TS_2.$$

■

The operator $\parallel_H$ defines the handshaking between two transition systems. Handshaking is commutative, but not associative in general. That is, in general we have

$$TS_1 \parallel_H (TS_2 \parallel_{H'} TS_3) \neq (TS_1 \parallel_H TS_2) \parallel_{H'} TS_3 \quad \text{for } H \neq H'.$$

However, for a fixed set H of handshake actions over which all processes synchronize, the operator $\parallel_H$ is associative. Let

$$TS \; = \; TS_1 \parallel_H TS_2 \parallel_H \ldots \parallel_H TS_n,$$

denote the parallel composition of transition systems TS_1 through TS_n where $H \subseteq Act_1 \cap \ldots \cap Act_n$ is a subset of the set of actions Act_i of all transition systems. This form of *multiway* handshaking is appropriate to model broadcasting, a communication form in which a process can transmit a datum to several other processes simultaneously.

In many cases, processes communicate in a pairwise fashion over their common actions. Let $TS_1 \| \ldots \| TS_n$ denote the parallel composition of TS_1 through TS_n (with $n > 0$) where TS_i and TS_j $(0 < i \neq j \leqslant n)$ synchronize over the set of actions $H_{i,j} = Act_i \cap Act_j$ such that $H_{i,j} \cap Act_k = \varnothing$ for $k \notin \{ i, j \}$. It is assumed that $\tau \notin H_{i,j}$. The formal definition of $TS_1 \| \ldots \| TS_n$ is analogous to Definition 2.26. The state space of $TS_1 \| \ldots \| TS_n$ results from the Cartesian product of the state spaces of TS_i. The transition relation $\rightarrow$ is defined by the following rules:

- for $\alpha \in Act_i \setminus (\bigcup_{\substack{0<j\leqslant n \\ i\neq j}} H_{i,j})$ and $0 < i \leqslant n$:

$$\frac{s_i \xrightarrow{\alpha}_i s_i'}{\langle s_1, \ldots, s_i, \ldots, s_n \rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots s_n \rangle}$$

- for $\alpha \in H_{i,j}$ and $0 < i < j \leqslant n$:

$$\frac{s_i \xrightarrow{\alpha}_i s_i' \quad \wedge \quad s_j \xrightarrow{\alpha}_j s_j'}{\langle s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n \rangle \xrightarrow{\alpha} \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n \rangle}$$

According to the first rule, components can execute actions that are not subject to handshaking in a completely autonomous manner as in interleaving. The second rule states that processes TS_i and TS_j $(i \neq j)$ have to perform every handshaking action in $Act_i \cap Act_j$ together. These rules are in fact just generalizations of those given in Figure 2.11.

Example 2.28. Mutual Exclusion by Means of an Arbiter

An alternative solution to the mutual exclusion problem between processes P_1 and P_2 (as before) is to model the binary semaphore that regulates access to the critical section by a separate parallel process that interacts with P_1 and P_2 by means of handshaking. For simplicity, we ignore the waiting phase and assume that P_i simply alternates infinitely often between noncritical and critical sections. Assume (much simplified) transition system representations TS_1 and TS_2 with just two states: $crit_i$ and $noncrit_i$. The new process, named *Arbiter*, mimics a binary semaphore (see Figure 2.12). P_1 and P_2 communicate with the *Arbiter* via handshaking over the set $H = \{ request, rel \}$. Accordingly, the actions *request* (requesting to access the critical section) and *rel* (to leave the critical section) have to be executed synchronously with the *Arbiter*. The complete system

$$TS_{Arb} \;=\; (TS_1 \;|||\; TS_2) \parallel Arbiter$$

guarantees mutual exclusion since there are no states of TS_{Arb} where both P_1 and P_2 are in their critical section (see bottom part of Figure 2.12). Note that in the initial state of $TS_1 \,|||\, TS_2$, the *Arbiter* determines which process will enter the critical section next. ■

Example 2.29. Booking System

Consider a (strongly simplified) booking system at a cashier of a supermarket. The system consists of three processes: the bar code reader *BCR*, the actual booking program *BP*, and the printer *Printer*. The bar code reader reads a bar code and communicates the data of the just scanned product to the booking program. On receiving such data, the booking program transmits the price of the article to the printer that prints the article Id together with the price on the receipt. The interactions between the bar code reader and the booking program, and between the booking program and the printer is performed by handshaking. Each process consist of just two states, named 0 and 1 (see Figure 2.13 for the transitions systems of *BCR*, *BP*, and *Printer*).

The complete system is given by:

$$BCR \parallel BP \parallel Printer.$$

The transition system of the overall system is depicted in Figure 2.14 on page 52. The initial global state of this system is $\langle 0, 0, 0 \rangle$, or in short, 000. In global state 010, e.g., the

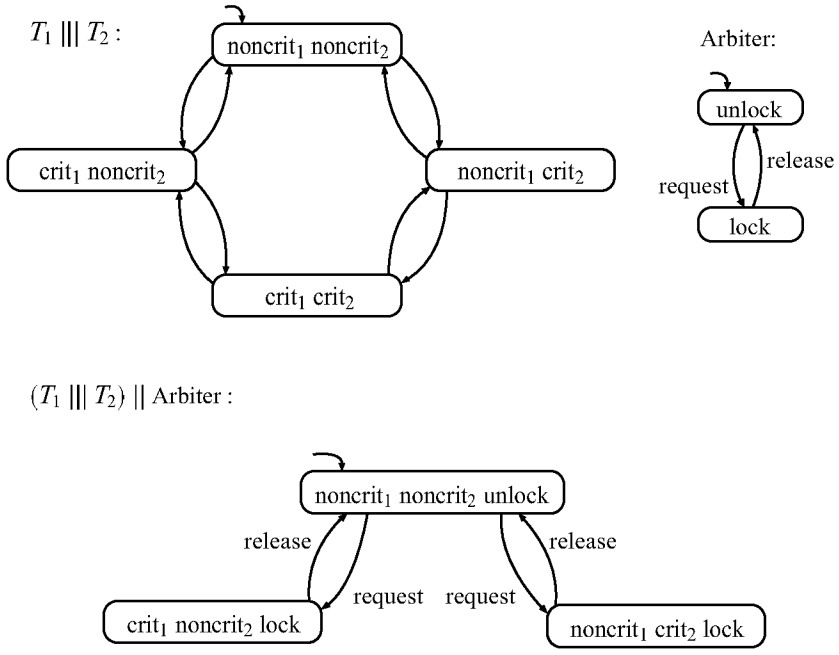


Figure 2.12: Mutual exclusion using handshaking with arbiter process.

nondeterminism stands for the concurrent execution of the actions scanning the bar code and the synchronous transfer of the price to the printer. ■

Example 2.30. Railroad Crossing

For a railroad crossing a control system needs to be developed that on receipt of a signal indicating that a train is approaching closes the gates, and only opens these gates after the train has sent a signal indicating that it crossed the road. The requirement that should be met by the control system is that the gates are always closed when the train is crossing the road. The complete system consists of the three components *Train*, *Gate*, and *Controller*:

$$\mathit{Train} \parallel \mathit{Gate} \parallel \mathit{Controller}.$$

Figure 2.15 depicts the transition systems of these components from left (modeling the *Train*) to right (modeling the *Gate*). For simplicity, it is assumed that all trains pass the relevant track section in the same direction—from left to right. The states of the transition system for the *Train* have the following intuitive meaning: in state *far* the train is not close to the crossing, in state *near* it is approaching the crossing and has just sent a signal to notify this, and in state *in* it is at the crossing. The states of the *Gate* have the obvious interpretation. The state changes of the *Controller* stand for handshaking with

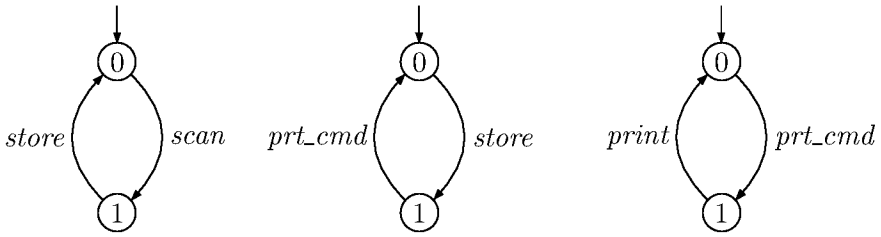


Figure 2.13: The components of the book keeping example.

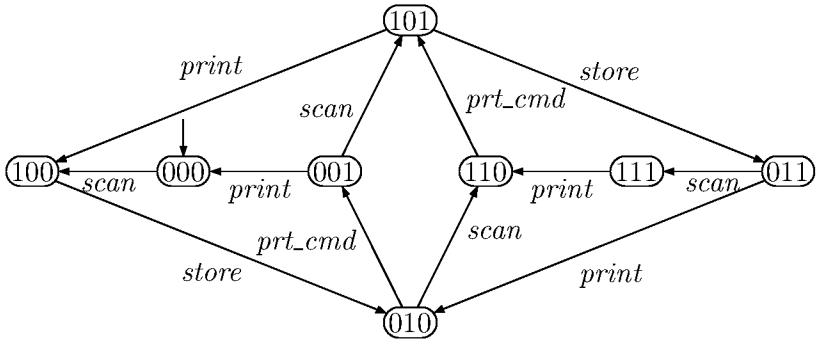


Figure 2.14: Transition system representation of the booking system.

the trains (via the actions *approach* and *exit*) and the *Gate* (via the actions *lower* and *raise* via which the *Controller* causes the gate to close or to open, respectively).

Figure 2.16 (above) illustrates the transition system of the overall system. A closer inspection of this transition system reveals that the system suffers from a design flaw. This can be seen from the following initial execution fragment:

$$\langle far, 0, up\rangle \xrightarrow{approach} \langle near, 1, up\rangle \xrightarrow{enter} \langle in, 1, up\rangle$$

in which the gate is about to close, while the train is (already) at the crossing. The nondeterminism in global state $\langle near, 1, up\rangle$ stands for concurrency: the train approaches the crossing, while the gate is being closed. In fact, the basic concept of the design is correct if and only if closing the gate does not take more time than the train needs to get to the crossing once it signals—"I am approaching". Such real-time constraints cannot be formulated by the concepts introduced so far. The interleaving representation for parallel systems is completely *time-abstract*. In Chapter 9, concepts and techniques will be introduced to specify and verify such real-time aspects. ■

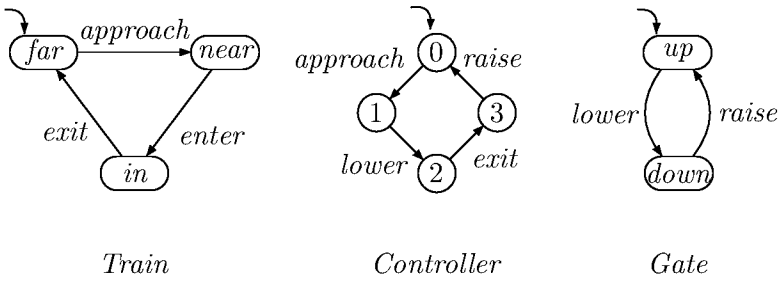


Figure 2.15: The components of the railroad crossing.

2.2.4 Channel Systems

This section introduces channel systems, parallel systems where processes communicate via so-called *channels*, i.e., first-in, first-out buffers that may contain messages. We consider channel systems that are closed. That is to say, processes may communicate with other processes in the system (via channels), but not with processes outside the system. Channel systems are popular for describing communication protocols and form the basis of PROMELA, the input language of the SPIN model checker.

Intuitively, a channel system consists of n (data-dependent) processes P_1 through P_n. Each P_i is specified by a program graph PG_i which is extended with *communication actions*. Transitions of these program graphs are either the usual conditional transitions (labeled with guards and actions) as before, or one of the communication actions with their respective intuitive meaning:

$c!v$	transmit the value v along channel c,
$c?x$	receive a message via channel c and assign it to variable x.

When considering channel c as buffer, the communication action $c!v$ puts value v (at the rear of) the buffer whereas $c?x$ retrieves an element from (the front of) the buffer while assigning it to x. It is assumed implicitly that variable x is of the right type, i.e., it has a type that is compatible to that of the messages that are put into channel c. Let

$$Comm \;=\; \Big\{\; c!v,\; c?x \;\mid\; c \in Chan,\; v \in dom(c),\; x \in Var \text{ with } \; dom(x) \supseteq dom(c) \;\Big\}$$

denote the set of communication actions where *Chan* is a finite set of channels with typical element c.

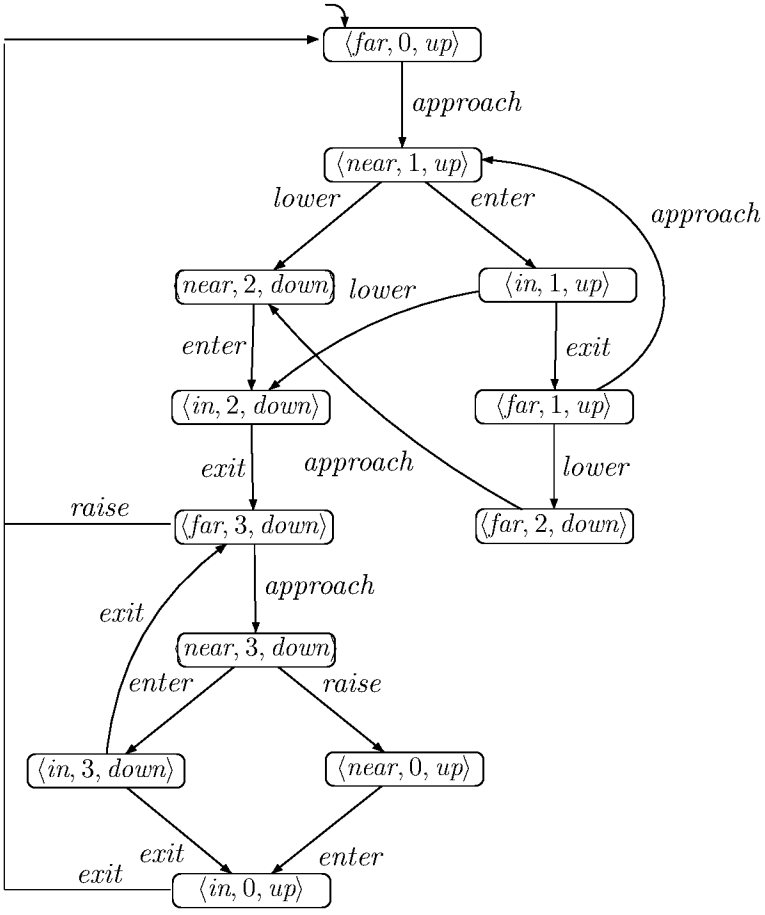


Figure 2.16: Transition system for the railroad crossing.

A channel c has a (finite or infinite) *capacity* indicating the maximum number of messages it can store, and a type (or *domain*) specifying the type of messages that can be transmitted over c. Each channel c has a capacity $cap(c) \in \mathbb{N} \cup \{\infty\}$, and a domain $dom(c)$. For a channel c that can only transfer bits, $dom(c) = \{0, 1\}$. If complete texts (of maximum length of 200, say) need to be transmitted over channel c, then another type of channel has to be used such that $dom(c) = \Sigma^{\leqslant 200}$, where Σ is the alphabet that forms the basis of the texts, e.g., Σ is the set of all letters and special characters used in German texts.

The capacity of a channel defines the size of the corresponding buffer, i.e., the number of messages not yet read that can be stored in the buffer. When $cap(c) \in \mathbb{N}$, c is a channel with finite capacity; $cap(c) = \infty$ indicates that c has an infinite capacity. Note that the special case $cap(c) = 0$ is permitted. In this case, channel c has *no* buffer. Communication via such a channel c corresponds to handshaking (simultaneous transmission and receipt, i.e., synchronous message passing) plus the exchange of some data. When $cap(c) > 0$, there is a "delay" between the transmission and the receipt of a message, i.e., sending and reading of the same message take place at different moments. This is called asynchronous message passing. Sending and reading a message from a channel with a nonzero capacity can never appear simultaneously. By means of channel systems, both synchronous and asynchronous message passing can thus be modeled.

Definition 2.31. Channel System

A program graph over $(Var, Chan)$ is a tuple

$$PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$$

according to Definition 2.13 (page 32) with the only difference that

$$\hookrightarrow \quad \subseteq \quad Loc \times (Cond(Var) \times (Act \cup Comm) \times Loc.$$

A *channel system* CS over $(Var, Chan)$ consists of program graphs PG_i over $(Var_i, Chan)$ (for $1 \leqslant i \leqslant n$) with $Var = \bigcup_{1 \leqslant i \leqslant n} Var_i$. We denote

$$CS \;=\; [PG_1 \mid \ldots \mid PG_n] \;.$$

■

The transition relation $\hookrightarrow$ of a program graph over $(Var, Chan)$ consists of two types of conditional transitions. As before, conditional transitions $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ are labeled with guards and actions. These conditional transitions can happen whenever the guard holds. Alternatively, conditional transitions may be labeled with communication actions. This yields conditional transitions of type $\ell \overset{g:c!v}{\hookrightarrow} \ell'$ (for sending v along c) and $\ell \overset{g:c?x}{\hookrightarrow} \ell'$ (for receiving a message along c). When can such conditional transitions happen? Stated

differently, when are these conditional transitions executable? This depends on the current variable evaluation and the capacity and content of the channel c. For the sake of simplicity assume in the following that the guard is satisfied.

- *Handshaking.* If $cap(c) = 0$, then process P_i can transmit a value v over channel c by performing
$$\ell_i \overset{c!v}{\hookrightarrow} \ell_i'$$
only if another process P_j, say, "offers" a complementary receive action, i.e., can perform
$$\ell_j \overset{c?x}{\hookrightarrow} \ell_j'.$$
P_i and P_j should thus be able to perform $c!v$ (in P_i) and $c?x$ (in P_j) simultaneously. Then, message passing can take place between P_i and P_j. The effect of message passing corresponds to the (distributed) assignment $x := v$.

  Note that when handshaking is only used for synchronization purposes and not for data transfer, the name of the channel as well as the value v are not of any relevance.

- *Asynchronous message passing.* If $cap(c) > 0$, then process P_i can perform the conditional transition
$$\ell_i \overset{c!v}{\hookrightarrow} \ell_i'$$
if and only if channel c is not full, i.e., if less than $cap(c)$ messages are stored in c. In this case, v is stored at the rear of the buffer c. Channels are thus considered as first-in, first-out buffers. Accordingly, P_j may perform
$$\ell_j \overset{c?x}{\hookrightarrow} \ell_j'$$
if and only if the buffer of c is not empty. In this case, the first element v of the buffer is extracted and assigned to x (in an atomic manner). This is summarized in Table 2.1.

	executable if ...	effect
$c!v$	c is not "full"	$Enqueue(c, v)$
$c?x$	c is not empty	$\langle x := Front(c)\,;\ Dequeue(c)\rangle$;

Table 2.1: Enabledness and effect of communication actions if $cap(c) > 0$.

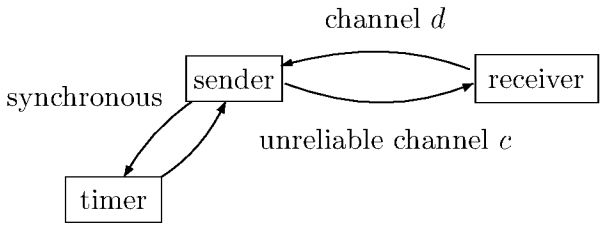


Figure 2.17: Schematic view of the alternating bit protocol.

Channel systems are often used to model communication protocols. One of the most elementary and well-known protocols is the alternating bit protocol.

Example 2.32. Alternating Bit Protocol (ABP)

Consider a system essentially consisting of a sender S and a receiver R that communicate with each other over channels c and d, see Figure 2.17. It is asssumed that both channels have an unlimited buffer, i.e., $cap(c) = cap(d) = \infty$. Channel c is unreliable in the sense that data may get lost when being transmitted from the sender S to channel c. Once messages are stored in the buffer of channel c, they are neither corrupted nor lost. Channel d is assumed to be perfect. The goal is to design a communication protocol that ensures any *distinct* transmitted datum by S to be delivered to R. To ensure this in the presence of possible message losses, sender S resorts to retransmissions. Messages are transmitted one by one, i.e., S starts sending a new message once the transmission of the previous message has been successful. This is a simple flow control principle, known as "send-and-wait".

We abstract from the real activities of S and R and, instead, concentrate on a simplified representation of the communication structure of the system. S sends the successive messages $m_0, m_1, \ldots$ together with control bits $b_0, b_1, \ldots$ over channel c to R. Transmitted messages are thus pairs:

$$\langle m_0, 0\rangle, \langle m_1, 1\rangle, \langle m_2, 0\rangle, \langle m_3, 1\rangle, \ldots$$

On receipt of $\langle m, b\rangle$ (along channel c), R sends an acknowledgment (ack) consisting of the control bit b just received. On receipt of ack $\langle b\rangle$, S transmits a new message m' with control bit $\neg b$. If, however, S has to wait "too long" for the ack, it timeouts and retransmits $\langle m, b\rangle$. The program graphs for S and R are sketched in Figure 2.18 and 2.19. For simplicity, the data that is transmitted is indicated by m instead of m_i.

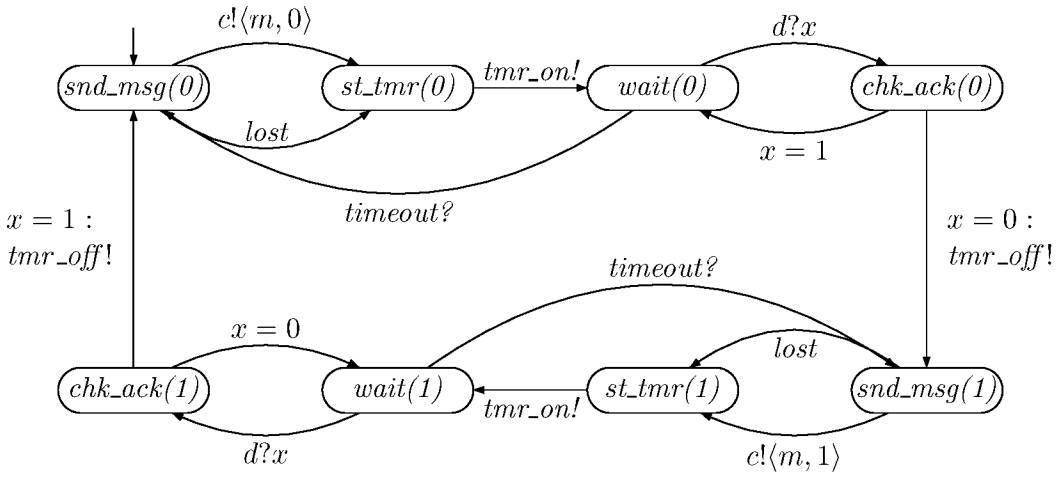


Figure 2.18: Program graph of ABP sender S.

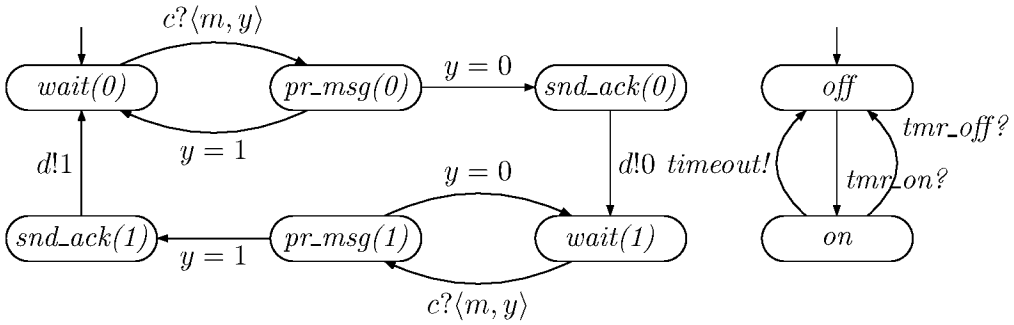


Figure 2.19: Program graph of (left) ABP receiver R and (right) *Timer*.

Control bit b—also called the alternating bit—is thus used to distinguish retransmissions of m from transmissions of subsequent (and previous) messages. Due to the fact that the transmission of a new datum is initiated only when the last datum has been received correctly (and this is acked), a single bit is sufficient for this purpose and notions like, e.g., sequence numbers, are not needed.

The timeout mechanism of S is modeled by a *Timer* process. S activates this timer on sending a message (along c), and it stops the timer on receipt of an ack. When raising a timeout, the timer signals to S that a retransmission should be initiated. (Note that due to this way of modeling, so-called premature timeouts may occur, i.e., a timeout may occur whereas an ack is still on its way to S.) The communication between the timer and S is modeled by means of handshaking, i.e., by means of channels with capacity 0.

The complete system can now be represented as the following channel system over $Chan = \{ c, d, tmr\_on, tmr\_off, timeout \}$ and $Var = \{ x, y, m_i \}$:

$$ABP \;=\; [S \mid Timer \mid R\,]\,.$$

■

The following definition formalizes the successive behavior of a channel system by means of a transition system. The basic concept is similar to the mapping from a program graph onto a transition system. Let $CS = [PG_1 \mid \ldots \mid PG_n]$ be a channel system over $(Chan, Var)$. The (global) states of $TS(CS)$ are tuples of the form

$$\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle$$

where ℓ_i indicates the current location of component PG_i, η keeps track of the current values of the variables, and ξ records the current content of the various channels (as sequences). Formally, $\eta \in Eval(Var)$ is an evaluation of the variables (as we have encountered before), and ξ is a *channel evaluation*, i.e., a mapping from channel $c \in Chan$ onto a sequence $\xi(c) \in dom(c)^*$ such that the length of the sequence cannot exceed the capacity of c, i.e., $len(\xi(c)) \leqslant cap(c)$ where $len(\cdot)$ denotes the length of a sequence. $Eval(Chan)$ denotes the set of all channel evaluations. For initial states, the control components $\ell_i \in Loc_{0,i}$ must be initial and variable evaluation η must satisfy the initial condition g_0. In addition, every channel is initially assumed to be empty, denoted ε.

Before providing the details of the semantics of a transition system, let us introduce some notations. Channel evaluation $\xi(c) = v_1\, v_2\, \ldots v_k$ (where $cap(c) \geqslant k$) denotes that v_1 is at the front of the buffer of c, v_2 is the second element, etc., and v_k is the element at the rear of c. $len(\xi(c)) = k$ in this case. Let $\xi[c := v_1, \ldots, v_k]$ denote the channel evaluation where sequence $v_1, \ldots, v_k$ is assigned to c and all other channels are unaffected, i.e.,

$$\xi[c := v_1 \ldots v_k](c') \;=\; \begin{cases} \xi(c') & \text{if } c \neq c' \\ v_1 \ldots v_k & \text{if } c = c'. \end{cases}$$

The channel evaluation ξ_0 maps any channel to the empty sequence, denoted ε, i.e., $\xi_0(c) = \varepsilon$ for any channel c. Let $len(\varepsilon) = 0$. The set of actions of $TS(CS)$ consists of actions $\alpha \in Act_i$ of component PG_i and the distinguished symbol τ representing all communication actions in which data is exchanged.

Definition 2.33. Transition System Semantics of a Channel System

Let $CS = [PG_1 \mid \ldots \mid PG_n]$ be a channel system over $(Chan, Var)$ with

$$PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})\,, \qquad \text{for } 0 < i \leqslant n.$$

The transition system of CS, denoted $TS(CS)$, is the tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- $S \;=\; (Loc_1 \times \ldots \times Loc_n) \times Eval(Var) \times Eval(Chan)$,
- $Act \;=\; \biguplus_{0<i\leqslant n} Act_i \;\uplus\; \{\,\tau\,\}$,
- $\rightarrow$ is defined by the rules of Figure 2.20 (page 61),
- $I \;=\; \Big\{\; \langle \ell_1, \ldots, \ell_n, \eta, \xi_0 \rangle \;\mid\; \forall 0 < i \leqslant n.\; (\ell_i \in Loc_{0,i} \;\wedge\; \eta \models g_{0,i}) \;\Big\}$,
- $AP \;= \biguplus_{0<i\leqslant n} Loc_i \;\uplus\; Cond(Var)$,
- $L(\langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle) \;=\; \{\,\ell_1, \ldots, \ell_n\,\} \;\cup\; \{\, g \in Cond(Var) \mid \eta \models g \,\}$.

■

This definition is a formalization of the informal description of the interpretation of a channel system given before. Note that the labeling of the atomic propositions is similar to that for program graphs (see Definition 2.15). For the sake of simplicity, the above definition does not allow for propositions on channels. This could be accommodated by allowing for conditions on channels such as, e.g., "the channel c is empty" or "the channel c is full", and checking these conditions on the channel evaluation ξ in a state.

Example 2.34. Alternating Bit Protocol (Revisited)

Consider the alternating bit protocol that was modeled as a channel system in Example 2.32. The underlying transition system $TS(ABP)$ has, despite various simplifying assumptions, infinitely many states. This is, e.g., due to the fact that the timer may signal a timeout on each transmission of a datum by S resulting in infinitely many messages in channel c.

To clarify the functionality of the alternating bit protocol, consider two execution fragments represented by indicating the states of the various components (sender S, receiver R, the timer, and the contents of channels c and d). The first execution fragment shows the loss of a message. Here, R does not execute any action at all as it only acts if channel c contains at least one message:

sender S	timer	receiver R	channel c	channel d	event
$snd\_msg(0)$	*off*	$wait(0)$	$\varnothing$	$\varnothing$	
$st\_tmr(0)$	*off*	$wait(0)$	$\varnothing$	$\varnothing$	loss of message
$wait(0)$	*on*	$wait(0)$	$\varnothing$	$\varnothing$	
$snd\_msg(0)$	*off*	$wait(0)$	$\varnothing$	$\varnothing$	timeout
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

- interleaving for $\alpha \in Act_i$:

$$\frac{\ell_i \overset{g:\alpha}{\hookrightarrow} \ell_i' \quad \wedge \quad \eta \models g}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\alpha} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = \mathit{Effect}(\alpha, \eta)$.

- asynchronous message passing for $c \in \mathit{Chan}$, $\mathit{cap}(c) > 0$:
  - receive a value along channel c and assign it to variable x:

$$\frac{\ell_i \overset{g:c?x}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g \ \wedge \ \mathit{len}(\xi(c)) = k > 0 \ \wedge \ \xi(c) \ = \ v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi' \rangle}$$

    where $\eta' = \eta[x := v_1]$ and $\xi' = \xi[c := v_2 \ldots v_k]$.
  - transmit value $v \in \mathit{dom}(c)$ over channel c:

$$\frac{\ell_i \overset{g:c!v}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g \ \wedge \ \mathit{len}(\xi(c)) = k < \mathit{cap}(c) \ \wedge \ \xi(c) \ = \ v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta, \xi' \rangle}$$

    where $\xi' = \xi[c := v_1 \, v_2 \ldots v_k \, v]$.

- synchronous message passing over $c \in \mathit{Chan}$, $\mathit{cap}(c) = 0$:

$$\frac{\ell_i \overset{g_1:c?x}{\hookrightarrow} \ell_i' \ \wedge \ \eta \models g_1 \ \wedge \ \eta \models g_2 \ \wedge \ \ell_j \overset{g_2:c!v}{\hookrightarrow} \ell_j' \ \wedge \ i \neq j}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{\tau} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_j', \ldots, \ell_n, \eta', \xi \rangle}$$

  where $\eta' = \eta[x := v]$.

Figure 2.20: Rules for the transition relation of a channel system.

When the receiver R is in location *wait*(0) and receives a message, it anticipates receiving a message with either control bit 0 (as it expects) or with control bit 1, see left side of Figure 2.19. Symmetrically, in location *wait*(1) also the possibility is taken into account to receive the (unexpected) ack with control bit 0. The following execution fragment indicates why this unexpected possibility is essential to take into consideration. In a nutshell, the execution fragment shows that it may happen that R receives $\langle m, 0\rangle$, notifies this by means of sending an ack (with control bit 0), and switches to "mode 1"—waiting to receive a message with control bit 1. In the meanwhile, however, sender S has initiated a retransmission of $\langle m, 0\rangle$ (as it timed out). On receipt of this (unexpected) message, receiver R should act accordingly and ignore the message. This is exactly what happens. Note that if this possibility would not have been taken into consideration in the program graph of R, the system would have come to a halt.

sender S	timer	receiver R	channel c	channel d	event
snd_msg(0)	*off*	*wait*(0)	$\varnothing$	$\varnothing$	
st_tmr(0)	*off*	*wait*(0)	$\langle m, 0\rangle$	$\varnothing$	message with bit 0 transmitted
wait(0)	*on*	*wait*(0)	$\langle m, 0\rangle$	$\varnothing$	
snd_msg(0)	*off*	*wait*(0)	$\langle m, 0\rangle$	$\varnothing$	timeout
st_tmr(0)	*off*	*wait*(0)	$\langle m, 0\rangle\ \langle m, 0\rangle$	$\varnothing$	retransmission
st_tmr(0)	*off*	*pr_msg*(0)	$\langle m, 0\rangle$	$\varnothing$	receiver reads first message
st_tmr(0)	*off*	*snd_ack*(0)	$\langle m, 0\rangle$	$\varnothing$	
st_tmr(0)	*off*	*wait*(1)	$\langle m, 0\rangle$	0	receiver changes into mode-1
st_tmr(0)	*off*	*pr_msg*(1)	$\varnothing$	0	receiver reads retransmission
st_tmr(0)	*off*	*wait*(1)	$\varnothing$	0	and ignores it
⋮	⋮	⋮	⋮	⋮	

We conclude this example by pointing out a possible simplification of the program graph of the sender S. Since the transmission of acks (over channel d) is reliable, it is unnecessary (but not wrong) for S to verify the control bit of the ack in location *chk_ack*($\cdot$). If S is in location *wait*(0) and channel d is not empty, then the (first) message in d corresponds to the expected ack 0, since R acknowledges each message m exactly once regardless of how many times m is received. Therefore, the program graph of S could be simplified such that by the action sequence $d?x$; *timer_off*, it moves from location *wait*(0) to location *gen_msg*(1). Location *chk_ack*(0) may thus be omitted. By similar arguments, location *chk_ack*(1) may be omitted. If, however, channel d would be unreliable (like channel c), these locations are necessary. ■

Remark 2.35. Open Channel Systems

The rule for synchronous message passing is subject to the idea that there is a closed channel system that does not communicate with the environment. To model open channel systems, only the rule for handshaking has to be modified. If there is a channel c with $cap(c) = 0$ over which the channel system is communicating with the environment, the rules

$$\frac{\ell_i \overset{c!v}{\hookrightarrow} \ell_i'}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{c!v} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta, \xi \rangle}$$

and

$$\frac{\ell_i \overset{c?x}{\hookrightarrow} \ell_i' \quad \wedge \quad v \in dom(c)}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \xrightarrow{c?x} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta[x := v], \xi \rangle}$$

have to be used. The receipt of value v for variable x along channel c is modeled by means of nondeterminism that is resolved by the environment. That is to say, the environment selects value $v \in dom(c)$ in a purely nondeterministic way. ■

2.2.5 NanoPromela

The concepts that have been discussed in the previous sections (program graphs, parallel composition, channel systems) provide the mathematical basis for modeling reactive systems. However, for building automated tools for verifying reactive systems, one aims at simpler formalisms to specify the behavior of the system to be analyzed. On the one hand, such specification languages should be simple and easy to understand, such that nonexperts also are able to use them. On the other hand, they should be expressive enough to formalize the stepwise behavior of the processes and their interactions. Furthermore, they have to be equipped with a *formal semantics* which renders the intuitive meaning of the language commands in an unambiguous manner. In our case, the purpose of the formal semantics is to assign to each program of the specification language a transition system that can serve as a basis for the automated analysis, e.g., simulation or model checking against temporal logical specifications.

In this section, we present the core features of the language Promela, the input language for the prominent model checker SPIN by Holzmann [209]. Promela is short for "process metalanguage". Promela programs $\overline{\mathcal{P}}$ consist of a finite number of processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ to be executed concurrently. Promela supports communication over shared variables and message passing along either synchronous or buffered FIFO-channels. The formal semantics of a Promela-program can be provided by means of a *channel system*, which then can be unfolded into a transition system, as explained in Section 2.2.4. The stepwise behavior of the processes $\mathcal{P}_i$ is specified in Promela using a *guarded command language*

[130, 18] with several features of classical imperative programming languages (variable assignments, conditional and repetitive commands, sequential composition), communication actions where processes may send and receive messages from the channels, and atomic regions that avoid undesired interleavings. Guarded commands have already been used as labels for the edges of program graphs and channel systems. They consist of a condition (guard) and an action. Promela does not use action names, but specifies the effect of actions by statements of the guarded command language.

Syntax of nanoPromela We now explain the syntax and semantics of a fragment of Promela, called *nanoPromela*, which concentrates on the basic elements of Promela, but abstracts from details like variable declarations and skips several "advanced" concepts like abstract data types (arrays, lists, etc.) or dynamic process creation. A nanoPromela *program* consists of *statements* representing the stepwise behavior of the processes $\mathcal{P}_1, \ldots,$ $\mathcal{P}_n$ together with a Boolean condition on the initial values of the program variables. We write nanoPromela programs as:

$$\overline{\mathcal{P}} \;=\; [\mathcal{P}_1 | \ldots | \mathcal{P}_n].$$

The main ingredients of the statements that formalize the stepwise behavior of the processes $\mathcal{P}_i$ are the atomic commands `skip`, variable assignments $x :=$ `expr`, communication actions $c?x$ (reading a value for variable x from channel c) and $c!$`expr` (sending the current value of an expression over channel c), conditional commands (if-then-else) and (while)loops. Instead of the standard if-then-else constructs or whileloops, nanoPromela supports nondeterministic choices and allows specifying a finite number of guarded commands in conditional and repetitive commands.

Variables, Expressions and Boolean Expressions The variables in a nanoPromela program $\overline{\mathcal{P}}$ may be typed (integer, Boolean, char, real, etc.) and either global or local to some process of $\mathcal{P}_i$. Similarly, data domains have to be specified for the channels and they have to be declared to be synchronous or fifo-channels of a predefined capacity. We skip the details of variable and channel declarations, as they are irrelevant for the purposes of this chapter. As local variables can be renamed in case they occur in more than one process or as the name of a global variable, we may treat all variables as global variables. Thus, we assume that *Var* is a set of variables occurring in $\overline{\mathcal{P}}$ and that for any a variable name x the domain (type) of x is given as the set $dom(x)$. Furthermore, in the declaration part of a Promela program, the type of a channel is specified. We simply write here $dom(c)$ for the type (domain) of channel c and $cap(c)$ for its capacity. In addition, we assume that the variable declaration of program $\overline{\mathcal{P}}$ contains a Boolean expression that specifies the legal initial values for the variables $x \in Var$.

$$
\begin{array}{rcl}
\mathsf{stmt} & ::= & \texttt{skip} \mid x := \mathsf{expr} \mid c?x \mid c!\mathsf{expr} \mid \\
& & \mathsf{stmt}_1 \,;\, \mathsf{stmt}_2 \mid \texttt{atomic}\{\mathsf{assignments}\} \mid \\
& & \textbf{if} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{fi} \quad \mid \\
& & \textbf{do} \quad :: g_1 \Rightarrow \mathsf{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathsf{stmt}_n \quad \textbf{do}
\end{array}
$$

Figure 2.21: Syntax of nanoPromela-statements.

The intuitive meaning of an assignment $x := \mathsf{expr}$ is obvious: variable x is assigned the value of the expression expr given the current variable evaluation. The precise syntax of the expressions and Boolean expressions is not of importance here. We may assume that the expressions used in assignments for variable x are built by constants in $dom(x)$, variables y of the same type as x (or a subtype of x), and operators on $dom(x)$, such as Boolean connectives $\wedge$, $\vee$, $\neg$, etc. for $dom(x) = \{0, 1\}$ and arithmetic operators $+$, $*$, etc. for $dom(x) = \mathbb{R}$.[3] The guards are Boolean expressions that impose conditions on the values of the variables, i.e., we treat the guards as elements of $Cond(Var)$.

Statements The syntax of the *statements* that specify the behavior of the nanoPromela-processes is shown in Figure 2.21 on page 65. Here, x is a variable in Var, expr an expression, and c is a channel of arbitrary capacity. Type consistency of the variable x and the expression expr is required in assignments $x := \mathsf{expr}$. Similarly, for the message-passing actions $c?x$ and $c!\mathsf{expr}$ we require that $dom(c) \subseteq dom(x)$ and that the type of expr corresponds to the domain of c. The g_i's in **if**–**fi**- and **do**–**od**-statements are *guards*. As mentioned above, we assume that $g_i \in Cond(Var)$. The body assignments of an atomic region is a nonempty sequential composition of assignments, i.e., assignments has the form

$$x_1 := \mathsf{expr}_1 \,;\, x_2 := \mathsf{expr}_2 \,;\, \ldots \,;\, x_m := \mathsf{expr}_m$$

where $m \geqslant 1$, $x_1, \ldots, x_m$ are variables and $\mathsf{expr}_1, \ldots, \mathsf{expr}_m$ expressions such that the types of x_i and expr_i are compatible.

Intuitive Meaning of the Commands Before presenting the formal semantics, let us give some informal explanations on the meaning of the commands. `skip` stands for a process that terminates in one step, without affecting the values of the variables or contents of the channels. The meaning of assignments is obvious. $\mathsf{stmt}_1 \,;\, \mathsf{stmt}_2$ denotes sequential composition, i.e., stmt_1 is executed first and after its termination stmt_2 is executed. The concept of *atomic regions* is realized in nanoPromela by statements of the

[3] For simplicity, the operators are supposed to be total. For operators that require special arguments (e.g., division requires a nonzero second argument) we assume that the corresponding domain contains a special element with the meaning of "undefined".

form `atomic{stmt}`. The effect of atomic regions is that the execution of stmt is treated as an atomic step that cannot be interleaved with the activities of other processes. As a side effect atomic regions can also serve as a compactification technique that compresses the state space by ignoring the intermediate configurations that are passed during the executions of the commands inside an atomic region. The assumption that the body of an atomic region consists of a sequence of assignments will simplify the inference rules given below.

Statements build by **if**–**fi** or **do**–**od** are generalizations of standard if-then-else commands and whileloops. Let us first explain the intuitive meaning of conditional commands. The statement

$$\textbf{if } :: g_1 \Rightarrow \mathsf{stmt}_1 \ \ldots \ :: g_n \Rightarrow \mathsf{stmt}_n \textbf{ fi}$$

stands for a nondeterministic choice between the statements stmt_i for which the guard g_i is satisfied in the current state, i.e., g_i holds for the current valuation of the variables. We assume a *test-and-set semantics* where the evaluation of the guards, the choice between the enabled guarded commands and the execution of the first atomic step of the selected statement, are performed as an atomic unit that cannot be interleaved with the actions of concurrent processes. If none of the guards $g_1, \ldots, g_n$ is fulfilled in the current state, then the **if**–**fi**–command *blocks*. Blocking has to be seen in the context of the other processes that run in parallel and that might abolish the blocking by changing the values of shared variables such that one or more of the guards may eventually evaluate to true. For instance, the process given by the statement

$$\textbf{if } :: y > 0 \ \Rightarrow \ x := 42 \textbf{ fi}$$

in a state where y has the value 0 waits until another process assigns a nonzero value to y. Standard if-then-else commands, say "**if** g **then** stmt_1 **else** stmt_2 **fi**", of imperative programming languages can be obtained by:

$$\textbf{if } :: g \Rightarrow \mathsf{stmt}_1 \ :: \neg g \Rightarrow \mathsf{stmt}_2 \textbf{ fi},$$

while statements "**if** g **then** stmt_1 **fi**" without an else option are modeled by:

$$\textbf{if } :: g \Rightarrow \mathsf{stmt}_1 \ :: \neg g \Rightarrow \mathtt{skip} \textbf{ fi}.$$

In a similar way, the **do**–**od**-command generalizes whileloops. These specify the repetition of the body, as long as one of the guards is fulfilled. That is, statements of the form:

$$\textbf{do } :: g_1 \Rightarrow \mathsf{stmt}_1 \ \ldots \ :: g_n \Rightarrow \mathsf{stmt}_n \textbf{ od}$$

stand for the iterative execution of the nondeterministic choice among the guarded commands $g_i \Rightarrow \mathsf{stmt}_i$ where guard g_i holds in the current configuration. Unlike conditional commands, **do**–**od**-loops do not block in a state if all guards are violated. Instead, if

$g_1, \ldots, g_n$ do not hold in the current state, then the whileloop is aborted. In fact, a single guarded loop **do** :: $g \Rightarrow$ stmt **od** has the same effect as an ordinary whileloop "**while** g **do** stmt **od**" with body stmt and termination condition $\neg g$. (As opposed to Promela, loops are not terminated by the special command "break".)

Example 2.36. Peterson's Mutual Exclusion Algorithm

Peterson's mutual exclusion algorithm for two processes (see Example 2.25 on page 45) can be specified in nanoPromela as follows. We deal with two Boolean variables b_1 and b_2 and the variable x with domain $dom(x) = \{1, 2\}$ and two Boolean variables $crit_1$ and $crit_2$. The activities of the processes inside their noncritical sections are modeled by the action `skip`. For the critical section, we use the assignment $crit_i :=$ true. Initially, we have $b_1 = b_2 = crit_1 = crit_2 =$ false, while x is arbitrary. Then the nanoPromela-code of process $\mathcal{P}_1$ is given by the statement

do :: true $\Rightarrow$ `skip`;
`atomic`$\{b_1 :=$ true$; x := 2\}$;
if :: $(x = 1) \vee \neg b_2 \Rightarrow crit_1 :=$ true **fi**
`atomic`$\{crit_1 :=$ false$; b_1 :=$ false$\}$
od

The statement for modeling the second process is similar. The infinite repetition of the three phases "noncritical section", "waiting phase" and "critical section" is modeled by the **do**–**od**-loop with the trivial guard true. The request action corresponds to the statement `atomic`$\{b_1 :=$ true$; x := 2\}$ and the release action to the statement `atomic`$\{crit_1 :=$ false$; b_1 :=$ false$\}$. The waiting phase where process $\mathcal{P}_1$ has to await until $x = 1$ or $b_2 =$ false is modeled by the **if**–**fi**-command.

The use of atomic regions is not necessary, but serves here as a compactification technique. As we mentioned in Example 2.25, the request action can also be split into the two assignments $b_1 :=$ true and $x := 2$. As long as both assignments are inside an atomic region the order of the assignments $b_1 :=$ true and $x := 2$ is irrelevant. If, however, we drop the atomic region, then we have to use the order $b_1 :=$ true; $x := 2$, as otherwise the mutual exclusion property cannot be ensured. That is, the process

do :: true $\Rightarrow$ `skip`;
$x := 2$;
$b_1 :=$ true;
if :: $(x = 1) \vee \neg b_2 \Rightarrow crit_1 :=$ true **fi**
`atomic`$\{crit_1 :=$ false$; b_1 :=$ false$\}$
od

for $\mathcal{P}_1$ together with the symmetric protocol for $\mathcal{P}_2$ constitutes a nanoPromela program where the mutual exclusion property "never $crit_1 = crit_2 =$ true" cannot be guaranteed. ■

Example 2.37. Vending Machine

In the above example there are no nondeterministic choices caused by a conditional or repetitive command. For an example where nondeterminism arises through simultaneously enabled guarded commands of a loop, consider the beverage vending machine of Example 2.14 (page 33). The following nanoPromela program describes its behavior:

$$
\begin{array}{lll}
\textbf{do} & :: & \text{true} \Rightarrow \\
& & \quad \texttt{skip}; \\
& & \quad \textbf{if} \quad :: \; nsoda > 0 \;\Rightarrow\; nsoda := nsoda - 1 \\
& & \qquad\quad :: \; nbeer > 0 \;\Rightarrow\; nbeer := nbeer - 1 \\
& & \qquad\quad :: \; nsoda = nbeer = 0 \;\Rightarrow\; \texttt{skip} \\
& & \quad \textbf{fi} \\
& :: & \text{true} \;\Rightarrow\; \texttt{atomic}\{nbeer := max; nsoda := max\} \\
\textbf{od} & &
\end{array}
$$

In the starting location, there are two options that are both enabled. The first is the insertion of a coin by the user, modeled by the command `skip`. The first two options of the **if**–**fi**-command represent the cases where the user selects soda or beer, provided some bottles of soda and beer, respectively, are left. The third guarded command in the **if**–**fi** substatement applies to the case where neither soda nor beer is available anymore and the machine automatically returns to the initial state. The second alternative that is enabled in the starting location is the refill action whose effect is specified by the atomic region where the variables *nbeer* and *nsoda* are reset to *max*. ■

Semantics The *operational semantics* of a nanoPromela-statement with variables and channels from $(Var, Chan)$ is given by a *program graph* over $(Var, Chan)$. The program graphs $PG_1, \ldots, PG_n$ for the processes $\mathcal{P}_1, \ldots, \mathcal{P}_n$ of a nanoPromela program $\overline{\mathcal{P}} = [\mathcal{P}_1 | \ldots | \mathcal{P}_n]$ constitute a *channel system* over $(Var, Chan)$. The transition system semantics for channel systems (Definition 2.31 on page 55) then yields a transition system $TS(\overline{\mathcal{P}})$ that formalizes the stepwise behavior of $\overline{\mathcal{P}}$.

The program graph associated with a nanoPromela-statement stmt formalizes the control flow when executing stmt. That is, the substatements play the role of the locations. For modeling termination, a special location exit is used. Roughly speaking, any guarded command $g \Rightarrow$ stmt corresponds to an edge with the label $g : \alpha$ where α stands for the first action of stmt. For example, for the statement

$$
\mathsf{cond\_cmd} \;=\; \begin{array}{lll}
\textbf{if} & :: \; x > 1 & \Rightarrow \; y := x + y \\
& :: \; \text{true} & \Rightarrow \; x := 0; \; y := x \\
\textbf{fi} & &
\end{array}
$$

from cond_cmd – viewed as a location of a program graph – there are two edges: one with the guard $x > 1$ and action $y := x + y$ leading to exit, and one with the guard true and action $x := 0$ yielding the location for the statement $y := x$. Since $y := x$ is deterministic there is a single edge with guard true, action $y := x$ leading to location exit.

As another example, consider the statement

$$\begin{array}{lcllll} \mathsf{loop} & = & \textbf{do} & :: & x > 1 & \Rightarrow \quad y := x + y \\ & & & :: & y < x & \Rightarrow \quad x := 0;\ y := x \\ & & \textbf{od} & & & \end{array}$$

Here, the repetition semantics of the **do**–**od**-loop is modeled by returning the control to stmt whenever the body of the selected alternative has been executed. Thus, from location loop there are three outgoing edges, see Figure 2.22 on page 69. One is labeled with the guard $x > 1$ and action $y := x + y$ and leads back to location loop. The second edge has the guard $y < x$ and action $x := 0$ and leads to the statement $y := x$; loop. The third edge covers the case where the loop terminates. It has the guard $\neg(x > 1) \wedge \neg(y < x)$ and an action without any effect on the variables and leads to location exit.

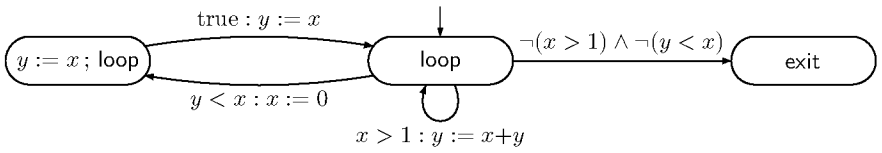


Figure 2.22: Program graph for a loop

The goal is now to formalize the ideas sketched above. We start with a formal definition of substatements of stmt. Intuitively, these are the potential locations of intermediate states during the execution of stmt.

Notation 2.38. Substatement

The set of substatements of a nanoPromela-statement stmt is recursively defined. For statements $\mathsf{stmt} \in \{\mathtt{skip}, x := \mathsf{expr}, c?x, c!\mathsf{expr}\}$ the set of substatements is $sub(\mathsf{stmt}) = \{\mathsf{stmt}, \mathsf{exit}\}$. For sequential composition let

$$sub(\mathsf{stmt}_1\,;\,\mathsf{stmt}_2) \;=$$

$$\{\,\mathsf{stmt}'\,;\,\mathsf{stmt}_2 \mid \mathsf{stmt}' \in sub(\mathsf{stmt}_1) \setminus \{\mathsf{exit}\}\,\} \cup sub(\mathsf{stmt}_2).$$

For conditional commands, the set of substatements is defined as the set consisting of the **if**–**fi**-statement itself and substatements of its guarded commands. That is, for cond_cmd

being **if** $:: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n$ **fi** we have

$$sub(\mathsf{cond\_cmd}) \;=\; \{\,\mathsf{cond\_cmd}\,\} \cup \bigcup_{1 \leqslant i \leqslant n} sub(\mathsf{stmt}_i).$$

The substatements of a loop loop given by **do** $:: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n$ **od** are defined similarly, but taking into account that control moves back to loop when guarded commands terminate. That is:

$$sub(\mathsf{loop}) \;=\;$$

$$\{\,\mathsf{loop},\, \mathsf{exit}\,\} \cup \bigcup_{1 \leqslant i \leqslant n} \{\,\mathsf{stmt}'\,;\, \mathsf{loop} \mid \mathsf{stmt}' \in sub(\mathsf{stmt}_i) \setminus \{\mathsf{exit}\}\,\}.$$

For atomic regions let $sub(\mathtt{atomic\{stmt\}}) \;=\; \{\,\mathtt{atomic\{stmt\}},\, \mathsf{exit}\,\}$. ■

The definition of $sub(\mathsf{loop})$ relies on the observation that the effect of a loop with a single guarded command, say "**do** $:: g \Rightarrow \mathsf{stmt}$ **od**", corresponds to the effect of

$$\textbf{if } g \textbf{ then } \;\mathsf{stmt}\,;\; \textbf{do } :: g \Rightarrow \mathsf{stmt} \textbf{ od } \;\textbf{ else } \mathtt{skip} \textbf{ fi}$$

An analogous characterization applies to loops with two or more guarded commands. Thus, the definition of the substatements of loop relies on combining the definitions of the sets of substatements for sequential composition and conditional commands.

The formal semantics of nanoPromela program $\overline{\mathcal{P}} = [\mathcal{P}_1 | \ldots | \mathcal{P}_n]$ where the behavior of the process $\mathcal{P}_i$ is specified by a nanoPromela statement is a channel system $[PG_1 | \ldots | PG_n]$ over $(Var, Chan)$ where Var is the set of variables and $Chan$ the set of channels that are declared in $\overline{\mathcal{P}}$. As mentioned before, a formal syntax for the variable and channel declarations will not be provided, and global and local variables will not be distinguished. We assume that the set Var of typed variables and the set $Chan$ of channels (together with a classification of the channels into synchronous and FIFO-channels of some capacity $cap(\cdot)$) are given. Hence, local variable and channel declarations for the processes $\mathcal{P}_i$ are not considered. It is assumed that they are given by a nanoPromela-statement over some fixed tuple $(Var, Chan)$.

We now provide inference rules for the nanoPromela constructs. The inference rules for the atomic commands (skip, assignment, communication actions) and sequential composition, conditional and repetitive commands give rise to the edges of a "large" program graph where the set of locations agrees with the set of nanoPromela-statements. Thus, the edges have the form

$$\mathsf{stmt} \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}' \quad \text{or} \quad \mathsf{stmt} \xrightarrow{g:comm} \mathsf{stmt}'$$

where stmt is a nanoPromela statement, stmt' a substatement of stmt, and g a guard, α an action, and $comm$ a communication action $c?x$ or $c!\mathsf{expr}$. The subgraph consisting of

the substatements of $\mathcal{P}_i$ then yields the program graph PG_i of process $\mathcal{P}_i$ as a component of the program $\overline{\mathcal{P}}$.

The semantics of the atomic statement `skip` is given by a single axiom formalizing that the execution of `skip` terminates in one step without affecting the variables

$$\frac{}{\texttt{skip} \xrightarrow{\text{true: } id} \mathsf{exit}}$$

where *id* denotes an action that does not change the values of the variables, i.e., $\mathit{Effect}(id, \eta) = \eta$ for all variable evaluations η. Similarly, the execution of a statement consisting of an assignment $x := \mathsf{expr}$ has the trivial guard and terminates in one step:

$$\frac{}{x := \mathsf{expr} \xrightarrow{\text{true} : \text{assign}(x, \mathsf{expr})} \mathsf{exit}}$$

where $\text{assign}(x, \mathsf{expr})$ denotes the action that changes the value of x according to the assignment $x := \mathsf{expr}$ and does not affect the other variables, i.e., if $\eta \in \mathit{Eval}(\mathit{Var})$ and $y \in \mathit{Var}$ then $\mathit{Effect}(\text{assign}(x, \mathsf{expr}), \eta)(y) = \eta(y)$ if $y \neq x$ and $\mathit{Effect}(\text{assign}(x, \mathsf{expr}), \eta)(x)$ is the value of expr when evaluated over η. For the communication actions $c!\mathsf{expr}$ and $c?x$ the following axioms apply:

$$\frac{}{c?x \overset{c?x}{\hookrightarrow} \mathsf{exit}} \qquad \frac{}{c!\mathsf{expr} \xrightarrow{c!\mathsf{expr}} \mathsf{exit}}$$

The effect of an atomic region $\texttt{atomic}\{x_1 := \mathsf{expr}_1; \ldots; x_m := \mathsf{expr}_m\}$ is the cumulative effect of the assignments $x_i := \mathsf{expr}_i$. It can be defined by the rule:

$$\frac{}{\texttt{atomic}\{x_1 := \mathsf{expr}_1; \ldots; x_m := \mathsf{expr}_m\} \xrightarrow{\text{true} : \alpha_m} \mathsf{exit}}$$

where $\alpha_0 = id$, $\alpha_i = \mathit{Effect}(\text{assign}(x_i, \mathsf{expr}_i), \mathit{Effect}(\alpha_{i-1}, \eta))$ for $1 \leqslant i \leqslant m$.

Sequential composition $\mathsf{stmt}_1; \mathsf{stmt}_2$ is defined by two rules that distinguish whether or not stmt_1 terminates in one step. If the first step of stmt_1 leads to a location (statement) different from exit, then the following rule applies:

$$\frac{\mathsf{stmt}_1 \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}_1' \neq \mathsf{exit}}{\mathsf{stmt}_1; \mathsf{stmt}_2 \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}_1'; \mathsf{stmt}_2}$$

If the computation of stmt_1 terminates in one step by executing action α, then control of $\mathsf{stmt}_1; \mathsf{stmt}_2$ moves to stmt_2 after executing α:

$$\frac{\mathsf{stmt}_1 \overset{g:\alpha}{\hookrightarrow} \mathsf{exit}}{\mathsf{stmt}_1; \mathsf{stmt}_2 \overset{g:\alpha}{\hookrightarrow} \mathsf{stmt}_2}$$

The effect of a conditional command $\mathsf{cond\_cmd} \;=\; \mathbf{if} :: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n\, \mathbf{fi}$ is formalized by means of the following rule:

$$\frac{\mathsf{stmt}_i \overset{h:\alpha}{\hookrightarrow} \mathsf{stmt}_i'}{\mathsf{cond\_cmd} \overset{g_i \wedge h:\alpha}{\longrightarrow} \mathsf{stmt}_i'}$$

This rule relies on the test-and-set semantics where choosing one of the enabled guarded commands and performing its first action are treated as an atomic step. The blocking of cond_cmd when none of its guards is enabled needs no special treatment. The reason is that cond_cmd has no other edges than the ones specified by the rule above. Thus, in a global state $s = \langle \ell_1, \ldots, \ell_n, \eta, \xi \rangle$ where the location ℓ_i of the ith process is $\ell_i = \mathsf{cond\_cmd}$ and all guards $g_1, \ldots, g_n$ evaluate to false, then there is no action of the ith process that is enabled in s. However, actions of other processes might be enabled. Thus, the ith process has to wait until the other processes modify the variables appearing in $g_1, \ldots, g_n$ such that one or more of the guarded commands $g_i \Rightarrow \mathsf{stmt}_i$ become enabled.

For loops, say $\mathsf{loop} \;=\; \mathbf{do} :: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n\, \mathbf{od}$, we deal with three rules. The first two rules are similar to the rule for conditional commands, but taking into account that control moves back to loop after the execution of the selected guarded command has been completed. This corresponds to the following rules:

$$\frac{\mathsf{stmt}_i \overset{h:\alpha}{\hookrightarrow} \mathsf{stmt}_i' \neq \mathsf{exit}}{\mathsf{loop} \overset{g_i \wedge h:\alpha}{\longrightarrow} \mathsf{stmt}_i'; \mathsf{loop}} \qquad \frac{\mathsf{stmt}_i \overset{h:\alpha}{\hookrightarrow} \mathsf{exit}}{\mathsf{loop} \overset{g_i \wedge h:\alpha}{\longrightarrow} \mathsf{loop}}$$

If none of the guards $g_1, \ldots, g_n$ holds in the current state then the **do**–**od**-loop will be aborted. This is formalized by the following axiom:

$$\frac{}{\mathsf{loop} \overset{\neg g_1 \wedge \ldots \wedge \neg g_n}{\longrightarrow} \mathsf{exit}}$$

Remark 2.39. Test-and-Set Semantics vs. Two-Step Semantics

The rules for **if**–**fi**- and **do**–**od**-statements formalize the so-called *test-and-set semantics* of guarded commands. This means that evaluating guard g_i and performing the first step of the selected enabled guarded command $g_i \Rightarrow \mathsf{stmt}_i$ are performed atomically. In contrast, SPIN's interpretation of Promela relies on a two-step-semantics where the selection of an enabled guarded command and the execution of its first action are split into two steps. The rule for a conditional command is formalized by the axiom

$$\frac{}{\mathbf{if} :: g_1 \Rightarrow \mathsf{stmt}_1 \ldots :: g_n \Rightarrow \mathsf{stmt}_n\, \mathbf{fi} \overset{g_i \,:\, id}{\longrightarrow} \mathsf{stmt}_i}$$

where *id* is an action symbol for an action that does not affect the variables. Similarly, the first two rules for loops have to be replaced for the two-step semantics by the following rule:

$$\frac{}{\mathsf{loop} \xrightarrow{g_i : id} \mathsf{stmt}_i; \mathsf{loop}}$$

The rule for terminating the loop remains unchanged.

As long as we consider the statements in isolation, the test-and-set semantics and the two-step semantics are equal. However, when running several processes in parallel, the interleaving might cause undesired side effects. For example, consider the semaphore-based solution of the mutual exclusion problem, modeled by a nanoPromela program where the behavior of $\mathcal{P}_i$ is given by the following nanoPromela-statement:

do :: true ⇒ `skip`;
if :: $y > 0 \Rightarrow$ $y := y - 1$;
$crit_i := \text{true}$
fi;
$y := y + 1$
od

The initial value of the semaphore y is zero. Under the two-step semantics the mutual exclusion property is not guaranteed as it allows the processes to verify that the guard $y > 0$ of the **if**–**fi**-statement holds, without decreasing the value of y, and moving control to the assignment $y := y - 1$. But from there the processes can enter their critical sections. However, the protocol works correctly for the test-and-set semantics since then checking $y > 0$ and decreasing y is an atomic step that cannot be interleaved by the actions of the other process. ■

Remark 2.40. Generalized Guards

So far we required that the guards in conditional or repetitive commands consist of Boolean conditions on the program variables. However, it is also often useful to ask for interaction facilities in the guards, e.g., to specify that a process has to wait for a certain input along a FIFO-channel by means of a conditional command **if** :: $c?x \Rightarrow$ stmt **fi**. The intuitive meaning of the above statement is that the process has to wait until the buffer for c is nonempty. If so, then it first performs the action $c?x$ and then executes stmt. The use of communication actions in the guards leads to a more general class of program graphs with guards consisting of Boolean conditions on the variables or communication actions. For the case of an asynchronous channel the rules in Figure 2.20 on page 61 then have to be extended by:

$$\frac{\ell_i \xrightarrow{c?x:\alpha} \ell_i' \wedge \mathit{len}(\xi(c)) = k > 0 \wedge \xi(c) = v_1 v_2 \ldots v_k}{\langle \ldots, \ell_i, \ldots, \eta, \xi \rangle \xrightarrow{\tau} \langle \ldots, \ell_i', \ldots, \eta', \xi' \rangle}$$

where $\eta' = \mathit{Effect}(\alpha, \eta[x := v_1])$ and $\xi[c := v_2 \ldots v_k]$, and

$$\frac{\ell_i \xrightarrow{c!v:\alpha} \ell_i' \ \wedge \ \mathit{len}(\xi(c)) = k < \mathit{cap}(c) \ \wedge \ \xi(c) = v_1 \ldots v_k}{\langle \ldots, \ell_i, \ldots, \eta, \xi \rangle \xrightarrow{\tau} \langle \ldots, \ell_i', \ldots, \eta', \xi' \rangle}$$

where $\eta' = \mathit{Effect}(\alpha, \eta)$ and $\xi[c := v_1 \ldots v_k v]$.

Another convenient concept is the special guard **else** which specifies configurations where no other guarded command can be taken. The intuitive semantics of:

$$\begin{array}{lllll} \textbf{if} & :: & g_1 & \Rightarrow & \mathsf{stmt}_1 \\ & & & \vdots & \\ & :: & g_n & \Rightarrow & \mathsf{stmt}_n \\ & :: & \textbf{else} & \Rightarrow & \mathsf{stmt}' \\ \textbf{fi} & & & & \end{array}$$

is that the else option is enabled if none of the guards $g_1, \ldots g_n$ evaluates to true. In this case, the execution evolves to a state in which the statement stmt' is to be executed. Here, the g_i's can be Boolean expressions on the variables or communication guards. For example,

$$\textbf{if} :: d?x \Rightarrow \mathsf{stmt} :: \textbf{else} \Rightarrow x := x + 1 \, \textbf{fi}$$

increases x unless a message is obtained from channel d. The else option used in loops leads to nonterminating behaviors. ■

Remark 2.41. Atomic Regions

The axiom for atomic regions yields that if $s = \langle \ell_1 \ldots, \ell_n, \eta, \xi \rangle$ is a state in the transition system for the channel system associated with $\overline{\mathcal{P}} = [\mathcal{P}_1 | \ldots | \mathcal{P}_n]$ and $\ell_i = \mathtt{atomic}\{x_1 := \mathsf{expr}_1; \ldots; x_m := \mathsf{expr}_m\}; \ldots$ then in state s the ith process can perform the sequence of assignments $x_1 := \mathsf{expr}_1; \ldots; x_m := \mathsf{expr}_m$ in a single transition. With this semantics we abstract from the intermediate states that are passed when having performed the first i assignments ($1 \leqslant i < m$) and avoid that other processes can interleave their activities with these assignments.

This concept can be generalized for atomic regions $\mathtt{atomic}\{\mathsf{stmt}\}$ where the body stmt is an arbitrary statement. The idea is that any terminating execution of stmt is collapsed into a single transition, leading from a state with location stmt to a state with location exit. For this more general approach, the semantic rule for atomic regions operates on execution sequences in the transition system rather than just edges in the program graph. This is not problematic as one could provide the meaning of the statements on the level of transition systems rather than program graph level. However, the semantics is less

obvious for, e.g., infinite executions inside atomic regions, synchronous communication actions inside atomic regions and blocking conditional commands inside atomic regions. One possibility is to insert transitions to a special deadlock state. Another possibility is to work with a semantics that represents also the intermediate steps of an atomic region (but avoids interleaving) and to abort atomic regions as soon as a synchronous communication is required or blocking configurations are reached. ■

As we mentioned in the beginning of the section, Promela provides many more features than nanoPromela, such as atomic regions with more complex statements than sequences of assignments, arrays and other data types, and dynamic process creation. These concepts will not be explained here and we refer to the literature on the model checker SPIN (see, e.g., [209]).

2.2.6 Synchronous Parallelism

When representing asynchronous systems by transition systems, there are no assumptions concerning the relative velocities of the processors on which the components are executed. The residence time of the system in a state and the execution time of the actions are completely ignored. For instance, in the example of the two independent traffic lights (see Example 2.17), no assumption has been made concerning the amount of time a light should stay red or green. The only assumption is that both time periods are finite. The concurrent execution of components is time-abstract.

This is opposed to synchronous systems where components evolve in a lock step fashion. This is a typical computation mechanism in synchronous hardware circuits, for example, where the different components (like adders, inverters, and multiplexers) are connected to a central clock and all perform a (possibly idle) step on each clock pulse. As clock pulses occur periodically with a fixed delay, these pulses may be considered in a discrete manner, and transition systems can be adequately used to describe these synchronous systems. Synchronous composition of two transition systems is defined as follows.

Definition 2.42. Synchronous Product

Let $TS_i = (S_i, Act, \rightarrow_i, I_i, AP_i, L_i)$, i=1, 2, be transition systems with the same set of actions Act. Further, let

$$Act \times Act \rightarrow Act, \quad (\alpha, \beta) \rightarrow \alpha * \beta$$

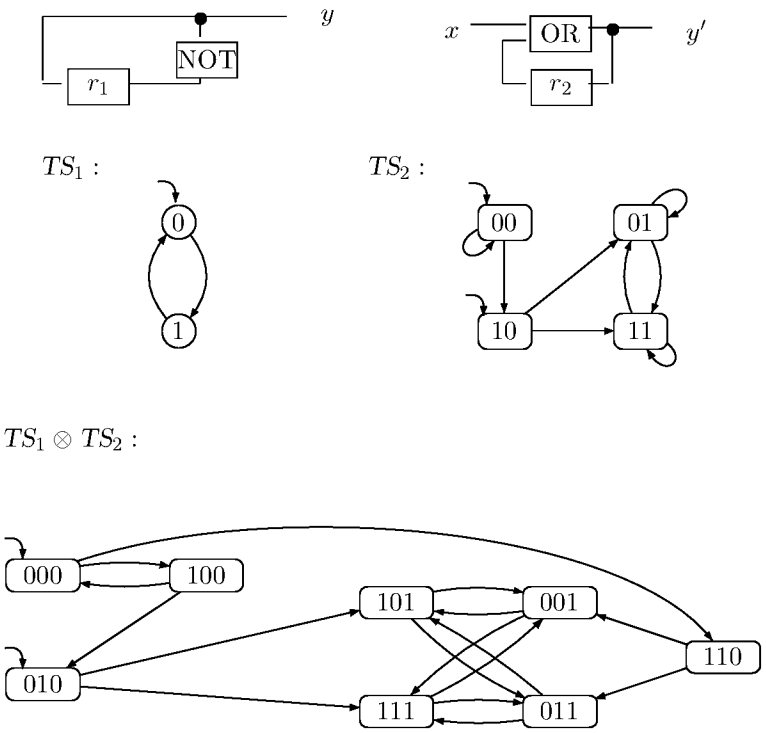


Figure 2.23: Synchronous composition of two hardware circuits.

be a mapping[4] that assigns to each pair of actions α, β, the action name $\alpha * \beta$. The *synchronous product* $TS_1 \otimes TS_2$ is given by:

$$TS_1 \otimes TS_2 \;=\; (S_1 \times S_2, Act, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L),$$

where the transition relation is defined by the following rule

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \quad \wedge \quad s_2 \xrightarrow{\beta}_2 s_2'}{\langle s_1, s_2 \rangle \xrightarrow{\alpha * \beta} \langle s_1', s_2' \rangle}$$

and the labeling function is defined by: $L(\langle s_1, s_2 \rangle) \;=\; L_1(s_1) \cup L_2(s_2)$. ■

Action $\alpha * \beta$ denotes the synchronous execution of actions α and β. Note that compared to the parallel operator $\|$ where components perform actions in common synchronously, and other action autonomously (i.e., asynchronously), in $TS_1 \otimes TS_2$, both transition systems have to perform all steps synchronously. There are no autonomous transitions of either TS_1 or TS_2.

[4]Operator $*$ is typically assumed to be commutative and associative.

Example 2.43. Synchronous Product of Two Circuits

Let C_1 be a circuit without input variables and with output variable y and register r. The control functions for output and register transitions are

$$\lambda_y \;=\; r_1, \quad \delta_{r_1} \;=\; \neg\, r_1.$$

Circuit C_2 has input variable x', output variable y', and register variable r_2 with the control functions

$$\lambda_{y'} \;=\; \delta_{r_2} \;=\; x' \;\vee\; r_2.$$

The transition system $TS_{C_1} \otimes TS_{C_2}$ is depicted in Figure 2.23 on page 76. Since action names are omitted for transition systems of circuits, action labels for $TS_{C_1} \otimes TS_{C_2}$ are irrelevant. $TS_{C_1} \otimes TS_{C_2}$ is thus the transition system of the circuit with input variable x', output variables y and y', and registers r_1 and r_2, whose control functions are λ_y, $\lambda_{y'}$, δ_{r_1}, and δ_{r_2}. ■

2.3 The State-Space Explosion Problem

The previous two sections have shown that various kinds of systems can be modeled using transition systems. This applies to program graphs representing data-dependent systems, and hardware circuits. Different communication mechanisms can be modeled in terms of appropriate operators on transition systems. This section considers the cardinality of the resulting transition systems, i.e., the number of states in these models. Verification techniques are based on systematically analyzing these transition systems. The runtimes of such verification algorithms are mainly determined by the number of states of the transition system to be analyzed. For many practical systems, the state space may be extremely large, and this is a major limitation for state-space search algorithms such as model checking. Chapter 8, Section 6.7, and Chapter 7 introduce a number of techniques to combat this problem.

Program Graph Representation Transition systems generated by means of "unfolding" a program graph may be extremely large, and in some cases—e.g., if there are infinitely many program locations or variables with infinite domains—even have infinitely many states. Consider a program graph over the set of variables *Var* with $x \in Var$. Recall that states of the unfolded transition system are of the form $\langle \ell, \eta \rangle$ with location ℓ and variable evaluation η. In case all variables in *Var* have a finite domain, like bits, or bounded integers, and the number of locations is finite, the number of states in the transition system is

$$|\,Loc\,| \cdot \prod_{x \in Var} |\,dom(x)\,| \;.$$

The number of states thus grows *exponentially* in the number of variables in the program graph: for N variables with a domain of k possible values, the number of states grows up to k^N. This exponential growth is also known as the *state-space explosion problem.*

It is important to realize that for even simple program graphs with just a small number of variables, this bound may already be rather excessive. For instance, a program graph with ten locations, three Boolean variables and five bounded integers (with domain in $\{0, \ldots, 9\}$) has $10 \cdot 2^3 \cdot 10^5 = 8,000,000$ states. If a single bit array of 50 bits is added to this program graph, for example, this bound grows even to $800{,}000 \cdot 2^{50}$! This observation clearly shows why the verification of data-intensive systems (with many variables or complex domains) is extremely hard. Even if there are only a few variables in a program, the state space that must be analyzed may be very large.

If $dom(x)$ is infinite for some $x \in Var$, as for reals or integers, the underlying transition system has infinitely many states as there are infinitely many values for x. Such program graphs usually yield undecidable verification problems. This is not to say that the verification of all transition systems with an infinite state space is undecidable, however.

It should be remarked that not only the state space of a transition system, but also the number of atomic propositions to represent program graphs (see Definition 2.15, page 34) may in principle be extremely large. Besides, any location, any condition on the variables in the program graph is allowed as an atomic proposition. However, in practice, only a small fragment of the possible atomic propositions is needed. An explicit representation of the labeling function is mostly not necessary, as the truth-values of the atomic formulae are typically derived from the state information. For these reasons, the number of atomic propositions plays only a secondary role.

For sequential hardware circuits (see page 26), states in the transition system are determined by the possible evaluations of the input variables and the registers. The size of the transition system thus grows exponentially in the number of registers and input variables. For N input variables and K registers, the total state space consists of 2^{N+K} states.

Parallelism In all variants of parallel operators for transition systems and program graphs, the state space of the complete system is built as the Cartesian product of the local state spaces S_i of the components. For example, for state space S of transition system

$$TS = TS_1 \,|||\, \ldots \,|||\, TS_n$$

we have $S = S_1 \times \ldots \times S_n$ where S_i denotes the state space of transition system TS_i. The state space of the parallel composition of a system with n states and a system with

k states yields $n{\cdot}k$ states. The total state space S is thus

$$|S_1| \cdot \ldots \cdot |S_n|.$$

The number of states in S is growing (at most) *exponentially* in the number of components: the parallel composition of N components of size k each yields k^N states. Even for small parallel systems this may easily run out of control.

Additionally, the variables (and their domains) represented in the transition system essentially influence the size of the state space. If one of the domains is infinite, then the state space is infinitely large. If the domains are finite, then the size of the state space grows exponentially in the number of variables (as we have seen before for program graphs).

The "exponential blowup" in the number of parallel components and the number of variables explains the enormous size of the state space of practically relevant systems. This observation is known under the heading *state explosion* and is another evidence for the fact that verification problems are particularly space-critical.

Channel Systems For the size of transition systems of channel systems, similar observations can be made as for the representation of program graphs. An important additional component for these systems is the size of the channels, i.e., their capacity. Clearly, if one of these channels has an infinite capacity, this may yield infinitely many states in the transition system. If all channels have finite capacity, however, the number of states is bound in the following way. Let $CS = [PG_1 \,|\ldots|\, PG_n]$ be a channel system over $Var = Var_1 \cup \ldots \cup Var_n$ and channels $Chan$. The state space of CS is of cardinality

$$\prod_{i=1}^{n} |\, PG_i \,| \cdot \prod_{c \in Chan} |\, dom(c) \,|^{cp(c)},$$

which can be rewritten as

$$\prod_{i=1}^{n} \left(|\, Loc_i \,| \cdot \prod_{x \in Var_i} |\, dom(x) \,| \right) \cdot \prod_{c \in Chan} |\, dom(c) \,|^{cp(c)}.$$

For L locations per component, K bit channels of capacity k each, and M variables x with $|\, dom(x) \,| \leqslant m$ totally, the total number of states in the transition system is $L^n{\cdot}m^M{\cdot}2^{K \cdot k}$. This is typically enormous.

Example 2.44. State-Space Size of the Alternating Bit Protocol

Consider a variant of the alternating bit protocol (see Example 2.32, page 57) where the

channels c and d have a fixed capacity, 10 say. Recall that along channel d, control bits are sent, and along channel c, data together with a control bit. Let us assume that data items are also simply bits. The timer has two locations, the sender eight, and the receiver six. As there are no further variables, we obtain that the total number of states is $2{\cdot}8{\cdot}6{\cdot}4^{10}{\cdot}2^{10}$, which equals $3{\cdot}2^{35}$. This is around 10^{11} states. ■

2.4 Summary

- Transition systems are a fundamental model for modeling software and hardware systems.
- An execution of a transition system is an alternating sequence of states and actions that starts in an initial state and that cannot be prolonged.
- Interleaving amounts to represent the evolvement of "simultaneous" activities of independent concurrent processes by the nondeterministic choice between these activities.
- In case of shared variable communication, parallel composition on the level of transition systems does not faithfully reflect the system's behavior. Instead, composition on program graphs has to be considered.
- Concurrent processes that communicate via handshaking on the set H of actions execute actions outside H autonomously whereas they execute actions in H synchronously.
- In channel systems, concurrent processes communicate via FIFO-buffers (i.e., channels). Handshaking communication is obtained when channels have capacity 0. For channels with a positive capacity, communication takes place asynchronously—sending and receiving a message takes place at different moments.
- The size of transition system representations grows exponentially in various components, such as the number of variables in a program graph or the number of components in a concurrent system. This is known as the state-space explosion problem.

2.5 Bibliographic Notes

Transition systems. Keller was one of the first researchers that explicitly used transition systems [236] for the verification of concurrent programs. Transition systems are used

as semantical models for a broad range of high-level formalisms for concurrent systems, such as process algebras [45, 57, 203, 298, 299], Petri nets [333], and statecharts [189]. The same is true for hardware synthesis and analysis, in which variants of finite-state automata (Mealy and Moore automata) play a central role; these variants can also be described by transition systems [246]. Program graphs and their unfolding into transition systems have been used extensively by Manna and Pnueli in their monograph(s) on temporal logic verification [283].

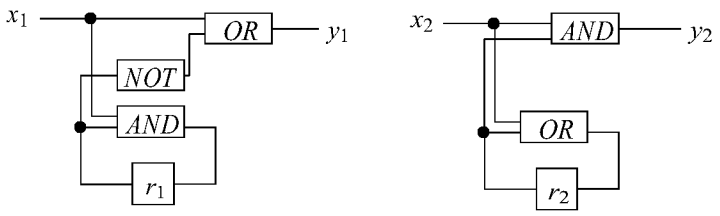
Synchronization paradigms. Shared variable "communication" dates back to the mid-sixties and is due to Dijkstra [126]. He also coined the term *interleaving* in 1971 [128]. Handshaking communication has been the main interaction paradigm in process algebras such as ACP [45], CCS [298, 299], CSP [202, 203], and LOTOS [57]. For a detailed account of process algebra we refer to [46]. The principle of synchronized parallelism has been advocated in Milner's synchronous variant of CCS, SCCS [297], and is used by Arnold to model the interaction between finite transition systems [19]. Synchronous parallelism is also at the heart of Lustre [183], a declarative programming language for reactive systems, and is used in many other hardware-oriented languages.

The interaction between concurrent processes by means of buffers (or channels) has first been considered by Dijkstra [129]. This paradigm has been adopted by specification languages for communication protocols, such as SDL (Specification and Description Language [37]) which is standardized by the ITU. The idea of guarded command languages goes back to Dijkstra [130]. The combination of guarded command languages in combination with channel-based communication is also used in Promela [205], the input language of the model checker SPIN [208]. The recent book by Holzmann [209] gives a detailed account of Promela and SPIN. Structured operational semantics has been introduced by Plotkin [334, 336] in 1981. Its origins are described in [335]. Atomic regions have been first discussed by Lipton [276], Lamport [257] and Owicki [317]. Further details on semantic rules for specification languages for reactive systems can be found, e.g., in [15, 18].

The examples. Most examples that have been provided in this chapter are rather classical. The problem of mutual exclusion was first proposed in 1962 by Dekker together with an algorithm that guarantees two-process mutual exclusion. Dijkstra's solution [126] was the first solution to mutual exclusion for an arbitrary number of processes. He also introduced the concept of semaphores [127] and their use for solving the mutual exclusion problem. A simpler and more elegant solution has been proposed by Lamport in 1977 [256]. This was followed up by Peterson's mutual exclusion protocol [332] in 1981. This algorithm is famous by now due to its beauty and simplicity. For other mutual exclusion algorithms, see e.g. [283, 280]. The alternating bit protocol stems from 1969 and is one of the first flow control protocols [34]. Holzmann gives a historical account of this and related protocols in his first book [205].

2.6 Exercises

EXERCISE 2.1. Consider the following two sequential hardware circuits:



Questions:

(a) Give the transition systems of both hardware circuits.

(b) Determine the reachable part of the transition system of the synchronous product of these transition systems. Assume that the initial values of the registers are r_1=0 and r_2=1.

EXERCISE 2.2. We are given three (primitive) processes P_1, P_2, and P_3 with shared integer variable x. The program of process $P_i$is as follows:

Algorithm 1 Process P_i

for $k_i = 1, \ldots, 10$ **do**
  LOAD(x);
  INC(x);
  STORE(x);
od

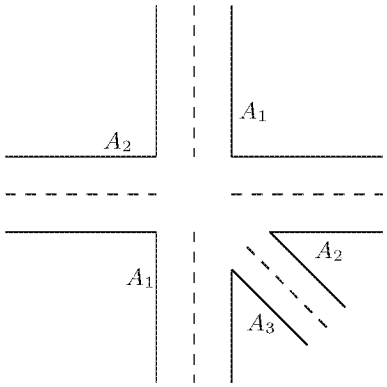
That is, P_i executes ten times the assignment $x := x{+}1$. The assignment $x := x{+}1$ is realized using the three actions LOAD(x), INC(x) and STORE(x). Consider now the parallel program:

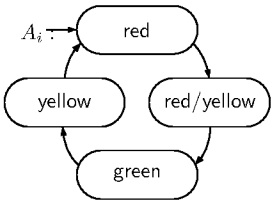
Algorithm 2 Parallel program P

$x := 0$;
$P_1 \parallel P_2 \parallel P_3$

Question: Does P have an execution that halts with the terminal value $x = 2$?

EXERCISE 2.3. Consider the following street junction with the specification of a traffic light as outlined on the right.





(a) Choose appropriate actions and label the transitions of the traffic light transition system accordingly.

(b) Give the transition system representation of a (reasonable) controller C that switches the green signal lamps in the following order: $A_1, A_2, A_3, A_1, A_2, A_3, \ldots$.
(*Hint: Choose an appropriate communication mechanism.*)

(c) Outline the transition system $A_1 \| A_2 \| A_3 \| C$.

EXERCISE 2.4. Show that the handshaking operator $\|$ that forces two transition systems to synchronize over their common actions (see Definition 2.26 on page 48) is associative. That is, show that

$$(TS_1 \| TS_2) \| TS_3 \quad = \quad TS_1 \| (TS_2 \| TS_3)$$

where TS_1, TS_2, TS_3 are arbitrary transition systems.

EXERCISE 2.5. The following program is a mutual exclusion protocol for two processes due to Pnueli [118]. There is a single shared variable s which is either 0 or 1, and initially 1. Besides, each process has a local Boolean variable y that initially equals 0. The program text for process P_i $(i = 0, 1)$ is as follows:

l0: **loop forever do**
  begin
  l1: Noncritical section
  l2: $(y_i, s) := (1, i)$;
  l3: **wait until** $((y_{1-i} = 0) \ \vee \ (s \neq i))$;
  l4: Critical section
  l5: $y_i := 0$
end.

Here, the statement $(y_i, s) := (1, i)$; is a *multiple assignment* in which variable $y_i := 1$ and $s := i$ is a single, atomic step.

Questions:

(a) Define the program graph of a process in Pnueli's algorithm.

(b) Determine the transition system for each process.

(c) Construct their parallel composition.

(d) Check whether the algorithm ensures mutual exclusion.

(e) Check whether the algorithm ensures starvation freedom.

The last two questions may be answered by inspecting the transition system.

EXERCISE 2.6. Consider a stack of nonnegative integers with capacity n (for some fixed n).

(a) Give a transition system representation of this stack. You may abstract from the values on the stack and use the operations *top*, *pop*, and *push* with their usual meaning.

(b) Sketch a transition system representation of the stack in which the concrete stack content is explicitly represented.

EXERCISE 2.7. Consider the following generalization of Peterson's mutual exclusion algorithm that is aimed at an arbitrary number n ($n \geqslant 2$) processes. The basic concept of the algorithm is that each process passes through n "levels" before acquiring access to the critical section. The concurrent processes share the bounded integer arrays $y[0..n{-}1]$ and $p[1..n]$ with $y[i] \in \{1, \ldots, n\}$ and $p[i] \in \{0, \ldots, n{-}1\}$. $y[j] = i$ means that process i has the lowest priority at level j, and $p[i] = j$ expresses that process i is currently at level j. Process i starts at level 0. On requesting access to the critical section, the process passes through levels 1 through $n{-}1$. Process i waits at level j until either all other processes are at a lower level (i.e., $p[k] < j$ for all $k \neq i$) or another process grants process i access to its critical section (i.e., $y[j] \neq i$). The behavior of process i is in pseudocode:

while true **do**
  ... noncritical section ...
  forall $j = 1, \ldots, n{-}1$ **do**
    $p[i] := j$;
    $y[j] := i$;
    wait until $(y[j] \neq i) \vee \left(\bigwedge_{0<k\leqslant n, k\neq i} p[k] < j\right)$
  od
  ... critical section ...
  $p[i] := 0$;
od

Questions:

(a) Give the program graph for process i.

(b) Determine the number of states (including the unreachable states) in the parallel composition of n processes.

(c) Prove that this algorithm ensures mutual exclusion for n processes.

(d) Prove that it is impossible that all processes are waiting in the for-iteration.

(e) Establish whether it is possible that a process that wants to enter the critical section waits ad infinitum.

EXERCISE 2.8. In channel systems, values can be transferred from one process to another process. As this is somewhat limited, we consider in this exercise an extension that allows for the transfer of *expressions*. That is to say, the send and receive statements $c!v$ and $c?x$ (where x and v are of the same type) are generalized into $c!expr$ and $c?x$, where for simplicity it is assumed that $expr$ is a correctly typed expression (of the same type as x). Legal expressions are, e.g., $x \wedge (\neg y \vee z)$ for Boolean variables x, y, and z, and channel c with $dom(c) = \{0, 1\}$. For integers x, y, and an integer-channel c, $|2x + (x - y)\text{div}17|$ is a legal expression.

Question: Extend the transition system semantics of channel systems such that expressions are allowed in send statements.
(Hint: Use the function η such that for expression expr, $\eta(expr)$ is the evaluation of expr under the variable valuation η.)

EXERCISE 2.9. Consider the following mutual exclusion algorithm that uses the shared variables y_1 and y_2 (initially both 0).

Process P_1:
while true **do**
... noncritical section ...
$y_1 := y_2 + 1$;
wait until $(y_2 = 0) \vee (y_1 < y_2)$
... critical section ...
$y_1 := 0$;
od

Process P_2:
while true **do**
... noncritical section ...
$y_2 := y_1 + 1$;
wait until $(y_1 = 0) \vee (y_2 < y_1)$
... critical section ...
$y_2 := 0$;
od

Questions:

(a) Give the program graph representations of both processes. (A pictorial representation suffices.)

(b) Give the reachable part of the transition system of $P_1 \parallel P_2$ where $y_1 \leqslant 2$ and $y_2 \leqslant 2$.

(c) Describe an execution that shows that the entire transition system is infinite.

(d) Check whether the algorithm indeed ensures mutual exclusion.

(e) Check whether the algorithm never reaches a state in which both processes are mutually waiting for each other.

(f) Is it possible that a process that wants to enter the critical section has to wait ad infinitum?

EXERCISE 2.10. Consider the following mutual exclusion algorithm that was proposed 1966 [221] as a simplification of Dijkstra's mutual exclusion algorithm in case there are just two processes:

```
 1 Boolean array b(0;1) integer k, i, j,
 2 comment  This is the program for computer i, which may be
      either 0 or 1, computer j =/= i is the other one, 1 or 0;
 3 C0: b(i) := false;
 4 C1: if k != i then begin
 5 C2: if not b(j) then go to C2;
 6     else k := i; go to C1 end;
 7     else critical section;
 8     b(i) := true;
 9     remainder of program;
10     go to C0;
11     end
```

Here `C0`, `C1`, and `C2` are program labels, and the word "computer" should be interpreted as process.

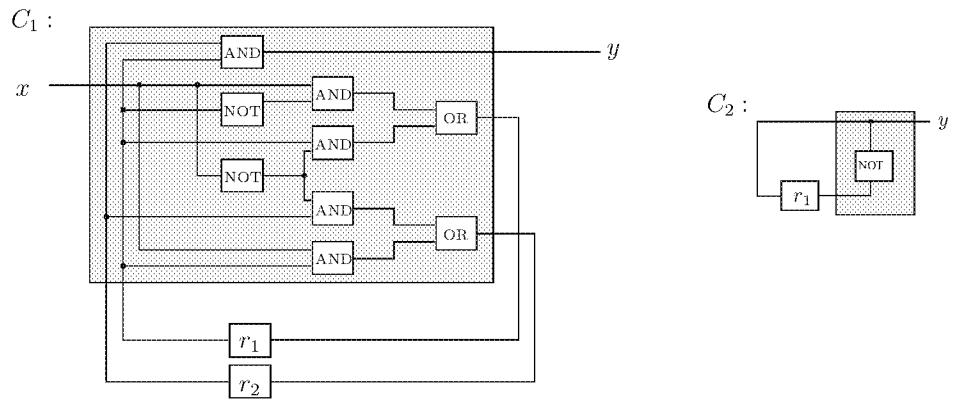
Questions:

(a) Give the program graph representations for a single process. (A pictorial representation suffices.)

(b) Give the reachable part of the transition system of $P_1 \parallel P_2$.

(c) Check whether the algorithm indeed ensures mutual exclusion.

EXERCISE 2.11. Consider the following two sequential hardware circuits C_1 and C_2:



(a) Give the transition system representation $TS(C_1)$ of the circuit C_1.

(b) Let $TS(C_2)$ be the transition system of the circuit C_2. Outline the transition system $TS(C_1) \otimes TS(C_2)$.

EXERCISE 2.12. Consider the following leader election algorithm: For $n \in \mathbb{N}$, n processes $P_1, \ldots, P_n$ are located in a ring topology where each process is connected by an unidirectional channel to its neighbor in a clockwise manner.

To distinguish the processes, each process is assigned a unique identifier $id \in \{1, \ldots, n\}$. The aim is to elect the process with the highest identifier as the leader within the ring. Therefore each process executes the following algorithm:

send (id); *initially set to process' id*
while (true) **do**
  receive (m);
  if $(m = id)$ **then stop**; *process is the leader*
  if $(m > id)$ **then send** (m); *forward identifier*
od

(a) Model the leader election protocol for n processes as a channel system.

(b) Give an initial execution fragment of $TS([P_1|P_2|P_3])$ such that at least one process has executed the send statement within the body of the whileloop. Assume for $0 < i \leqslant 3$, that process P_i has identifier $id_i = i$.

Chapter 3

Linear-Time Properties

For verification purposes, the transition system model of the system under consideration needs to be accompanied with a specification of the property of interest that is to be verified. This chapter introduces some important, though relatively simple, classes of properties. These properties are formally defined and basic model-checking algorithms are presented to check such properties in an automated manner. This chapter focuses on linear-time behavior and establishes relations between the different classes of properties and trace behavior. Elementary forms of fairness are introduced and compared.

3.1 Deadlock

Sequential programs that are not subject to divergence (i.e., endless loops) have a terminal state, a state without any outgoing transitions. For parallel systems, however, computations typically do not terminate—consider, for instance, the mutual exclusion programs treated so far. In such systems, terminal states are undesirable and mostly represent a design error. Apart from "trivial" design errors where it has been forgotten to indicate certain activities, in most cases such terminal states indicate a *deadlock*. A deadlock occurs if the complete system is in a terminal state, although at least one component is in a (local) nonterminal state. The entire system has thus come to a halt, whereas at least one component has the possibility to continue to operate. A typical deadlock scenario occurs when components mutually wait for each other to progress.

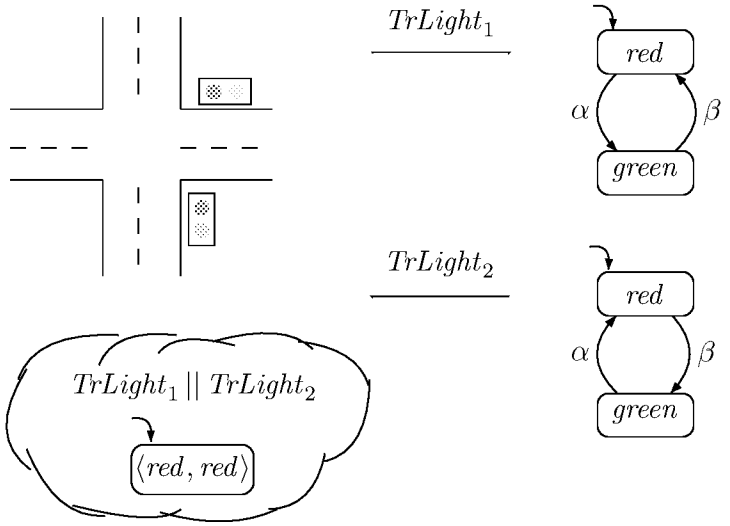


Figure 3.1: An example of a deadlock situation.

Example 3.1. Deadlock for Fault Designed Traffic Lights

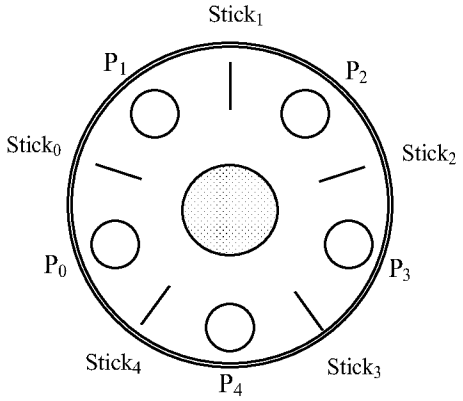
Consider the parallel composition of two transition systems

$$TrLight_1 \parallel TrLight_2$$

modeling the traffic lights of two intersecting roads. Both traffic lights synchronize by means of the actions α and β that indicate the change of light (see Figure 3.1). The apparently trivial error to let both traffic lights start with a red light results in a deadlock. While the first traffic light is waiting to be synchronized on action α, the second traffic light is blocked, since it is waiting to be synchronized with action β. ■

Example 3.2. Dining Philosophers

This example, originated by Dijkstra, is one of the most prominent examples in the field of concurrent systems.



Five philosophers are sitting at a round table with a bowl of rice in the middle. For the philosophers (being a little unworldly) life consists of thinking and eating (and waiting, as we will see). To take some rice out of the bowl, a philosopher needs two chopsticks. In between two neighboring philosophers, however, there is only a single chopstick. Thus, at any time only one of two neighboring philosophers can eat. Of course, the use of the chopsticks is exclusive and eating with hands is forbidden.

Note that a deadlock scenario occurs when all philosophers possess a single chopstick. The problem is to design a protocol for the philosophers, such that the complete system is deadlock-free, i.e., at least one philosopher can eat and think infinitely often. Additionally, a fair solution may be required with each philosopher being able to think and eat infinitely often. The latter characteristic is called freedom of *individual starvation.*

The following obvious design cannot ensure deadlock freedom. Assume the philosophers and the chopsticks are numbered from 0 to 4. Furthermore, assume all following calculations be "modulo 5", e.g., chopstick $i-1$ for $i=0$ denotes chopstick 4, and so on.

Philosopher i has stick i on his left and stick $i-1$ on his right side. The action $request_{i,i}$ express that stick i is picked up by philosopher i. Accordingly, $request_{i-1,i}$ denotes the action by means of which philosopher i picks up the $(i-1)$th stick. The actions $release_{i,i}$ and $release_{i-1,i}$ have a corresponding meaning.

The behavior of philosopher i (called process $Phil_i$) is specified by the transition system depicted in the left part of Figure 3.2. Solid arrows depict the synchronizations with the i-th stick, dashed arrows refer to communications with the $i-1$th stick. The sticks are modeled as independent processes (called $Stick_i$) with which the philosophers synchronize via actions *request* and *release*; see the right part of Figure 3.2 that represents the process of stick i. A stick process prevents philosopher i from picking up the ith stick when philosopher $i+1$ is using it.

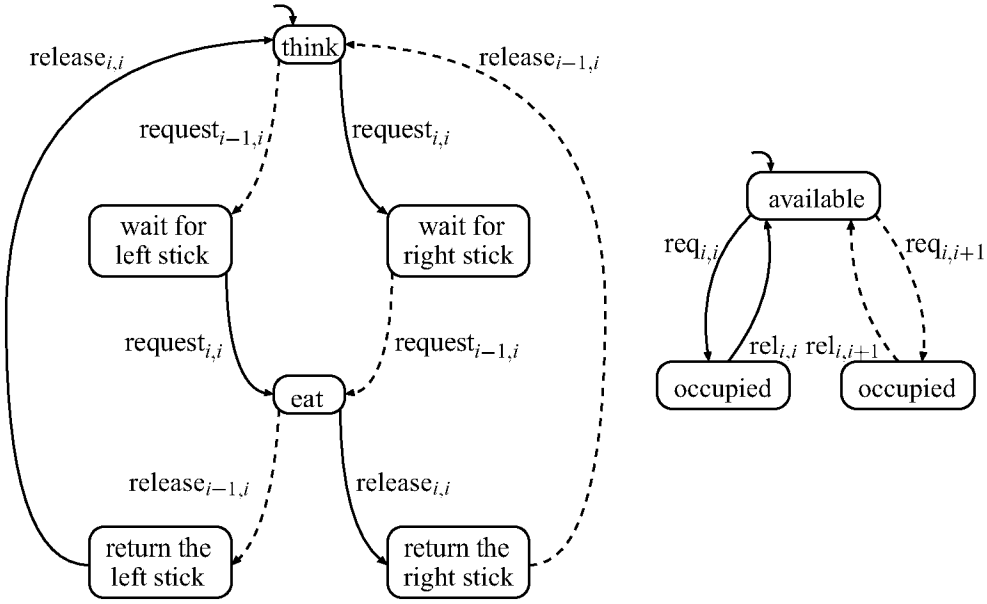


Figure 3.2: Transition systems for the ith philosopher and the ith stick.

The complete system is of the form:

$$Phil_4 \parallel Stick_3 \parallel Phil_3 \parallel Stick_2 \parallel Phil_2 \parallel Stick_1 \parallel Phil_1 \parallel Stick_0 \parallel Phil_0 \parallel Stick_4$$

This (initially obvious) design leads to a deadlock situation, e.g., if all philosophers pick up their left stick at the same time. A corresponding execution leads from the initial state

$$\langle think_4, avail_3, think_3, avail_2, think_2, avail_1, think_1, avail_0, think_0, avail_4 \rangle$$

by means of the action sequence $request_4$, $request_3$, $request_2$, $request_1$, $request_0$ (or any other permutation of these 5 request actions) to the terminal state

$$\langle wait_{4,0}, occ_{4,4}, wait_{3,4}, occ_{3,3}, wait_{2,3}, occ_{2,2}, wait_{1,2}, occ_{1,1}, wait_{0,1}, occ_{0,0} \rangle.$$

This terminal state represents a deadlock with each philosopher waiting for the needed stick to be released.

A possible solution to this problem is to make the sticks available for only one philosopher at a time. The corresponding chopstick process is depicted in the right part of Figure 3.3. In state $available_{i,j}$ only philosopher j is allowed to pick up the ith stick. The above-mentioned deadlock situation can be avoided by the fact that some sticks (e.g., the first, the third, and the fifth stick) start in state $available_{i,i}$, while the remaining sticks start in state $available_{i,i+1}$. It can be verified that this solution is deadlock- and starvation-free.

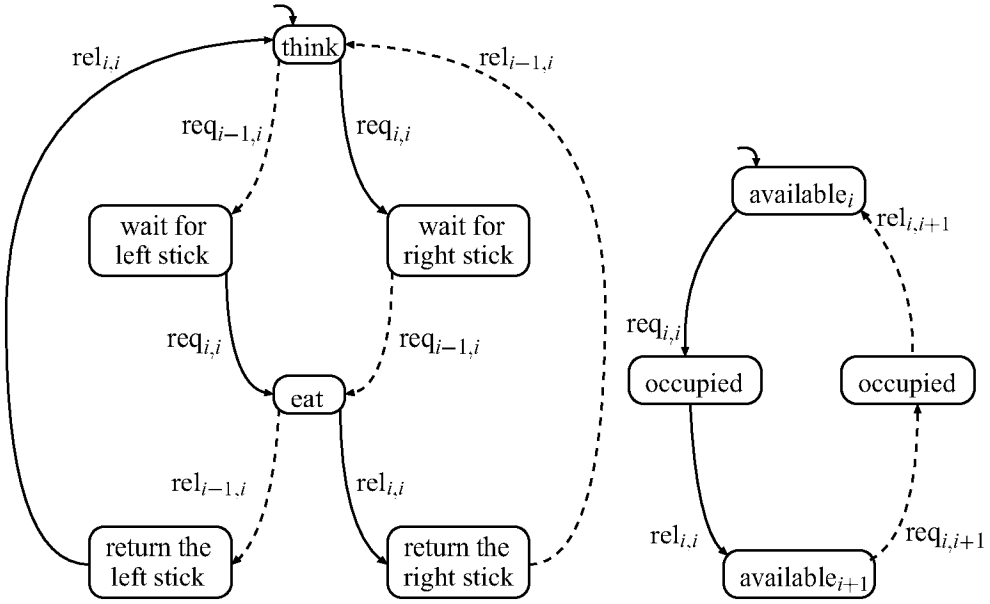


Figure 3.3: Improved variant of the ith philosopher and the ith stick.

A further characteristic often required for concurrent systems is robustness against failure of their components. In the case of the dining philosophers, robustness can be formulated in a way that ensures deadlock and starvation freedom even if one of the philosophers is "defective" (i.e., does not leave the think phase anymore).[1] The above-sketched deadlock- and starvation-free solution can be modified to a fault-tolerant solution by changing the transition systems of philosophers and sticks such that philosopher $i{+}1$ can pick up the ith stick even if philosopher i is thinking (i.e., does not need stick i) independent of whether stick i is in state $\mathit{available}_{i,i}$ or $\mathit{available}_{i,i+1}$. The corresponding is also true when the roles of philosopher i and $i{+}1$ are reversed. This can be established by adding a single Boolean variable x_i to philosopher i (see Figure 3.4). The variable x_i informs the neighboring philosophers about the current location of philosopher i. In the indicated sketch, x_i is a Boolean variable which is true if and only if the ith philosopher is thinking. Stick i is made available to philosopher i if stick i is in location $\mathit{available}_i$ (as before), or if stick i is in location $\mathit{available}_{i+1}$ while philosopher $i{+}1$ is thinking.

Note that the above description is at the level of program graphs. The complete system is a channel system with *request* and *release* actions standing for handshaking over a channel of capacity 0. ■

[1] Formally, we add a loop to the transition system of a defective philosopher at state think_i.

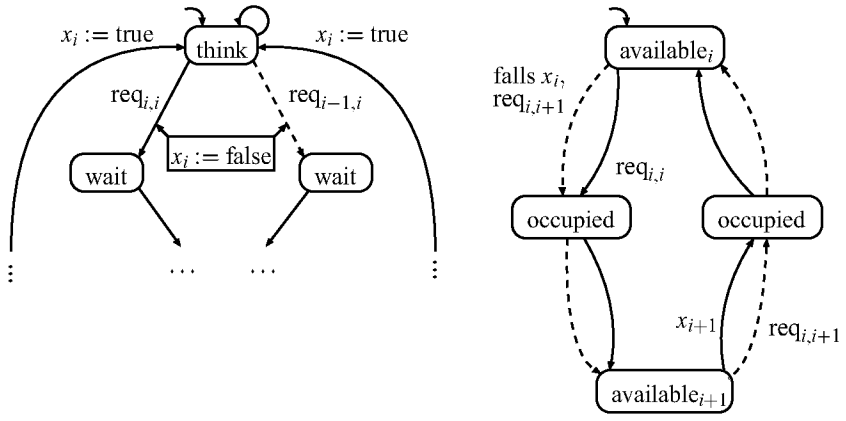


Figure 3.4: Fault-tolerant variant of the dining philosophers.

3.2 Linear-Time Behavior

To analyze a computer system represented by a transition system, either an action-based or a state-based approach can be followed. The state-based approach abstracts from actions; instead, only labels in the state sequences are taken into consideration. In contrast, the action-based view abstracts from states and refers only to the action labels of the transitions. (A combined action- and state-based view is possible, but leads to more involved definitions and concepts. For this reason it is common practice to abstract from either action or state labels.) Most of the existing specification formalisms and associated verification methods can be formulated in a corresponding way for both perspectives.

In this chapter, we mainly focus on the state-based approach. Action labels of transitions are only necessary for modeling communication; thus, they are of no relevance in the following chapters. Instead, we use the atomic propositions of the states to formulate system properties. Therefore, the verification algorithms operate on the *state graph* of a transition system, the digraph originating from a transition system by abstracting from action labels.

3.2.1 Paths and State Graph

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system.

Definition 3.3. State Graph

The *state graph* of TS, notation $G(TS)$, is the digraph (V, E) with vertices $V = S$ and edges $E = \{(s, s') \in S \times S \mid s' \in Post(s)\}$. ■

The state graph of transition system TS has a vertex for each state in TS and an edge between vertices s and s' whenever s' is a direct successor of s in TS for some action α. It is thus simply obtained from TS by omitting all state labels (i.e., the atomic propositions), all transition labels (i.e., the actions), and by ignoring the fact whether a state is initial or not. Moreover, multiple transitions (that have different action labels) between states are represented by a single edge. This seems to suggest that the state labels are no longer of any use; later on, we will see how these state labels will be used to check the validity of properties.

Let $Post^*(s)$ denote the states that are reachable in state graph $G(TS)$ from s. This notion is generalized toward sets of states in the usual way (i.e., pointwise extension): for $C \subseteq S$ let

$$Post^*(C) \;=\; \bigcup_{s \in C} Post^*(s).$$

The notations $Pre^*(s)$ and $Pre^*(C)$ have analogous meaning. The set of states that are reachable from some initial state, notation $Reach(TS)$, equals $Post^*(I)$.

As explained in Chapter 2, the possible behavior of a transition system is defined by an execution fragment. Recall that an execution fragment is an alternating sequence of states and actions. As we consider a state-based approach, the actions are not of importance and are omitted. The resulting "runs" of a transition system are called *paths*. The following definitions define path fragments, initial and maximal path fragments, and so on. These notions are easily obtained from the same notions for executions by omitting the actions.

Definition 3.4. Path Fragment

A *finite* path fragment $\widehat{\pi}$ of TS is a finite state sequence $s_0\, s_1 \ldots s_n$ such that $s_i \in Post(s_{i-1})$ for all $0 < i \leqslant n$, where $n \geqslant 0$. An *infinite* path fragment π is an infinite state sequence $s_0\, s_1\, s_2 \ldots$ such that $s_i \in Post(s_{i-1})$ for all $i > 0$. ■

We adopt the following notational conventions for infinite path fragment $\pi = s_0\, s_1 \ldots$. The initial state of π is denoted by $first(\pi) = s_0$. For $j \geqslant 0$, let $\pi[j] = s_j$ denote the jth state of

π and $\pi[..j]$ denote the jth prefix of π, i.e., $\pi[..j] = s_0\, s_1 \ldots s_j$. Similarly, the jth suffix of π, notation $\pi[j..]$, is defined as $\pi[j..] = s_j\, s_{j+1} \ldots$. These notions are defined analogously for finite paths. Besides, for finite path $\widehat{\pi} = s_0\, s_1 \ldots s_n$, let $\mathit{last}(\widehat{\pi}) = s_n$ denote the last state of $\widehat{\pi}$, and $\mathit{len}(\widehat{\pi}) = n$ denote the length of $\widehat{\pi}$. For infinite path π these notions are defined by $\mathit{len}(\pi) = \infty$ and $\mathit{last}(\pi) = \bot$, where $\bot$ denotes "undefined".

Definition 3.5. Maximal and Initial Path Fragment

A *maximal* path fragment is either a finite path fragment that ends in a terminal state, or an infinite path fragment. A path fragment is called *initial* if it starts in an initial state, i.e., if $s_0 \in I$. ■

A maximal path fragment is a path fragment that cannot be prolonged: either it is infinite or it is finite but ends in a state from which no transition can be taken. Let $\mathit{Paths}(s)$ denote the set of maximal path fragments π with $\mathit{first}(\pi) = s$, and $\mathit{Paths}_{\mathit{fin}}(s)$ denote the set of all finite path fragments $\widehat{\pi}$ with $\mathit{first}(\widehat{\pi}) = s$.

Definition 3.6. Path

A *path* of transition system TS is an initial, maximal path fragment.[2] ■

Let $\mathit{Paths}(TS)$ denote the set of all paths in TS, and $\mathit{Paths}_{\mathit{fin}}(TS)$ the set of all initial, finite path fragments of TS.

Example 3.7. Beverage Vending Machine

Consider the beverage vending machine of Example 2.2 on page 21. For convenience, its transition system is repeated in Figure 3.5. As the state labeling is simply $L(s) = \{ s \}$ for each state s, the names of states may be used in paths (as in this example), as well as atomic propositions (as used later on). Example path fragments of this transition system are

$$\begin{aligned} \pi_1 &= \mathit{pay\ select\ soda\ pay\ select\ soda} \ldots \\ \pi_2 &= \mathit{select\ soda\ pay\ select\ beer} \ldots \\ \widehat{\pi} &= \mathit{pay\ select\ soda\ pay\ select\ soda}\,. \end{aligned}$$

These path fragments result from the execution fragments indicated in Example 2.8 on page 25. Only π_1 is a path. The infinite path fragment π_2 is maximal but not initial. $\widehat{\pi}$ is initial but not maximal since it is finite while ending in a state that has outgoing

[2] It is important to realize the difference between the notion of a path in a transition system and the notion of a path in a digraph. A path in a transition system is maximal, whereas a path in a digraph in the graph-theoretical sense is not always maximal. Besides, paths in a digraph are usually required to be finite whereas paths in transition systems may be infinite.

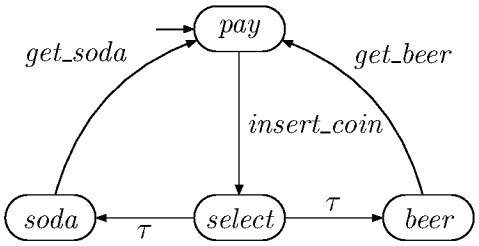


Figure 3.5: A transition system of a simple beverage vending machine.

transitions. We have that $last(\widehat{\pi}) = soda$, $first(\pi_2) = select$, $\pi_1[0] = pay$, $\pi_1[3] = pay$, $\pi_1[..5] = \widehat{\pi}$, $\widehat{\pi}[..2] = \widehat{\pi}[3..]$, $len(\widehat{\pi}) = 5$, and $len(\pi_1) = \infty$. ■

3.2.2 Traces

Executions (as introduced in Chapter 2) are alternating sequences consisting of states and actions. Actions are mainly used to model the (possibility of) interaction, be it synchronous or asynchronous communication. In the sequel, interaction is not our prime interest, but instead we focus on the states that are visited during executions. In fact, the states themselves are not "observable", but just their atomic propositions. Thus, rather than having an execution of the form $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots$ we consider sequences of the form $L(s_0)\,L(s_1)\,L(s_2)\ldots$ that register the (set of) atomic propositions that are valid along the execution. Such sequences are called *traces*.

The traces of a transition system are thus words over the alphabet 2^{AP}. In the following it is assumed that a transition system has no terminal states. In this case, all traces are infinite words. (Recall that the traces of a transition system have been defined as traces induced by its initial maximal path fragments. See also Appendix A.2, page 912). This assumption is made for simplicity and does not impose any serious restriction. First of all, prior to checking any (linear-time) property, a reachability analysis could be carried out to determine the set of terminal states. If indeed some terminal state is encountered, the system contains a deadlock and has to be repaired before any further analysis. Alternatively, each transition system TS (that probably has a terminal state) can be extended such that for each terminal state s in TS there is a new state s_{stop}, transition $s \to s_{stop}$, and s_{stop} is equipped with a self-loop, i.e., $s_{stop} \to s_{stop}$. The resulting "equivalent" transition system obviously has no terminal states.[3]

[3] A further alternative is to adapt the linear-time framework for transition systems with terminal states. The main concepts of this chapter are still applicable, but require some adaptions to distinguish nonmax-

Definition 3.8. Trace and Trace Fragment

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states. The *trace* of the infinite path fragment $\pi = s_0\, s_1 \ldots$ is defined as $trace(\pi) = L(s_0)\, L(s_1) \ldots$. The trace of the finite path fragment $\widehat{\pi} = s_0\, s_1 \ldots s_n$ is defined as $trace(\widehat{\pi}) = L(s_0)\, L(s_1) \ldots L(s_n)$. ■

The trace of a path fragment is thus the induced finite or infinite word over the alphabet 2^{AP}, i.e., the sequence of sets of atomic propositions that are valid in the states of the path. The set of traces of a set Π of paths is defined in the usual way:

$$trace(\Pi) \;=\; \{\, trace(\pi) \mid \pi \in \Pi \,\}.$$

A trace of state s is the trace of an infinite path fragment π with $first(\pi) = s$. Accordingly, a finite trace of s is the trace of a finite path fragment that starts in s. Let $Traces(s)$ denote the set of traces of s, and $Traces(TS)$ the set of traces of the initial states of transition system TS:

$$Traces(s) = trace(Paths(s)) \quad \text{and} \quad Traces(TS) = \bigcup_{s \in I} Traces(s).$$

In a similar way, the finite traces of a state and of a transition system are defined:

$$Traces_{fin}(s) = trace(Paths_{fin}(s)) \quad \text{and} \quad Traces_{fin}(TS) = \bigcup_{s \in I} Traces_{fin}(s).$$

Example 3.9. Semaphore-Based Mutual Exclusion

Consider the transition system TS_{Sem} as depicted in Figure 3.6. This two-process mutual exclusion example has been described before in Example 2.24 (page 43).

Assume the available atomic propositions are $crit_1$ and $crit_2$, i.e.,

$$AP = \{\, crit_1, crit_2 \,\}.$$

The proposition $crit_1$ holds in any state of the transition system TS_{Sem} where the first process (called P_1) is in its critical section. Proposition $crit_2$ has the same meaning for the second process (i.e., P_2).

Consider the execution in which the processes P_1 and P_2 enter their critical sections in an alternating fashion. Besides, they only request to enter the critical section when the

imal and maximal finite paths.

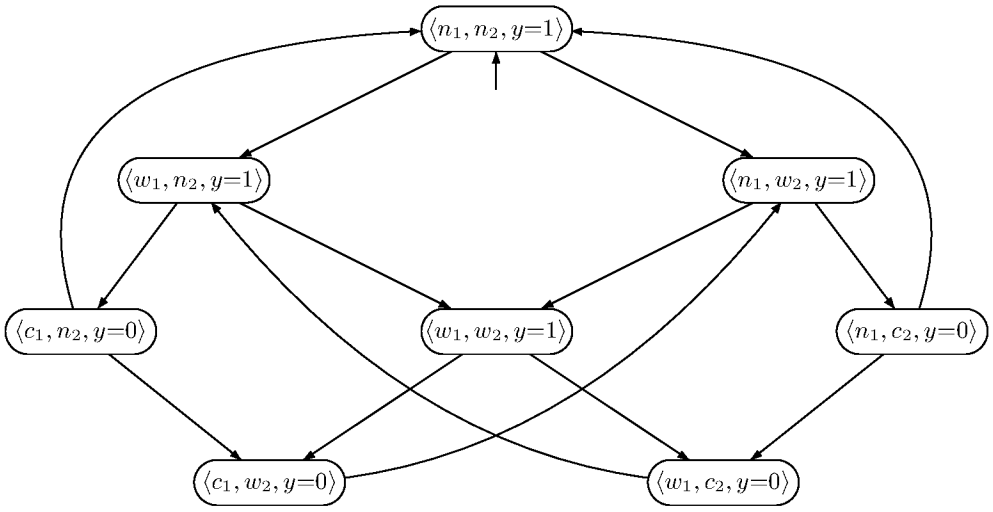


Figure 3.6: Transition system of semaphore-based mutual exclusion algorithm.

other process is no longer in its critical section. Situations in which one process is in its critical section whereas the other is moving from the noncritical state to the waiting state are impossible.

The path π in the state graph of TS_{Sem} where process P_1 is the first to enter its critical section is of the form

$$\begin{aligned}\pi \quad = \quad & \langle n_1, n_2, y=1\rangle \to \langle w_1, n_2, y=1\rangle \to \langle c_1, n_2, y=0\rangle \to \\ & \langle n_1, n_2, y=1\rangle \to \langle n_1, w_2, y=1\rangle \to \langle n_1, c_2, y=0\rangle \to \ldots\end{aligned}$$

The trace of this path is the infinite word:

$$\mathit{trace}(\pi) \;=\; \varnothing\,\varnothing\,\{\,\mathit{crit}_1\,\}\,\varnothing\,\varnothing\,\{\,\mathit{crit}_2\,\}\,\varnothing\,\varnothing\,\{\,\mathit{crit}_1\,\}\,\varnothing\,\varnothing\,\{\,\mathit{crit}_2\,\}\ldots.$$

The trace of the finite path fragment

$$\begin{aligned}\widehat{\pi} \quad = \quad & \langle n_1, n_2, y=1\rangle \to \langle w_1, n_2, y=1\rangle \to \langle w_1, w_2, y=1\rangle \to \\ & \langle w_1, c_2, y=0\rangle \to \langle w_1, n_2, y=1\rangle \to \langle c_1, n_2, y=0\rangle\end{aligned}$$

is $\mathit{trace}(\widehat{\pi}) \;=\; \varnothing\,\varnothing\,\varnothing\,\{\,\mathit{crit}_2\,\}\,\varnothing\,\{\,\mathit{crit}_1\,\}$. ■

3.2.3 Linear-Time Properties

Linear-time properties specify the traces that a transition system should exhibit. Informally speaking, one could say that a linear-time property specifies the admissible (or desired) behavior of the system under consideration. In the following we provide a formal definition of such properties. This definition is rather elementary, and gives a good basic understanding of what a linear-time property is. In Chapter 5, a logical formalism will be introduced that allows for the specification of linear-time properties.

In the following, we assume a fixed set of propositions AP. A linear-time (LT) property is a requirement on the traces of a transition system. Such property can be understood as a requirement over all words over AP, and is defined as the set of words (over AP) that are admissible:

Definition 3.10. LT Property

A *linear-time property* (LT property) over the set of atomic propositions AP is a subset of $(2^{AP})^{\omega}$. ■

Here, $(2^{AP})^{\omega}$ denotes the set of words that arise from the infinite concatenation of words in 2^{AP}. An LT property is thus a language (set) of infinite words over the alphabet 2^{AP}. Note that it suffices to consider infinite words only (and not finite words), as transition systems without terminal states are considered. The fulfillment of an LT property by a transition system is defined as follows.

Definition 3.11. Satisfaction Relation for LT Properties

Let P be an LT property over AP and $TS = (S, Act, \rightarrow, I, AP, L)$ a transition system without terminal states. Then, $TS = (S, Act, \rightarrow, I, AP, L)$ *satisfies* P, denoted $TS \models P$, iff $\mathit{Traces}(TS) \subseteq P$. State $s \in S$ satisfies P, notation $s \models P$, whenever $\mathit{Traces}(s) \subseteq P$. ■

Thus, a transition system satisfies the LT property P if all its traces respect P, i.e., if all its behaviors are admissible. A state satisfies P whenever all traces starting in this state fulfill P.

Example 3.12. Traffic Lights

Consider two simplified traffic lights that only have two possible settings: red and green. Let the propositions of interest be

$$AP = \{ red_1, green_1, red_2, green_2 \}$$

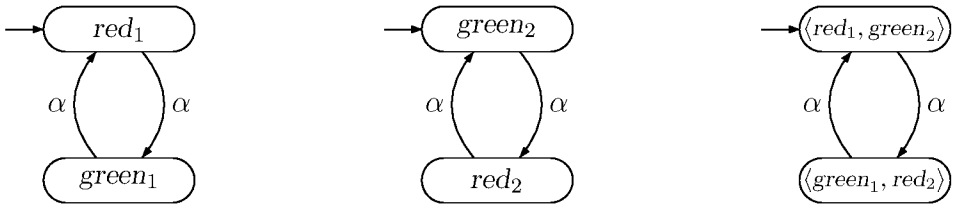


Figure 3.7: Two fully synchronized traffic lights (left and middle) and their parallel composition (right).

We consider two LT properties of these traffic lights and give some example words that are contained by such properties. First, consider the property P that states:

"The first traffic light is infinitely often green".

This LT property corresponds to the set of infinite words of the form $A_0\,A_1\,A_2\,\ldots$ over 2^{AP}, such that $green_1 \in A_i$ holds for infinitely many i. For example, P contains the infinite words

$$\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\,\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\ldots,$$
$$\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\ldots$$
$$\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\ldots \quad \text{and}$$
$$\{\, green_1, green_2 \,\}\,\{\, green_1, green_2 \,\}\,\{\, green_1, green_2 \,\}\,\{\, green_1, green_2 \,\}\ldots$$

The infinite word $\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\varnothing\,\varnothing\,\varnothing\,\varnothing\ldots$ is not in P as it contains only finitely many occurrences of $green_1$.

As a second LT property, consider P':

"The traffic lights are never both green simultaneously".

This property is formalized by the set of infinite words of the form $A_0\,A_1\,A_2\ldots$ such that either $green_1 \notin A_i$ or $green_2 \notin A_i$, for all $i \geqslant 0$. For example, the following infinite words are in P':

$$\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\,\{\, red_1, green_2 \,\}\,\{\, green_1, red_2 \,\}\ldots,$$
$$\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\,\{\, green_1 \,\}\,\varnothing\ldots \quad \text{and}$$
$$\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\,\{\, red_1, green_1 \,\}\ldots,$$

whereas the infinite word $\{\, red_1\, green_2 \,\}\,\{\, green_1, green_2 \,\},\ldots$ is not in P'.

The traffic lights depicted in Figure 3.7 are at intersecting roads and their switching is synchronized, i.e., if one light switches from red to green, the other switches from green to

red. In this way, the lights always have complementary colors. Clearly, these traffic lights satisfy both P and P'. Traffic lights that switch completely autonomously will neither satisfy P—there is no guarantee that the first traffic light is green infinitely often—nor P'. ■

Often, an LT property does not refer to all atomic propositions occurring in a transition system, but just to a relatively small subset thereof. For a property P over a set of propositions $AP' \subseteq AP$, only the labels in AP' are relevant. Let $\widehat{\pi}$ be a finite path fragment of TS. We write $trace_{AP'}(\widehat{\pi})$ to denote the finite trace of $\widehat{\pi}$ where only the atomic propositions in AP' are considered. Accordingly, $trace_{AP'}(\pi)$ denotes the trace of an infinite path fragment π by focusing on propositions in AP' Thus, for $\pi = s_0\, s_1\, s_2 \ldots$, we have

$$trace_{AP'}(\pi) \;=\; L'(s_0)\, L'(s_1) \ldots \;=\; (L(s_0) \cap AP')\; (L(s_1) \cap AP') \ldots$$

Let $Traces_{AP'}(TS)$ denote the set of traces $trace_{AP'}(Paths(TS))$. Whenever the set AP' of atomic propositions is clear from the context, the subscript AP' is omitted. In the rest of this chapter, the restriction to a relevant subset of atomic propositions is often implicitly made.

Example 3.13. The Mutual Exclusion Property

In Chapter 2, several mutual exclusion algorithms have been considered. For specifying the mutual exclusion property—always at most one process is in its critical section—it suffices to only consider the atomic propositions $crit_1$ and $crit_2$. Other atomic propositions are not of any relevance for this property. The formalization of the mutual exclusion property is given by the LT property

$$P_{mutex} \;=\; \text{set of infinite words } A_0\, A_1\, A_2 \ldots \text{ with } \{\, crit_1, crit_2 \,\} \not\subseteq A_i \text{ for all } 0 \leqslant i.$$

For example, the infinite words

$$\{\, crit_1 \,\}\, \{\, crit_2 \,\}\, \{\, crit_1 \,\}\, \{\, crit_2 \,\}\, \{\, crit_1 \,\}\, \{\, crit_2 \,\} \ldots, \quad \text{and}$$
$$\{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\}\, \{\, crit_1 \,\} \ldots, \quad \text{and}$$
$$\varnothing\, \varnothing\, \varnothing\, \varnothing\, \varnothing\, \varnothing\, \varnothing \ldots$$

are all contained in P_{mutex}. However, this does not apply to words of the form

$$\{\, crit_1 \,\}\, \varnothing\, \{\, crit_1, crit_2 \,\} \ldots$$

The transition system $TS_{Arb} = (TS_1 \,|||\, TS_2) \parallel Arbiter$ described in Example 2.28 (page 50) fulfills the mutex property, i.e.,

$$TS_{Arb} \;\models\; P_{mutex}.$$

It is left to the reader to check that the mutex property is also fulfilled by the semaphore-based mutual exclusion algorithm (see Figure 3.6 on page 99) and Peterson's algorithm (see Example 2.25 on page 45). ■

Example 3.14. Starvation Freedom

Guaranteeing mutual exclusion is a significant property of mutual exclusion algorithms, but is not the only relevant property. An algorithm that never allows a process to enter its critical section will do, but is certainly not intended. Besides, a property is imposed that requires a process that wants to enter the critical section to be able to eventually do so. This property prevents a process from waiting ad infinitum and is formally specified as the LT property $P_{finwait}$ = set of infinite words $A_0\,A_1\,A_2\ldots$ such that

$$\forall j.\, l\mathit{wait}_i \in A_j \;\Rightarrow\; \exists k \geqslant j.\, \mathit{wait}_i \in A_k \quad \text{for each } i \in \{1,2\}.$$

Here, we assumed the set of propositions to be:

$$AP \;=\; \{\, \mathit{wait}_1, \mathit{crit}_1, \mathit{wait}_2, \mathit{crit}_2 \,\}.$$

Property $P_{finwait}$ expresses that each of the two processes enters its critical section eventually if they are waiting. That is, a process has to wait some finite amount before entering the critical section. It does not express that a process that waits often, is often entering the critical section.

Consider the following variant. The LT property $P_{nostarve}$ = set of infinite words $A_0\,A_1\,A_2\ldots$ such that:

$$(\forall k \geqslant 0.\, \exists j \geqslant k.\; \mathit{wait}_i \in A_j\;) \;\Rightarrow\; (\forall k \geqslant 0.\, \exists j \geqslant k.\; \mathit{crit}_i \in A_j\;) \qquad \text{for each } i \in \{1,2\}.$$

In abbreviated form we write:

$$\left(\overset{\infty}{\exists}\, j.\; \mathit{wait}_i \in A_j\right) \;\Rightarrow\; \left(\overset{\infty}{\exists}\, j.\; \mathit{crit}_i \in A_j\right) \qquad \text{for each } i \in \{1,2\}$$

where $\overset{\infty}{\exists}$ stands for "there are infinitely many".

Property $P_{nostarve}$ expresses that each of the two processes enters its critical section infinitely often if they are waiting infinitely often. This natural requirement is, however, *not* satisfied for the semaphore-based solution, since

$$\varnothing\;(\{\,\mathit{wait}_2\,\}\,\{\,\mathit{wait}_1, \mathit{wait}_2\,\}\,\{\,\mathit{crit}_1, \mathit{wait}_2\,\})^{\omega}$$

is a possible trace of the transition system but does not belong to $P_{nostarve}$. This trace represents an execution in which only the first process enters its critical section infinitely often. In fact, the second process waits infinitely long to enter its critical section.

It is left to the reader to check that the transition system modeling Peterson's algorithm (see Example 2.25, page 45) does indeed satisfy $P_{nostarve}$. ■

3.2.4 Trace Equivalence and Linear-Time Properties

LT properties specify the (infinite) traces that a transition system should exhibit. If transition systems TS and TS' have the same traces, one would expect that they satisfy the same LT properties. Clearly, if $TS \models P$, then all traces of TS are contained in P, and when $\mathit{Traces}(TS) = \mathit{Traces}(TS')$, the traces of TS' are also contained in P. Otherwise, whenever $TS \not\models P$, there is a trace in $\mathit{Traces}(TS)$ that is prohibited by P, i.e., not included in the set P of traces. As $\mathit{Traces}(TS) = \mathit{Traces}(TS')$, also TS' exhibits this prohibited trace, and thus $TS' \not\models P$. The precise relationship between trace equivalence, trace inclusion, and the satisfaction of LT properties is the subject of this section.

We start by considering trace inclusion and its importance in concurrent system design. Trace inclusion between transition systems TS and TS' requires that all traces exhibited by TS can also be exhibited by TS', i.e., $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$. Note that transition system TS' may exhibit more traces, i.e., may have some (linear-time) behavior that TS does not have. In stepwise system design, where designs are successively refined, trace inclusion is often viewed as an implementation relation in the sense that

$$\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS') \text{ means } TS \text{ "is a correct implementation of" } TS'.$$

For example, let TS' be a (more abstract) design where parallel composition is modeled by interleaving, and TS its realization where (some of) the interleaving is resolved by means of some scheduling mechanism. TS may thus be viewed as an "implementation" of TS', and clearly, $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$.

What does trace inclusion have to do with LT properties? The following theorem shows that trace inclusion is *compatible* with requirement specifications represented as LT properties.

Theorem 3.15. Trace Inclusion and LT Properties

Let TS and TS' be transition systems without terminal states and with the same set of propositions AP. Then the following statements are equivalent:

(a) $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$

(b) For any LT property P: $TS' \models P$ *implies* $TS \models P$.

Proof: (a) $\Longrightarrow$ (b): Assume $\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS')$, and let P be an LT property such that $TS' \models P$. From Definition 3.11 it follows that $\mathit{Traces}(TS') \subseteq P$. Given $\mathit{Traces}(TS) \subseteq$

$Traces(TS')$, it now follows that $Traces(TS) \subseteq P$. By Definition 3.11 it follows that $TS \models P$.

(b) $\Longrightarrow$ (a): Assume that for all LT properties it holds that: $TS' \models P$ implies $TS \models P$. Let $P = Traces(TS')$. Obviously, $TS' \models P$, as $Traces(TS') \subseteq Traces(TS')$. By assumption, $TS \models P$. Hence, $Traces(TS) \subseteq Traces(TS')$. ■

This simple observation plays a decisive role for the design by means of successive refinement. If TS' is the transition system representing a preliminary design and TS is a transition system originating from a refinement of TS' (i.e., a more detailed design), then it can immediately—without explicit proof—be concluded from the relation $Traces(TS) \subseteq Traces(TS')$ that any LT property that holds in TS' also holds for TS.

Example 3.16. Refining the Semaphore-Based Mutual Exclusion Algorithm

Let $TS' = TS_{Sem}$, the transition system representing the semaphore-based mutual exclusion algorithm (see Figure 3.6 on page 99) and let TS be the transition system obtained from TS' by removing the transition

$$\langle wait_1, wait_2, y = 1 \rangle \ \rightarrow \ \langle wait_1, crit_2, y = 0 \rangle.$$

Stated in words, from the situation in which both processes are waiting, it is no longer possible that the second process (P_2) acquires access to the critical section. This thus yields a model that assigns higher priority to process P_1 than to process P_2 when both processes are competing to access the critical section. As a transition is removed, it immediately follows that $Traces(TS) \subseteq Traces(TS')$. Consequently, by the fact that TS' ensures mutual exclusion, i.e., $TS' \models P_{mutex}$, it follows by Theorem 3.15 that $TS \models P_{mutex}$. ■

Transition systems are said to be trace-equivalent if they have the same set of traces:

Definition 3.17. Trace Equivalence

Transition systems TS and TS' are *trace-equivalent* with respect to the set of propositions AP if $Traces_{AP}(TS) = Traces_{AP}(TS')$. [4] ■

Theorem 3.15 implies equivalence of two trace-equivalent transition systems with respect to requirements formulated as LT properties.

[4] Here, we assume two transition systems with sets of propositions that include AP.

Corollary 3.18. Trace Equivalence and LT Properties

Let TS and TS' be transition systems without terminal states and with the same set of atomic propositions. Then:

$$Traces(TS) = Traces(TS') \iff TS \text{ and } TS' \text{ satisfy the same LT properties.}$$

There thus does not exist an LT property that can distinguish between trace-equivalent transition systems. Stated differently, in order to establish that the transition systems TS and TS' are *not* trace-equivalent it suffices to find one LT property that holds for one but not for the other.

Example 3.19. Two Beverage Vending Machines

Consider the two transition systems in Figure 3.8 that both model a beverage vending

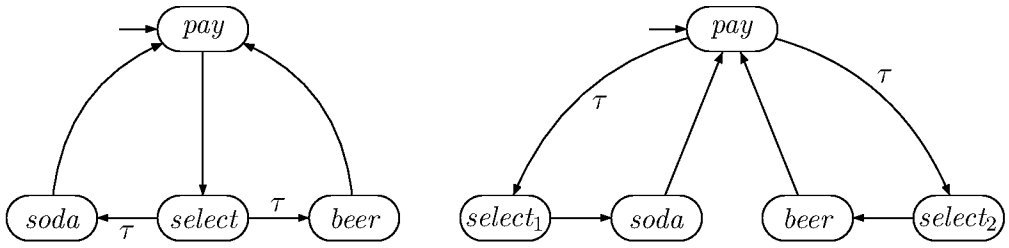


Figure 3.8: Two beverage vending machines.

machine. For simplicity, the observable action labels of transitions have been omitted. Both machines are able to offer soda and beer. The left transition system models a beverage machine that after insertion of a coin nondeterministically chooses to either provide soda or beer. The right one, however, has two selection buttons (one for each beverage), and after insertion of a coin, nondeterministically blocks one of the buttons. In either case, the user has no control over the beverage obtained—the choice of beverage is under full control of the vending machine.

Let $AP = \{ pay, soda, beer \}$. Although the two vending machines behave differently, it is not difficult to see that they exhibit the same traces when considering AP, as for both machines traces are alternating sequences of *pay* and either *soda* or *beer*. The vending machines are thus trace-equivalent. By Corollary 3.18 both vending machines satisfy exactly the same LT properties. Stated differently, it means that there does not exist an LT property that distinguishes between the two vending machines. ■

3.3 Safety Properties and Invariants

Safety properties are often characterized as "nothing bad should happen". The mutual exclusion property—always at most one process is in its critical section—is a typical safety property. It states that the bad thing (having two or more processes in their critical section simultaneously) never occurs. Another typical safety property is deadlock freedom. For the dining philosophers (see Example 3.2, page 90), for example, such deadlock could be characterized as the situation in which all philosophers are waiting to pick up the second chopstick. This bad (i.e., unwanted) situation should never occur.

3.3.1 Invariants

In fact, the above safety properties are of a particular kind: they are *invariants*. Invariants are LT properties that are given by a condition Φ for the states and require that Φ holds for all reachable states.

Definition 3.20. Invariant

An LT property P_{inv} over AP is an *invariant* if there is a propositional logic formula[5] Φ over AP such that

$$P_{inv} = \Big\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \forall j \geqslant 0.\ A_j \models \Phi \Big\}.$$

Φ is called an invariant condition (or state condition) of P_{inv}. ■

Note that

$$\begin{array}{lll} TS \models P_{inv} & \text{iff} & trace(\pi) \in P_{inv} \text{ for all paths } \pi \text{ in } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \text{ that belong to a path of } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \in Reach(TS). \end{array}$$

Thus, the notion "invariant" can be explained as follows: the condition Φ has to be fulfilled by all initial states and satisfaction of Φ is invariant under all transitions in the reachable fragment of the given transition system. The latter means that if Φ holds for the source state s of a transition $s \xrightarrow{a} s'$, then Φ holds for the target state s' too.

Let us return to the examples of mutual exclusion and deadlock freedom for the dining philosophers. The mutual exclusion property can be described by an invariant using the

[5] The basic principles of propositional logic are treated in Appendix A.3.

propositional logic formula

$$\Phi \;=\; \neg \mathit{crit}_1 \vee \neg \mathit{crit}_2.$$

For deadlock freedom of the dining philosophers, the invariant ensures that at least one of the philosophers is not waiting to pick up the chopstick. This can be established using the propositional formula:

$$\Phi \;=\; \neg \mathit{wait}_0 \vee \neg \mathit{wait}_1 \vee \neg \mathit{wait}_2 \vee \neg \mathit{wait}_3 \vee \neg \mathit{wait}_4.$$

Here, the proposition wait_i characterizes the state(s) of philosopher i in which he is waiting for a chopstick.

How do we check whether a transition system satisfies an invariant? As checking an invariant for the propositional formula Φ amounts to checking the validity of Φ in every state that is reachable from some initial state, a slight modification of standard graph traversal algorithms like depth-first search (DFS) or breadth-first search (BFS) will do, provided the given transition system TS is *finite*.

Algorithm 3 on page 109 summarizes the main steps for checking the invariant condition Φ by means of a forward depth-first search in the state graph $G(TS)$. The notion *forward search* means that we start from the initial states and investigate all states that are reachable from them. If at least one state s is visited where Φ does not hold, then the invariance induced by Φ is violated. In Algorithm 3, R stores all visited states, i.e., if Algorithm 3 terminates, then $R = \mathit{Reach}(TS)$ contains all reachable states. Furthermore, U is a stack that organizes all states that still have to be visited, provided they are not yet contained in R. The operations *push*, *pop*, and *top* are the standard operations on stacks. The symbol ε is used to denote the empty stack. Alternatively, a *backward search* could have been applied that starts with all states where Φ does not hold and calculates (by a DFS or BFS) the set $\bigcup_{s \in S, s \not\models \Phi} \mathit{Pre}^*(s)$.

Algorithm 3 could be slightly improved by aborting the computation once a state s is encountered that does not fulfill Φ. This state is a "bad" state as it makes the transition system refute the invariant and could be returned as an error indication. Such error indication, however, is not very helpful.

Instead, an initial path fragment $s_0\, s_1\, s_2 \ldots s_n$ in which all states (except the last one) satisfy Φ and $s_n \not\models \Phi$ would be more useful. Such a path fragment indicates a possible behavior of the transition system that violates the invariant. Algorithm 3 can be easily adapted such that a counterexample is provided on encountering a state that violates Φ. To that end we exploit the (depth-first search) stack U. When encountering s_n that violates Φ, the stack content, read from bottom to top, contains the required initial path fragment. Algorithm 4 on page 110 thus results.

Algorithm 3 Naïve invariant checking by forward depth-first search

Input: finite transition system TS and propositional formula Φ
Output: true if TS satisfies the invariant "always Φ", otherwise false

```
set of state R := ∅;                          (* the set of visited states *)
stack of state U := ε;                        (* the empty stack *)
bool b := true;                               (* all states in R satisfy Φ *)
for all s ∈ I do
  if s ∉ R then
    visit(s)                                  (* perform a dfs for each unvisited initial state *)
  fi
od
return b
```

```
procedure visit (state s)
  push(s, U);                                 (* push s on the stack *)
  R := R ∪ { s };                             (* mark s as reachable *)
  repeat
    s' := top(U);
    if Post(s') ⊆ R then
      pop(U);
      b := b ∧ (s' ⊨ Φ);                      (* check validity of Φ in s' *)
    else
      let s'' ∈ Post(s') \ R
      push(s'', U);
      R := R ∪ { s'' };                       (* state s'' is a new reachable state *)
    fi
  until (U = ε)
endproc
```

Algorithm 4 Invariant checking by forward depth-first search

Input: finite transition system TS and propositional formula Φ
Output: "yes" if $TS \models$ "always Φ", otherwise "no" plus a counterexample

```
set of states R := ∅;                                  (* the set of reachable states *)
stack of states U := ε;                                (* the empty stack *)
bool b := true;                                        (* all states in R satisfy Φ *)
while (I \ R ≠ ∅ ∧ b) do
  let s ∈ I \ R;                                       (* choose an arbitrary initial state not in R *)
  visit(s);                                            (* perform a DFS for each unvisited initial state *)
od
if b then
  return("yes")                                        (* TS ⊨ "always Φ" *)
else
  return("no", reverse(U))                             (* counterexample arises from the stack content *)
fi
```

```
procedure visit (state s)
  push(s, U);                                          (* push s on the stack *)
  R := R ∪ { s };                                      (* mark s as reachable *)
  repeat
    s' := top(U);
    if Post(s') ⊆ R then
      pop(U);
      b := b ∧ (s' ⊨ Φ);                               (* check validity of Φ in s' *)
    else
      let s'' ∈ Post(s') \ R
      push(s'', U);
      R := R ∪ { s'' };                                (* state s'' is a new reachable state *)
    fi
  until ((U = ε) ∨ ¬b)
endproc
```

The worst-case time complexity of the proposed invariance checking algorithm is dominated by the cost for the DFS that visits all reachable states. The latter is linear in the number of states (nodes of the state graph) and transitions (edges in the state graph), provided we are given a representation of the state graph where the direct successors $s' \in Post(s)$ for any state s can be encountered in time $\Theta(|Post(s)|)$. This holds for a representation of the sets $Post(s)$ by adjacency lists. An explicit representation of adjacency lists is not adequate in our context where the state graph of a complex system has to be analyzed. Instead, the adjacency lists are typically given in an *implicit* way, e.g., by a syntactic description of the concurrent processes, such as program graphs or higher-level description languages with a program graph semantics such as nanoPromela, see Section 2.2.5, page 63). The direct successors of a state s are then obtained by the axioms and rules for the transition relation for the composite system. Besides the space for the syntactic descriptions of the processes, the space required by Algorithm 4 is dominated by the representation of the set R of visited states (this is typically done by appropriate hash techniques) and stack U. Hence, the additional space complexity of invariant checking is linear in the number of reachable states.

Theorem 3.21. Time Complexity of Invariant Checking

The time complexity of Algorithm 4 is $\mathcal{O}(\, N * (1 + |\Phi|) + M \,)$ *where* N *denotes the number of reachable states, and* $M = \sum_{s \in S} |Post(s)|$ *the number of transitions in the reachable fragment of* TS.

Proof: The time complexity of the forward reachability on the state graph $G(TS)$ is $\mathcal{O}(N + M)$. The time needed to check $s \models \Phi$ for some state s is linear in the length of Φ.[6] As for each state s it is checked whether Φ holds, this amounts to a total of $N + M + N * (1 + |\Phi|)$ operations. ■

3.3.2 Safety Properties

As we have seen in the previous section, invariants can be viewed as state properties and can be checked by considering the reachable states. Some safety properties, however, may impose requirements on finite path fragments, and cannot be verified by considering the reachable states only. To see this, consider the example of a cash dispenser, also known as an automated teller machine (ATM). A natural requirement is that money can only be withdrawn from the dispenser once a correct personal identifier (PIN) has been provided. This property is not an invariant, since it is not a state property. It is, however, considered

[6] To cover the special case where Φ is an atomic proposition, in which case $|\Phi| = 0$, we deal with $1 + |\Phi|$ for the cost to check whether Φ holds for a given state s.

to be a safety property, as any infinite run violating the requirement has a finite prefix that is "bad", i.e., in which money is withdrawn without issuing a PIN before.

Formally, safety property P is defined as an LT property over AP such that any infinite word σ where P does not hold contains a *bad prefix*. The latter means a finite prefix $\widehat{\sigma}$ where the bad thing has happened, and thus no infinite word that starts with this prefix $\widehat{\sigma}$ fulfills P.

Definition 3.22. Safety Properties, Bad Prefixes

An LT property P_{safe} over AP is called a *safety* property if for all words $\sigma \in (2^{AP})^\omega \setminus P_{safe}$ there exists a finite prefix $\widehat{\sigma}$ of σ such that

$$P_{safe} \cap \left\{ \sigma' \in (2^{AP})^\omega \mid \widehat{\sigma} \text{ is a finite prefix of } \sigma' \right\} \;=\; \varnothing.$$

Any such finite word $\widehat{\sigma}$ is called a *bad prefix* for P_{safe}. A *minimal* bad prefix for P_{safe} denotes a bad prefix $\widehat{\sigma}$ for P_{safe} for which no proper prefix of $\widehat{\sigma}$ is a bad prefix for P_{safe}. In other words, minimal bad prefixes are bad prefixes of minimal length. The set of all bad prefixes for P_{safe} is denoted by $\mathit{BadPref}(P_{safe})$, the set of all minimal bad prefixes by $\mathit{MinBadPref}(P_{safe})$. ■

Let us first observe that any invariant is a safety property. For propositional formula Φ over AP and its invariant P_{inv}, all finite words of the form

$$A_0\, A_1\, \ldots\, A_n \;\in\; (2^{AP})^+$$

with $A_0 \models \Phi, \ldots, A_{n-1} \models \Phi$ and $A_n \not\models \Phi$ constitute the minimal bad prefixes for P_{inv}. The following two examples illustrate that there are safety properties that are not invariants.

Example 3.23. A Safety Property for a Traffic Light

We consider a specification of a traffic light with the usual three phases "red", "green", and "yellow". The requirement that each red phase should be immediately preceded by a yellow phase is a safety property but not an invariant. This is shown in the following.

Let *red*, *yellow*, and *green* be atomic propositions. Intuitively, they serve to mark the states describing a red (yellow or green) phase. The property "always at least one of the lights is on" is specified by:

$$\{\, \sigma = A_0\, A_1 \ldots \mid A_j \subseteq AP \;\wedge\; A_j \neq \varnothing \,\}.$$

The bad prefixes are finite words that contain $\varnothing$. A minimal bad prefix ends with $\varnothing$. The property "it is never the case that two lights are switched on at the same time" is specified

by

$$\{\, \sigma = A_0\, A_1 \ldots \mid A_j \subseteq AP \;\wedge\; |A_j| \leqslant 1 \,\}.$$

Bad prefixes for this property are words containing sets such as $\{\, red, green \,\}$, $\{\, red, yellow \,\}$, and so on. Minimal bad prefixes end with such sets.

Now let $AP' = \{\, red, yellow \,\}$. The property "a red phase must be preceded immediately by a yellow phase" is specified by the set of infinite words $\sigma = A_0\, A_1 \ldots$ with $A_i \subseteq \{\, red, yellow \,\}$ such that for all $i \geqslant 0$ we have that

$$red \in A_i \text{ implies } i > 0 \text{ and } yellow \in A_{i-1}.$$

The bad prefixes are finite words that violate this condition. An example of bad prefixes that are minimal is:

$$\varnothing\, \varnothing\, \{\, red \,\} \quad \text{and} \quad \varnothing\, \{\, red \,\}.$$

The following bad prefix is not minimal:

$$\{\, yellow \,\}\, \{\, yellow \,\}\, \{\, red \,\}\, \{\, red \,\}\, \varnothing\, \{\, red \,\}$$

since it has a proper prefix $\{\, yellow \,\}\, \{\, yellow \,\}\, \{\, red \,\}\, \{\, red \,\}$ which is also a bad prefix.

The minimal bad prefixes of this safety property are regular in the sense that they constitute a regular language. The finite automaton in Figure 3.9 accepts precisely the minimal bad prefixes for the above safety property.[7] Here, $\neg yellow$ should be read as either $\varnothing$ or $\{\, red \,\}$. Note the other properties given in this example are also regular. ■

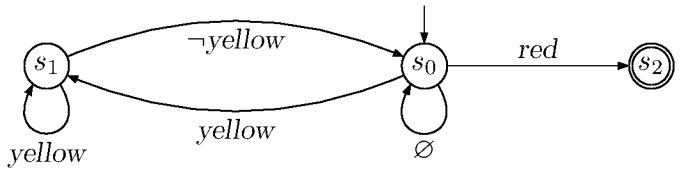


Figure 3.9: A finite automaton for the minimal bad prefixes of a regular safety property.

Example 3.24. A Safety Property for a Beverage Vending Machine

For a beverage vending machine, a natural requirement is that

"The number of inserted coins is always at least the number of dispensed drinks."

[7] The main concepts of a finite automaton as acceptors for languages over finite words are summarized in Section 4.1.

Using the set of propositions $\{ pay, drink \}$ and the obvious labeling function, this property could be formalized by the set of infinite words $A_0 A_1 A_2 \ldots$ such that for all $i \geqslant 0$ we have

$$| \{ 0 \leqslant j \leqslant i \mid pay \in A_j \} | \;\geqslant\; | \{ 0 \leqslant j \leqslant i \mid drink \in A_j \} |$$

Bad prefixes for this safety property are, for example

$$\varnothing \{ pay \} \{ drink \} \{ drink \} \quad \text{and}$$
$$\varnothing \{ pay \} \{ drink \} \varnothing \{ pay \} \{ drink \} \{ drink \}$$

It is left to the interested reader to check that both beverage vending machines from Figure 3.8 satisfy the above safety property. ■

Safety properties are requirements for the finite traces which is formally stated in the following lemma:

Lemma 3.25. Satisfaction Relation for Safety Properties

For transition system TS without terminal states and safety property P_{safe}*:*

$$TS \models P_{safe} \;\; \textit{if and only if} \;\; Traces_{fin}(TS) \cap BadPref(P_{safe}) = \varnothing.$$

Proof: "if": By contradiction. Let $Traces_{fin}(TS) \cap BadPref(P_{safe}) = \varnothing$ and assume that $TS \not\models P_{safe}$. Then, $trace(\pi) \notin P_{safe}$ for some path π in TS. Thus, $trace(\pi)$ starts with a bad prefix $\widehat{\sigma}$ for P_{safe}. But then, $\widehat{\sigma} \in Traces_{fin}(TS) \cap BadPref(P_{safe})$. Contradiction.

"only if": By contradiction. Let $TS \models P_{safe}$ and assume that $\widehat{\sigma} \in Traces_{fin}(TS) \cap BadPref(P_{safe})$. The finite trace $\widehat{\sigma} = A_1 \ldots A_n \in Traces_{fin}(TS)$ can be extended to an infinite trace $\sigma = A_1 \ldots A_n A_{n+1} A_{n+2} \ldots \in Traces(TS)$. Then, $\sigma \notin P_{safe}$ and thus, $TS \not\models P_{safe}$. ■

We conclude this section with an alternative characterization of safety properties by means of their closure.

Definition 3.26. Prefix and Closure

For trace $\sigma \in (2^{AP})^\omega$, let $pref(\sigma)$ denote the set of finite prefixes of σ, i.e.,

$$pref(\sigma) = \{ \widehat{\sigma} \in (2^{AP})^* \mid \widehat{\sigma} \text{ is a finite prefix of } \sigma \}.$$

that is, if $\sigma = A_0\, A_1\, \ldots$ then $\mathit{pref}(\sigma) = \{\varepsilon, A_0, A_0A_1, A_0A_1A_2, \ldots\}$ is an infinite set of finite words. This notion is lifted to sets of traces in the usual way. For property P over AP:

$$\mathit{pref}(P) = \bigcup_{\sigma \in P} \mathit{pref}(\sigma).$$

The *closure* of LT property P is defined by

$$\mathit{closure}(P) \;=\; \{\sigma \in (2^{AP})^\omega \mid \mathit{pref}(\sigma) \subseteq \mathit{pref}(P)\}.$$

■

For instance, for infinite trace $\sigma = ABABAB\ldots$ (where $A, B \subseteq AP$) we have $\mathit{pref}(\sigma) = \{\varepsilon, A, AB, ABA, ABAB, \ldots\}$ which equals the regular language given by the regular expression $(AB)^*(A + \varepsilon)$.

The *closure* of an LT property P is the set of infinite traces whose finite prefixes are also prefixes of P. Stated differently, infinite traces in the closure of P do not have a prefix that is not a prefix of P itself. As we will see below, the closure is a key concept in the characterization of safety and liveness properties.

Lemma 3.27. Alternative Characterization of Safety Properties

Let P be an LT property over AP. Then, P is a safety property iff $\mathit{closure}(P) = P$.

Proof: "if": Let us assume that $\mathit{closure}(P) = P$. To show that P is a safety property, we take an element $\sigma \in (2^{AP})^\omega \setminus P$ and show that σ starts with a bad prefix for P. Since $\sigma \notin P = \mathit{closure}(P)$ there exists a finite prefix $\widehat{\sigma}$ of σ with $\widehat{\sigma} \notin \mathit{pref}(P)$. By definition of $\mathit{pref}(P)$, none of the words $\sigma' \in (2^{AP})^\omega$ where $\widehat{\sigma} \in \mathit{pref}(\sigma')$ belongs to P. Hence, $\widehat{\sigma}$ is a bad prefix for P, and by definition, P is a safety property.

"only if": Let us assume that P is a safety property. We have to show that $P = \mathit{closure}(P)$. The inclusion $P \subseteq \mathit{closure}(P)$ holds for all LT properties. It remains to show that $\mathit{closure}(P) \subseteq P$. We do so by contradiction. Let us assume that there is some $\sigma = A_1\, A_2\, \ldots \in \mathit{closure}(P) \setminus P$. Since P is a safety property and $\sigma \notin P$, σ has a finite prefix

$$\widehat{\sigma} = A_1\, \ldots\, A_n \in \mathit{BadPref}(P).$$

As $\sigma \in \mathit{closure}(P)$ we have $\widehat{\sigma} \in \mathit{pref}(\sigma) \subseteq \mathit{pref}(P)$. Hence, there exists a word $\sigma' \in P$ of the form

$$\sigma' = \underbrace{A_1\, \ldots\, A_n}_{\text{bad prefix}}\, B_{n+1}\, B_{n+2}\, \ldots$$

This contradicts the fact that P is a safety property. ■

3.3.3 Trace Equivalence and Safety Properties

We have seen before that there is a strong relationship between trace inclusion of transition systems and the satisfaction of LT properties (see Theorem 3.15, page 104):

$$\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS') \quad \text{if and only if} \quad \begin{array}{l}\text{for all LT properties } P\text{:}\\ TS' \models P \text{ implies } TS \models P\end{array}$$

for transition systems TS and TS' without terminal states. Note that this result considers all *infinite* traces. The above thus states a relationship between infinite traces of transition systems and the validity of LT properties. When considering only finite traces instead of infinite ones, a similar connection with the validity of safety properties can be established, as stated by the following theorem.

Theorem 3.28. Finite Trace Inclusion and Safety Properties

Let TS and TS' be transition systems without terminal states and with the same set of propositions AP. Then the following statements are equivalent:

(a) $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$,

(b) For any safety property P_{safe}*:* $TS' \models P_{\mathit{safe}}$ *implies* $TS \models P_{\mathit{safe}}$.

Proof:

(a) $\Longrightarrow$ (b): Let us assume that $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$ and let P_{safe} be a safety property with $TS' \models P_{\mathit{safe}}$. By Lemma 3.25, we have $\mathit{Traces}_{\mathit{fin}}(TS') \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing$, and hence, $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing$. Again by Lemma 3.25, we get $TS \models P_{\mathit{safe}}$.

(b) $\Longrightarrow$ (a): Assume that (b) holds. Let $P_{\mathit{safe}} = \mathit{closure}(\mathit{Traces}(TS'))$. Then, P_{safe} is a safety property and we have $TS' \models P_{\mathit{safe}}$ (see Exercise 3.9, page 147). Hence, (b) yields $TS \models P_{\mathit{safe}}$, i.e.,

$$\mathit{Traces}(TS) \subseteq \mathit{closure}(\mathit{Traces}(TS')).$$

From this, we may derive

$$\begin{array}{rcl}\mathit{Traces}_{\mathit{fin}}(TS) & = & \mathit{pref}(\mathit{Traces}(TS))\\ & \subseteq & \mathit{pref}(\mathit{closure}(\mathit{Traces}(TS')))\\ & = & \mathit{pref}(\mathit{Traces}(TS'))\\ & = & \mathit{Traces}_{\mathit{fin}}(TS').\end{array}$$

Here we use the property that for any P it holds that $\mathit{pref}(\mathit{closure}(P)) = \mathit{pref}(P)$ (see Exercise 3.10, page 147). ■

Theorem 3.28 is of relevance for the gradual design of concurrent systems. If a preliminary design (i.e., a transition system) TS' is refined to a design TS such that

$$Traces(TS) \not\subseteq Traces(TS'),$$

then the LT properties of TS' cannot be carried over to TS. However, if the finite traces of TS are finite traces of TS' (which is a weaker requirement than full trace inclusion of TS and TS'), i.e.,

$$Traces_{fin}(TS) \subseteq Traces_{fin}(TS'),$$

then all safety properties that have been established for TS' also hold for TS. Other requirements for TS, i.e., LT properties that fall outside the scope of safety properties, need to be checked using different techniques.

Corollary 3.29. Finite Trace Equivalence and Safety Properties

Let TS and TS' be transition systems without terminal states and with the same set AP of atomic propositions. Then, the following statements are equivalent:

(a) $Traces_{fin}(TS) = Traces_{fin}(TS')$,

(b) For any safety property P_{safe} over AP: $TS \models P_{safe} \Longleftrightarrow TS' \models P_{safe}$.

A few remarks on the difference between finite trace inclusion and trace inclusion are in order. Since we assume transition systems without terminal states, there is only a slight difference between trace inclusion and finite trace inclusion. For *finite* transition systems TS and TS' without terminal states, trace inclusion and finite trace inclusion coincide. This can be derived from the following theorem.

Theorem 3.30. Relating Finite Trace and Trace Inclusion

Let TS and TS' be transition systems with the same set AP of atomic propositions such that TS has no terminal states and TS' is finite. Then:

$$Traces(TS) \subseteq Traces(TS') \iff Traces_{fin}(TS) \subseteq Traces_{fin}(TS').$$

Proof: The implication from left to right follows from the monotonicity of $pref(\cdot)$ and the fact that $Traces_{fin}(TS) = pref(Traces(TS))$ for any transition system TS.

It remains to consider the proof for the implication $\Longleftarrow$. Let us assume that $Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$. As TS has no terminal states, all traces of TS are infinite. Let $A_0 A_1 \ldots \in$

$\mathit{Traces}(TS)$. To prove that $A_0A_1\ldots \in \mathit{Traces}(TS')$ we have to show that there exists a path in TS', say $s_0\,s_1\ldots$, that generates this trace, i.e., $\mathit{trace}(s_0\,s_1\ldots) = A_0\,A_1\ldots$.

Any finite prefix $A_0\,A_1\ldots A_m$ of the infinite trace $A_0A_1\ldots$ is in $\mathit{Traces}_{\mathit{fin}}(TS)$, and as $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$, also in $\mathit{Traces}_{\mathit{fin}}(TS')$. Thus, for any natural number m, there exists a finite path $\pi^m = s_0^m s_1^m \ldots s_m^m$ in TS' such that

$$\mathit{trace}(\pi^m) \;=\; L(s_0^m)L(s_1^m)\ldots L(s_m^m) = A_0A_1\ldots A_m$$

where L denotes the labeling function of TS'. Thus, $L(s_j^m) = A_j$ for all $0 \leqslant j \leqslant m$.

Although $A_0\ldots A_m$ is a prefix of $A_0\ldots A_{m+1}$, it is not guaranteed that path π^m is a prefix of π^{m+1}. Due to the finiteness of TS', however, there is an infinite subsequence $\pi^{m_0}\,\pi^{m_1}\,\pi^{m_2}\ldots$ of $\pi^0\,\pi^1\,\pi^2\ldots$ such that π^{m_i} and $\pi^{m_{i+1}}$ agree on the first i states. Thus, $\pi^{m_0}\,\pi^{m_1}\,\pi^{m_2}\ldots$ induces an infinite path π in TS' with the desired property.

This is formally proven using a so-called *diagonalization technique*. This goes as follows. Let $I_0, I_1, I_2, \ldots$ be an infinite series of infinite sets of indices (i.e., natural numbers) with $I_n \subseteq \{\, m \in \mathbb{N} \mid m \geqslant n \,\}$ and $s_0, s_1, \ldots$ be states in TS' such that for all natural numbers n it holds that

(1) $n \geqslant 1$ implies $I_{n-1} \supseteq I_n$,

(2) $s_0\,s_1\,s_2\,\ldots\,s_n$ is an initial, finite path fragment in TS',

(3) for all $m \in I_n$ it holds that $s_0\,\ldots\,s_n = s_0^m\,\ldots\,s_n^m$.

The definition of the sets I_n and states s_n is by induction on n.

Base case ($n = 0$): As $\{\, s_0^m \mid m \in \mathbb{N} \,\}$ is finite (since it is a subset of the finite set of initial states of TS'), there exists an initial state s_0 in TS' and an infinite index set I_0 such that $s_0 = s_0^m$ for all $m \in I_0$.

Induction step $n \Longrightarrow n{+}1$. Assume that the index sets $I_0, \ldots, I_n$ and states $s_0, \ldots, s_n$ are defined. Since TS' is finite, $\mathit{Post}(s_n)$ is finite. Furthermore, by the induction hypothesis $s_n = s_n^m$ for all $m \in I_n$, and thus

$$\{\, s_{n+1}^m \mid m \in I_n, m \geqslant n{+}1 \,\} \;\subseteq\; \mathit{Post}(s_n).$$

Since I_n is infinite, there exists an infinite subset $I_{n+1} \subseteq \{\, m \in I_n \mid m \geqslant n{+}1 \,\}$ and a state $s_{n+1} \in \mathit{Post}(s_n)$ such that $s_{n+1}^m = s_{n+1}$ for all $m \in I_{n+1}$. It follows directly that the above properties (1) through (3) are fulfilled.

We now consider the state sequence $s_0\, s_1\, \ldots$ in TS'. Obviously, this state sequence is a path in TS' satisfying $trace(s_0\, s_1\, \ldots) = A_0\, A_1\, \ldots$. Consequently, $A_0\, A_1\, \ldots \in Traces(TS')$. ■

Remark 3.31. Image-Finite Transition Systems

The result stated in Theorem 3.30 also holds under slightly weaker conditions: it suffices to require that TS has no terminal states (as in Theorem 3.30) and that TS' is AP image-finite (rather than being finite).

Let $TS' = (S, Act, \rightarrow, I, AP, L)$. Then, TS' is called AP image-finite (or briefly *image-finite*) if

(i) for all $A \subseteq AP$, the set $\{\, s_0 \in I \mid L(s_0) = A \,\}$ is finite and

(ii) for all states s in TS' and all $A \subseteq AP$, the set of successors $\{\, s' \in Post(s) \mid L(s') = A \,\}$ is finite.

Thus, any finite transition system is image-finite. Moreover, any transition system that is AP-deterministic is image-finite. (Recall that AP-determinism requires $\{\, s_0 \in I \mid L(s_0) = A \,\}$ and $\{\, s' \in Post(s) \mid L(s') = A \,\}$ to be either singletons or empty sets; see Definition 2.5, page 24.)

In fact, a careful inspection of the proof of Theorem 3.30 shows that (i) and (ii) for TS' are used in the construction of the index sets I_n and states s_n. Hence, we have $Traces(TS) \subseteq Traces(TS')$ iff $Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$, provided TS has no terminal states and TS' is image-finite. ■

Trace and finite trace inclusion, however, coincide neither for infinite transition systems nor for finite ones which have terminal states.

Example 3.32. Finite vs. Infinite Transition System

Consider the transition systems sketched in Figure 3.10, where b stands for an atomic proposition. Transition system TS (on the left) is finite, whereas TS' (depicted on the right) is infinite and not image-finite, because of the infinite branching in the initial state. It is not difficult to observe that

$$Traces(TS) \not\subseteq Traces(TS') \quad \text{and} \quad Traces_{fin}(TS) \subseteq Traces_{fin}(TS').$$

This stems from the fact that TS can take the self-loop infinitely often and never reaches a b-state, whereas TS' does not exhibit such behavior. Moreover, any finite trace of TS is

of the form $(\varnothing)^n$ for $n \geqslant 0$ and is also a finite trace of TS'. Consequently, LT properties of TS' do not carry over to TS (and those of TS may not hold for TS'). For example, the LT property "eventually b" holds for TS', but not for TS. Similarly, the LT property "never b" holds for TS, but not for TS'.

Although these transition systems might seem rather artificial, this is not the case: TS could result from an infinite loop in a program, whereas TS' could model the semantics of a program fragment that nondeterministically chooses a natural number k and then performs k steps. ■

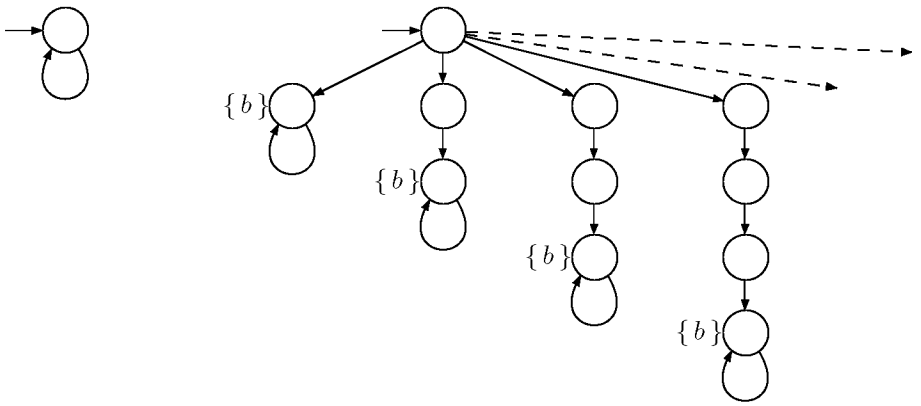


Figure 3.10: Distinguishing trace inclusion from finite trace inclusion.

3.4 Liveness Properties

Informally speaking, safety properties specify that "something bad never happens". For the mutual exclusion algorithm, the "bad" thing is that more than one process is in its critical section, while for the traffic light the "bad" situation is whenever a red light phase is not preceded by a yellow light phase. An algorithm can easily fulfill a safety property by simply doing nothing as this will never lead to a "bad" situation. As this is usually undesired, safety properties are complemented by properties that require some progress. Such properties are called "liveness" properties (or sometimes "progress" properties). Intuitively, they state that "something good" will happen in the future. Whereas safety properties are violated in finite time, i.e., by a finite system run, liveness properties are violated in infinite time, i.e., by infinite system runs.

3.4.1 Liveness Properties

Several (nonequivalent) notions of liveness properties have been defined in the literature. We follow here the approach of Alpern and Schneider [5, 6, 7]. They provided a formal notion of liveness properties which relies on the view that liveness properties do not constrain the finite behaviors, but require a certain condition on the infinite behaviors. A typical example for a liveness property is the requirement that certain events occur infinitely often. In this sense, the "good event" of a liveness property is a condition on the infinite behaviors, while the "bad event" for a safety property occurs in a finite amount of time, if it occurs at all.

In our approach, a liveness property (over AP) is defined as an LT property that does not rule out any prefix. This entails that the set of finite traces of a system are of no use at all to decide whether a liveness property holds or not. Intuitively speaking, it means that any finite prefix can be extended such that the resulting infinite trace satisfies the liveness property under consideration. This is in contrast to safety properties where it suffices to have one finite trace (the "bad prefix") to conclude that a safety property is refuted.

Definition 3.33. Liveness Property

LT property P_{live} over AP is a *liveness* property whenever $pref(P_{live}) = (2^{AP})^*$. ■

Thus, a liveness property (over AP) is an LT property P such that each finite word can be extended to an infinite word that satisfies P. Stated differently, P is a liveness property if and only if for all finite words $w \in (2^{AP})^*$ there exists an infinite word $\sigma \in (2^{AP})^\omega$ satisfying $w\sigma \in P$.

Example 3.34. Repeated Eventually and Starvation Freedom

In the context of mutual exclusion algorithms the natural safety property that is required ensures the mutual exclusion property stating that the processes are never simultaneously in their critical sections. (This is even an invariant.) Typical liveness properties that are desired assert that

- (eventually) each process will eventually enter its critical section;
- (repeated eventually) each process will enter its critical section infinitely often;
- (starvation freedom) each waiting process will eventually enter its critical section.

Let's see how these liveness properties are formalized as LT properties and let us check that

they are liveness properties. As in Example 3.14, we will deal with the atomic propositions $wait_1, crit_1, wait_2, crit_2$ where $wait_i$ characterizes the states where process P_i has requested access to its critical section and is in its waiting phase, while $crit_i$ serves as a label for the states where P_i has entered its critical section. We now formalize the three properties by LT properties over $AP = \{wait_1, crit_1, wait_2, crit_2\}$. The first property (eventually) consists of all infinite words $A_0\, A_1 \ldots$ with $A_j \subseteq AP$ such that

$$(\exists j \geqslant 0.\ crit_1 \in A_j) \ \wedge \ (\exists j \geqslant 0.\ crit_2 \in A_j)$$

which requires that P_1 and P_2 are in their critical sections at least once. The second property (repeated eventually) poses the condition

$$(\forall k \geqslant 0.\, \exists j \geqslant k.\ crit_1 \in A_j) \ \wedge \ (\forall k \geqslant 0.\, \exists j \geqslant k.\ crit_2 \in A_j)$$

stating that P_1 and P_2 are infinitely often in their critical sections. This formula is often abbreviated by

$$\left(\overset{\infty}{\exists}\ j \geqslant 0.\ crit_1 \in A_j\right) \ \wedge \ \left(\overset{\infty}{\exists}\ j \geqslant 0.\ crit_2 \in A_j\right).$$

The third property (starvation freedom) requires that

$$\begin{aligned} &\forall j \geqslant 0.\ (wait_1 \in A_j \ \Rightarrow \ (\exists k > j.\, crit_1 \in A_k)) \ \wedge \\ &\forall j \geqslant 0.\ (wait_2 \in A_j \ \Rightarrow \ (\exists k > j.\, crit_2 \in A_k)). \end{aligned}$$

It expresses that each process that is waiting will acquire access to the critical section at some later time point. Note that here we implicitly assume that a process that starts waiting to acquire access to the critical section does not "give up" waiting, i.e., it continues waiting until it is granted access.

All aforementioned properties are liveness properties, as any finite word over AP is a prefix of an infinite word where the corresponding condition holds. For instance, for starvation freedom, a finite trace in which a process is waiting but never acquires access to its critical section can always be extended to an infinite trace that satisfies the starvation freedom property (by, e.g., providing access in an strictly alternating fashion from a certain point on). ■

3.4.2 Safety vs. Liveness Properties

This section studies the relationship between liveness and safety properties. In particular, it provides answers to the following questions:

- Are safety and liveness properties disjoint?, and

- Is any linear-time property a safety or liveness property?

As we will see, the first question will be answered affirmatively while the second question will result in a negative answer. Interestingly enough, though, for any LT property P an equivalent LT property P' does exist which is a combination (i.e., intersection) of a safety and a liveness property. All in all, one could say that the identification of safety and liveness properties thus provides an essential characterization of linear-time properties.

The first result states that safety and liveness properties are indeed almost disjoint. More precisely, it states that the only property that is both a safety and a liveness property is nonrestrictive, i.e., allows all possible behaviors. Logically speaking, this is the equivalent of "true".

Lemma 3.35. Intersection of Safety and Liveness Properties

The only LT property over AP that is both a safety and a liveness property is $(2^{AP})^\omega$.

Proof: Assume P is a liveness property over AP. By definition, $pref(P) = (2^{AP})^*$. It follows that $closure(P) = (2^{AP})^\omega$. If P is a safety property too, $closure(P) = P$, and hence $P = (2^{AP})^\omega$. ■

Recall that the closure of property P (over AP) is the set of infinite words (over 2^{AP}) for which all prefixes are also prefixes of P. In order to show that an LT property can be considered as a conjunction of a liveness and a safety property, the following result is helpful. It states that the closure of the union of two properties equals the union of their closures.

Lemma 3.36. Distributivity of Union over Closure

For any LT properties P *and* P'*:*

$$closure(P) \cup closure(P') = closure(P \cup P').$$

Proof: $\subseteq$: As $P \subseteq P'$ implies $closure(P) \subseteq closure(P')$, we have $P \subseteq P \cup P'$ implies $closure(P) \subseteq closure(P \cup P')$. In a similar way it follows that $closure(P') \subseteq closure(P \cup P')$. Thus, $closure(P) \cup closure(P') \subseteq closure(P \cup P')$.

$\supseteq$: Let $\sigma \in closure(P \cup P')$. By definition of closure, $pref(\sigma) \subseteq pref(P \cup P')$. As $pref(P \cup P') = pref(P) \cup pref(P')$, any finite prefix of σ is in $pref(P)$ or in $pref(P')$ (or in both).

As $\sigma \in (2^{AP})^\omega$, σ has infinitely many prefixes. Thus, infinitely many finite prefixes of σ belong to $pref(P)$ or to $pref(P')$ (or to both). W.l.o.g., assume $pref(\sigma) \cap pref(P)$ to be infinite. Then $pref(\sigma) \subseteq pref(P)$, which yields $\sigma \in closure(P)$, and thus $\sigma \in closure(P) \cup closure(P')$. The fact that $pref(\sigma) \subseteq pref(P)$ can be shown by contraposition. Assume $\widehat{\sigma} \in pref(\sigma) \setminus pref(P)$. Let $|\widehat{\sigma}| = k$. As $pref(\sigma) \cap pref(P)$ is infinite, there exists $\widehat{\sigma'} \in pref(\sigma) \cap pref(P)$ with length larger than k. But then, there exists $\sigma' \in P$ with $\widehat{\sigma'} \in pref(\sigma')$. It then follows that $\widehat{\sigma} \in pref(\widehat{\sigma'})$ (as both $\widehat{\sigma}$ and $\widehat{\sigma'}$ are prefixes of σ) and as $pref(\widehat{\sigma'}) \subseteq pref(P)$, it follows that $\widehat{\sigma} \in pref(P)$. This contradicts $\widehat{\sigma} \in pref(\sigma) \setminus pref(P)$. ■

Consider the beverage vending machine of Figure 3.5 (on page 97), and the following property:

"the machine provides beer infinitely often
after initially providing soda three times in a row"

In fact, this property consists of two parts. On the one hand, it requires beer to be provided infinitely often. As any finite trace can be extended to an infinite trace that enjoys this property it is a liveness property. On the other hand, the first three drinks it provides should all be soda. This is a safety property, since any finite trace in which one of the first three drinks provided is beer violates it. The property is thus a combination (in fact, a conjunction) of a safety and a liveness property. The following result shows that *every* LT property can be decomposed in this way.

Theorem 3.37. Decomposition Theorem

For any LT property P over AP there exists a safety property P_{safe} and a liveness property P_{live} (both over AP) such that

$$P = P_{safe} \cap P_{live}.$$

Proof: Let P be an LT property over AP. It is easy to see that $P \subseteq closure(P)$. Thus: $P = closure(P) \cap P$, which by set calculus can be rewritten into:

$$P \;=\; \underbrace{closure(P)}_{=P_{safe}} \;\cap\; \underbrace{\Big(P \cup \Big((2^{AP})^\omega \setminus closure(P)\Big)\Big)}_{=P_{live}}$$

By definition, $P_{safe} = closure(P)$ is a safety property. It remains to prove that $P_{live} = P \cup \Big((2^{AP})^\omega \setminus closure(P)\Big)$ is a liveness property. By definition, P_{live} is a liveness property whenever $pref(P_{live}) = (2^{AP})^*$. This is equivalent to $closure(P_{live}) = (2^{AP})^\omega$. As for any

LT property P, $\mathit{closure}(P) \subseteq (2^{AP})^\omega$ holds true, it suffices to showing that $(2^{AP})^\omega \subseteq \mathit{closure}(P_{\mathit{live}})$. This goes as follows:

$$\begin{aligned}
\mathit{closure}(P_{\mathit{live}}) &= \mathit{closure}\Big(P \cup ((2^{AP})^\omega \setminus \mathit{closure}(P))\Big) \\
&\overset{\text{Lemma 3.36}}{=} \mathit{closure}(P) \cup \mathit{closure}\Big((2^{AP})^\omega \setminus \mathit{closure}(P)\Big) \\
&\supseteq \mathit{closure}(P) \cup \Big((2^{AP})^\omega \setminus \mathit{closure}(P)\Big) \\
&= (2^{AP})^\omega
\end{aligned}$$

where in the one-but-last step in the derivation, we exploit the fact that $\mathit{closure}(P') \supseteq P'$ for all LT properties P'. ■

The proof of Theorem 3.37 shows that $P_{\mathit{safe}} = \mathit{closure}(P)$ is a safety property and $P_{\mathit{live}} = P \cup ((2^{AP})^\omega \setminus \mathit{closure}(P))$ a liveness property with $P = P_{\mathit{safe}} \cap P_{\mathit{live}}$. In fact, this decomposition is the "sharpest" one for P since P_{safe} is the *strongest* safety property and P_{live} the *weakest* liveness property that can serve for a decomposition of P:

Lemma 3.38. Sharpest Decomposition

Let P be an LT property and $P = P_{\mathit{safe}} \cap P_{\mathit{live}}$ where P_{safe} is a safety property and P_{live} a liveness property. We then have

1. $\mathit{closure}(P) \subseteq P_{\mathit{safe}}$,
2. $P_{\mathit{live}} \subseteq P \cup ((2^{AP})^\omega \setminus \mathit{closure}(P))$.

Proof: See Exercise 3.12, page 147. ■

A summary of the classification of LT properties is depicted as a Venn diagram in Figure 3.11. The circle denotes the set of all LT properties over a given set of atomic propositions.

Remark 3.39. Topological Characterizations of Safety and Liveness

Let us conclude this section with a remark for readers who are familiar with basic notions of topological spaces. The set $(2^{AP})^\omega$ can be equipped with the distance function given by $d(\sigma_1, \sigma_2) = 1/2^n$ if σ_1, σ_2 are two distinct infinite words $\sigma_1 = A_1 A_2 \ldots$ and $\sigma_2 = B_1 B_2 \ldots$ and n is the length of the longest common prefix. Moreover, we put $d(\sigma, \sigma) = 0$. Then, d is a metric on $(2^{AP})^\omega$, and hence induces a topology on $(2^{AP})^\omega$. Under this topology,

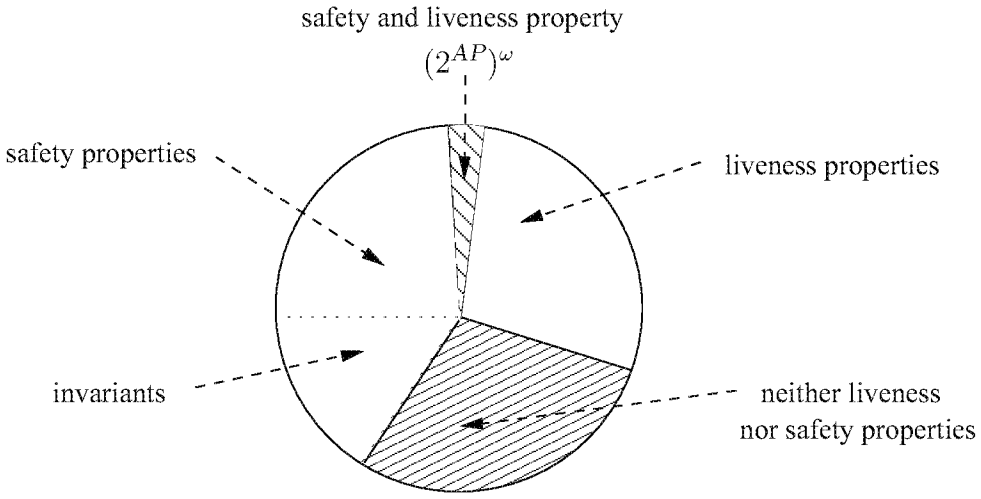


Figure 3.11: Classification of linear-time properties.

the safety properties are exactly the closed sets, while the liveness properties agree with the dense sets. In fact, *closure*(P) is the topological closure of P, i.e., the smallest closed set that contains P. The result stated in Theorem 3.37 then follows from the well-known fact that any subset of a topological space (of the kind described above) can be written as the intersection of its closure and a dense set. ■

3.5 Fairness

An important aspect of reactive systems is fairness. Fairness assumptions rule out infinite behaviors that are considered unrealistic, and are often necessary to establish liveness properties. We illustrate the concept of fairness by means of a frequently encountered problem in concurrent systems.

Example 3.40. Process Fairness

Consider N processes $P_1, \ldots, P_N$ which require a certain service. There is one server process *Server* that is expected to provide services to these processes. A possible strategy that *Server* can realize is the following. Check the processes starting with P_1, then P_2, and so on, and serve the first thus encountered process that requires service. On finishing serving this process, repeat this selection procedure once again starting with checking P_1.

Now suppose that P_1 is continuously requesting service. Then this strategy will result in *Server* always serving P_1. Since in this way another process has to wait infinitely long before being served, this is called an unfair strategy. In a fair serving strategy it is required that the server eventually responds to any request by any one of the processes. For instance, a round-robin scheduling strategy where each process is only served for a limited amount of time is a fair strategy: after having served one process, the next (in the round-robin order) is checked and, if needed, served. ■

When verifying concurrent systems one is often only interested in paths in which enabled transitions (statements) are executed in some "fair" manner. Consider, for instance, a mutual exclusion algorithm for two processes. In order to prove starvation freedom, the situation in which a process that wants to enter its critical section has to wait infinitely long, we want to exclude those paths in which the competitor process is always being selected for execution. This type of fairness is also known as *process fairness*, since it concerns the fair scheduling of the execution of processes. If we were to consider unfair paths when proving starvation freedom, we would usually fail, since there always exists an unfair strategy according to which some process is always neglected, and thus can never make progress. One might argue that such unfair strategy is unrealistic and should be avoided.

Example 3.41. Starvation Freedom

Consider the transition systems TS_{Sem} and TS_{Pet} for the semaphore-based mutual exclusion algorithms (see Example 2.24 on page 43) and Peterson's algorithm. The starvation freedom property

"Once access is requested, a process does not have to wait infinitely long before acquiring access to its critical section"

is violated by transition system TS_{Sem} while it permits only one of the processes to proceed, while the other process is starving (or only acquiring access to the critical section finitely often). The transition system TS_{Pet} for Peterson's algorithm, however, fulfills this property.

The property

"Each of the processes is infinitely often in its critical section"

is violated by both transition systems as none of them excludes the fact that a process would never (or only finitely often) request to enter the critical section. ■

Process fairness is a particular form of fairness. In general, fairness assumptions are needed to prove liveness or other properties stating that the system makes some progress ("something good will eventually happen"). This is of vital importance if the transition system to be checked contains nondeterminism. Fairness is then concerned with resolving nondeterminism in such a way that it is not biased to consistently ignore a possible option. In the above example, the scheduling of processes is nondeterministic: the choice of the next process to be executed (if there are at least two processes that can be potentially selected) is arbitrary. Another prominent example where fairness is used to "resolve" nondeterminism is in modeling concurrent processes by means of interleaving. Interleaving is equivalent to modeling the concurrent execution of two independent processes by enumerating all the possible orders in which activities of the processes can be executed (see Chapter 2).

Example 3.42. Independent Traffic Lights

Consider the transition system

$$TS \;=\; TrLight_1 \;|||\; TrLight_2$$

for the two independent traffic lights described in Example 2.17 (page 36). The liveness property

"Both traffic lights are infinitely often green"

is not satisfied, since

$$\{\, red_1, red_2 \,\}\, \{\, green_1, red_2 \,\}\, \{\, red_1, red_2 \,\}\, \{\, green_1, red_2 \,\} \ldots$$

is a trace of TS where only the first traffic light is infinitely often green. ■

What is wrong with the above examples? In fact, nothing. Let us explain this. In the traffic light example, the information whether each traffic light switches color infinitely often is lost by means of interleaving. The trace in which only the first traffic light is acting while the second light seems to be completely stopped is formally a trace of the transition system $TrLight_1 \,|||\, TrLight_2$. However, it does not represent a realistic behavior as in practice no traffic light is infinitely faster than another.

For the semaphore-based mutual exclusion algorithm, the difficulty is the degree of abstraction. A semaphore is not a willful individual that arbitrarily chooses a process which is authorized to enter its critical section. Instead, the waiting processes are administered in a queue (or another "fair" medium). The required liveness can be proven in one of the following refinement steps, in which the specification of the behavior of the semaphore is sufficiently detailed.

3.5.1 Fairness Constraints

The above considerations show that we—to obtain a realistic picture of the behavior of a parallel system modeled by a transition system—need a more alleviated form of satisfaction relation for LT properties, which implies an "adequate" resolution of the nondeterministic decisions in a transition system. In order to rule out the unrealistic computations, *fairness constraints* are imposed.

In general, a fair execution (or trace) is characterized by the fact that certain fairness constraints are fulfilled. Fairness constraints are used to rule out computations that are considered to be unreasonable for the system under consideration. Fairness constraints come in different flavors:

- *Unconditional fairness*: e.g.,"Every process gets its turn infinitely often."
- *Strong fairness*: e.g., "Every process that is enabled infinitely often gets its turn infinitely often."
- *Weak fairness:* e.g., "Every process that is continuously enabled from a certain time instant on gets its turn infinitely often."

Here, the term "is enabled" has to be understood in the sense of "ready to execute (a transition)". Similarly, "gets its turn" stands for the execution of an arbitrary transition. This can, for example, be a noncritical action, acquiring a shared resource, an action in the critical section, or a communication action.

An execution fragment is unconditionally fair with respect to, e.g., "a process enters its critical section" or "a process gets its turn", if these properties hold infinitely often. That is to say, a process enters its critical section infinitely often, or, in the second example, a process gets its turn infinitely often. Note that no condition (such as "a process is enabled") is expressed that constrains the circumstances under which a process gets its turn infinitely often. Unconditional fairness is sometimes referred to as impartiality.

Strong fairness means that if an activity is infinitely often enabled—but not necessarily always, i.e., there may be finite periods during which the activity is not enabled—then it will be executed infinitely often. An execution fragment is strongly fair with respect to activity α if it is not the case that α is infinitely often enabled without being taken beyond a certain point. Strong fairness is sometimes referred to as compassion.

Weak fairness means that if an activity, e.g., a transition in a process or an entire process itself, is continuously enabled—no periods are allowed in which the activity is not

enabled—then it has to be executed infinitely often. An execution fragment is weakly fair with respect to some activity, α say, if it is not the case that α is always enabled beyond some point without being taken beyond this point. Weak fairness is sometimes referred to as justice.

How to express these fairness constraints? There are different ways to formulate fairness requirements. In the sequel, we adopt the action-based view and define strong fairness for (sets of) actions. (In Chapter 5, also state-based notions of fairness will be introduced and the relationship between action-based and state-based fairness is studied in detail.) Let A be a set of actions. The execution fragment ρ is said to be strongly A-fair if the actions in A are not continuously ignored under the circumstance that they can be executed infinitely often. ρ is unconditionally A-fair if some action in A is infinitely often executed in ρ. Weak fairness is defined in a similar way as strong fairness (see below).

In order to formulate these fairness notions formally, the following auxiliary notion is convenient. For state s, let $Act(s)$ denote the set of actions that are executable in state s, that is,

$$Act(s) \;=\; \{\alpha \in Act \mid \exists s' \in S.\, s \xrightarrow{\alpha} s' \;\}.$$

Definition 3.43. Unconditional, Strong, and Weak Fairness

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states, $A \subseteq Act$, and infinite execution fragment $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of TS:

1. ρ is *unconditionally A-fair* whenever $\overset{\infty}{\exists}\, j.\ \alpha_j \in A$.
2. ρ is *strongly A-fair* whenever
$$\Big(\overset{\infty}{\exists}\, j.\ Act(s_j) \cap A \neq \varnothing\Big) \;\Longrightarrow\; \Big(\overset{\infty}{\exists}\, j.\ \alpha_j \in A\Big).$$
3. ρ is *weakly A-fair* whenever
$$\Big(\overset{\infty}{\forall}\, j.\ Act(s_j) \cap A \neq \varnothing\Big) \;\Longrightarrow\; \Big(\overset{\infty}{\exists}\, j.\ \alpha_j \in A\Big).$$

■

Here, $\overset{\infty}{\exists}\, j$ stands for "there are infinitely many j" and $\overset{\infty}{\forall}\, j$ for "for nearly all j" in the sense of "for all, except for finitely many j". The variable j, of course, ranges over the natural numbers.

To check whether a run is unconditionally A-fair it suffices to consider the actions that occur along the execution, i.e., it is not necessary to check which actions in A are enabled in visited states. However, in order to decide whether a given execution is strongly or weakly A-fair, it does not suffice to only consider the actions actually occurring in the execution. Instead, also the enabled actions in all visited states need to be considered. These enabled actions are possible in the visited states, but do not necessarily have to be taken along the considered execution.

Example 3.44. A Simple Shared-Variable Concurrent Program

Consider the following two processes that run in parallel and share an integer variable x that initially has value 0:

$$\begin{aligned} \textbf{proc}\ \mathsf{Inc} &= \textbf{while}\ \langle\, x \geqslant 0\ \textbf{do}\ x := x + 1 \,\rangle\ \textbf{od} \\ \textbf{proc}\ \mathsf{Reset} &= x := -1 \end{aligned}$$

The pair of brackets $\langle \ldots \rangle$ embraces an atomic section, i.e., process Inc performs the check whether x is positive and the increment of x (if the guard holds) as one atomic step. Does this parallel program terminate? When no fairness constraints are imposed, it is possible that process Inc is permanently executing, i.e., process Reset never gets its turn, and the assignment $x = -1$ is not executed. In this case, termination is thus not guaranteed, and the property is refuted. If, however, we require unconditional process fairness, then every process gets its turn, and termination is guaranteed. ■

An important question now is: given a verification problem, which fairness notion to use? Unfortunately, there is no clear answer to this question. Different forms of fairness do exist—the above is just a small, though important, fragment of all possible fairness notions—and there is no single favorite notion. For verification purposes, fairness constraints are crucial, though. Recall that the purpose of fairness constraints is to rule out certain "unreasonable" computations. If the fairness constraint is too strong, relevant computations may not be considered. In case a property is satisfied (for a transition system), it might well be the case that some reasonable computation that is not considered (as it is ruled out by the fairness constraint) refutes this property. On the other hand, if the fairness constraint is too weak, we may fail to prove a certain property as some unreasonable computations (that are not ruled out) refute it.

The relationship between the different fairness notions is as follows. Each unconditionally A-fair execution fragment is strongly A-fair, and each strongly A-fair execution fragment is weakly A-fair. In general, the reverse direction does not hold. For instance, an execution fragment that solely visits states in which no A-actions are possible is strongly A-fair (as the premise of strong A-fairness does not hold), but not unconditionally A-fair. Besides,

an execution fragment that only visits finitely many states in which some A-actions are enabled but never executes an A-action is weakly A-fair (as the premise of weak A-fairness does not hold), but not strongly A-fair. Summarizing, we have

$$\text{unconditional } A\text{-fairness} \implies \text{strong } A\text{-fairness} \implies \text{weak } A\text{-fairness}$$

where the reverse implication in general does not hold.

Example 3.45. Fair Execution Fragments

Consider the transition system TS_{Sem} for the semaphore-based mutual exclusion solution. We label the transitions with the actions req_i, $enter_i$ (for i=1, 2), and *rel* in the obvious way, see Figure 3.12.

In the execution fragment

$$\langle n_1, n_2, y=1\rangle \xrightarrow{req_1} \langle w_1, n_2, y=1\rangle \xrightarrow{enter_1} \langle c_1, n_2, y=0\rangle \xrightarrow{rel} \langle n_1, n_2, y=1\rangle \xrightarrow{req_1} \ldots$$

only the first process gets its turn. This execution fragment is indicated by the dashed arrows in Figure 3.12. It is *not* unconditionally fair for the set of actions

$$A = \{\, enter_2 \,\}.$$

It is, however, strongly A-fair, since no state is visited in which the action $enter_2$ is executable, and hence the premise of strong fairness is vacuously false. In the alternative execution fragment

$$\langle n_1, n_2, y=1\rangle \xrightarrow{req_2} \langle n_1, w_2, y=1\rangle \xrightarrow{req_1} \langle w_1, w_2, y=1\rangle \xrightarrow{enter_1}$$
$$\langle c_1, w_2, y=0\rangle \xrightarrow{rel} \langle n_1, w_2, y=1\rangle \xrightarrow{req_1} \ldots$$

the second process requests to enter its critical section but is ignored forever. This execution fragment is indicated by the dotted arrows in Figure 3.12. It is not strongly A-fair: although the action $enter_2$ is infinitely often enabled (viz. every time when visiting the state $\langle w_1, w_2, y = 1\rangle$ or $\langle n_1, w_2, y = 1\rangle$), it is never taken. It is, however, weakly A-fair, since the action $enter_2$ is not continuously enabled—it is not enabled in the state $\langle c_1, w_2, y = 0\rangle$. ■

A fairness constraint imposes a requirement on all actions in a set A. In order to enable different fairness constraints to be imposed on different, possibly nondisjoint, sets of actions, fairness *assumptions* are used. A fairness assumption for a transition system may require different notions of fairness with respect to several sets of actions.

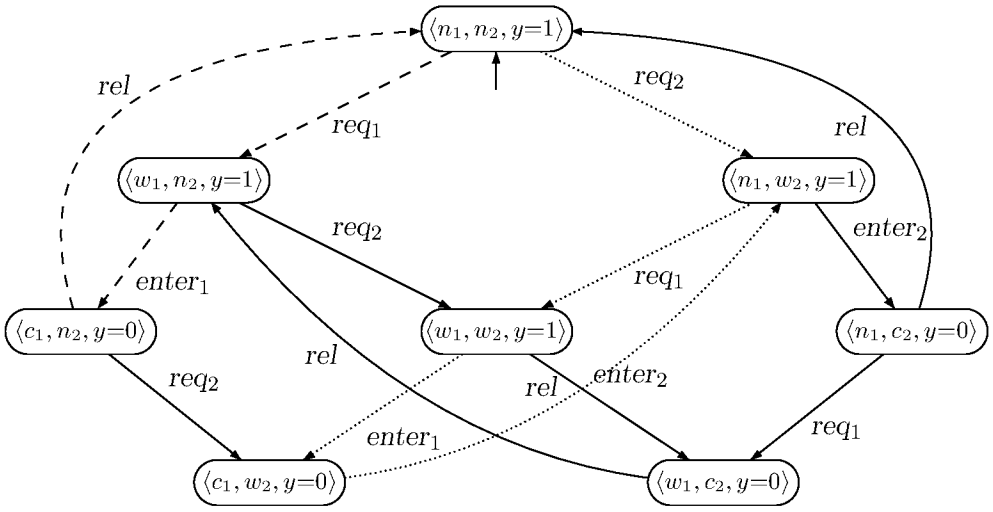


Figure 3.12: Two examples of fair execution fragments of the semaphore-based mutual exclusion algorithm.

Definition 3.46. Fairness Assumption

A *fairness assumption* for Act is a triple

$$\mathcal{F} \;=\; (\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak})$$

with $\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak} \subseteq 2^{Act}$. Execution ρ is $\mathcal{F}$-fair if

- it is unconditionally A-fair for all $A \in \mathcal{F}_{ucond}$,
- it is strongly A-fair for all $A \in \mathcal{F}_{strong}$, and
- it is weakly A-fair for all $A \in \mathcal{F}_{weak}$.

If the set $\mathcal{F}$ is clear from the context, we use the term fair instead of $\mathcal{F}$-fair. ■

Intuitively speaking, a fairness assumption is a triple of sets of (typically different) action sets, one such set of action sets is treated in a strongly fair manner, one in a weakly fair manner, and one in an unconditionally fair way. This is a rather general definition that allows imposing different fairness constraints on different sets of actions. Quite often, only a single type of fairness constraint suffices. In the sequel, we use the casual notations for these fairness assumptions. For $\mathcal{F} \subseteq 2^{Act}$, a strong fairness assumption denotes the fairness assumption $(\varnothing, \mathcal{F}, \varnothing)$. Weak, and unconditional fairness assumptions are used in a similar way.

The notion of $\mathcal{F}$-fairness as defined on execution fragments is lifted to traces and paths in the obvious way. An infinite trace σ is $\mathcal{F}$-fair if there is an $\mathcal{F}$-fair execution ρ with $trace(\rho) = \sigma$. $\mathcal{F}$-fair (infinite) path fragments and $\mathcal{F}$-fair paths are defined analogously.

Let $FairPaths_{\mathcal{F}}(s)$ denote the set of $\mathcal{F}$-paths of s (i.e., infinite $\mathcal{F}$-fair path fragments that start in state s), and $FairPaths_{\mathcal{F}}(TS)$ the set of $\mathcal{F}$-fair paths that start in some initial state of TS. Let $FairTraces_{\mathcal{F}}(s)$ denote the set of $\mathcal{F}$-fair traces of s, and $FairTraces_{\mathcal{F}}(TS)$ the set of $\mathcal{F}$-fair traces of the initial states of transition system TS:

$$\begin{aligned} FairTraces_{\mathcal{F}}(s) &= trace(FairPaths_{\mathcal{F}}(s)) \quad \text{and} \\ FairTraces_{\mathcal{F}}(TS) &= \bigcup_{s \in I} FairTraces_{\mathcal{F}}(s). \end{aligned}$$

Note that it does not make much sense to define these notions for finite traces as any finite trace is fair by default.

Example 3.47. Mutual Exclusion Again

Consider the following fairness requirement for two-process mutual exclusion algorithms:

"process P_i acquires access to its critical section infinitely often"

for any $i \in \{1, 2\}$. What kind of fairness assumption is appropriate to achieve this? Assume each process P_i has three states n_i (noncritical), w_i (waiting), and c_i (critical). As before, the actions req_i, $enter_i$, and rel are used to model the request to enter the critical section, the entering itself, and the release of the critical section. The strong-fairness assumption

$$\{\, \{enter_1, enter_2\} \,\}$$

ensures that one of the actions $enter_1$ or $enter_2$, is executed infinitely often. A behavior in which one of the processes gets access to the critical section infinitely often while the other gets access only finitely many times is strongly fair with respect to this assumption. This is, however, not intended. The strong-fairness assumption

$$\{\, \{\, enter_1 \,\}, \{\, enter_2 \,\} \,\}$$

indeed realizes the above requirement. This assumption should be viewed as a requirement on how to resolve the contention when both processes are awaiting to get access to the critical section. ■

Fairness assumptions can be *verifiable* properties whenever all infinite execution fragments are fair. For example, it can be verified that the transition system for Peterson's algorithm

satisfies the strong-fairness assumption

$$\mathcal{F}_{strong} \;=\; \{\, \{\, enter_1 \,\}, \{\, enter_2 \,\} \,\}.$$

But in many cases it is necessary to *assume* the validity of the fairness conditions to verify liveness properties.

A transition system TS satisfies the LT property P under fairness assumption $\mathcal{F}$ if all $\mathcal{F}$-fair paths fulfill the property P. However, no requirements whatsoever are imposed on the unfair paths. This is formalized as follows.

Definition 3.48. Fair Satisfaction Relation for LT Properties

Let P be an LT property over AP and $\mathcal{F}$ a fairness assumption over Act. Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ *fairly satisfies* P, notation $TS \models_{\mathcal{F}} P$, if and only if $FairTraces_{\mathcal{F}}(TS) \subseteq P$. ■

For a transition system that satisfies the fairness assumption $\mathcal{F}$ (i.e., *all* paths are $\mathcal{F}$-fair), the satisfaction relation $\models$ without fairness assumptions (see Definition 3.11, page 100) corresponds with the fair satisfaction relation $\models_{\mathcal{F}}$. In this case, the fairness assumption does not rule out any trace. However, in case a transition system has traces that are not $\mathcal{F}$-fair, then in general we are confronted with a situation

$$TS \;\models_{\mathcal{F}}\; P \quad \text{whereas} \quad TS \;\not\models\; P.$$

By restricting the validity of a property to the set of fair paths, the verification can be restricted to "realistic" executions.

Before turning to some examples, a few words on the relationship between unconditional, strong, and weak fairness are (again) in order. As indicated before, we have that the set of unconditional A-fair executions is a subset of all strong A-fair executions. In a similar way, the latter set of executions is a subset of all weak A-fair executions. Stated differently, unconditional fairness rules out more behaviors than strong fairness, and strong excludes more behaviors than weak fairness. For $\mathcal{F} = \{\, A_1, \ldots, A_k \,\}$, let fairness assumption $\mathcal{F}_{ucond} = (\mathcal{F}, \varnothing, \varnothing)$, $\mathcal{F}_{strong} = (\varnothing, \mathcal{F}, \varnothing)$, and $\mathcal{F}_{weak} = (\varnothing, \varnothing, \mathcal{F})$. Then for any transition system TS and LT property P it follows that:

$$TS \models_{\mathcal{F}_{weak}} P \;\Rightarrow\; TS \models_{\mathcal{F}_{strong}} P \;\Rightarrow\; TS \models_{\mathcal{F}_{ucond}} P.$$

Example 3.49. Independent Traffic Lights

Consider again the independent traffic lights. Let action *switch2green* denote the switching to green. Similarly *switch2red* denotes the switching to red. The fairness assumption

$$\mathcal{F} \;=\; \{\,\{\,switch2green_1, switch2red_1\,\}, \{\,switch2green_2, switch2red_2\,\}\,\}$$

expresses that both traffic lights infinitely often switch color. In this case, it is irrelevant whether strong, weak, or unconditional fairness is required.

Note that in this example $\mathcal{F}$ is *not* a verifiable system property (as it is not guaranteed to hold), but a natural property which is satisfied for a practical implementation of the system (with two independent processors). Obviously,

$$TrLight_1 \;|||\; TrLight_2 \;\models_{\mathcal{F}}\; \text{“each traffic light is green infinitely often”}$$

while the corresponding proposition for the nonfair relation $\models$ is refuted. ■

Example 3.50. Fairness for Mutual Exclusion Algorithms

Consider again the semaphore-based mutual exclusion algorithm, and assume the fairness assumption $\mathcal{F}$ consists of

$$\mathcal{F}_{weak} \;=\; \{\{\,req_1\,\}, \{\,req_2\,\}\} \quad \text{and} \quad \mathcal{F}_{strong} \;=\; \{\{\,enter_1\,\}, \{\,enter_2\,\}\}$$

and $\mathcal{F}_{ucond} = \varnothing$. The strong fairness constraint requires each process to enter its critical section infinitely often when it infinitely often gets the opportunity to do so. This does not forbid a process to never leave its noncritical section. To avoid this unrealistic scenario, the weak fairness constraint requires that any process infinitely often requests to enter the critical section. In order to do so, each process has to leave the noncritical section infinitely often. It follows that $TS_{Sem} \models_{\mathcal{F}} P$ where P stands for the property “every process enters its critical section infinitely often”.

Weak fairness is sufficient for request actions, as such actions are not critical: if req_i is executable in (global) state s, then it is executable in all direct successor states of s that are reached by an action that differs from req_i.

Peterson’s algorithm satisfies the strong fairness property

“Every process that requests access to the critical section
will eventually be able to do so”.

We can, however, not ensure that a process will ever leave its noncritical section and request the critical section. That is, the property P is refuted. This can be “repaired” by imposing the weak fairness constraint $\mathcal{F}_{weak} \;=\; \{\,\{\,req_1\,\}, \{\,req_2\,\}\,\}$. We now have $TS_{Pet} \models_{\mathcal{F}_{weak}} P$. ■

3.5.2 Fairness Strategies

The examples in the previous section indicate that fairness assumptions may be necessary to verify liveness properties of transition system TS. In order to rule out the "unrealistic" computations, fairness assumptions are imposed on the traces of TS, and it is checked whether $TS \models_{\mathcal{F}} P$ as opposed to checking $TS \models P$ (without fairness). Which fairness assumptions are appropriate to check P? Many model-checking tools provide the possibility to work with built-in fairness assumptions. Roughly speaking, the intention is to rule out executions that cannot occur in a realistic implementation. But what does that exactly mean? In order to give some insight into this, we consider several fairness assumptions for synchronizing concurrent systems. The aim is to establish a fair communication mechanism between the various processes involved. A rule of thumb is: Strong fairness is needed to obtain an adequate resolution of contentions (between processes), while weak fairness suffices for sets of actions that represent the concurrent execution of independent actions (i.e., interleaving).

For modeling *asynchronous* concurrency by means of transition systems, the following rule of thumb can be adopted:

concurrency = interleaving (i.e., nondeterminism) + fairness

Example 3.51. Fair Concurrency with Synchronization

Consider the concurrent transition system:

$$TS = TS_1 \parallel TS_2 \parallel \ldots \parallel TS_n \quad ,$$

where $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, for $1 \leqslant i \leqslant n$, is a transition system without terminal states. Recall that each pair of processes TS_i and TS_j (for $i \neq j$) has to synchronize on their common sets of actions, i.e., $Syn_{i,j} = Act_i \cap Act_j$. It is assumed that $Syn_{i,j} \cap Act_k = \varnothing$ for any $k \neq i, j$. For simplicity, it is assumed that TS has no terminal states. (In case there are terminal states, each finite execution is considered to be fair.)

We consider several fairness assumptions on the transition system TS. First, consider the strong fairness assumption

$$\{Act_1, Act_2, \ldots, Act_n\}$$

which ensures that each transition system TS_i executes an action infinitely often, provided the composite system TS is infinitely often in a (global) state with a transition being executable in which TS_i participates. This fairness assumption, however, cannot ensure that a communication will ever occur—it is possible for each TS_i to only execute local actions ad infinitum.

In order to force a synchronization to take place every now and then, the strong fairness assumption

$$\{\,\{\,\alpha\,\} \mid \alpha \in Syn_{i,j}, 0 < i < j \leqslant n\,\} \tag{3.1}$$

could be imposed. It forces every synchronization action to happen infinitely often. Alternatively, a somewhat weaker fairness assumption can be imposed by requiring every pair of processes to synchronize—regardless of the synchronization action—infinitely often. The corresponding strong fairness assumption is

$$\{\,Syn_{i,j} \mid 0 < i < j \leqslant n\,\}. \tag{3.2}$$

Whereas (3.2) allows processes to always synchronize on the same action, (3.1) does not permit this. The strong fairness assumption:

$$\{\bigcup_{0<i<j\leqslant n} Syn_{i,j}\,\}$$

goes even one step further as it only requires a synchronization to take place infinitely often, regardless of the process involved. This fairness assumption does not rule out executions in which always the same synchronization takes place or in which always the same pair of processes synchronizes.

Note that all fairness assumptions in this example so far are strong. This requires that infinitely often a synchronization is enabled. As the constituting transition systems TS_i may execute internal actions, synchronizations are not continuously enabled, and hence weak fairness is in general inappropriate.

If the internal actions should be fairly considered, too, then we may use, e.g., the strong fairness assumption

$$\{\,Act_1 \setminus Syn_1, \ldots, Act_n \setminus Syn_n\,\} \;\cup\; \{\,\{\,\alpha\,\} \mid \alpha \in Syn\,\},$$

where $Syn_i = \bigcup_{j\neq i} Syn_{i,j}$ denotes the set of all synchronization actions of TS_i and $Syn = \bigcup_i Syn_i$.

Under the assumption that in every (local) state either only internal actions or only synchronization actions are executable, it suffices to impose the *weak* fairness constraint

$$\{\,Act_1 \setminus Syn_1, \ldots, Act_n \setminus Syn_n\,\}.$$

Weak fairness is appropriate for the internal actions $\alpha \in Act_i \setminus Syn_i$, as the ability to perform an internal action is preserved until it will be executed. ■

As an example of another form of fairness we consider the following sequential hardware circuit.

Example 3.52. Circuit Fairness

For sequential circuits we have modeled the environmental behavior, which provides the input bits, by means of nondeterminism. It may be necessary to impose fairness assumptions on the environment in order to be able to verify liveness properties, such as "the values 0 and 1 are output infinitely often". Let us illustrate this by means of a concrete example. Consider a sequential circuit with input variable x, output variable y, and register r. Let the transition function and the output function be defined as

$$\lambda_y \;=\; \delta_r = \; x \leftrightarrow \neg r.$$

That is, the circuit inverts the register and output evaluation if and only if the input bit is set. If x=0, then the register evaluation remains unchanged. The value of the register is output. Suppose all transitions leading to a state with a register evaluation of the form $[r = 1, \ldots]$ are labeled with the action *set*. Imposing the unconditional fairness assumption $\{\{\, set \,\}\}$ ensures that the values 0 and 1 are output infinitely often. ■

3.5.3 Fairness and Safety

While fairness assumptions may be necessary to verify liveness properties, they are irrelevant for verifying safety properties, provided that they can always be ensured by means of an appropriate scheduling strategy. Such fairness assumptions are called *realizable* fairness assumptions. A fairness assumption cannot be realized in a transition system whenever there exists a reachable state from where *no* fair path begins. In this case, it is impossible to design a scheduler that resolves the nondeterminism such that only fair paths remain.

Example 3.53. A Nonrealizable Fairness Assumption

Consider the transition system depicted in Figure 3.13, and suppose the unconditional fairness assumption $\{\{\, \alpha \,\}\}$ is imposed. As the α-transition can only be taken once, it is evident that the transition system can never guarantee this form of fairness. As there is a reachable state from which no unconditional fair path exists, this fairness assumption is nonrealizable. ■

Definition 3.54. Realizable Fairness Assumption

Let TS be a transition system with the set of actions Act and $\mathcal{F}$ a fairness assumption for Act. $\mathcal{F}$ is called *realizable* for TS if for every reachable state s: $FairPaths_{\mathcal{F}}(s) \neq \varnothing$. ■

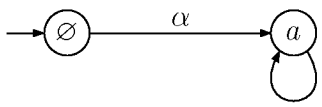


Figure 3.13: Unconditional fairness.

Stated in words, a fairness assumption is realizable in a transition system TS whenever in any reachable state at least one fair execution is possible. This entails that every initial finite execution fragment of TS can be completed to a fair execution. Note that there is no requirement on the unreachable states.

The following theorem shows the irrelevance of realizable fairness assumptions for the verification of safety properties. The *suffix property* of fairness assumptions is essential for its proof. This means the following. If

$$\rho \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

is an (infinite) execution fragment, then ρ is fair if and only if every suffix

$$s_j \xrightarrow{\alpha_{j+1}} s_{j+1} \xrightarrow{\alpha_{j+2}} s_{j+2} \xrightarrow{\alpha_{j+3}} \ldots$$

of ρ is fair too. Conversely, every fair execution fragment ρ (as above) starting in state s_0 can be preceded by an arbitrary finite execution fragment

$$s'_0 \xrightarrow{\beta_1} s'_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s'_n = s_0$$

ending in s_0. Proceeding in s_0 by execution ρ yields the fair execution fragment:

$$\underbrace{s'_0 \xrightarrow{\beta_1} s'_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s'_n}_{\text{arbitrary starting fragment}} = \underbrace{s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots}_{\text{fair continuation}}$$

Theorem 3.55. Realizable Fairness is Irrelevant for Safety Properties

Let TS be a transition system with set of propositions AP, $\mathcal{F}$ a realizable fairness assumption for TS, and P_{safe} a safety property over AP. Then:

$$TS \models P_{safe} \quad \textit{if and only if} \quad TS \models_{\mathcal{F}} P_{safe}.$$

Proof: "⇒": Assume $TS \models P_{safe}$. Then, by definition of $\models$ and the fact that the fair traces of TS are a subset of the traces of TS, we have

$$\mathit{FairTraces}_{\mathcal{F}}(TS) \;\subseteq\; \mathit{Traces}(TS) \;\subseteq\; P_{safe}.$$

Thus, by definition of $\models_{\mathcal{F}}$ it follows that $TS \models_{\mathcal{F}} P_{safe}$.

"$\Leftarrow$": Assume $TS \models_{\mathcal{F}} P_{safe}$. It is to be shown $TS \models P_{safe}$, i.e., $\mathit{Traces}(TS) \subseteq P_{safe}$. This is done by contraposition. Let $\sigma \in \mathit{Traces}(TS)$ and assume $\sigma \notin P_{safe}$. As $\sigma \notin P_{safe}$, there is a bad prefix of σ, $\widehat{\sigma}$ say, for P_{safe}. Hence, the set of properties that has $\widehat{\sigma}$ as a prefix, i.e.,

$$P \;=\; \Big\{\sigma' \in \Big(2^{AP}\Big)^{\omega} \;\mid\; \widehat{\sigma} \in \mathit{pref}(\sigma')\Big\},$$

satisfies $P \cap P_{safe} = \varnothing$. Further, let $\widehat{\pi} = s_0\, s_1 \ldots s_n$ be a finite path fragment of TS with

$$\mathit{trace}(\widehat{\pi}) \;=\; \widehat{\sigma}.$$

Since $\mathcal{F}$ is a realizable fairness assumption for TS and $s_n \in \mathit{Reach}(TS)$, there is an $\mathcal{F}$-fair path starting in s_n. Let

$$s_n\, s_{n+1}\, s_{n+2} \ldots \in \mathit{FairPaths}_{\mathcal{F}}(s_n).$$

The path $\pi = s_0 \ldots s_n\, s_{n+1}\, s_{n+2} \ldots$ is in $\mathit{FairPaths}_{\mathcal{F}}(TS)$ and thus,

$$\mathit{trace}(\pi) \;=\; L(s_0) \ldots L(s_n)\, L(s_{n+1})\, L(s_{n+2}) \ldots \;\in\; \mathit{FairTraces}_{\mathcal{F}}(TS) \;\subseteq\; P_{safe}.$$

On the other hand, $\widehat{\sigma} = L(s_0) \ldots L(s_n)$ is a prefix of $\mathit{trace}(\pi)$. Thus, $\mathit{trace}(\pi) \in P$. This contradicts $P \cap P_{safe} = \varnothing$. ■

Theorem 3.55 does not hold if arbitrary (i.e., possibly nonrealizable) fairness assumptions are permitted. This is illustrated by the following example.

Example 3.56. Nonrealizable Fairness may harm Safety Properties

Consider the transition system TS in Figure 3.14 and suppose the unconditional fairness assumption $\mathcal{F} = \{\{\alpha\}\}$ is imposed. $\mathcal{F}$ is not realizable for TS, as the noninitial state (referred to as state s_1), is reachable, but has no $\mathcal{F}$-fair execution. Obviously, TS has only one fair path (namely the path that never leaves the initial state s_0). In contrast, paths of the form $s_0 \ldots s_0\, s_1\, s_1\, s_1 \ldots$ are not fair, since α is only executed finitely often. Accordingly, we have that

$$TS \models_{\mathcal{F}} \text{"never } a\text{"} \qquad \text{but} \qquad TS \not\models \text{"never } a\text{"}.$$

■

3.6 Summary

- The set of reachable states of a transition system TS can be determined by a search algorithm on the state graph of TS.

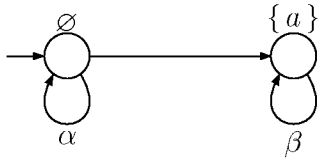


Figure 3.14: Unconditional fairness may be relevant for safety properties.

- A trace is a sequence of sets (!) of atomic propositions. The traces of a transition system TS are obtained from projecting the paths to the sequence of state labels.
- A linear-time (LT, for short) property is a set of infinite words over the alphabet 2^{AP}.
- Two transition systems are trace-equivalent (i.e., they exhibit the same traces) if and only if they satisfy the same LT properties.
- An invariant is an LT property that is purely state-based and requires a propositional logic formula Φ to hold for all reachable states. Invariants can be checked using a depth-first search where the depth-first search stack can be used to provide a counterexample in case an invariant is refuted.
- Safety properties are generalizations of invariants. They constrain the finite behaviors. The formal definition of safety properties can be provided by means of their bad prefixes in the sense that each trace that refutes a safety property has a finite prefix, the bad prefix, that causes this.
- Two transition systems exhibit the same finite traces if and only if they satisfy the same safety properties.
- A liveness property is an LT property if it does not rule out any finite behavior. It constrains infinite behavior.
- Any LT property is equivalent to an LT property that is a conjunction of a safety and a liveness property.
- Fairness assumptions serve to rule out traces that are considered to be unrealistic. They consist of unconditional, strong, and weak fairness constraints, i.e., constraints on the actions that occur along infinite executions.
- Fairness assumptions are often necessary to establish liveness properties, but they are—provided they are realizable—irrelevant for safety properties.

3.7 Bibliographic Notes

The dining philosophers example discussed in Example 3.2 has been developed by Dijkstra [128] in the early seventies to illustrate the intricacies of concurrency. Since then it has become one of the standard examples for reasoning about parallel systems.

The depth-first search algorithm that we used as a basis for the invariance checking algorithm goes back to Tarjan [387]. Further details about graph traversal algorithms can be found in any textbook on algorithms and data structures, e.g. [100], or on graph algorithms [188].

Traces. Traces have been introduced by Hoare [202] to describe the linear-time behavior of transition systems and have been used as the initial semantical model for the process algebra CSP. Trace theory has further been developed by, among others, van de Snepscheut [403] and Rem [354] and has successfully been used to design and analyze fine-grained parallel programs that occur in, e.g., asynchronous hardware circuits. Several extensions to traces and their induced equivalences have been proposed, such as failures [65] where a trace is equipped with information about which actions are rejected after execution of such trace. The FDR model checker [356] supports the automated checking of failure-divergence refinement and the checking of safety properties. A comprehensive survey of these refined notions of trace equivalence and trace inclusion has recently been given by Bruda [68].

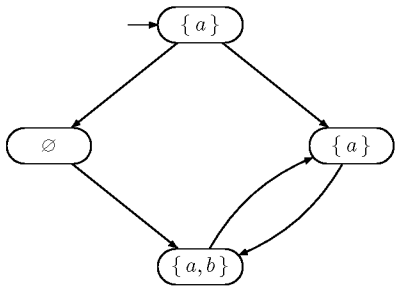
Safety and liveness. The specification of linear-time properties using sets of infinite sequences of states (and their topological characterizations) goes back to Alpern and Schneider [5, 6, 7]. An earlier approach by Gerth [164] was based on finite sequences. Lamport [257] categorized properties as either safety, liveness, or properties that are neither. Alternative characterizations have been provided by Rem [355] and Gumm [178]. Subclasses of liveness and safety properties in the linear-time framework have been identified by Sistla [371], and Chang, Manna, and Pnueli [80]. Other definitions of liveness properties have been provided by Dederichs and Weber [119] and Naumovich and Clarke [312] (for linear-time properties), and Manolios and Trefler [285, 286] (for branching-time properties). A survey of safety and liveness has been given by Kindler [239].

Fairness. Fairness has implicitly been introduced by Dijkstra [126, 127] by assuming that one should abstract from the speed of processors and that each process gets its turn once it is initiated. Park [321] studied the notion of fairness in providing a semantics to data-flow languages. Weak and strong fairness have been introduced by Lehmann, Pnueli, and Stavi [267] in the context of shared variable concurrent programs. Queille and Sifakis [348] consider fairness for transition systems. An overview of the fairness notions has been provided by Kwiatkowska [252]. An extensive treatment of fairness can be found

in the monograph by Francez [155]. A recent characterization of fairness in terms of topology, language theory, and game theory has been provided by Völzer, Varacca, and Kindler [415].

3.8 Exercises

EXERCISE 3.1. Give the traces on the set of atomic propositions $\{ a, b \}$ of the following transition system:



EXERCISE 3.2. On page 97, a transformation is described of a transition system TS with possible terminal states into an "equivalent" transition system TS^* without terminal states. Questions:

(a) Give a formal definition of this transformation $TS \mapsto TS^*$

(b) Prove that the transformation preserves trace-equivalence, i.e., show that if TS_1, TS_2 are transition systems (possibly with terminal states) such that $\mathit{Traces}(TS_1) = \mathit{Traces}(TS_2)$, then $\mathit{Traces}(TS_1^*) = \mathit{Traces}(TS_2^*)$.[8]

EXERCISE 3.3. Give an algorithm (in pseudocode) for invariant checking such that in case the invariant is refuted, a *minimal* counterexample, i.e., a counterexample of minimal length, is provided as an error indication.

EXERCISE 3.4. Recall the definition of AP-deterministic transition systems (Definition 2.5 on page 24). Let TS and TS' be transition systems with the same set of atomic propositions AP. Prove the following relationship between trace inclusion and finite trace inclusion:

(a) For AP-deterministic TS and TS':

$$\mathit{Traces}(TS) = \mathit{Traces}(TS') \text{ if and only if } \mathit{Traces}_{\mathit{fin}}(TS) = \mathit{Traces}_{\mathit{fin}}(TS').$$

[8] If TS is a transition system with terminal states, then $\mathit{Traces}(TS)$ is defined as the set of all words $\mathit{trace}(\pi)$ where π is an initial, maximal path fragment in TS.

(b) Give concrete examples of TS and TS' where at least one of the transition systems is not AP-deterministic, but

$$\mathit{Traces}(TS) \not\subseteq \mathit{Traces}(TS') \quad \text{and} \quad \mathit{Traces}_{\mathit{fin}}(TS) = \mathit{Traces}_{\mathit{fin}}(TS').$$

EXERCISE 3.5. Consider the set AP of atomic propositions defined by $AP = \{ x = 0, x > 1 \}$ and consider a nonterminating sequential computer program P that manipulates the variable x. Formulate the following informally stated properties as LT properties:

(a) false

(b) initially x is equal to zero

(c) initially x differs from zero

(d) initially x is equal to zero, but at some point x exceeds one

(e) x exceeds one only finitely many times

(f) x exceeds one infinitely often

(g) the value of x alternates between zero and two

(h) true

(This exercise has been adopted from [355].) Determine which of the provided LT properties are safety properties. Justify your answers.

EXERCISE 3.6. Consider the set $AP = \{ A, B \}$ of atomic propositions. Formulate the following properties as LT properties and characterize each of them as being either an invariance, safety property, or liveness property, or none of these.

(a) A should never occur,

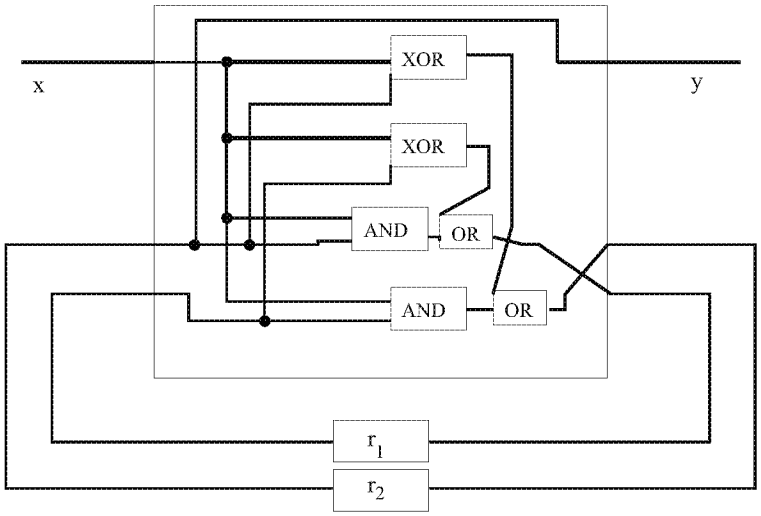
(b) A should occur exactly once,

(c) A and B alternate infinitely often,

(d) A should eventually be followed by B.

(This exercise has been inspired by [312].)

EXERCISE 3.7. Consider the following sequential hardware circuit:



The circuit has input variable x, output variable y, and registers r_1 and r_2 with initial values $r_1 = 0$ and $r_2 = 1$. The set AP of atomic propositions equals $\{ x, r_1, r_2, y \}$. Besides, consider the following informally formulated LT properties over AP:

P_1 : Whenever the input x is continuously high (i.e., x=1), then the output y is infinitely often high.

P_2 : Whenever currently r_2=0, then it will never be the case that after the next input, r_1=1.

P_3 : It is never the case that two successive outputs are high.

P_4 : The configuration with x=1 and r_1=0 never occurs.

Questions:

(a) Give for each of these properties an example of an infinite word that belongs to P_i. Do the same for the property $(2^{AP})^{\omega} \setminus P_i$, i.e., the complement of P_i.

(b) Determine which properties are satisfied by the hardware circuit that is given above.

(c) Determine which of the properties are safety properties. Indicate which properties are invariants.

  (i) For each safety property P_i, determine the (regular) language of bad prefixes.

  (ii) For each invariant, provide the propositional logic formula that specifies the property that should be fulfilled by each state.

EXERCISE 3.8. Let LT properties P and P' be equivalent, notation $P \cong P'$, if and only if *pref*(P) = *pref*(P'). Prove or disprove: $P \cong P'$ if and only if *closure*(P) = *closure*(P').

EXERCISE 3.9. Show that for any transition system TS, the set $\mathit{closure}(\mathit{Traces}(TS))$ is a safety property such that $TS \models \mathit{closure}(\mathit{Traces}(TS))$.

EXERCISE 3.10. Let P be an LT property. Prove: $\mathit{pref}(\mathit{closure}(P)) = \mathit{pref}(P)$.

EXERCISE 3.11. Let P and P' be liveness properties over AP. Prove or disprove the following claims:

(a) $P \cup P'$ is a liveness property,

(b) $P \cap P'$ is a liveness property.

Answer the same question for P and P' being safety properties.

EXERCISE 3.12. Prove Lemma 3.38 on page 125.

EXERCISE 3.13. Let $AP = \{ a, b \}$ and let P be the LT property of all infinite words $\sigma = A_0 A_1 A_2 \ldots \in \left(2^{AP}\right)^{\omega}$ such that there exists $n \geqslant 0$ with $a \in A_i$ for $0 \leqslant i < n$, $\{ a, b \} = A_n$ and $b \in A_j$ for infinitely many $j \geqslant 0$. Provide a decomposition $P = P_{safe} \cap P_{live}$ into a safety and a liveness property.

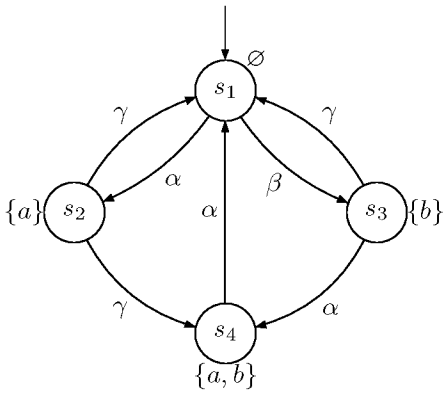
EXERCISE 3.14. Let TS_{Sem} and TS_{Pet} be the transition systems for the semaphore-based mutual exclusion algorithm (Example 2.24 on page 43) and Peterson's algorithm (Example 2.25 on page 45), respectively. Let $AP = \{ \mathit{wait}_i, \mathit{crit}_i \mid i = 1, 2 \}$. Prove or disprove:

$$\mathit{Traces}(TS_{Sem}) = \mathit{Traces}(TS_{Pet}).$$

If the property does not hold, provide an example trace of one transition system that is not a trace of the other one.

EXERCISE 3.15. Consider the transition system TS outlined on the right and the sets of actions $B_1 = \{ \alpha \}$, $B_2 = \{ \alpha, \beta \}$, and $B_3 = \{ \beta \}$. Further, let E_b, E_a and E' be the following LT properties:

- E_b = the set of all words $A_0 A_1 \cdots \in \left(2^{\{a,b\}}\right)^\omega$ with $A_i \in \{\{a,b\},\{b\}\}$ for infinitely many i (i.e., infinitely often b).
- E_a = the set of all words $A_0 A_1 \cdots \in \left(2^{\{a,b\}}\right)^\omega$ with $A_i \in \{\{a,b\},\{a\}\}$ for infinitely many i (i.e., infinitely often a).
- E' = set of all words $A_0 A_1 \cdots \in \left(2^{\{a,b\}}\right)^\omega$ for which there does not exist an $i \in \mathbb{N}$ s.t. $A_i = \{a\}$, $A_{i+1} = \{a,b\}$ and $A_{i+2} = \varnothing$.



Questions:

(a) For which sets of actions B_i ($i \in \{1,2,3\}$) and LT properties $E \in \{E_a, E_b, E'\}$ it holds that $TS \models_{\mathcal{F}_i} E$? Here, $\mathcal{F}_i$ is a strong fairness condition with respect to B_i that does not impose any unconditional or weak fairness conditions (i.e., $\mathcal{F}_i = (\varnothing, \{B_i\}, \varnothing)$).

(b) Answer the same question in the case of weak fairness (instead of strong fairness, i.e., $\mathcal{F}_i = (\varnothing, \varnothing, \{B_i\})$).

EXERCISE 3.16. Let TS_i (for i=1, 2) be the transition system $(S_i, Act, \rightarrow_i, I_i, AP_i, L_i)$ and $\mathcal{F} = (\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak})$ be a fairness assumption with $\mathcal{F}_{ucond} = \varnothing$. Prove or disprove (i.e., give a counterexample for) the following claims:

(a) $Traces(TS_1) \subseteq Traces(TS_1 \parallel TS_2)$ where $Syn \subseteq Act$

(b) $Traces(TS_1) \subseteq Traces(TS_1 \,|||\, TS_2)$

(c) $Traces(TS_1 \parallel TS_2) \subseteq Traces(TS_1)$ where $Syn \subseteq Act$

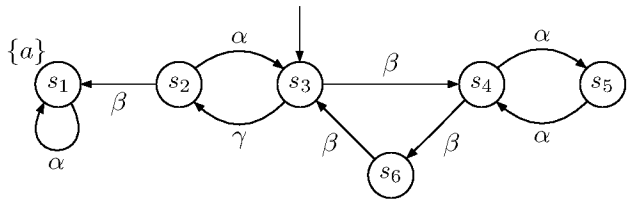
(d) $Traces(TS_1) \subseteq Traces(TS_2) \;\Rightarrow\; FairTraces_{\mathcal{F}}(TS_1) \subseteq FairTraces_{\mathcal{F}}(TS_2)$

(e) For liveness property P with $TS_2 \models_{\mathcal{F}} P$ we have

$$Traces(TS_1) \subseteq Traces(TS_2) \quad \Rightarrow \quad TS_1 \models_{\mathcal{F}} P.$$

Assume that in items (a) through (c), we have $AP_2 = \varnothing$ and that $TS_1 \parallel TS_2$ and $TS_1 \,|||\, TS_2$, respectively, have $AP = AP_1$ as atomic propositions and $L(\langle s, s' \rangle) = L_1(s)$ as labeling function. In items (d) and (e) you may assume that $AP_1 = AP_2$.

EXERCISE 3.17. Consider the following transition system TS with the set of atomic propositions $\{a\}$:

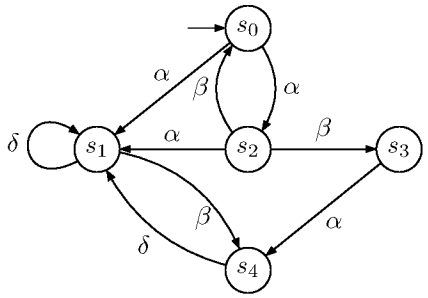


Let the fairness assumption

$$\mathcal{F} = (\varnothing, \{\{\alpha\}, \{\beta\}\}, \{\{\beta\}\}).$$

Determine whether $TS \models_{\mathcal{F}}$ "eventually a". Justify your answer!

EXERCISE 3.18. Consider the following transition system TS (without atomic propositions):



Decide which of the following fairness assumptions $\mathcal{F}_i$ are realizable for TS. Justify your answers!

(a) $\mathcal{F}_1 = (\{\{\alpha\}\}, \{\{\delta\}\}, \{\{\alpha, \beta\}\})$

(b) $\mathcal{F}_2 = (\{\{\delta, \alpha\}\}, \{\{\alpha, \beta\}\}, \{\{\delta\}\})$

(c) $\mathcal{F}_3 = (\{\{\alpha, \delta\}, \{\beta\}\}, \{\{\alpha, \beta\}\}, \{\{\delta\}\})$

EXERCISE 3.19. Let $AP = \{a, b\}$.

(a) P_1 denotes the LT property that consists of all infinite words $\sigma = A_0 A_1 A_2 \ldots \in (2^{AP})^\omega$ such that there exists $n \geqslant 0$ with

$$\forall j < n.\ A_j = \varnothing \quad \wedge \quad A_n = \{a\} \quad \wedge \quad \forall k > n.\ (A_k = \{a\} \Rightarrow A_{k+1} = \{b\}).$$

(i) Give an ω–regular expression for P_1.

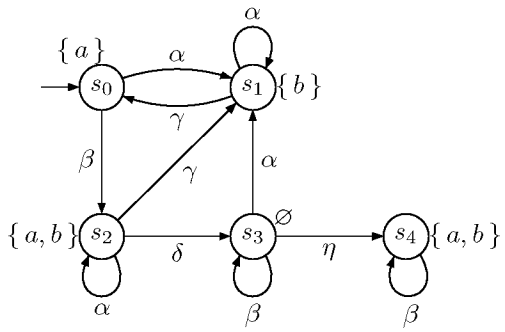
(ii) Apply the decomposition theorem and give expressions for P_{safe} and P_{live}.

(iii) Justify that P_{live} is a liveness and that P_{safe} is a safety property.

(b) Let P_2 denote the set of traces of the form $\sigma = A_0 A_1 A_2 \ldots \in \big(2^{AP}\big)^\omega$ such that

$$\overset{\infty}{\exists}\, k.\ A_k = \{\, a, b \,\} \quad \wedge \quad \exists n \geqslant 0.\ \forall k > n.\ \big(a \in A_k \Rightarrow b \in A_{k+1}\big).$$

Consider the following transition system TS:



Consider the following fairness assumptions:

(a) $\mathcal{F}_1 = \Big(\big\{\{\alpha\}\big\}, \big\{\{\beta\}, \{\delta, \gamma\}, \{\eta\}\big\}, \varnothing\Big)$. Decide whether $TS \models_{\mathcal{F}_1} P_2$.

(b) $\mathcal{F}_2 = \Big(\big\{\{\alpha\}\big\}, \big\{\{\beta\}, \{\gamma\}\big\}, \big\{\{\eta\}\big\}\Big)$. Decide whether $TS \models_{\mathcal{F}_2} P_2$.

Justify your answers.

EXERCISE 3.20. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and let $A_1, \ldots, A_k,\ A'_1, \ldots, A'_l \subseteq Act$.

(a) Let $\mathcal{F}$ be the fairness assumption $\mathcal{F} = (\varnothing, \mathcal{F}_{strong}, \mathcal{F}_{weak})$ where

$$\mathcal{F}_{strong} = \{A_1, \ldots, A_k\} \text{ and } \mathcal{F}_{weak} = \{A'_1, \ldots, A'_l\}.$$

Provide a sketch of a scheduling algorithm that resolves the nondeterminism in TS in an $\mathcal{F}$-fair way.

(b) Let $\mathcal{F}_{ucond} = \{A_1, \ldots, A_k\}$, viewed as an unconditional fairness assumption for TS. Design a (scheduling) algorithm that checks whether $\mathcal{F}_{ucond}$ for TS is realizable, and if so, generates an $\mathcal{F}_{ucond}$-fair execution for TS.

Chapter 4

Regular Properties

This chapter treats some elementary algorithms to verify important classes of safety properties, liveness properties, and a wide range of other linear-time properties. We first consider regular safety properties, i.e., safety properties whose bad prefixes constitute a regular language, and hence can be recognized by a finite automaton. The algorithm to check a safety property P_{safe} for a given finite transition system TS relies on a reduction to the invariant-checking problem in a certain product construction of TS with a finite automaton that recognizes the bad prefixes of P_{safe}.

We then generalize this automaton-based verification algorithm to a larger class of linear-time properties, the so-called ω-regular properties. This class of properties covers regular safety properties, but also many other relevant properties such as various liveness properties. ω-regular properties can be represented by so-called Büchi automata, a variant of finite automata that accept infinite (rather than finite) words. Büchi automata will be the key concept to verify ω-regular properties via a reduction to persistence checking. The latter is a variant of invariant checking and aims to show that a certain state-condition holds continuously from some moment on.

4.1 Automata on Finite Words

Definition 4.1. Nondeterministic Finite Automaton (NFA)

A *nondeterministic finite automaton* (NFA) $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function,
- $Q_0 \subseteq Q$ is a set of initial states, and
- $F \subseteq Q$ is a set of *accept* (or: final) states.

The size of $\mathcal{A}$, denoted $|\mathcal{A}|$, is the number of states and transitions in $\mathcal{A}$, i.e.,

$$|\mathcal{A}| \;=\; |Q| + \sum_{q \in Q} \sum_{A \in \Sigma} |\delta(q, A)|.$$

■

Σ defines the symbols on which the automaton is defined. The (possibly empty) set Q_0 defines the states in which the automaton may start. The transition function δ can be identified with the relation $\to \subseteq Q \times \Sigma \times Q$ given by

$$q \xrightarrow{A} q' \text{ iff } q' \in \delta(q, A).$$

Thus, often the notion of transition relation (rather than transition function) is used for δ. Intuitively, $q \xrightarrow{A} q'$ denotes that the automaton can move from state q to state q' when reading the input symbol A.

Example 4.2. An Example of a Finite-State Automaton

An example of an NFA is depicted in Figure 4.1. Here, $Q = \{ q_0, q_1, q_2 \}$, $\Sigma = \{ A, B \}$, $Q_0 = \{ q_0 \}$, $F = \{ q_2 \}$, and the transition function δ is defined by

$$\begin{array}{lcllcl} \delta(q_0, A) & = & \{ q_0 \} & \delta(q_0, B) & = & \{ q_0, q_1 \} \\ \delta(q_1, A) & = & \{ q_2 \} & \delta(q_1, B) & = & \{ q_2 \} \\ \delta(q_2, A) & = & \varnothing & \delta(q_2, B) & = & \varnothing \end{array}$$

This corresponds to the transitions $q_0 \xrightarrow{A} q_0$, $q_0 \xrightarrow{B} q_0$, $q_0 \xrightarrow{B} q_1$, $q_1 \xrightarrow{A} q_2$, and $q_1 \xrightarrow{B} q_2$. The drawing conventions for an NFA are the same as for labeled transition systems. Accept states are distinguished from other states by drawing them with a double circle. ■

The intuitive operational behavior of an NFA is as follows. The automaton starts in one of the states in Q_0, and then is fed with an input word $w \in \Sigma^*$. The automaton reads this

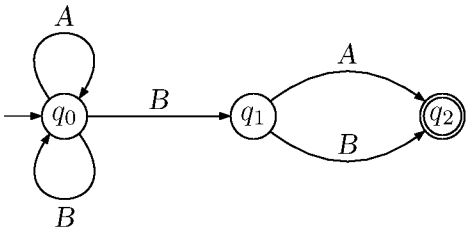


Figure 4.1: An example of a finite-state automaton.

word character by character from the left to the right. (The reader may assume that the input word is provided on a tape from which the automaton reads the input symbols from the left to the right by moving a cursor from the current position i to the next position $i{+}1$ when reading the ith input symbol. However, the automaton can neither write on the tape nor move the cursor in another way than one position to the right.) After reading an input symbol, the automaton changes its state according to the transition relation δ. That is, if the current input symbol A is read in the state q, the automaton chooses one of the possible transitions $q \xrightarrow{A} q'$ (i.e., one state $q' \in \delta(q, A)$) and moves to q' where the next input symbol will be consumed. If $\delta(q, A)$ contains two or more states, then the decision for the next state is made nondeterministically. An NFA cannot perform any transition when its current state q does not have an outgoing transition that is labeled with the current input symbol A. In that case, i.e., if $\delta(q, A) = \varnothing$, the automaton is stuck and no further progress can be made. The input word is said to be *rejected*. When the complete input word has been read, the automaton halts. It *accepts* whenever the current state is an accept state, and it *rejects* otherwise.

This intuitive explanation of the possible behaviors of an NFA for a given input word $w = A_1 \dots A_n$ is formalized by means of runs for w (see Definition 4.3 below). For any input word w there might be several possible behaviors (runs); some of them might be accepting, some of them might be rejecting. Word w is accepted by $\mathcal{A}$ if *at least one* of its runs is accepting, i.e., succeeds in reading the whole word and ends in a final state. This relies on the typical nondeterministic acceptor criterion which assumes an oracle for resolving the nondeterminism such that, whenever possible, an accepting run will be generated.

Definition 4.3. Runs, Accepted Language of an NFA

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA and $w = A_1 \dots A_n \in \Sigma^*$ a finite word. A *run* for w in $\mathcal{A}$ is a finite sequence of states $q_0\, q_1\, \dots\, q_n$ such that

- $q_0 \in Q_0$ and

- $q_i \xrightarrow{A_{i+1}} q_{i+1}$ for all $0 \leqslant i < n$.

Run $q_0\, q_1\, \ldots\, q_n$ is called *accepting* if $q_n \in F$. A finite word $w \in \Sigma^*$ is called *accepted* by $\mathcal{A}$ if there exists an accepting run for w. The *accepted language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of finite words in Σ^* accepted by $\mathcal{A}$, i.e.,

$$\mathcal{L}(\mathcal{A}) = \{ w \in \Sigma^* \mid \text{ there exists an accepting run for } w \text{ in } \mathcal{A} \}.$$

■

Example 4.4. Runs and Accepted Words

Example runs of the automaton $\mathcal{A}$ in Figure 4.1 are q_0 for the empty word ε, $q_0\, q_1$ for the word consisting of the symbol B, $q_0\, q_0\, q_0\, q_0$ for, e.g., the words ABA and BBA and $q_0\, q_1\, q_2$ for the words BA, and BB. Accepting runs are runs that finish in the final state q_2. For instance, the runs $q_0\, q_1\, q_2$ for BA and BB and $q_0\, q_0\, q_1\, q_2$ for the words ABB, ABA, BBA, and BBB are accepting. Thus, these words belong to $\mathcal{L}(\mathcal{A})$. The word AAA is not accepted by $\mathcal{A}$ since it only has single run, namely $q_0\, q_0\, q_0\, q_0$, which is not accepting.

The accepted language $\mathcal{L}(\mathcal{A})$ is given by the regular expression $(A + B)^* B (A + B)$. Thus, $\mathcal{L}(\mathcal{A})$ is the set of words over $\{ A, B \}$ where the last but one symbol is B. ■

The special cases $Q_0 = \varnothing$ or $F = \varnothing$ are allowed. In both cases, $\mathcal{L}(\mathcal{A}) = \varnothing$. If $F = \varnothing$, then there are no accepting runs. If there are no initial states, then there are no runs at all. Intuitively, the automaton rejects any input word immediately.

An equivalent alternative characterization of the accepted language of an NFA $\mathcal{A}$ is as follows. Let $\mathcal{A}$ be an NFA as above. We extend the transition function δ to the function $\delta^* : Q \times \Sigma^* \to 2^Q$ as follows: $\delta^*(q, \varepsilon) = \{ q \}$, $\delta^*(q, A) = \delta(q, A)$, and

$$\delta^*(q, A_1 A_2 \ldots A_n) = \bigcup_{p \in \delta(q, A_1)} \delta^*(p, A_2 \ldots A_n).$$

Stated in words, $\delta^*(q, w)$ is the set of states that are reachable from q for the input word w. In particular, $\bigcup_{q_0 \in Q_0} \delta^*(q_0, w)$ is the set of all states where a run for w in $\mathcal{A}$ can end. If one of these states is final, then w has an accepting run. Vice versa, if $w \notin \mathcal{L}(\mathcal{A})$, then none of these states is final. Hence, we have the following alternative characterization of the accepted language of an NFA by means of the extended transition function δ^*:

Lemma 4.5. Alternative Characterization of the Accepted Language

Let $\mathcal{A}$ be an NFA. Then:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \varnothing \text{ for some } q_0 \in Q_0\}.$$

It can be shown that the language accepted by an NFA constitutes a regular language. In fact, there are algorithms that for a given NFA $\mathcal{A}$ generate a regular expression for the language $\mathcal{L}(\mathcal{A})$. Vice versa, for any regular expression E, an NFA can be constructed that accepts $\mathcal{L}(\mathsf{E})$. Hence, the class of regular languages agrees with the class of languages accepted by an NFA.

An example of a language that is nonregular (but context-free) is $\{ A^n B^n \mid n \geqslant 0 \}$. There does not exist an NFA that accepts it. The intuitive argument for this is that one needs to be able to count the number of A's so as to be able to determine the number of B's that are to follow.

Since NFAs serve to represent (regular) languages we may identify those NFA that accept the same language:

Definition 4.6. Equivalence of NFAs

Let $\mathcal{A}$ and $\mathcal{A}'$ be NFAs with the same alphabet. $\mathcal{A}$ and $\mathcal{A}'$ are called *equivalent* if $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. ■

A fundamental issue in automata theory is to decide for a given NFA $\mathcal{A}$ whether its accepted language is *empty*, i.e., whether $\mathcal{L}(\mathcal{A}) = \varnothing$. This is known as the *emptiness problem*. From the acceptance condition, it follows directly that $\mathcal{L}(\mathcal{A})$ is nonempty if and only if there is at least one run that ends in some final state. Thus, nonemptiness of $\mathcal{L}(\mathcal{A})$ is equivalent to the existence of an accept state $q \in F$ which is reachable from an initial state $q_0 \in Q_0$. This can easily be determined in time $\mathcal{O}(|\mathcal{A}|)$ using a depth-first search traversal that encounters all states that are reachable from the initial states and checks whether one of them is final. For state $q \in Q$, let $\mathit{Reach}(q) = \bigcup_{w \in \Sigma^*} \delta^*(q, w)$; that is, $\mathit{Reach}(q)$ is the set of states q' that are reachable via an arbitrary run starting in state q.

Theorem 4.7. Language Emptiness is Equivalent to Reachability

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. Then, $\mathcal{L}(\mathcal{A}) \neq \varnothing$ if and only if there exists $q_0 \in Q_0$ and $q \in F$ such that $q \in \mathit{Reach}(q_0)$.

Regular languages exhibit some interesting closure properties, e.g., the union of two regular languages is regular. The same applies to concatenation and Kleene star (finite repetition). This is immediate from the definition of regular languages as those languages that can be generated by regular expressions. They are also closed under intersection and complementation, i.e., if $\mathcal{L}$, $\mathcal{L}_1, \mathcal{L}_2$ are regular languages over the alphabet Σ, then so are $\overline{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}$ and $\mathcal{L}_1 \cap \mathcal{L}_2$.

Let us briefly sketch the proofs for this. In both cases, we may proceed on the basis of finite automata and assume a representation of the given regular languages by NFA $\mathcal{A}$, $\mathcal{A}_1$, and $\mathcal{A}_2$ with the input alphabet Σ that accept the regular languages $\mathcal{L}$, $\mathcal{L}_1$, and $\mathcal{L}_2$, respectively. Intersection can be realized by a product construction $\mathcal{A}_1 \otimes \mathcal{A}_2$ which can be viewed as a parallel composition with synchronization over all symbols $A \in \Sigma$. In fact, the formal definition of $\otimes$ is roughly the same as the synchronization operator $\|$; see Definition 2.26 on page 48. The idea is simply that for the given input word, we run the two automata in parallel and reject as soon as one automaton cannot read the current input symbol, but accept if the input word is fully consumed and both automata accept (i.e., are in a final state).

Definition 4.8. Synchronous Product of NFAs

For NFA $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, Q_{0,i}, F_i)$, with i=1, 2, the *product automaton*

$$\mathcal{A}_1 \otimes \mathcal{A}_2 \;=\; (Q_1 \times Q_2, \Sigma, \delta, Q_{0,1} \times Q_{0,2}, F_1 \times F_2)$$

where δ is defined by

$$\frac{q_1 \xrightarrow{A}_1 q_1' \;\wedge\; q_2 \xrightarrow{A}_2 q_2'}{(q_1, q_2) \xrightarrow{A} (q_1', q_2')}.$$

■

It follows that this product construction of automata corresponds indeed to the intersection of their accepting languages, i.e., $\mathcal{L}(\mathcal{A}_1 \otimes \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

Let us now consider the complementation operator. Given an NFA $\mathcal{A}$ with the input alphabet Σ, we aim to construct an NFA for the complement language $\Sigma^* \setminus \mathcal{L}(\mathcal{A})$. The main step to do so is first to construct an equivalent *deterministic* finite automaton $\mathcal{A}_{det}$ which can be complemented in a quite simple way.

Definition 4.9. Deterministic Finite Automaton (DFA)

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. $\mathcal{A}$ is called *deterministic* if $|Q_0| \leqslant 1$ and $|\delta(q, A)| \leqslant 1$ for all states $q \in Q$ and all symbols $A \in \Sigma$. We will use the abbreviation DFA for a deterministic finite automaton.

DFA $\mathcal{A}$ is called *total* if $|Q_0| = 1$ and $|\delta(q, A)| = 1$ for all $q \in Q$ and all $A \in \Sigma$. ■

Stated in words, an NFA is deterministic if it has at most a single initial state and if for each symbol A the successor state of each state q is either uniquely defined (if $|\delta(q, A)| = 1$) or undefined (if $\delta(q, A) = \varnothing$). Total DFAs provide unique successor states, and thus, unique runs for each input word. Any DFA can be turned into an equivalent total DFA by simply adding a nonfinal trap state, q_{trap} say, that is equipped with a self-loop for any symbol $A \in \Sigma$. From any state $q \neq q_{trap}$, there is a transition to q_{trap} for any symbol A for which q has no A-successor in the given nontotal DFA.

Total DFA are often written in the form $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where q_0 stands for the unique initial state and δ is a (total) transition function $\delta : Q \times \Sigma \to Q$. Also, the extended transition function δ^* of a total DFA can be viewed as a total function $\delta^* : Q \times \Sigma^* \to Q$, which for given state q and finite word w returns the unique state $p = \delta^*(q, w)$ that is reached from state q for the input word w. In particular, the accepted language of a total DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is given by

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

The observation that total DFAs have *exactly one run* for each input word allows complementing a total DFA $\mathcal{A}$ by simply declaring all states to be final that are nonfinal in $\mathcal{A}$ and vice versa. Formally, if $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is a total DFA then $\overline{\mathcal{A}} = (Q, \Sigma, \delta, q_0, Q \setminus F)$ is a total DFA with $\mathcal{L}(\overline{\mathcal{A}}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$. Note that the operator $\mathcal{A} \mapsto \overline{\mathcal{A}}$ applied to a nontotal DFA (or NFA with proper nondeterministic choices) fails to provide an automaton for the complement language (why?).

It remains to explain how to construct for a given NFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ an equivalent total DFA $\mathcal{A}_{det}$. This can be done by a *powerset construction*, also often called a *subset construction*, since the states of $\mathcal{A}_{det}$ are the subsets of Q. This allows $\mathcal{A}_{det}$ to simulate $\mathcal{A}$ by moving the prefixes $A_1 \dots A_i$ of the given input word $w = A_1 \dots A_n \in \Sigma$ to the set of states that are reachable in $\mathcal{A}$ for $A_1 \dots A_i$. That is, $\mathcal{A}_{det}$ starts in Q_0, the set of initial states in $\mathcal{A}$. If $\mathcal{A}_{det}$ is in state Q' (which is a subset of $\mathcal{A}$'s state space Q), then $\mathcal{A}_{det}$ moves the input symbol A to $Q'' = \bigcup_{q \in Q'} \delta(q, A)$. If the input word has been consumed and $\mathcal{A}_{det}$ is in a state Q' that contains a state in $\mathcal{A}$'s set of accept states, then $\mathcal{A}_{det}$ accepts. The latter means that there exists an accepting run in $\mathcal{A}$ for the given input word w that ends in an accept state, and hence, $w \in \mathcal{L}(\mathcal{A})$. The formal definition of $\mathcal{A}_{det}$ is $\mathcal{A}_{det} = (2^Q, \Sigma, \delta_{det}, Q_0, F_{det})$ where

$$F_{det} = \{Q' \subseteq Q \mid Q' \cap F \neq \varnothing\}$$

and where the total transition function $\delta_{det} : 2^Q \times \Sigma \to 2^Q$ is defined by

$$\delta_{det}(Q', A) = \bigcup_{q \in Q'} \delta(q, A).$$

Clearly, $\mathcal{A}_{det}$ is a total DFA and, for all finite words $w \in \Sigma^*$, we have

$$\delta^*_{det}(Q_0, w) = \bigcup_{q_0 \in Q_0} \delta^*(q_0, w).$$

Thus, by Lemma 4.5, $\mathcal{L}(\mathcal{A}_{det}) = \mathcal{L}(\mathcal{A})$.

Example 4.10. Determinizing a Nondeterministic Finite Automaton

Consider the NFA depicted in Figure 4.1 on page 153. This automaton is not deterministic as on input symbol B in state q_0 the next state is not uniquely determined. The total DFA that is obtained through the powerset construction is depicted in Figure 4.2. ■

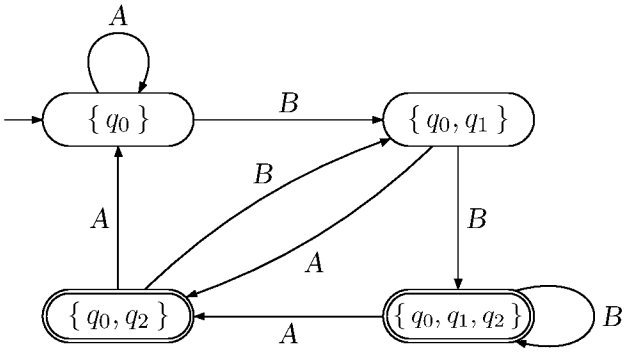


Figure 4.2: A DFA accepting $\mathcal{L}((A + B)^*B(A + B))$.

The powerset construction yields a total DFA that is exponentially larger than the original NFA. In fact, although DFAs and NFAs have the same power (both are equivalent formalisms for regular languages), NFAs can be much more efficient. The regular language given by the regular expression $\mathsf{E}_k = (A + B)^*B(A + B)^k$ (where k is a natural number) is accepted by an NFA with k+2 states (namely?), but it can be shown that there is no equivalent DFA with less than 2^k states. The intuitive argument for the latter is that each DFA for $\mathcal{L}(\mathsf{E}_k)$ needs to "remember" the positions of the symbol B among the last k input symbols which yields $\Omega(2^k)$ states.

We finally mention that for any regular language $\mathcal{L}$ there is a unique DFA $\mathcal{A}$ with $\mathcal{L} = \mathcal{L}(\mathcal{A})$ where the number of states is minimal under all DFAs for $\mathcal{L}$. Uniqueness is understood up to isomorphism, i.e., renaming of the states. (This does not hold for NFA. Why?) There is an algorithm to minimize a given DFA with N states into its equivalent minimal DFA which is based on partition refinement and takes $\mathcal{O}(N \cdot \log N)$ time in the worst case. The concepts of this minimization algorithm are outside the scope of this monograph and can be found in any textbook on automata theory. However, in Chapter 7 a very similar partitioning-refinement algorithm will be presented for bisimulation minimization.

4.2 Model-Checking Regular Safety Properties

In this section, it will be shown how NFAs can be used to check the validity of an important class of safety properties. The main characteristic of these safety properties is that all their bad prefixes constitute a regular language. The bad prefixes of these so-called *regular* safety properties can thus be recognized by an NFA. The main result of this section is that checking a regular safety property on a finite transition system can be reduced to invariant checking on the product of TS and an NFA $\mathcal{A}$ for the bad prefixes. Stated differently, if one wants to check whether a regular safety property holds for TS, it suffices to perform a reachability analysis in the product $TS \otimes \mathcal{A}$ to check a corresponding invariant on $TS \otimes \mathcal{A}$.

4.2.1 Regular Safety Properties

Recall that safety properties are LT properties, i.e., sets of infinite words over 2^{AP}, such that every trace that violates a safety property has a bad prefix that causes a refutation (cf. Definition 3.22 on page 112). Bad prefixes are finite, and thus the set of bad prefixes constitutes a language of finite words over the alphabet $\Sigma = 2^{AP}$. That is, the input symbols $A \in \Sigma$ of the NFA are now sets of atomic propositions. For instance, if $AP = \{ a, b \}$, then $\Sigma = \{ A_1, A_2, A_3, A_4 \}$ consists of the four input symbols $A_1 = \{\}$, $A_2 = \{ a \}$, $A_3 = \{ b \}$, and $A_4 = \{ a, b \}$.[1]

Definition 4.11. Regular Safety Property

Safety property P_{safe} over AP is called *regular* if its set of bad prefixes constitutes a regular language over 2^{AP}. ■

Every invariant is a regular safety property. If Φ is the state condition (propositional formula) of the invariant that should be satisfied by all reachable states, then the language of bad prefixes consists of the words $A_0 A_1 \ldots A_n$ such that $A_i \not\models \Phi$ for some $0 \leqslant i \leqslant n$. Such languages are regular, since they can be characterized by the (casually written) regular notation

$$\Phi^* (\neg \Phi) \, \mathrm{true}^*.$$

Here, Φ stands for the set of all $A \subseteq AP$ with $A \models \Phi$, $\neg\Phi$ for the set of all $A \subseteq AP$ with $A \not\models \Phi$, while true means the set of all subsets A of AP. For instance, if $AP = \{ a, b \}$ and $\Phi = a \vee \neg b$, then

[1] The symbol {} denotes the empty subset of AP which serves as symbol in the alphabet $\Sigma = 2^{AP}$. It must be distinguished from the regular expression $\varnothing$ representing the empty language.

- Φ stands for the regular expression $\{\} + \{a\} + \{a,b\}$,
- $\neg\Phi$ stands for the regular expression consisting of the symbol $\{b\}$,
- true stands for the regular expression $\{\} + \{a\} + \{b\} + \{a,b\}$.

The bad prefixes of the invariant over condition $a \vee \neg b$ are given by the regular expression:

$$\mathsf{E} = \underbrace{(\{\} + \{a\} + \{a,b\})^*}_{\Phi^*} \underbrace{\{b\}}_{\neg\Phi} \underbrace{(\{\} + \{a\} + \{b\} + \{a,b\})^*}_{\text{true}^*}.$$

Thus, $\mathcal{L}(\mathsf{E})$ consists of all words $A_1 \ldots A_n$ such that $A_i = \{b\}$ for some $1 \leqslant i \leqslant n$. Note that, for $A \subseteq AP = \{a,b\}$, we have $A \not\models a \vee \neg b$ if and only if $A = \{b\}$. Hence, $\mathcal{L}(\mathsf{E})$ agrees with the set of bad prefixes for the invariant induced by the condition Φ.

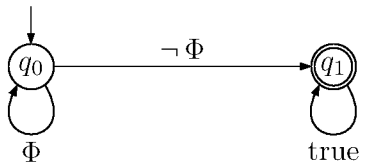


Figure 4.3: NFA accepting all bad prefixes of the invariant over the condition Φ.

In fact, for any invariant P_{inv}, the language of all bad prefixes can be represented by an NFA with two states, as shown in Figure 4.3. Here and in the sequel, we use symbolic notations in the pictures for NFAs over the alphabet 2^{AP}. We use propositional formulae over AP as labels for the edges. Thus, an edge leading from state q to state q' labeled with formula Ψ means that there are transitions $q \xrightarrow{A} q'$ for *all* $A \subseteq AP$ where $A \models \Psi$. E.g., if $AP = \{a,b\}$ and $\Phi = a \vee \neg b$, then Figure 4.3 is a representation for an NFA with two states q_0, q_1 and the transitions

$$q_0 \xrightarrow{\{\}} q_0, \quad q_0 \xrightarrow{\{a\}} q_0, \quad q_0 \xrightarrow{\{a,b\}} q_0, \quad q_0 \xrightarrow{\{b\}} q_1$$

and

$$q_1 \xrightarrow{\{\}} q_1, \quad q_1 \xrightarrow{\{a\}} q_1, \quad q_1 \xrightarrow{\{b\}} q_1, \quad q_1 \xrightarrow{\{a,b\}} q_1.$$

For the invariant over $AP = \{a,b\}$ induced by the condition $\Phi = a \vee \neg b$, the minimal bad prefixes are described by the regular expression $(\{\} + \{a\} + \{a,b\})^*\{b\}$. Hence, the minimal bad prefixes constitute a regular language too. An automaton that recognizes all minimal bad prefixes, which are given by the regular expression $\Phi^*(\neg\Phi)$, is obtained from Figure 4.3 by omitting the self-loop of state q_1. In fact, for the definition of regular safety properties it is irrelevant whether the regularity of the set of all bad prefixes or of the set of all minimal bad prefixes is required:

Lemma 4.12. Criterion for Regularity of Safety Properties

The safety property P_{safe} is regular if and only if the set of minimal bad prefixes for P_{safe} is regular.

Proof: "if": Let $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ be an NFA for $\mathit{MinBadPref}(P_{safe})$. Then, an NFA for $\mathit{BadPref}(P_{safe})$ is obtained by adding self-loops $q \xrightarrow{A} q$ for all states $q \in F$ and all $A \subseteq AP$. It is easy to check that the modified NFA accepts the language consisting of all bad prefixes for P_{safe}. Thus, $\mathit{BadPref}(P_{safe})$ is regular, which yields the claim.

"only if": Let $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ be a DFA for $\mathit{BadPref}(P_{safe})$. For $\mathit{MinBadPref}(P_{safe})$, a DFA is obtained by removing all outgoing transitions from the accept states in $\mathcal{A}$. Let $\mathcal{A}'$ be the modified DFA and let us check that $\mathcal{L}(\mathcal{A}') = \mathit{MinBadPref}(P_{safe})$.

If we are given a word $w = A_1 \ldots A_n \in \mathcal{L}(\mathcal{A}')$, then $w \in \mathcal{L}(\mathcal{A})$ since the run $q_0 q_1 \ldots q_n$ in $\mathcal{A}'$ for w is also an accepting run in $\mathcal{A}$. Therefore, w is a bad prefix for P_{safe}. Distinguish two cases.

Assume w is not a minimal bad prefix. Then there exists a proper prefix $A_1 \ldots A_i$ of w that is a bad prefix for P_{safe}. Thus, $A_1 \ldots A_i \in \mathcal{L}(\mathcal{A})$. Since $\mathcal{A}$ is deterministic, $q_0 q_1 \ldots q_i$ is the (unique) run for $A_1 \ldots A_i$ in $\mathcal{A}$ and $q_i \in F$. Since $i < n$ and q_i has no outgoing transitions in $\mathcal{A}'$, $q_0 \ldots q_i \ldots q_n$ cannot be a run for $A_1 \ldots A_i \ldots A_n$ in $\mathcal{A}'$. This contradicts the assumption and shows that $A_1 \ldots A_n$ is a minimal bad prefix for P_{safe}.

Vice versa, if w is a minimal bad prefix for P_{safe}, then

(1) $A_1 \ldots A_n \in \mathit{BadPref}(P_{safe}) = \mathcal{L}(\mathcal{A})$ and

(2) $A_1 \ldots A_i \notin \mathit{BadPref}(P_{safe}) = \mathcal{L}(\mathcal{A})$ for all $1 \leqslant i < n$.

Let $q_0 \ldots q_n$ be the unique run for w in $\mathcal{A}$. Then, (2) yields $q_i \notin F$ for $1 \leqslant i < n$, while $q_n \in F$ by (1). Thus, $q_0 \ldots q_n$ is an accepting run for w in $\mathcal{A}'$ which yields $w \in \mathcal{L}(\mathcal{A}')$. ■

Example 4.13. Regular Safety Property for Mutual Exclusion Algorithms

Consider a mutual exclusion algorithm such as the semaphore-based one or Peterson's algorithm. The bad prefixes of the safety property P_{mutex} ("there is always at most one process in its critical section") constitute the language of all finite words $A_0\, A_1 \ldots A_n$ such that

$$\{\, crit_1, crit_2 \,\} \subseteq A_i$$

for some index i with $0 \leqslant i \leqslant n$. If $i{=}n$ is the smallest such index, i.e., $\{ crit_1, crit_2 \} \subseteq A_n$ and $\{ crit_1, crit_2 \} \not\subseteq A_j$ for $0 \leqslant j < n$, then $A_0 \ldots A_n$ is a minimal bad prefix. The language of all (minimal) bad prefixes is regular. An NFA recognizing all minimal bad prefixes is depicted in Figure 4.4. ■

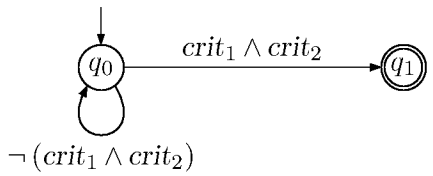


Figure 4.4: Minimal bad prefixes that refute the mutual exclusion property.

Example 4.14. Regular Safety Property for the Traffic Light

Consider a traffic light with three possible colors: red, yellow and green. The property "a red phase must be preceded immediately by a yellow phase" is specified by the set of infinite words $\sigma = A_0\, A_1 \ldots$ with $A_i \subseteq \{ red, yellow \}$ such that for all $i \geqslant 0$ we have that

$$red \in A_i \text{ implies } i > 0 \text{ and } yellow \in A_{i-1}.$$

The bad prefixes are finite words that violate this condition. Examples of bad prefixes that are minimal are

$$\{\}\,\{\}\,\{ red \} \quad \text{and} \quad \{\}\,\{ red \}.$$

In general, the minimal bad prefixes are words of the form $A_0\, A_1 \ldots A_n$ such that $n > 0$, $red \in A_n$, and $yellow \notin A_{n-1}$. The NFA in Figure 4.5 accepts these minimal bad prefixes. Recall the meaning of the edge labels in the pictorial representations of an NFA over the

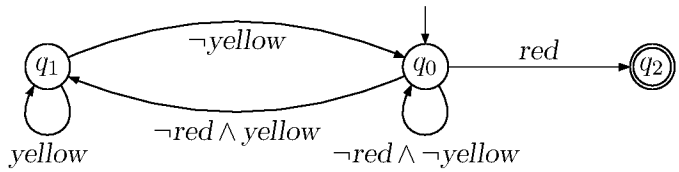


Figure 4.5: Minimal bad prefixes in which red is not preceded by yellow.

alphabet $\Sigma = 2^{AP}$ where $AP = \{ yellow,red \}$. For instance, the edge-label *yellow* in the self-loop of state q_1 denotes a formula, namely the positive literal $yellow \in AP$. This stands for all sets $A \subseteq AP = \{ yellow,red \}$ where the literal *yellow* holds, that is, the sets $\{ yellow \}$ and $\{ yellow, red \}$. Hence, the self-loop of q_1 in the picture stands for the two

transitions:

$$q_1 \xrightarrow{\{yellow\}} q_1 \quad \text{and} \quad q_1 \xrightarrow{\{yellow,red\}} q_1.$$

Similarly, the edge label $\neg yellow$ in the transition from q_1 to q_0 stands for the negative literal $\neg yellow$, and thus represents the transitions:

$$q_1 \xrightarrow{\{red\}} q_0 \quad \text{and} \quad q_1 \xrightarrow{\{\}} q_0.$$

In the same way the label *red* of the transition from q_0 to q_2 represents two transitions (with the labels $\{ red \}$ and $\{ red, yellow \}$), while the edge labels $\neg red \wedge yellow$ and $\neg red \wedge \neg yellow$ denote only a single symbol in 2^{AP}, namely $\{ yellow \}$ and $\{\}$, respectively. ■

Example 4.15. A Nonregular Safety Property

Not all safety properties are regular. As an example of a nonregular safety property, consider:

"The number of inserted coins is always at least the number of dispensed drinks."

(See also Example 3.24 on page 113). Let the set of propositions be $\{ pay, drink \}$. Minimal bad prefixes for this safety property constitute the language

$$\{ pay^n \, drink^{n+1} \mid n \geqslant 0 \}$$

which is not a regular, but a context-free language. Such safety properties fall outside the scope of the following verification algorithm. ■

4.2.2 Verifying Regular Safety Properties

Let P_{safe} be a *regular* safety property over the atomic propositions AP and $\mathcal{A}$ an NFA recognizing the (minimal) bad prefixes of P_{safe}. (Recall that by Lemma 4.12 on page 161 it is irrelevant whether $\mathcal{A}$ accepts all bad prefixes for P_{safe} or only the minimal ones.) For technical reasons, we assume that $\varepsilon \notin \mathcal{L}(\mathcal{A})$. In fact, this is not a severe restriction since otherwise all finite words over 2^{AP} are bad prefixes, and hence, $P_{safe} = \varnothing$. In this case, $TS \models P_{safe}$ if and only if TS has no initial state.

Furthermore, let TS be a *finite* transition system without terminal states with corresponding set of propositions AP. In this section, we aim to establish an algorithmic method for verifying whether TS satisfies *regular* safety property P_{safe}, i.e., to check whether

$TS \models P_{safe}$ holds. According to Lemma 3.25 on page 114 we have

$$TS \models P_{safe} \quad \text{if and only if} \quad \mathit{Traces}_{fin}(TS) \cap \mathit{BadPref}(P_{safe}) = \varnothing$$

$$\text{if and only if} \quad \mathit{Traces}_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing.$$

Thus, it suffices to check whether $\mathit{Traces}_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$ to establish $TS \models P_{safe}$.

To do so, we adopt a similar strategy as for checking whether two NFAs intersect. Recall that in order to check whether the NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ do intersect, it suffices to consider their product automaton, so

$$\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2) \;=\; \varnothing \quad \text{if and only if} \quad \mathcal{L}(\mathcal{A}_1 \otimes \mathcal{A}_2) \;=\; \varnothing.$$

The question whether two automata do intersect is thus reduced to a simple reachability problem in the product automaton.

This is now exploited as follows. In order to check whether $\mathit{Traces}_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$, we first build a product of transition system TS and NFA $\mathcal{A}$ in the same vein as the synchronous product of NFA. This yields the transition system $TS \otimes \mathcal{A}$. For this transition system, an invariant can be given using a propositional logic formula Φ –derived from the accept states of $\mathcal{A}$– such that $\mathit{Traces}_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$ if and only if $TS \otimes \mathcal{A} \models$ "always Φ". In this way, the verification of a regular safety property is reduced to invariant checking. Recall that for checking invariants, Algorithm 4 (see page 110) can be exploited.

We start by formally defining the product between a transition system TS and an NFA $\mathcal{A}$, denoted $TS \otimes \mathcal{A}$. Let $TS = (S, Act, \rightarrow, I, AP, L)$ and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ with $Q_0 \cap F = \varnothing$. Recall that the alphabet of $\mathcal{A}$ consists of sets of atomic proposition in TS. Transition system $TS \otimes \mathcal{A}$ has state space $S \times Q$ and a transition relation such that each path fragment $\pi = s_0\, s_1 \ldots s_n$ in TS can be extended to a path fragment

$$\langle s_0, q_1 \rangle \, \langle s_1, q_2 \rangle \ldots \langle s_n, q_{n+1} \rangle$$

in $TS \otimes \mathcal{A}$ which has an initial state $q_0 \in Q_0$ for which

$$q_0 \xrightarrow{L(s_0)} q_1 \xrightarrow{L(s_1)} q_2 \xrightarrow{L(s_2)} \ldots \xrightarrow{L(s_n)} q_{n+1}$$

is a run—not necessarily accepting—of NFA $\mathcal{A}$ that generates the word

$$\mathit{trace}(\pi) \;=\; L(s_0)\, L(s_1) \ldots L(s_n).$$

Finally, labels of states are state names of $\mathcal{A}$. These considerations lead to the following definition:

Definition 4.16. Product of Transition System and NFA

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ an NFA with the alphabet $\Sigma = 2^{AP}$ and $Q_0 \cap F = \varnothing$. The product transition system $TS \otimes \mathcal{A}$ is defined as follows:

$$TS \otimes \mathcal{A} \;=\; (S', Act, \rightarrow', I', AP', L')$$

where

- $S' = S \times Q$,
- $\rightarrow'$ is the smallest relation defined by the rule

$$\frac{s \xrightarrow{\alpha} t \;\wedge\; q \xrightarrow{L(t)} p}{\langle s, q\rangle \xrightarrow{\alpha}' \langle t, p\rangle},$$

- $I' = \{\, \langle s_0, q\rangle \mid s_0 \in I \;\wedge\; \exists q_0 \in Q_0.\; q_0 \xrightarrow{L(s_0)} q \,\}$,
- $AP' = Q$, and
- $L' : S \times Q \to 2^Q$ is given by $L'(\langle s, q\rangle) \;=\; \{\, q \,\}$.

■

Remark 4.17. Terminal States

For the definition of LT properties (and thus of invariants) we have assumed transition systems to have no terminal states. It is, however, not guaranteed that $TS \otimes \mathcal{A}$ possesses this property, even if TS does. This stems from the fact that in NFA $\mathcal{A}$ there may be a state q, say, that has no direct successor states for some set A of atomic propositions, i.e., with $\delta(q, A) = \varnothing$. This technical problem can be treated by either requiring $\delta(q, A) \neq \varnothing$ for all states $q \in Q$ and $A \subseteq AP$ or by extending the notion of invariants to arbitrary transition systems. Note that imposing the requirement $\delta(q, A) \neq \varnothing$ is not a severe restriction, as any NFA can be easily transformed into an equivalent one that satisfies this property by introducing a state q_{trap} and adding transition $q \xrightarrow{A} q_{trap}$ to $\mathcal{A}$ whenever $\delta(q, A) = \varnothing$ or $q = q_{trap}$. We finally remark that for the algorithm for invariant checking, it is not of any relevance whether terminal states exist or not. ■

Example 4.18. A Product Automaton

The language of the minimal bad prefixes of the safety property "each red light phase

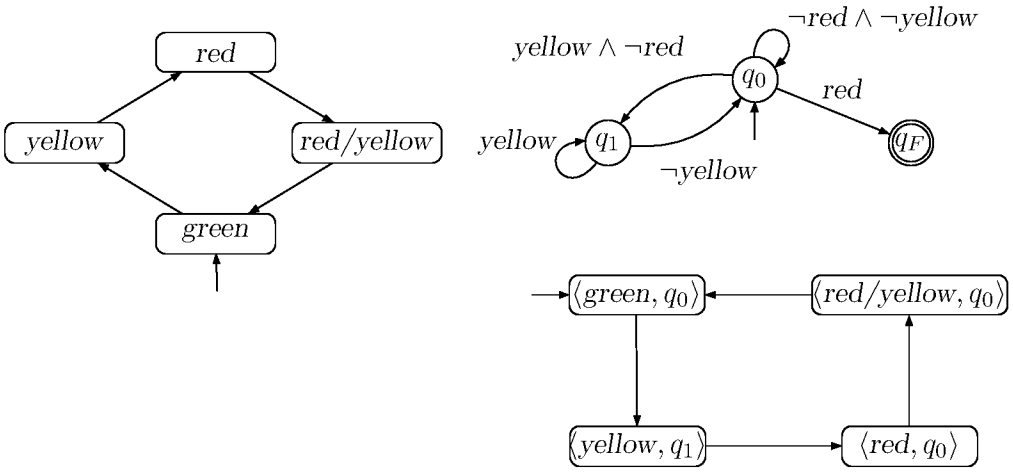


Figure 4.6: German traffic light (left upper figure), an NFA (right upper figure), and their product (lower figure).

is preceded by a yellow light phase" is accepted by the DFA $\mathcal{A}$ indicated in Example 4.14 (page 162). We consider a German traffic light, which besides the usual possible colors red, green, and yellow, has the possibility to indicate red and yellow simultaneously indicating "green light soon". The transition system *GermanTrLight* thus has four states with the usual transitions *red* $\rightarrow$ *red+yellow*, *red+yellow* $\rightarrow$ *green*, *green* $\rightarrow$ *yellow*, and *yellow* $\rightarrow$ *red*. Let $AP = \{ red, yellow \}$ indicating the corresponding light phases. The labeling is defined as follows: $L(red) = \{ red \}$, $L(yellow) = \{ yellow \}$, $L(green) = \varnothing = L(red{+}yellow)$. The product transition system $GermanTrLight \otimes \mathcal{A}$ consists of four reachable states (see Figure 4.6). As action labels are not relevant here, they are omitted. ■

The following theorem shows that the verification of a regular safety property can be reduced to checking an invariant in the product.

Let TS and $\mathcal{A}$ be as before. Let $P_{inv(\mathcal{A})}$ be the invariant over $AP' = 2^Q$ which is defined by the propositional formula

$$\bigwedge_{q \in F} \neg\, q.$$

In the sequel, we often write $\neg\, F$ as shorthand for $\bigwedge_{q \in F} \neg\, q$. Stated in words, $\neg F$ holds in all nonaccept states.

Theorem 4.19. Verification of Regular Safety Properties

For transition system TS over AP, NFA $\mathcal{A}$ with alphabet 2^{AP} as before, and regular safety property P_{safe} over AP such that $\mathcal{L}(\mathcal{A})$ equals the set of (minimal) bad prefixes of P_{safe}, the following statements are equivalent:

(a) $TS \models P_{safe}$

(b) $Traces_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) = \varnothing$

(c) $TS \otimes \mathcal{A} \models P_{inv(A)}$

Proof: Let $TS = (S, Act, \rightarrow, I, AP, L)$ and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$.

The equivalence of (a) and (b) follows immediately by Lemma 3.25 (page 114). To establish the equivalence of (a), (b), and (c), we show

$$(c) \Longrightarrow (a): \quad TS \not\models P_{safe} \quad \text{implies} \quad TS \otimes \mathcal{A} \not\models P_{inv(A)}$$

and

$$(b) \Longrightarrow (c): \quad TS \otimes \mathcal{A} \not\models P_{inv(A)} \quad \text{implies} \quad Traces_{fin}(TS) \cap \mathcal{L}(\mathcal{A}) \neq \varnothing.$$

Proof of "(c) $\Longrightarrow$ (a)": If $TS \not\models P_{safe}$, then there is a finite initial path fragment $\widehat{\pi} = s_0\, s_1 \ldots s_n$ in TS with

$$trace(\widehat{\pi}) \quad = \quad L(s_0)\, L(s_1) \ldots L(s_n) \quad \in \quad \mathcal{L}(\mathcal{A}).$$

Since $trace(\widehat{\pi}) \in \mathcal{L}(\mathcal{A})$, there exists an accepting run $q_0\, q_1 \ldots q_{n+1}$ of $\mathcal{A}$ for $trace(\widehat{\pi})$. Accordingly

$$q_0 \in Q_0 \text{ and } q_i \xrightarrow{L(s_i)} q_{i+1} \text{ for all } 0 \leqslant i \leqslant n, \text{ and } q_{n+1} \in F.$$

Thus, $\langle s_0, q_1\rangle\, \langle s_1, q_2\rangle \ldots \langle s_n, q_{n+1}\rangle$ is an initial path fragment in $TS \otimes \mathcal{A}$ with

$$\langle s_n, q_{n+1}\rangle \not\models \neg F.$$

It thus follows that $TS \otimes \mathcal{A} \not\models P_{inv(A)}$.

Proof of "(b) $\Longrightarrow$ (c)": Let $TS \otimes \mathcal{A} \not\models P_{inv(A)}$. Then there exists an initial path fragment

$$\langle s_0, q_1\rangle \ldots \langle s_n, q_{n+1}\rangle$$

in $TS \otimes \mathcal{A}$ with $q_{n+1} \in F$, and $q_1, \ldots, q_n \notin F$. Besides, $s_0\, s_1 \ldots s_n$ is an initial path fragment in TS. Further,

$$q_i \xrightarrow{L(s_i)} q_{i+1} \text{ for all } 0 \leqslant i \leqslant n.$$

Since $\langle s_0, q_1 \rangle$ is an initial state of $TS \otimes \mathcal{A}$, there is an initial state q_0 in $\mathcal{A}$ such that $q_0 \xrightarrow{L(s_0)} q_1$. Sequence $q_0\, q_1 \ldots q_{n+1}$ is thus an accepting run for $\mathit{trace}(s_0\, s_1 \ldots s_n)$. Thus,

$$\mathit{trace}(s_0\, s_1 \ldots s_n) \in \mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A})$$

which yields $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathcal{L}(\mathcal{A}) \neq \varnothing$.

■

Stated in words, Theorem 4.19 yields that in order to check the transition system TS versus the regular safety property P_{safe}, it suffices to check whether no state $\langle s, q \rangle$ in $TS \otimes \mathcal{A}$ is reachable where the $\mathcal{A}$-component q is an accept state in $\mathcal{A}$. This invariant "visit never an accept state in $\mathcal{A}$" (formally given by the invariant condition $\Phi = \neg F$) can be checked using a depth-first search approach as described in detail in Algorithm 4 (page 110). Note that in case the safety property is refuted, the invariant checking algorithm provides a counterexample. This counterexample is in fact a finite path fragment $\langle s_0, q_1 \rangle\, \langle s_1, q_2 \rangle \ldots \langle s_n, q_{n+1} \rangle$ in the transition system $TS \otimes \mathcal{A}$ that leads to an accept state. The projection to the states in TS yields an initial finite path fragment $s_0\, s_1 \ldots s_n$ in TS where the induced trace $\mathit{trace}(s_0 s_1 \ldots s_n) \in (2^{AP})^*$ is accepted by $\mathcal{A}$ (since it has an accepting run of the form $q_0\, q_1 \ldots q_{n+1}$). Thus, $\mathit{trace}(s_0\, s_1 \ldots s_n)$ is a bad prefix for P_{safe}. Hence, $s_0\, s_1 \ldots s_n$ yields a useful error indication since $\mathit{trace}(\pi) \notin P_{safe}$ for all paths π in TS that start with the prefix $s_0\, s_1 \ldots s_n$.

Corollary 4.20.

Let TS, $\mathcal{A}$, and P_{safe} be as in Theorem 4.19. Then, for each initial path fragment $\langle s_0, q_1 \rangle \ldots \langle s_n, q_{n+1} \rangle$ of $TS \otimes \mathcal{A}$:

$$q_1, \ldots, q_n \notin F \text{ and } q_{n+1} \in F \quad \textit{implies} \quad \mathit{trace}(s_0\, s_1 \ldots s_n) \in \mathcal{L}(\mathcal{A}).$$

As a result, the skeleton in Algorithm 5 can be used to check a regular safety property against a transition system and to report a counterexample (i.e., finite initial path fragment in TS inducing a bad prefix) as diagnostic feedback if the safety property does not hold for TS.

Example 4.21. Checking a Regular Safety Property for the Traffic Light

Consider again the German traffic light system and the regular safety property P_{safe}

Algorithm 5 Model-checking algorithm for regular safety properties

Input: finite transition system TS and regular safety property P_{safe}
Output: true if $TS \models P_{safe}$. Otherwise false plus a counterexample for P_{safe}.

Let NFA $\mathcal{A}$ (with accept states F') be such that $\mathcal{L}(\mathcal{A})$ = bad prefixes of P_{safe}
Construct the product transition system $TS \otimes \mathcal{A}$
Check the invariant $P_{inv(\mathcal{A})}$ with proposition $\neg F' = \bigwedge_{q \in F} \neg q$ on $TS \otimes \mathcal{A}$.

if $TS \otimes \mathcal{A} \models P_{inv(\mathcal{A})}$ **then**
  return true
else
  Determine an initial path fragment $\langle s_0, q_1 \rangle \ldots \langle s_n, q_{n+1} \rangle$ of $TS \otimes \mathcal{A}$ with $q_{n+1} \in F'$
  return (false, $s_0\, s_1 \ldots s_n$)
fi

that each red light phase should be immediately preceded by a yellow light phase. The transition system of the traffic light, the NFA accepting the bad prefixes of the safety property, as well as their product automaton, are depicted in Figure 4.6 (page 166). To check the validity of P_{safe}, only the second component of the states $\langle s, q \rangle$ is relevant. The fact that no state of the form $\langle \ldots, q_F \rangle$ is reachable ensures the invariant $\neg q_F$ to hold in all reachable states. Thus $GermanTrLight \models P_{safe}$.

If the traffic light is modified such that the state "red" is the initial state (instead of "green"), then we obtain a transition system that violates P_{safe}. Actually, in this case the invariant $\neg q_F$ is already violated in the initial state of the resulting product transition system that has the following form:

$$\langle red, \delta(q_0, \{red\}) \rangle \;=\; \langle red, q_F \rangle.$$

■

We conclude this part by considering the worst-case time and space complexity of the automata-based algorithm for checking regular safety properties.

Theorem 4.22. Complexity of Verifying Regular Safety Properties

The time and space complexity of Algorithm 5 is in $\mathcal{O}(|TS| \cdot |\mathcal{A}|)$ *where* $|TS|$ *and* $|\mathcal{A}|$ *denote the number of states and transitions in* TS *and* $\mathcal{A}$*, respectively.*

Assuming an generation of the reachable states of TS from a syntactic description of the processes, the above bound also holds if $|TS|$ denotes the size of the reachable fragment of TS.

Proof: Follows directly from the fact that the number of states in the product automaton $TS \otimes \mathcal{A}$ is in $\mathcal{O}(|S| \cdot |Q|)$ (where S and Q denote the state space of TS and $\mathcal{A}$, respectively) and the fact that the time and space complexity of invariant checking is linear in the number of states and transitions of the transition system $TS \otimes \mathcal{A}$. (Thus, we can even establish the bound $\mathcal{O}(|S| \cdot |Q| + | \rightarrow | \cdot |\delta|)$ for the runtime where $| \rightarrow |$ denotes the number of transitions in TS and $|\delta|$ the number of transitions in $\mathcal{A}$.) ■

4.3 Automata on Infinite Words

Finite-state automata accept finite words, i.e., sequences of symbols of finite length, and yield the basis for checking regular safety properties. In this and the following sections, these ideas are generalized toward a more general class of LT properties. These include regular safety, various liveness properties, but also many other properties that are relevant to formalize the requirements for "realistic" systems. The rough idea is to consider variants of NFAs, called nondeterministic Büchi automata (NBAs), which serve as acceptors for languages of infinite words. It will be established that if we are given a nondeterministic Büchi automaton $\mathcal{A}$ that specifies the "bad traces" (i.e., that accepts the complement of the LT property P to be verified), then a graph analysis in the product of the given transition system TS and the automaton $\mathcal{A}$ suffices to either establish or disprove $TS \models P$. Whereas for regular safety properties a reduction to invariant checking (i.e., a depth-first search) is possible, the graph algorithms needed here serve to check a so-called persistence property. Such properties state that eventually for ever a certain proposition holds.

We first introduce the "ω-counterpart" to regular languages, both by introducing ω-regular expressions (Section 4.3.1) and nondeterministic Büchi automata (see Section 4.3.2). Variants of nondeterministic Büchi automata will be discussed in Sections 4.3.3 and 4.3.4.

4.3.1 ω-Regular Languages and Properties

Infinite words over the alphabet Σ are infinite sequences $A_0\, A_1\, A_2 \ldots$ of symbols $A_i \in \Sigma$. Σ^ω denotes the set of all infinite words over Σ. As before, the Greek letter σ will be used for infinite words, while w, v, u range over finite words. Any subset of Σ^ω is called a language of infinite words, sometimes also called an ω-language. In the sequel, the notion of a language will be used for any subset of $\Sigma^* \cup \Sigma^\omega$. Languages will be denoted by the symbol $\mathcal{L}$.

To reason about languages of infinite words, the basic operations of regular expressions (union, concatenation, and finite repetition) are extended by *infinite* repetition, denoted by the Greek letter ω.[2] For instance, the infinite repetition of the finite word AB yields the infinite word $ABABABABAB\ldots$ (ad infinitum) and is denoted by $(AB)^\omega$. For the special case of the empty word, we have $\varepsilon^\omega = \varepsilon$. For an infinite word, infinite repetition has no effect, that is, $\sigma^\omega = \sigma$ if $\sigma \in \Sigma^\omega$. Note that the finite repetition of a word results in a language of finite words, i.e., a subset of Σ^*, whereas infinite repetition of a (finite or infinite) word results in a single word.

Infinite repetition can be lifted to languages as follows. For language $\mathcal{L} \subseteq \Sigma^*$, let $\mathcal{L}^\omega$ be the set of words in $\Sigma^* \cup \Sigma^\omega$ that arise from the infinite concatenation of (arbitrary) words in Σ, i.e.,

$$\mathcal{L}^\omega \;=\; \{w_1 w_2 w_3 \ldots \mid w_i \in \mathcal{L}, i \geqslant 1\}.$$

The result is an ω-language, provided that $\mathcal{L} \subseteq \Sigma^+$, i.e., $\mathcal{L}$ does not contain the empty word ε. However, in the sequel, we only need the ω-operator applied to languages of finite words that do not contain the empty word. In this case, i.e., for $\mathcal{L} \subseteq \Sigma^+$, we have $\mathcal{L}^\omega \subseteq \Sigma^\omega$.

In the following definition, the concatenation operator $\mathcal{L}_1.\mathcal{L}_2$ is used that combines a language $\mathcal{L}_1$ of finite words with a language $\mathcal{L}_2$ of infinite words. It is defined by $\mathcal{L}_1.\mathcal{L}_2 = \{w\sigma \mid w \in \mathcal{L}_1, \sigma \in \mathcal{L}_2\}$.

Definition 4.23. **ω-Regular Expression**

An *ω-regular expression* G over the alphabet Σ has the form

$$\mathsf{G} = \mathsf{E}_1.\mathsf{F}_1^\omega + \ldots + \mathsf{E}_n.\mathsf{F}_n^\omega$$

where $n \geqslant 1$ and $\mathsf{E}_1, \ldots, \mathsf{E}_n, \mathsf{F}_1, \ldots, \mathsf{F}_n$ are regular expressions over Σ such that $\varepsilon \notin \mathcal{L}(\mathsf{F}_i)$, for all $1 \leqslant i \leqslant n$.

The semantics of the ω-regular expression G is a language of infinite words, defined by

$$\mathcal{L}_\omega(\mathsf{G}) \;=\; \mathcal{L}(\mathsf{E}_1).\mathcal{L}(\mathsf{F}_1)^\omega \cup \ldots \cup \mathcal{L}(\mathsf{E}_n).\mathcal{L}(\mathsf{F}_n)^\omega$$

where $\mathcal{L}(\mathsf{E}) \subseteq \Sigma^*$ denotes the language (of finite words) induced by the regular expression E (see page 914).

Two ω-regular expressions G_1 and G_2 are *equivalent*, denoted $\mathsf{G}_1 \equiv \mathsf{G}_2$, if $\mathcal{L}_\omega(\mathsf{G}_1) = \mathcal{L}_\omega(\mathsf{G}_2)$. ■

[2] The symbol ω denotes the first infinite ordinal. It already appeared in the notation Σ^ω for the set of infinite words over the alphabet Σ.

Examples for ω-regular expressions over the alphabet $\Sigma = \{ A, B, C \}$ are

$$(A + B)^* A (AAB + C)^\omega \quad \text{or} \quad A(B + C)^* A^\omega + B(A + C)^\omega.$$

If E is a regular expression with $\varepsilon \notin \mathcal{L}(\mathsf{E})$, then also E^ω can be viewed as an ω-regular expression since it can be identified with $\mathsf{E}.\mathsf{E}^\omega$ or $\varepsilon.\mathsf{E}^\omega$. Note that we have $\mathcal{L}(\mathsf{E})^\omega = \mathcal{L}(\mathsf{E}.\mathsf{E}^\omega) = \mathcal{L}(\varepsilon.\mathsf{E}^\omega)$.

Definition 4.24. ω-Regular Language

A language $\mathcal{L} \subseteq \Sigma^\omega$ is called ω-regular if $\mathcal{L} = \mathcal{L}_\omega(\mathsf{G})$ for some ω-regular expression G over Σ. ■

For instance, the language consisting of all infinite words over $\{A, B\}$ that contain infinitely many A's is ω-regular since it is given by the ω-regular expression $(B^*A)^\omega$. The language consisting of all infinite words over $\{A, B\}$ that contain only finitely many A's is ω-regular too. A corresponding ω-regular expression is $(A+B)^*B^\omega$. The empty set is ω-regular since it is obtained, e.g., by the ω-regular expression $\varnothing^\omega$. More generally, if $\mathcal{L} \subseteq \Sigma^*$ is regular and $\mathcal{L}'$ is ω-regular, then $\mathcal{L}^\omega$ and $\mathcal{L}.\mathcal{L}'$ are ω-regular.

ω-Regular languages possess several closure properties: they are closed under union, intersection, and complementation. The argument for union is obvious from the definition by ω-regular expressions. The proof for the intersection will be provided later; see Corollary 4.60 on page 198. The more advanced proof for complementation is not provided in this monograph. We refer the interested reader to [174] that covers also other properties of ω-regular languages and various other automata models.

The concepts of ω-regular languages play an important role in verification since most relevant LT properties are ω-regular:

Definition 4.25. ω-Regular Properties

LT property P over AP is called *ω-regular* if P is an ω-regular language over the alphabet 2^{AP}. ■

For instance, for $AP = \{ a, b \}$, the invariant P_{inv} induced by the proposition $\Phi = a \vee \neg b$ is an ω-regular property since

$$\begin{aligned} P_{inv} &= \Big\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \forall i \geqslant 0. \, (a \in A_i \text{ or } b \notin A_i) \Big\} \\ &= \Big\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \forall i \geqslant 0. \, (A_i \in \{\{\}, \{a\}, \{a, b\}\}) \Big\} \end{aligned}$$

is given by the ω-regular expression $\mathsf{E} = (\{\} + \{a\} + \{a,b\})^\omega$ over the alphabet $\Sigma = 2^{AP} = \{\{\}, \{a\}, \{b\}, \{a,b\}\}$. In fact, any invariant over AP is ω-regular (the set AP of atomic propositions is arbitrary) as it can be described by the ω-regular expression Φ^ω where Φ denotes the underlying propositional formula (that has to hold for all reachable states) and is identified with the regular expression given by the sum of all $A \subseteq AP$ with $A \models \Phi$.

Also, any regular safety property P_{safe} is an ω-regular property. This follows from the fact that the complement language

$$(2^{AP})^\omega \setminus P_{safe} \;=\; \underbrace{\mathit{BadPref}(P_{safe})}_{\text{regular}} . (2^{AP})^\omega$$

is an ω-regular language. The result that ω-regular languages are closed under complementation (stated above, in the end of Section 4.3.1 on page 172) yields the claim.

Example 4.26. Mutual Exclusion

Another example of an ω-regular property is the property given by the informal statement "process $\mathcal{P}$ visits its critical section infinitely often" which, for $AP = \{ \mathit{wait}, \mathit{crit} \}$, can be formalized by the ω-regular expression:

$$((\underbrace{\{\} + \{ \mathit{wait} \}}_{\text{negative literal } \neg \mathit{crit}})^* . (\underbrace{\{ \mathit{crit} \} + \{ \mathit{wait}, \mathit{crit} \}}_{\text{positive literal } \mathit{crit}}))^\omega .$$

When allowing a somewhat sloppy notation using propositional formulae, the above expression may be rewritten into $((\neg \mathit{crit})^* . \mathit{crit})^\omega$.

Starvation freedom in the sense of "whenever process $\mathcal{P}$ is waiting then it will enter its critical section eventually later" is an ω-regular property as it can be described by

$$((\neg \mathit{wait})^* . \mathit{wait} . \mathrm{true}^* . \mathit{crit})^\omega \;+\; ((\neg \mathit{wait})^* . \mathit{wait} . \mathrm{true}^* . \mathit{crit})^* . (\neg \mathit{wait})^\omega$$

which is a short form for the ω-regular expression over $AP = \{ \mathit{wait}, \mathit{crit} \}$ that results by replacing $\neg \mathit{wait}$ with $\{\} + \{\mathit{crit}\}$, wait with $\{\mathit{wait}\} + \{\mathit{wait}, \mathit{crit}\}$, true with $\{\} + \{\mathit{crit}\} + \{\mathit{wait}\} + \{\mathit{wait}, \mathit{crit}\}$, and crit with $\{\mathit{crit}\} + \{\mathit{wait}, \mathit{crit}\}$. Intuitively, the first summand in the above expression stands for the case where $\mathcal{P}$ requests and enters its critical section infinitely often, while the second summand stands for the case where $\mathcal{P}$ is in its waiting phase only finitely many times. ■

4.3.2 Nondeterministic Büchi Automata

The issue now is to provide a kind of automaton that is suited for accepting ω-regular languages. Finite automata are not adequate for this purpose as they operate on finite

words, while we need an acceptor for infinite words. Automata models that recognize languages of infinite words are called ω-automata. The accepting runs of an ω-automaton have to "check" the entire input word (and not just a finite prefix thereof), and thus have to be infinite. This implies that acceptance criteria for infinite runs are needed.

In this monograph, the simplest variant of ω-automata, called *nondeterministic Büchi automata* (NBAs), suffices. The syntax of NBAs is exactly the same as for nondeterministic finite automata (NFAs). NBAs and NFAs differ, however, in their semantics: the accepted language of an NFA $\mathcal{A}$ is a language of finite words, i.e., $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$, whereas the accepted language of NBA $\mathcal{A}$ (denoted $\mathcal{L}_\omega(\mathcal{A})$ is an ω-language, i.e., $\mathcal{L}_\omega(\mathcal{A}) \subseteq \Sigma^\omega$. The intuitive meaning of the acceptance criterion named after Büchi is that the accept set of $\mathcal{A}$ (i.e., the set of accept states in $\mathcal{A}$) has to be visited infinitely often. Thus, the accepted language $\mathcal{L}_\omega(\mathcal{A})$ consists of all infinite words that have a run in which some accept state is visited infinitely often.

Definition 4.27. Nondeterministic Büchi Automaton (NBA)

A *nondeterministic Büchi automaton* (NBA) $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function,
- $Q_0 \subseteq Q$ is a set of initial states, and
- $F \subseteq Q$ is a set of *accept* (or: final) states, called the *acceptance set*.

A run for $\sigma = A_0 A_1 A_2 \ldots \in \Sigma^\omega$ denotes an infinite sequence $q_0\, q_1\, q_2 \ldots$ of states in $\mathcal{A}$ such that $q_0 \in Q_0$ and $q_i \xrightarrow{A_i} q_{i+1}$ for $i \geqslant 0$. Run $q_0\, q_1\, q_2 \ldots$ is *accepting* if $q_i \in F$ for infinitely many indices $i \in \mathbb{N}$. The *accepted language* of $\mathcal{A}$ is

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \{\, \sigma \in \Sigma^\omega \mid \text{there exists an accepting run for } \sigma \text{ in } \mathcal{A} \,\}.$$

The size of $\mathcal{A}$, denoted $|\mathcal{A}|$, is defined as the number of states and transitions in $\mathcal{A}$. ■

As for an NFA, we identify the transition function δ with the induced transition relation $\to \subseteq Q \times \Sigma \times Q$ which is given by

$$q \xrightarrow{A} p \;\text{ if and only if }\; p \in \delta(q, A).$$

Since the state space Q of an NBA $\mathcal{A}$ is finite, each run for an infinite word $\sigma \in \Sigma^\omega$ is infinite, and hence visits some state $q \in Q$ infinitely often. Acceptance of a run depends on whether or not the set of all states that appear infinitely often in the given run contains an accept state. The definition of an NBA allows for the special case where $F = \varnothing$, which means that there are no accept states. Clearly, in this case, no run is accepting. Thus $\mathcal{L}_\omega(\mathcal{A}) = \varnothing$ if $F = \varnothing$. There are also no accepting runs whenever, $Q_0 = \varnothing$ as in this case, no word has a run.

Example 4.28.

Consider the NBA of Figure 4.7 with the alphabet $\Sigma = \{A, B, C\}$. The word C^ω has only

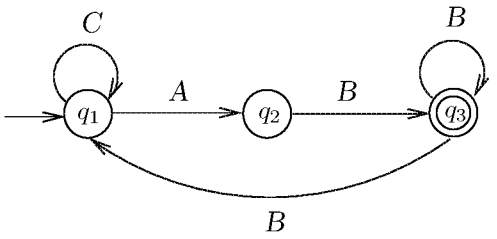


Figure 4.7: An example of an NBA.

one run in $\mathcal{A}$, namely $q_1\, q_1\, q_1\, q_1\, \ldots$, or in short, q_1^ω. Some other runs are $q_1\, q_2\, q_3^\omega$ for the word AB^ω, $(q_1\, q_1\, q_2\, q_3)^\omega$ for the word $(CABB)^\omega$, and $(q_1\, q_2\, q_3)^n\, q_1^\omega$ for the word $(ABB)^n C^\omega$ where $n \geqslant 0$.

The runs that go infinitely often through the accept state q_3 are accepting. For instance, $q_1\, q_2\, q_3^\omega$ and $(q_1\, q_1\, q_2\, q_3)^\omega$ are accepting runs. q_1^ω is not an accepting run as it never visits the accept state q_3, while runs of the form $(q_1\, q_2\, q_3)^n\, q_1^\omega$ are not accepting as they visit the accept state q_3 only finitely many times. The language accepted by this NBA is given by the ω-regular expression:

$$C^* AB \left(B^+ \;+\; BC^* AB\right)^\omega$$

■

Later in this chapter (page 198 ff.), NBAs are used for the verification of ω-regular properties – in the same vein as NFAs were exploited for the verification of regular safety properties. In that case, Σ is of the form $\Sigma = 2^{AP}$. As explained on page 159, propositional logic formulae are used as a shorthand notation for the transitions of such NBAs. For instance, if $AP = \{a, b\}$, then the label $a \vee b$ for an edge from q to p means that there are three transitions from q to p: one for the symbol $\{a\}$, one for the symbol $\{b\}$, and one for the symbol $\{a, b\}$.

Example 4.29. Infinitely Often Green

Let $AP = \{ green, red \}$ or any other set containing the proposition *green*. The language of words $\sigma = A_0 A_1 \ldots \in 2^{AP}$ satisfying the LT property "infinitely often *green*" is accepted by the NBA $\mathcal{A}$ depicted in Figure 4.8.

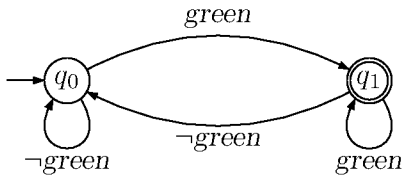


Figure 4.8: An NBA accepting "infinitely often *green*".

The automaton $\mathcal{A}$ is in the accept state q_1 if and only if the last input set of symbols (i.e., the last set A_i) contains the propositional symbol *green*. Therefore, $\mathcal{L}_\omega(\mathcal{A})$ is exactly the set of all infinite words $A_0 A_1 \ldots$ with infinitely many sets A_i with $green \in A_i$. For example, for the input word

$$\sigma \;=\; \{ green \}\, \{\}\, \{ green \}\, \{\}\, \{ green \}\, \{\} \ldots$$

we obtain the accepting run $q_0\, q_1\, q_0\, q_1 \ldots$. The same run $q_0\, q_1\, q_0\, q_1 \ldots$ is obtained for the word

$$\sigma' \;=\; (\{ green,\, red \}\, \{\}\, \{ green \}\, \{ red \})^\omega$$

or any other word $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega$ with $green \in A_{2j}$ and $green \notin A_{2j+1}$ for all $j \geqslant 0$. ■

Example 4.30. Request Response

Many liveness properties are of the form

"Whenever some event a occurs,
some event b will eventually occur in the future"

For example, the property "once a request is provided, eventually a response occurs" is of this form. An associated NBA with propositions *req* and *resp* is indicated in Figure 4.9. It is assumed that $\{ req, resp \} \subseteq AP$, i.e., we assume the NBA to have alphabet 2^{AP} with AP containing at least *req* and *resp*. It is not difficult to see that this NBA accepts exactly those sequences in which each request is always eventually followed by a response. Note that an infinite trace in which only responses occur, but never a request (or finitely many requests) is also accepting. ■

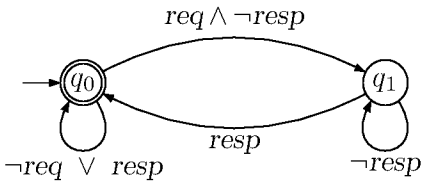


Figure 4.9: An NBA accepting "on each request, eventually a response is provided".

Remark 4.31. NBA and Regular Safety Properties

In Section 4.2, we have seen that there is a strong relationship between bad prefixes of regular safety properties and NFAs. In fact, there is also a strong relationship between NBAs and regular safety properties. This can be seen as follows. Let P_{safe} be a regular safety property over AP and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ an NFA recognizing the language of all bad prefixes of P_{safe}. Each accept state $q_F \in F$ may be assumed to be a trapping state, i.e., $q_F \xrightarrow{A} q_F$ for all $A \subseteq AP$. This assumption is justified since each extension of a bad prefix is a bad prefix. (As a bad prefix contains a "bad" event that causes the violation of P_{safe}, each extension of this prefix contains this event.)

When interpreting $\mathcal{A}$ as an NBA, it accepts exactly the infinite words $\sigma \in (2^{AP})^\omega$ that violate P_{safe}, i.e.,

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; (2^{AP})^\omega \setminus P_{safe}.$$

Here, it is important that $\mathcal{A}$ accepts all bad prefixes, and not just the minimal ones (see Exercise 4.18).

If $\mathcal{A}$ is a total deterministic automaton, i.e., in each state there is a single possible transition for each input symbol, then the NBA obtained by

$$\overline{\mathcal{A}} \;=\; \Big(Q, 2^{AP}, \delta, Q_0, Q \setminus F\Big)$$

accepts the language $\mathcal{L}_\omega\left(\overline{\mathcal{A}}\right) = P_{safe}$.

This is exemplified by means of a concrete case. Consider again the property "a red light phase should be immediately preceded by a yellow light phase" for a traffic light system. We have seen before (see Example 4.13 on page 161) that the bad prefixes of this safety property constitute a regular language and are accepted by the NFA shown in Figure 4.10. Note that this NFA is total. Applying the procedure described just above to this automaton yields the NBA depicted in Figure 4.11. It is easy to see that the infinite language accepted by this NBA consists exactly of all sequences of the form $\sigma = A_0\, A_1\, A_2 \ldots$ such that $\mathit{red} \in A_j$ implies $j > 0$ and $\mathit{yellow} \in A_{j-1}$. ■

The accepted languages of the NBA examples have so far been ω-regular. It is now shown

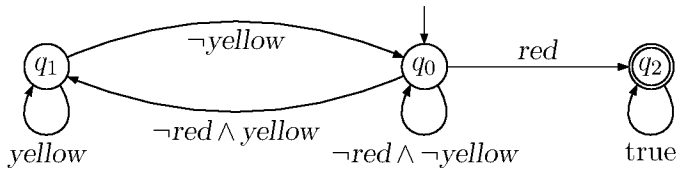


Figure 4.10: An NFA for the set of all bad prefixes of P_{safe}.

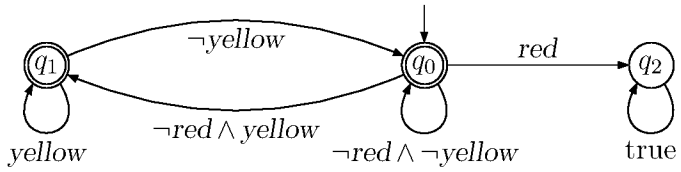


Figure 4.11: An NBA for the LT property "red should be preceded by yellow".

that this holds for any NBA. Moreover, it will be shown that any ω-regular language can be described by an NBA. Thus, *NBAs are as expressive as ω-regular languages.* This result is analogous to the fact that NFAs are as expressive as regular languages, and thus may act as an alternative formalism to describe regular languages. In the same spirit, NBAs are an alternative formalism for describing ω-regular languages. This is captured by the following theorem.

Theorem 4.32. NBAs and ω-Regular Languages

The class of languages accepted by NBAs agrees with the class of ω-regular languages.

The proof of Theorem 4.32 amounts to showing that (1) any ω-regular language is recognized by an NBA (see Corollary 4.38 on page 182) and (2) that the language $\mathcal{L}_\omega(\mathcal{A})$ accepted by the NBA $\mathcal{A}$ is ω-regular (see Lemma 4.39 on page 183).

We first consider the statement that ω-regular languages are contained in the class of languages recognized by an NBA. The proof of this fact is divided into the following three steps that rely on operations for NBAs to mimic the building blocks of ω-regular expressions:

(1) For any NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ there exists an NBA accepting $\mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$.

(2) For any regular language $\mathcal{L}$ (of finite words) with $\varepsilon \notin \mathcal{L}$ there exists an NBA accepting $\mathcal{L}^\omega$.

(3) For regular language $\mathcal{L}$ and NBA $\mathcal{A}'$ there exists an NBA accepting $\mathcal{L}.\mathcal{L}_\omega(\mathcal{A}')$.

These three results that are proven below form the basic ingredients to construct an NBA for a given ω-regular expression $\mathsf{G} = \mathsf{E}_1.\mathsf{F}_1^\omega + \ldots + \mathsf{E}_n.\mathsf{F}_n^\omega$ with $\varepsilon \notin \mathsf{F}_i$. This works as follows. As an initial step, (2) is exploited to construct NBA $\mathcal{A}'_1, \ldots, \mathcal{A}'_n$ for the expressions $\mathsf{F}_1^\omega, \ldots, \mathsf{F}_n^\omega$. Then, (3) is used to construct an NBA for the expressions $\mathsf{E}_i.\mathsf{F}_i^\omega$, for $1 \leqslant i \leqslant n$. Finally, these NBA are combined using (1) to obtain an NBA for G.

Let us start with the union operator on two NBAs. Let $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, Q_{0,1}, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, Q_{0,2}, F_2)$ be NBAs over the same alphabet Σ. Without loss of generality, it may be assumed that the state spaces Q_1 and Q_2 of $\mathcal{A}_1$ and $\mathcal{A}_2$ are disjoint, i.e., $Q_1 \cap Q_2 = \varnothing$. Let $\mathcal{A}_1 + \mathcal{A}_2$ be the NBA with the joint state spaces of $\mathcal{A}_1$ and $\mathcal{A}_2$, and with all transitions in $\mathcal{A}_1$ and $\mathcal{A}_2$. The initial states of $\mathcal{A}$ are the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$, and similarly, the accept states of $\mathcal{A}$ are the accept states of $\mathcal{A}_1$ and $\mathcal{A}_2$. That is,

$$\mathcal{A}_1 + \mathcal{A}_2 \;=\; (Q_1 \cup Q_2, \Sigma, \delta, Q_{0,1} \cup Q_{0,2}, F_1 \cup F_2)$$

where $\delta(q, A) = \delta_i(q, A)$ if $q \in Q_i$ for i=1, 2. Clearly, any accepting run in $\mathcal{A}_i$ is also an accepting run in $\mathcal{A}_1 + \mathcal{A}_2$, and vice versa, each accepting run in $\mathcal{A}_1 + \mathcal{A}_2$ is an accepting run in either $\mathcal{A}_1$ or $\mathcal{A}_2$. This yields $\mathcal{L}_\omega(\mathcal{A}_1 + \mathcal{A}_2) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$. We thus obtain:

Lemma 4.33. Union Operator on NBA

For NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ (both over the alphabet Σ) there exists an NBA $\mathcal{A}$ such that:

$$\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2) \quad \textit{and} \quad |\mathcal{A}| = \mathcal{O}(|\mathcal{A}_1| + |\mathcal{A}_2|).$$

Now consider (2). We will show that for any regular language $\mathcal{L} \subseteq \Sigma^*$ there exists an NBA over the alphabet Σ that accepts the ω-regular language $\mathcal{L}^\omega$. To do so, we start with a representation of $\mathcal{L}$ by an NFA $\mathcal{A}$.

Lemma 4.34. ω-Operator for NFA

For each NFA $\mathcal{A}$ with $\varepsilon \notin \mathcal{L}(\mathcal{A})$ there exists an NBA $\mathcal{A}'$ such that

$$\mathcal{L}_\omega(\mathcal{A}') = \mathcal{L}(\mathcal{A})^\omega \quad \textit{and} \quad |\mathcal{A}'| = \mathcal{O}(|\mathcal{A}|).$$

Proof: Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA with $\varepsilon \notin \mathcal{L}(\mathcal{A})$. Without loss of generality, we may assume that all initial states in $\mathcal{A}$ have no incoming transitions and are not accepting.

Any $\mathcal{A}$ that does not possess this property, can be modified into an equivalent NFA as follows. Add a new initial (nonaccept) state q_{new} to Q with the transitions $q_{new} \xrightarrow{A} q$ if and only if $q_0 \xrightarrow{A} q$ for some initial state $q_0 \in Q_0$. All other transitions, as well as the accept states, remain unchanged. The state q_{new} is the single initial state of the modified NFA, is not accept, and, clearly, has no incoming transitions. This modification neither affects the accepted language nor the asymptotic size of $\mathcal{A}$.

In the sequel, we assume that $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ is an NFA such that the states in Q_0 do not have any incoming transitions and $Q_0 \cap F = \varnothing$. We now construct an NBA $\mathcal{A}' = (Q, \Sigma, \delta', Q_0', F')$ with $\mathcal{L}_\omega(\mathcal{A}') = \mathcal{L}(\mathcal{A})^\omega$. The basic idea of the construction of $\mathcal{A}'$ is to add for any transition in $\mathcal{A}$ that leads to an accept state new transitions leading to the initial states of $\mathcal{A}$. Formally, the transition relation δ' in the NBA $\mathcal{A}'$ is given by

$$\delta'(q, A) \;=\; \begin{cases} \delta(q, A) & \text{if } \delta(q, A) \cap F = \varnothing \\ \delta(q, A) \cup Q_0 & \text{otherwise.} \end{cases}$$

The initial states in the NBA $\mathcal{A}'$ agree with the initial states in $\mathcal{A}$, i.e., $Q_0' = Q_0$. These are also the accept states in $\mathcal{A}'$, i.e., $F' = Q_0$.

Let us check that $\mathcal{L}_\omega(\mathcal{A}') = \mathcal{L}(\mathcal{A})^\omega$. This is proven as follows.

$\subseteq$: Assume that $\sigma \in \mathcal{L}_\omega(\mathcal{A}')$ and let $q_0 q_1 q_2 \ldots$ be an accepting run for σ in $\mathcal{A}'$. Hence, $q_i \in F' = Q_0$ for infinitely many indices i. Let $i_0 = 0 < i_1 < i_2 < \ldots$ be the strictly increasing sequence of natural numbers with $\{ q_{i_0}, q_{i_1}, q_{i_2}, \ldots \} \subseteq Q_0$ and $q_j \notin Q_0$ for all $j \in \mathbb{N} \setminus \{ i_0, i_1, i_2, \ldots \}$. The word σ can be divided into infinitely many nonempty finite subwords $w_i \in \Sigma^*$ yielding $\sigma = w_1 w_2 w_3 \ldots$ such that $q_{i_k} \in \delta'^*(q_{i_{k-1}}, w_k)$ for all $k \geqslant 1$. (The extension of δ' to a function $\delta'^* : Q \times \Sigma^* \to 2^Q$ is as for an NFA, see page 154.) By definition of $\mathcal{A}'$ and since the states $q_{i_k} \in Q_0$ do not have any predecessor in $\mathcal{A}$, we get $\delta^*(q_{i_{k-1}}, w_k) \cap F \neq \varnothing$. This yields $w_k \in \mathcal{L}(\mathcal{A})$ for all $k \geqslant 1$, which gives us $\sigma \in \mathcal{L}(\mathcal{A})^\omega$.

$\supseteq$: Let $\sigma = w_1 w_2 w_3 \ldots \in \Sigma^\omega$ such that $w_k \in \mathcal{L}(\mathcal{A})$ for all $k \geqslant 1$. For each k, we choose an accepting run $q_0^k q_1^k \ldots q_{n_k}^k$ for w_k in $\mathcal{A}$. Hence, $q_0^k \in Q_0$ and $q_{n_k}^k \in F$. By definition of $\mathcal{A}'$, we have $q_0^{k+1} \in \delta'^*(q_0^k, w_k)$ for all $k \geqslant 1$. Thus,

$$q_0^1 \ldots q_{n_1-1}^1 q_0^2 \ldots q_{n_2-1}^2 q_0^3 \ldots q_{n_3-1}^3 \cdots$$

is an accepting run for σ in $\mathcal{A}'$. Hence, $\sigma \in \mathcal{L}_\omega(\mathcal{A}')$. ∎

Example 4.35. ω-Operator for an NFA

Consider the NFA depicted in the left upper part of Figure 4.12. It accepts the language $A^* B$. In order to obtain an NBA recognizing $(A^* B)^\omega$, we first apply the transformation

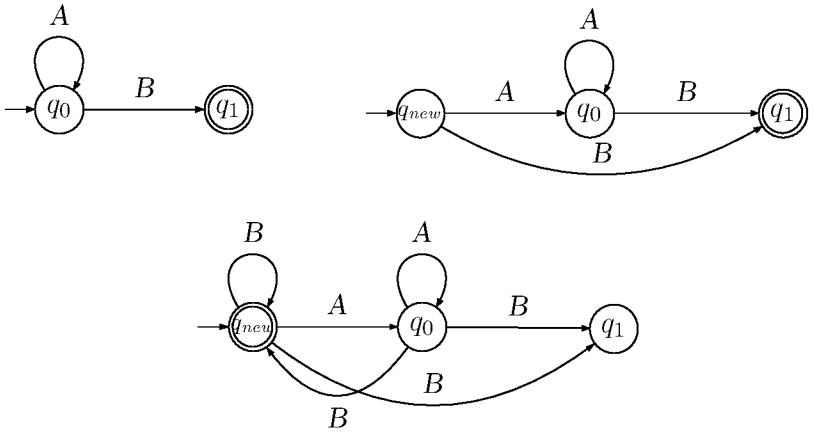


Figure 4.12: From an NFA accepting A^*B to an NBA accepting $(A^*B)^\omega$.

as described in the proof of Lemma 4.34 to remove initial states that have an incoming transition. This yields the NFA depicted in the right upper part of Figure 4.12. This automaton can now be used to apply the construction of the required NBA as detailed in the proof of Lemma 4.34. This yields the NBA depicted in the lower part of Figure 4.12. ■

It remains to provide a construction for task (3) above. Assume that we have NFA $\mathcal{A}$ for the regular language $\mathcal{L}(\mathcal{A})$ and a given NBA $\mathcal{A}'$ at our disposal. The proof of the following lemma will describe a procedure to obtain an NBA for the ω-language $\mathcal{L}(\mathcal{A}).\mathcal{L}_\omega(\mathcal{A}')$.

Lemma 4.36. Concatenation of an NFA and an NBA

For NFA $\mathcal{A}$ and NBA $\mathcal{A}'$ (both over the alphabet Σ), there exists an NBA $\mathcal{A}''$ with

$$\mathcal{L}_\omega(\mathcal{A}'') = \mathcal{L}(\mathcal{A}).\mathcal{L}_\omega(\mathcal{A}') \quad \textit{and} \quad |\mathcal{A}''| = \mathcal{O}(|\mathcal{A}| + |\mathcal{A}'|).$$

Proof: Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NFA and $\mathcal{A}' = (Q', \Sigma, \delta', Q_0', F')$ an NBA with $Q \cap Q' = \varnothing$. Let $\mathcal{A}'' = (Q'', \Sigma, \delta'', Q_0'', F'')$ be the following NBA. The state space is $Q'' = Q \cup Q'$. The set of initial and accept states are given by

$$Q_0'' = \begin{cases} Q_0 & \text{if } Q_0 \cap F = \varnothing \\ Q_0 \cup Q_0' & \text{otherwise,} \end{cases}$$

and $F'' = F'$ (set of accept state in the NBA $\mathcal{A}'$). The transition function δ'' is given by

$$\delta''(q,A) \;=\; \begin{cases} \delta(q,A) & \text{if } q \in Q \text{ and } \delta(q,A) \cap F = \varnothing \\ \delta(q,A) \cup Q'_0 & \text{if } q \in Q \text{ and } \delta(q,A) \cap F \neq \varnothing \\ \delta'(q,A) & \text{if } q \in Q' \end{cases}$$

It is now easy to check that $\mathcal{A}''$ fulfills the desired conditions. ■

Example 4.37. Concatenation of an NFA and an NBA

Consider the NFA $\mathcal{A}$ and the NBA $\mathcal{A}'$ depicted in the left and right upper part of Figure 4.13, respectively. We have $\mathcal{L}(\mathcal{A}) = (AB)^*$ and $\mathcal{L}(\mathcal{A}') = (A{+}B)^*BA^\omega$. Applying the transformation as described in Lemma 4.36 yields the NBA depicted in the lower part of Figure 4.13. It is not difficult to assess that this NBA accepts indeed the concatenated language $(AB)^*(A{+}B)^*BA^\omega$. ■

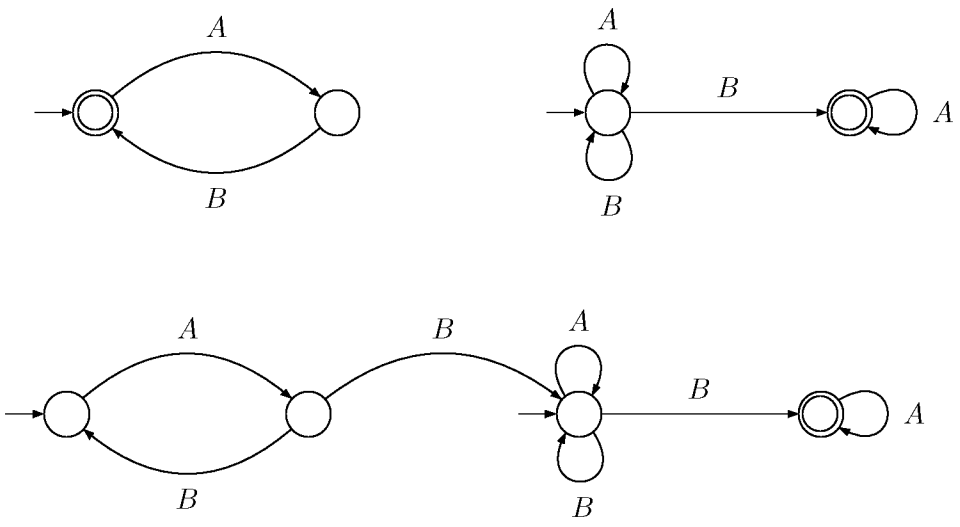


Figure 4.13: Concatenation of an NFA and an NBA.

By Lemmas 4.33, 4.34, and 4.36 we obtain the first part for the proof of Theorem 4.32:

Corollary 4.38. NBA for ω-Regular Languages

For any ω-regular language $\mathcal{L}$ there exists an NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}$.

The proof of the following lemma shows that the languages accepted by NBA can be described by an ω-regular expression.

Lemma 4.39. NBAs Accept ω-Regular Languages

For each NBA $\mathcal{A}$, the accepted language $\mathcal{L}_\omega(\mathcal{A})$ is ω-regular.

Proof: Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. For states $q, p \in Q$, let $\mathcal{A}_{qp}$ be the NFA $(Q, \Sigma, \delta, \{q\}, \{p\})$. Then, $\mathcal{A}_{qp}$ recognizes the regular language consisting of all finite words $w \in \Sigma^*$ that have a run in $\mathcal{A}$ leading from q to p, that is,

$$\mathcal{L}_{qp} \stackrel{\text{def}}{=} \mathcal{L}(\mathcal{A}_{qp}) = \{\, w \in \Sigma^* \mid p \in \delta^*(q, w) \,\}.$$

Consider a word $\sigma \in \mathcal{L}_\omega(\mathcal{A})$ and an accepting run $q_0 q_1 \ldots$ for σ in $\mathcal{A}$. Some accept state $q \in F$ appears infinitely often in this run. Hence, we may split σ into nonempty finite subwords $w_0, w_1, w_2, w_3, \ldots \in \Sigma^*$ such that $w_0 \in \mathcal{L}_{q_0 q}$ and $w_k \in \mathcal{L}_{qq}$ for all $k \geqslant 1$ and

$$\sigma \;=\; \underbrace{w_0}_{\in \mathcal{L}_{q_0 q}} \underbrace{w_1}_{\in \mathcal{L}_{qq}} \underbrace{w_2}_{\in \mathcal{L}_{qq}} \underbrace{w_3}_{\in \mathcal{L}_{qq}} \cdots\cdots$$

On the other hand, any infinite word σ which has the form $\sigma = w_0 w_1 w_2 \ldots$ where the w_k's are nonempty finite words with $w_0 \in \mathcal{L}_{q_0 q}$ for some initial state $q_0 \in Q_0$ and $\{w_1, w_2, w_3, \ldots\} \subseteq \mathcal{L}_{qq}$ for some accept state $q \in F$ has an accepting run in $\mathcal{A}$. This yields

$$\sigma \in \mathcal{L}_\omega(\mathcal{A}) \quad \text{if and only if} \quad \exists q_0 \in Q_0 \; \exists q \in F.\;\; \sigma \in \mathcal{L}_{q_0 q} \, (\mathcal{L}_{qq} \setminus \{\varepsilon\})^\omega \,.$$

Hence, $\mathcal{L}_\omega(\mathcal{A})$ agrees with the language

$$\bigcup_{q_0 \in Q_0, q \in F} \mathcal{L}_{q_0 q} . \, (\mathcal{L}_{qq} \setminus \{\varepsilon\})^\omega$$

which is ω-regular. ∎

Example 4.40. From NBA to ω-Regular Expression

For the NBA $\mathcal{A}$ shown in Figure 4.7 on page 175, a corresponding ω-regular expression is obtained by

$$\mathcal{L}_{q_1 q_3} . (\mathcal{L}_{q_3 q_3} \setminus \{\varepsilon\})^\omega$$

since q_1 is the unique initial state and q_3 the unique accept state in $\mathcal{A}$. The regular language $\mathcal{L}_{q_3 q_3} \setminus \{\varepsilon\}$ can be described by the expression $(B^+ + BC^*AB)^+$, while $\mathcal{L}_{q_1 q_3}$ is given by $(C^*AB(B^* + BC^*AB)^*B)^*C^*AB$. Hence, $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathsf{G})$ where G is the ω-regular expression:

$$\mathsf{G} \;=\; \underbrace{(C^*AB(B^* + BC^*AB)^*B)^*C^*AB}_{\mathcal{L}_{q_1 q_3}} \underbrace{((B^+ + BC^*AB)^+)^\omega}_{(\mathcal{L}_{q_3 q_3} \setminus \{\varepsilon\})^\omega} .$$

The thus obtained expression G can be simplified to the equivalent expression:

$$C^*AB(B^+ + BC^*AB)^\omega.$$

■

The above lemma, together with Corollary 4.38, completes the proof of Theorem 4.32 stating the equivalence of the class of languages accepted by NBAs and the class of all ω-regular languages. Thus, NBAs and ω-regular languages are equally expressive.

A fundamental question for any type of automata model is the question whether for a given automaton $\mathcal{A}$ the accepted language is empty. For nondeterministic Büchi automata, an analysis of the underlying directed graph by means of standard graph algorithms is sufficient, as we will show now.

Lemma 4.41. Criterion for the Nonemptiness of an NBA

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. Then, the following two statements are equivalent:

(a) $\mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$,

(b) There exists a reachable accept state q that belongs to a cycle in $\mathcal{A}$. Formally,

$$\exists q_0 \in Q_0 \; \exists q \in F \; \exists w \in \Sigma^* \; \exists v \in \Sigma^+. \;\; q \in \delta^*(q_0, w) \cap \delta^*(q, v).$$

Proof: (a) $\Longrightarrow$ (b): Let $\sigma = A_0 A_1 A_2 \ldots \in \mathcal{L}_\omega(\mathcal{A})$ and let $q_0 q_1 q_2 \ldots$ be an accepting run for σ in $\mathcal{A}$. Let $q \in F$ be an accept state with $q = q_i$ for infinitely many indices i. Let i and j be two indices with $0 \leqslant i < j$ and $q_i = q_j = q$. We consider the finite words $w = A_0 A_1 \ldots A_{i-1}$ and $v = A_i A_{i+1} \ldots A_{j-1}$ and obtain $q = q_i \in \delta^*(q_0, w)$ and $q = q_j \in \delta^*(q_i, v) = \delta^*(q, v)$. Hence, (b) holds.

(b) $\Longrightarrow$ (a): Let q_0, q, w, v be as in statement (b). Then, the infinite word $\sigma = wv^\omega$ has a run of the form $q_0 \ldots q \ldots q \ldots q \ldots$ that infinitely often contains q. Since $q \in F$ this run is accepting which yields $\sigma \in \mathcal{L}_\omega(\mathcal{A})$, and thus, $\mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. ■

By the above lemma, the emptiness problem for NBAs can be solved by means of graph algorithms that explore all reachable states and check whether they belong to a cycle. One possibility to do so is to calculate the strongly connected components of the underlying directed graph of $\mathcal{A}$ and to check whether there is at least one nontrivial strongly con-

nected component[3] that is reachable (from at least one of the initial states) and contains an accept state. Since the strongly connected components of a (finite) directed graph can be computed in time linear in the number of states and edges, the time complexity of this algorithm for the emptiness check of NBA $\mathcal{A}$ is linear in the size of $\mathcal{A}$. An alternative algorithm that also runs in time linear in the size of $\mathcal{A}$, but avoids the explicit computation of the strongly connected components, can be derived from the results stated in Section 4.4.2.

Theorem 4.42. Checking Emptiness for NBA

The emptiness problem for NBA $\mathcal{A}$ can be solved in time $\mathcal{O}(|\mathcal{A}|)$.

Since NBAs serve as a formalism for ω-regular languages, we may identify two Büchi automata for the same language:

Definition 4.43. Equivalence of NBA

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two NBAs with the same alphabet. $\mathcal{A}_1$ and $\mathcal{A}_2$ are called *equivalent*, denoted $\mathcal{A}_1 \equiv \mathcal{A}_2$, if $\mathcal{L}_\omega(\mathcal{A}_1) = \mathcal{L}_\omega(\mathcal{A}_2)$. ■

Example 4.44. Equivalent NBA

As for other finite automata, equivalent NBAs can have a totally different structure. For example, consider the NBA shown in Figure 4.14 over the alphabet 2^{AP} where $AP = \{a, b\}$. Both NBAs represent the liveness property "infinitely often a and infinitely often b", and thus, they are equivalent. ■

Remark 4.45. NFA vs. NBA Equivalence

It is interesting to consider more carefully the relationship between the notions of equivalence of NFAs and NBAs. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two automata that we can regard as either NFA or as NBA. To distinguish the equivalence symbol $\equiv$ for NFAs from that for NBAs we will write in this example $\equiv_{NFA}$ to denote the equivalence relation for NFA and the symbol $\equiv_{NBA}$ to denote the equivalence relation for NBA, i.e., $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$ iff $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$ and $\mathcal{A}_1 \equiv_{NBA} \mathcal{A}_2$ iff $\mathcal{L}_\omega(\mathcal{A}_1) = \mathcal{L}_\omega(\mathcal{A}_2)$.

1. If $\mathcal{A}_1$ and $\mathcal{A}_2$ accept the same finite words, i.e., $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$, then this does not mean that they also accept the same infinite words. The following two automata examples show this:

[3] A strongly connected component is nontrivial if it contains at least one edge. In fact, any cycle is contained in a nontrivial strongly connected component, and vice versa, any nontrivial strongly connected component contains a cycle that goes through all its states.

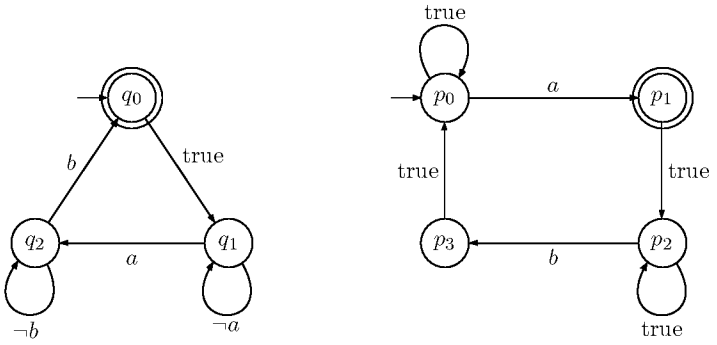
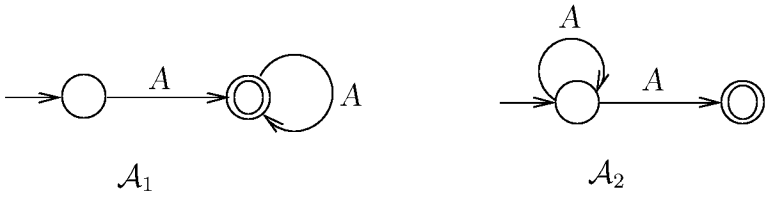
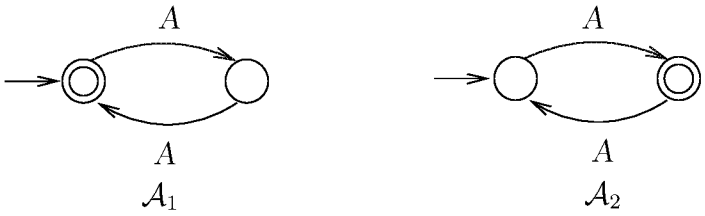


Figure 4.14: Two equivalent NBA.



We have $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2) = \{ A^n \mid n \geqslant 1 \}$, but $\mathcal{L}_\omega(\mathcal{A}_1) = \{ A^\omega \}$ and $\mathcal{L}_\omega(\mathcal{A}_2) = \varnothing$. Thus, $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$ but $\mathcal{A}_1 \not\equiv_{NBA} \mathcal{A}_2$.

2. If $\mathcal{A}_1$ and $\mathcal{A}_2$ accept the same infinite words, i.e., $\mathcal{A}_1 \equiv_{NBA} \mathcal{A}_2$, then one might expect that they would also accept the same finite words. This also turns out not to be true. The following example shows this:



We have $\mathcal{L}_\omega(\mathcal{A}_1) = \mathcal{L}_\omega(\mathcal{A}_2) = \{ A^\omega \}$, but $\mathcal{L}(\mathcal{A}_1) = \{ A^{2n} \mid n \geqslant 0 \}$ and $\mathcal{L}(\mathcal{A}_2) = \{ A^{2n+1} \mid n \geqslant 0 \}$.

3. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are both deterministic (see Definition 4.9 on page 156), then $\mathcal{A}_1 \equiv_{NFA} \mathcal{A}_2$ implies $\mathcal{A}_1 \equiv_{NBA} \mathcal{A}_2$. The reverse is, however, not true, as illustrated by the previous example.

■

For technical reasons, it is often comfortable to assume for an NBA that for each state q and for each input symbol A, there is a possible transition. Such an NBA can be seen to be nonblocking since no matter how the nondeterministic choices are resolved, the automaton cannot fail to consume the current input symbol.

Definition 4.46. Nonblocking NBA

Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. $\mathcal{A}$ is called *nonblocking* if $\delta(q, A) \neq \varnothing$ for all states q and all symbols $A \in \Sigma$. ■

Note that for a given nonblocking NBA $\mathcal{A}$ and input word $\sigma \in \Sigma^\omega$, there is at least one (infinite) possibly nonaccepting run for σ in $\mathcal{A}$. The following remark demonstrates that it is not a restriction to assume a nonblocking NBA.

Remark 4.47. Nonblocking NBA

For each NBA $\mathcal{A}$ there exists a nonblocking NBA $trap(\mathcal{A})$ with $|trap(\mathcal{A})| = \mathcal{O}(|\mathcal{A}|)$ and $\mathcal{A} \equiv trap(\mathcal{A})$.

Let us see how such a nonblocking NBA can be derived from $\mathcal{A}$. NBA $trap(\mathcal{A})$ is obtained from $\mathcal{A}$ by inserting a nonaccept trapping state q_{trap} equipped with a self-loop for each symbol in the alphabet Σ. For every symbol $A \in \Sigma$ for which state q in $\mathcal{A}$ does not have an outgoing transition, a transition to q_{trap} is added. Formally, if $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$, then $trap(\mathcal{A}) = (Q', \Sigma, \delta', Q_0', F')$ as follows. Here, $Q' = Q \cup \{ q_{trap} \}$ where q_{trap} is a new state (not in Q) that will be reached in $\mathcal{A}'$ whenever $\mathcal{A}$ does not have a corresponding transition. Formally, the transition relation δ' of $trap(\mathcal{A})$ is defined by:

$$\delta'(q, A) = \begin{cases} \delta(q, A) & \text{if } q \in Q \text{ and } \delta(q, A) \neq \varnothing \\ \{ q_{trap} \} & \text{otherwise} \end{cases}$$

The initial and accept states are unchanged, i.e., $Q_0' = Q_0$ and $F' = F$. By definition, $trap(\mathcal{A})$ is nonblocking and – since the new trap state is nonaccepting – is equivalent to $\mathcal{A}$. ■

We conclude this subsection on automata over infinite words with a few more comments on Büchi automata and ω-regular languages. We first study the subclass of deterministic Büchi automata (Section 4.3.3 below) and then in Section 4.3.4 the class of NBA with a more general acceptance condition consisting of several acceptance sets that have to be visited infinitely often.

4.3.3 Deterministic Büchi Automata

An important difference between finite-state automata and Büchi automata is the expressive power of deterministic and nondeterministic automata. While for languages of finite words, DFAs and NFAs have the same expressiveness, this does not hold for Büchi automata.

The definition of a deterministic Büchi automaton is the same as for a DFA:

Definition 4.48. Deterministic Büchi Automaton (DBA)
Let $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ be an NBA. $\mathcal{A}$ is called *deterministic*, if

$$|Q_0| \leqslant 1 \quad \text{and} \quad |\delta(q, A)| \leqslant 1$$

for all $q \in Q$ and $A \in \Sigma$. $\mathcal{A}$ is *total* if $|Q_0| = 1$ and $|\delta(q, A)| = 1$ for all $q \in Q$ and $A \in \Sigma$. ■

Obviously, the behavior of a DBA for a given input word is deterministic: either eventually the DBA will get stuck in some state as it fails to consume the current input symbol or there is a unique (infinite) run for the given input word. Total DBAs rule out the first alternative and ensure the existence of a unique run for every input word $\sigma \in \Sigma^\omega$.

Example 4.49. DBA for LT Properties
Figure 4.15 shows the DBA $\mathcal{A}'$ (on the left) and the NBA $\mathcal{A}$ (on the right) over the alphabet $\Sigma = 2^{AP}$ where $AP = \{a, b\}$. These automata are equivalent since both represent the LT property "always b and infinitely often a". Let δ be the transition function of $\mathcal{A}$ and δ'

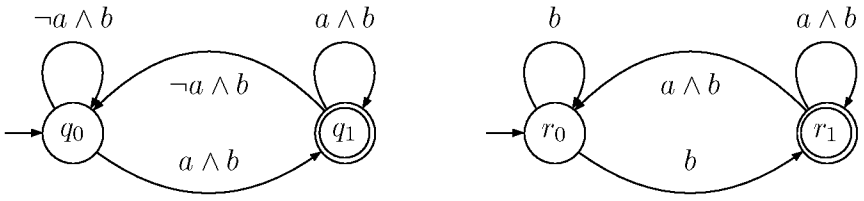


Figure 4.15: An equivalent DBA $\mathcal{A}'$ (left) and NBA $\mathcal{A}$ (right).

the transition function of $\mathcal{A}'$. The NBA $\mathcal{A}$ is not deterministic since for all input symbols containing a b, there is the possibility to move to either state r_0 or r_1. The DBA $\mathcal{A}'$ is

deterministic. Note that both $\mathcal{A}'$ and $\mathcal{A}$ are blocking, e.g., any state is blocking on an input symbol containing $\neg b$. ■

As for deterministic finite automata, the usual notation is $q' = \delta(q, A)$ (instead of $\{q'\} = \delta(q, A)$) and $\delta(q, A) = \perp$ (undefined), if $\delta(q, A) = \varnothing$. Thus, the transition relation of a DBA is understood as partial function $\delta : Q \times \Sigma \to Q$. Total DBAs are often written in the form $(Q, \Sigma, \delta, q_0, F)$ where q_0 is the unique initial state and δ is viewed as a total function $Q \times \Sigma \to Q$. Since DBA can always be extended by a nonaccept trapping state without changing the accepting language, it can be assumed without restriction that the transition relation is total. For instance, Figure 4.16 shows an equivalent total DBA for the DBA $\mathcal{A}'$ in Figure 4.15 that is obtained by adding such a trapping state.

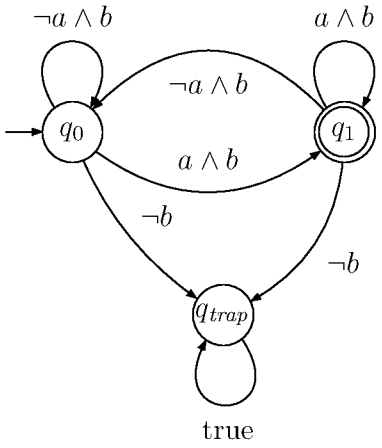


Figure 4.16: A total DBA for "always b and infinitely often a".

The transition function δ of a total DBA can be expanded to a total function $\delta^* : Q \times \Sigma^* \to Q$ in the obvious way; see also page 157 for the transition function of a total DFA. That is, let $\delta^*(q, \varepsilon) = q$, $\delta^*(q, A) = \delta(q, A)$ and

$$\delta^*(q, A_1 A_2 \ldots A_n) = \delta^*(\, \delta(q, A_1),\ A_2 \ldots A_n).$$

Then, for every infinite word $\sigma = A_0\, A_1\, A_2\, \ldots \in \Sigma^\omega$, the run $q_0\, q_1\, q_2 \ldots$ in $\mathcal{A}$ belonging to σ is given by $q_{i+1} = \delta^*(q_0, A_0 \ldots A_i)$ for all $i \geqslant 0$, where q_0 is the unique initial state of $\mathcal{A}$. In particular, for total DBA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ the accepted language is given by

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \{\; A_0\, A_1\, A_2 \ldots \in \Sigma^\omega \mid\; \delta^*(q_0, A_0 \ldots A_i) \in F \text{ for infinitely many } i \;\}$$

As we have seen before, NFAs are as expressive as deterministic ones. However, *NBAs are more expressive than deterministic ones.* That is, there do exist NBA for which there

does not exist an equivalent deterministic one. Stated differently, while any ω-language accepted by a DBA is ω-regular, there do exist ω-regular languages for which there does not exist a DBA accepting it. An example of such ω-regular language is the language given by the expression $(A+B)^*B^\omega$.

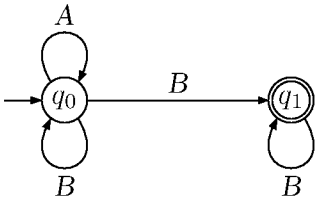


Figure 4.17: NBA for the ω-regular expression $(A + B)^*B^\omega$.

In fact, the language $\mathcal{L}_\omega(\,(A+B)^*B^\omega\,)$ is accepted by a rather simple NBA, shown in Figure 4.17. The idea of this NBA is that given an input word $\sigma = wB^\omega$ where $w \in \{A, B\}^*$ the automaton may stay in q_0 and guess nondeterministically when the suffix consisting of B's starts and then moves to the accept state q_1. This behavior, however, cannot be simulated by a DBA as formally shown in the following theorem.

Theorem 4.50. NBAs are More Powerful than DBAs

There does not exist a DBA $\mathcal{A}$ such that $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega((A + B)^*B^\omega)$.

Proof: By contradiction. Assume that $\mathcal{L}_\omega((A + B)^*B^\omega) = \mathcal{L}_\omega(\mathcal{A})$ for some DBA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with $\Sigma = \{\, A, B \,\}$. Note that since $\mathcal{A}$ is deterministic, δ^* can be considered as a function of type $Q \times \Sigma^* \to Q$.

Since the word $\sigma_1 = B^\omega$ belongs to $\mathcal{L}_\omega((A + B)^*B^\omega) = \mathcal{L}_\omega(\mathcal{A})$, there exists an accepting state $q_1 \in F$ and a $n_1 \in \mathbb{N}_{\geqslant 1}$ such that

$$(1) \qquad \delta^*\,(q_0, B^{n_1}) \;=\; q_1 \;\in F \quad .$$

(Since $\mathcal{A}$ is deterministic, q_1 is uniquely determined.) Now consider the word $\sigma_2 = B^{n_1}AB^\omega \;\in \mathcal{L}_\omega((A + B)^*B^\omega) = \mathcal{L}_\omega(\mathcal{A})$. Since σ_2 is accepted by $\mathcal{A}$, there exists an accepting state $q_2 \in F$ and $n_2 \in \mathbb{N}_{\geqslant 1}$, such that

$$(2) \qquad \delta^*\,(q_0, B^{n_1}AB^{n_2}) \;=\; q_2 \in F \quad .$$

The word $B^{n_1}AB^{n_2}AB^\omega$ is in $\mathcal{L}_\omega((A + B)^*B^\omega)$, and, thus, is accepted by $\mathcal{A}$. So, there is an accepting state $q_3 \in F$ and $n_3 \in \mathbb{N}_{\geqslant 1}$ with

$$\delta^*\,(q_0, B^{n_1}AB^{n_2}AB^{n_3}) \;=\; q_3 \in F. \tag{3}$$

Continuing this process, we obtain a sequence $n_1, n_2, n_3, \ldots$ of natural numbers $\geqslant 1$ and a sequence $q_1, q_2, q_3, \ldots$ of accepting states such that

$$\delta^*\,(q_0, B^{n_1}AB^{n_2}A \ldots B^{n_{i-1}}AB^{n_i}) \;=\; q_i \in F, \quad i \geqslant 1 \ldots$$

Since there are only finitely many states, there exist $i < j$ such that

$$\delta^*\,(q_0, B^{n_1}A \ldots AB^{n_i}) \;=\; \delta^*\,(q_0, B^{n_1}A \ldots AB^{n_i} \ldots AB^{n_j})$$

Thus $\mathcal{A}$ has an accepting run on

$$B^{n_1}A \ldots AB^{n_i}\,(AB^{n_{i+1}}A \ldots AB^{n_j})^{\omega}\,.$$

But this word has infinitely many occurrences of A, and thus does not belong to $\mathcal{L}_\omega((A+B)^*B^\omega)$. Contradiction. ■

Example 4.51. The Need for Nondeterminism

In Examples 4.29 and 4.30, we provided DBAs for LT properties. To represent liveness properties of the form "eventually forever", the concept of nondeterminism is, however, necessary. Consider the property "eventually forever a", where a is some atomic proposition. Let $\{\,a\,\} = AP$, i.e., $2^{AP} = \{A, B\}$ where $A = \{\}$ and $B = \{a\}$. Then, the linear-time property "eventually forever a" is given by the ω-regular expression

$$(A + B)^*B^\omega \;=\; (\{\} + \{a\})^*\{a\}^\omega.$$

By Theorem 4.50, there is no DBA for "eventually forever a". On the other hand, this property can be described by the NBA $\mathcal{A}$ depicted in Figure 4.18. (Note that state q_2 could be omitted, as there is no accepting run that starts in q_2.) Intuitively, $\mathcal{A}$ nondeterministically decides (by means of an omniscient oracle) from which instant the proposition a is continuously true. This behavior cannot be mimicked by a DBA. ■

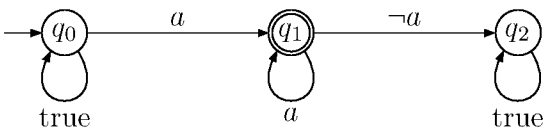


Figure 4.18: An NBA accepting "eventually forever a".

The reader might wonder why the powerset construction known for finite automata (see page 157) fails for Büchi automata. The deterministic automaton $\mathcal{A}_{det}$ obtained through

the powerset construction allows simulating the given nondeterministic automaton $\mathcal{A}$ by keeping track of the set Q' of states that are reachable in $\mathcal{A}$ for any finite prefix of the given input word. (This set Q' is a state in $\mathcal{A}_{det}$.) What is problematic here is the acceptance condition: while for NFAs the information whether an accept state is reachable is sufficient, for infinite words we need one single run that passes an accept state infinitely often. The latter is not equivalent to the requirement that a state Q' with $Q' \cap F \neq \varnothing$ is visited infinitely often, since there might be infinitely many possibilities (runs) to enter F at different time points, i.e., for different prefixes of the input word. This, in fact, is the case for the NBA in Figure 4.17. For the input word $\sigma = ABABA\ldots = (AB)^\omega$, the automaton in Figure 4.17 can enter the accept state q_1 after the second, fourth, sixth, etc., symbol by staying in q_0 for the first $2n-1$ symbols and moving with the nth B to state q_1 (for $n = 1, 2, \ldots$). Thus, at infinitely many positions there is the possibility to enter F, although there is no run that visits q_1 infinitely often, since whenever q_1 has been entered the automaton $\mathcal{A}$ rejects when reading the next A. In fact, the powerset construction applied to the NBA $\mathcal{A}$ in Figure 4.17 yields a DBA $\mathcal{A}_{det}$ with two reachable states (namely $\{q_0\}$ and $\{q_0, q_1\}$) for the language consisting of all infinite words with infinitely many B's, but not for the language given by $(A + B)^* B^\omega$.

Another example that illustrates why the powerset construction fails for Büchi automata is provided in Exercise 4.16 (page 225).

4.3.4 Generalized Büchi Automata

In several applications, other ω-automata types are useful as automata models for ω-regular languages. In fact, there are several variants of ω-automata that are equally expressive as nondeterministic Büchi automata, although they use more general acceptance conditions than the Büchi acceptance condition "visit infinitely often the acceptance set F". For some of these ω-automata types, the deterministic version has the full power of ω-regular languages. These automata types are not relevant for the remaining chapters of this monograph and will not be treated here. [4]

For the purposes of this monograph, it suffices to consider a slight variant of nondeterministic Büchi automata, called *generalized* nondeterministic Büchi automata, or GNBA for short. The difference between an NBA and a GNBA is that the acceptance condition for a GNBA requires to visit several sets $F_1, \ldots, F_k$ infinitely often. Formally, the syntax of a GNBA is as for an NBA, except that the acceptance condition is a set $\mathcal{F}$ consisting of *finitely many acceptance sets* $F_1, \ldots, F_k$ with $F_i \subseteq Q$. That is, if Q is the state space of the automaton then the acceptance condition of a GNBA is an element $\mathcal{F}$ of 2^{2^Q}. Recall

[4] In Chapter 10, deterministic Rabin automata will be used for representing ω-regular properties.

that for an NBA, it is an element $F \in 2^Q$. The accepted language of a GNBA $\mathcal{G}$ consists of all infinite words which have an infinite run in $\mathcal{G}$ that visits *all* sets $F_i \in \mathcal{F}$ infinitely often. Thus, the acceptance criterion in a generalized Büchi automaton can be understood as the conjunction of a number of Büchi acceptance conditions.

Definition 4.52. Generalized NBA (GNBA)

A *generalized NBA* is a tuple $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ where Q, Σ, δ, Q_0 are defined as for an NBA (see Definition 4.27 on page 174) and $\mathcal{F}$ is a (possibly empty) subset of 2^Q.

The elements $F \in \mathcal{F}$ are called *acceptance sets*. Runs in a GNBA are defined as for an NBA. That is, a run in $\mathcal{G}$ for the infinite word $A_0\, A_1 \ldots \in \Sigma^\omega$ is an infinite state sequence $q_0\, q_1\, q_2 \ldots \in Q^\omega$ such that $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, A_i)$ for all $i \geqslant 0$.

The infinite run $q_0\, q_1\, q_2 \ldots$ is called *accepting* if

$$\forall\, F \in \mathcal{F}.\ \left(\overset{\infty}{\exists}\ j \in \mathbb{N}.\ q_j \in F \right).$$

The accepted language of $\mathcal{G}$ is:

$$\mathcal{L}_\omega(\mathcal{G}) \ = \ \{\, \sigma \in \Sigma^\omega \mid \text{there exists an accepting run for } \sigma \text{ in } \mathcal{G} \,\}.$$

■

Equivalence of GNBAs and the size of a GNBA are defined as for NBAs. Thus, GNBA $\mathcal{G}$ and $\mathcal{G}'$ are equivalent if $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}')$. The size of GNBA $\mathcal{G}$, denoted $|\mathcal{G}|$, equals the number of states and transitions in $\mathcal{G}$.

Example 4.53. GNBA

Figure 4.19 shows a GNBA $\mathcal{G}$ over the alphabet 2^{AP} where $AP = \{\, crit_1, crit_2 \,\}$ with the acceptance sets $F_1 = \{\, q_1 \,\}$ and $F_2 = \{\, q_2 \,\}$. That is, $\mathcal{F} = \{\{\, q_1 \,\}, \{\, q_2 \,\}\}$. The accepted language is the LT property P_{live} consisting of all infinite words $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega$ such that the atomic propositions $crit_1$ and $crit_2$ hold infinitely often (possibly at different positions), i.e.,

$$\overset{\infty}{\exists}\ j \geqslant 0.\, crit_1 \in A_j \quad \text{and} \quad \overset{\infty}{\exists}\ j \geqslant 0.\, crit_2 \in A_j.$$

Thus, P_{live} formalizes the property "both processes are infinitely often in their critical section". Let us justify that indeed $\mathcal{L}_\omega(\mathcal{G}) = P_{live}$. This goes as follows.

"$\subseteq$": Each accepting run has to pass infinitely often through the edges (labeled with $crit_1$ or $crit_2$) leading to the states q_1 and q_2. Thus, in every accepted word $\sigma = A_0 A_1 A_2 \ldots \in$

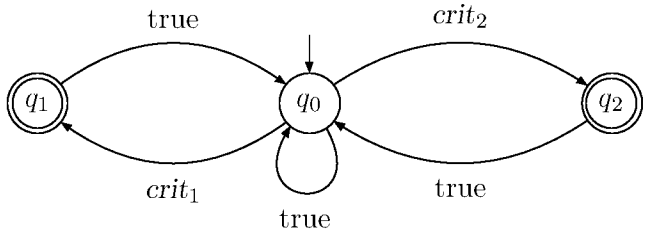


Figure 4.19: GNBA for "infinitely often processes 1 and 2 are in their critical section".

$\mathcal{L}_\omega(\mathcal{G})$ the atomic propositions $crit_1$ and $crit_2$ occur infinitely often as elements of the sets $A_i \in 2^{AP}$. Thus, $\sigma \in P_{live}$.

"$\supseteq$": Let $\sigma = A_0\, A_1\, A_2 \ldots \in P_{live}$. Since both propositions $crit_1$ and $crit_2$ occur infinitely often in the symbols A_i, the GNBA $\mathcal{G}$ can behave for the input word σ as follows. $\mathcal{G}$ remains in state q_0 until the first input symbol A_i with $crit_1 \in A_i$ appears. The automaton then moves to state q_1. From there, $\mathcal{G}$ consumes the next input symbol A_{i+1} and returns to q_0. It then waits in q_0 until a symbol A_j with $crit_2 \in A_j$ occurs, in which case the automaton moves to state q_2 for the symbol A_j and returns to q_0 on the next symbol A_{j+1}. Now the whole procedure restarts, i.e., $\mathcal{G}$ stays in q_0 while reading the symbols $A_{j+1}, \ldots, A_{\ell-1}$ and moves to q_1 as soon as the current input symbol A_ℓ contains $crit_1$. And so on. In this way, $\mathcal{G}$ generates an accepting run of the form

$$q_0^{k_1}\, q_1\, q_0^{k_2}\, q_2\, q_0^{k_3}\, q_1\, q_0^{k_4} q_2\, q_0^{k_5} \ldots$$

for the input word σ. These considerations show that $P_{live} \subseteq \mathcal{L}_\omega(\mathcal{G})$. ■

Remark 4.54. No Acceptance Set

The set $\mathcal{F}$ of acceptance sets of a GNBA may be empty. If $\mathcal{F} = \varnothing$ then $\sigma \in \mathcal{L}_\omega(\mathcal{G})$ if and only if there exists an infinite run for σ in $\mathcal{G}$. We like to stress the difference with NBA with an empty set of accepting states. For an NBA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \varnothing)$ there are *no* accepting runs. Therefore, the language $\mathcal{L}_\omega(\mathcal{A})$ is empty. Contrary to that, *every* infinite run of a GNBA $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \varnothing)$ is accepting.

In fact, every GNBA $\mathcal{G}$ is equivalent to a GNBA $\mathcal{G}'$ having *at least one* acceptance set. This is due to the fact that the state space Q can always be added to the set $\mathcal{F}$ of the acceptance sets without affecting the accepted language of the GNBA. Formally, for GNBA $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ let GNBA $\mathcal{G}' = (Q, \Sigma, \delta, Q_0, \mathcal{F} \cup \{Q\})$. Then it easily follows that: $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}')$. ■

Remark 4.55. Nonblocking GNBA

As for NBAs, each GNBA $\mathcal{G}$ can be replaced with an equivalent GNBA $\mathcal{G}'$, in which all possible behaviors for a given infinite input word yield an infinite run. Such a GNBA $\mathcal{G}'$ can be constructed by inserting a nonaccept trapping state, as we did for NBA in the remark on page 187. ■

Obviously, every NBA can be understood as a GNBA with exactly one acceptance set. Conversely, every GNBA can be transformed into an equivalent NBA:

Theorem 4.56. From GNBA to NBA

For each GNBA $\mathcal{G}$ there exists an NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{A})$ and $|\mathcal{A}| = \mathcal{O}(|\mathcal{G}| \cdot |\mathcal{F}|)$ where $\mathcal{F}$ denotes the set of acceptance sets in $\mathcal{G}$.

Proof: Let $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ be a GNBA. According to the remark on page 194, we may assume without loss of generality that $\mathcal{F} \neq \varnothing$. Let $\mathcal{F} = \{F_1, \ldots, F_k\}$ where $k \geqslant 1$. The basic idea of the construction of $\mathcal{A}$ is to create k copies of $\mathcal{G}$ such that the acceptance set F_i of the ith copy is connected to the corresponding states of the $(i{+}1)$th copy. The accepting

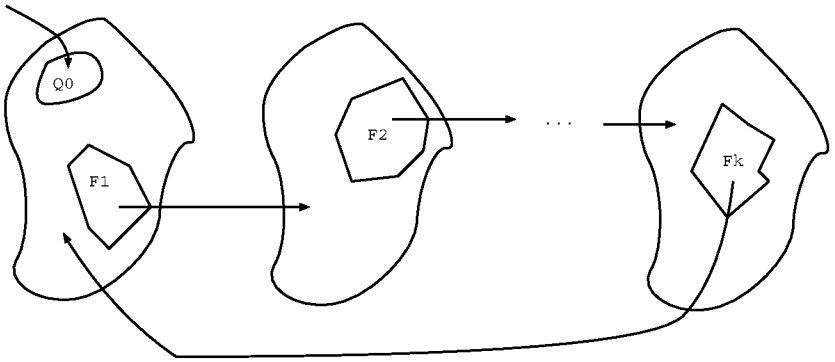


Figure 4.20: Idea for transforming a GNBA into an NBA.

condition for $\mathcal{A}$ consists of the requirement that an accepting state of the first copy is visited infinitely often. This ensures that all other accepting sets F_i of the k copies are visited infinitely often too, see Figure 4.20 on page 195. Formally, let $\mathcal{A} = (Q', \Sigma, \delta', Q'_0, F')$ where:

$$Q' = Q \times \{1, \ldots, k\},$$

$$Q'_0 = Q_0 \times \{1\} = \{\langle q_0, 1\rangle \mid q_0 \in Q_0\}, \text{ and}$$

$$F' = F_1 \times \{1\} = \{\langle q_F, 1\rangle \mid q_F \in F_1\}.$$

The transition function δ' is given by

$$\delta'(\langle q, i\rangle, A) = \begin{cases} \{\, \langle q', i\rangle \mid q' \in \delta(q, A) \,\} & \text{if } q \notin F_i \\ \{\, \langle q', i{+}1\rangle \mid q' \in \delta(q, A) \,\} & \text{otherwise.} \end{cases}$$

We thereby identify $\langle q, k{+}1\rangle$ and $\langle q, 1\rangle$. It is not difficult to check that $\mathcal{A}$ can be constructed in time and space $\mathcal{O}(|\mathcal{G}| \cdot |\mathcal{F}|)$ where $|\mathcal{F}| = k$ is the number of acceptance sets in $\mathcal{G}$. The fact $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{A})$ can be seen as follows.

$\supseteq$: For a run of $\mathcal{A}$ to be accepting it has to visit some state $\langle q, 1\rangle$ infinitely often, where $q \in F_1$. As soon as a run reaches $\langle q, 1\rangle$, the NBA $\mathcal{A}$ moves to the second copy. From the second copy the next copy can be reached by visiting $\langle q', 2\rangle$ with $q' \in F_2$. NBA $\mathcal{A}$ can only return to $\langle q, 1\rangle$ if it goes through all k copies. This is only possible if it reaches an accept state in each copy since that is the only opportunity to move to the next copy. So, for a run to visit $\langle q, 1\rangle$ infinitely often it has to visit some accept state in each copy infinitely often.

$\subseteq$: By a similar reasoning it can be deduced that every word in $\mathcal{L}_\omega(\mathcal{G})$ is also accepting in $\mathcal{A}$. ■

Example 4.57. Transformation of a GNBA into an NBA

Consider the GNBA $\mathcal{G}$ described in Example 4.53 on page 193. The construction indicated in the proof of Theorem 4.56 provides an NBA consisting of two copies (as there are two accept sets) of $\mathcal{G}$, see Figure 4.21. For example,

$$\delta'(\langle q_0, 1\rangle, \{\, crit_1 \,\}) = \{\, \langle q_0, 1\rangle, \langle q_1, 1\rangle \,\},$$

since $q_0 \notin F_1 = \{\, q_1 \,\}$ and $\delta'(\langle q_2, 2\rangle, A) = \{\, \langle q_0, 1\rangle \,\}$, since $F_2 = \{\, q_2 \,\}$. Thereby, $A \subseteq \{\, crit_1, crit_2 \,\}$ is arbitrary. ■

Any NBA can be considered as a GNBA by simply replacing the acceptance set F of the NBA with the singleton set $\mathcal{F} = \{\, F \,\}$ for the corresponding GNBA. Using this fact, together with the result that NBAs are equally expressive as ω-regular languages (see Theorem 4.32 on page 178), we obtain by Theorem 4.56:

Corollary 4.58. GNBA and ω-Regular Languages

The class of languages accepted by GNBAs agrees with the class of ω-regular languages.

As we have seen before, ω-regular languages are closed under union. This is immediate from the definition of ω-regular expressions and can also simply be proven by means of

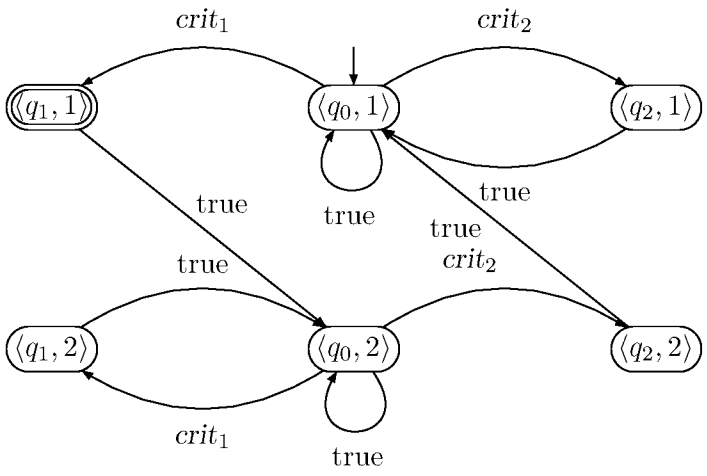


Figure 4.21: Example for the transformation of a GNBA into an equivalent NBA.

NBA representations (see Lemma 4.33 on page 179). We now use GNBAs to show that ω-regular languages are closed under intersection too.

Lemma 4.59. Intersection of GNBA

For GNBA $\mathcal{G}_1$ and $\mathcal{G}_2$ (both over the alphabet Σ), there exists a GNBA $\mathcal{G}$ with

$$\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}_1) \cap \mathcal{L}_\omega(\mathcal{G}_2) \quad \textit{and} \quad |\mathcal{G}| = \mathcal{O}(|\mathcal{G}_1| \cdot |\mathcal{G}_2|).$$

Proof: Let $\mathcal{G}_1 = (Q_1, \Sigma, \delta_1, Q_{0,1}, \mathcal{F}_1)$ and $\mathcal{G}_2 = (Q_2, \Sigma, \delta_2, Q_{0,2}, \mathcal{F}_2)$ where without loss of generality $Q_1 \cap Q_2 = \varnothing$. Let $\mathcal{G}$ be the GNBA that results from $\mathcal{G}_1$ and $\mathcal{G}_2$ by a synchronous product construction (as for NFA) and "lifts" the acceptance sets $F \in \mathcal{F}_1 \cup \mathcal{F}_2$ to acceptance sets in $\mathcal{G}$. Formally,

$$\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2 = (Q_1 \times Q_2, \Sigma, \delta, Q_{0,1} \times Q_{0,2}, \mathcal{F})$$

where the transition relation δ is defined by the rule

$$\frac{q_1 \xrightarrow{A}_1 q_1' \ \wedge \ q_2 \xrightarrow{A}_2 q_2'}{(q_1, q_2) \xrightarrow{A} (q_1', q_2')}.$$

The acceptance condition in $\mathcal{G}$ is given by

$$\mathcal{F} = \{F_1 \times Q_2 \mid F_1 \in \mathcal{F}_1\} \cup \{Q_1 \times F_2 \mid F_2 \in \mathcal{F}_2\}.$$

It is now easy to verify that $\mathcal{G}$ has the desired properties. ■

The same result also holds for union, that is, given two GNBAs $\mathcal{G}_1$ and $\mathcal{G}_2$ with the same alphabet Σ there is a GNBA $\mathcal{G}$ with $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{G}_1) \cup \mathcal{L}_\omega(\mathcal{G}_2)$ and $|\mathcal{G}| = \mathcal{O}(|\mathcal{G}_1| + |\mathcal{G}_2|)$. The argument is the same as for NBA (Lemma 4.33): we simply may take the disjoint union of the two GNBAs and decide nondeterministically which of them is chosen to "scan" the given input word.

Since GNBA yield an alternative characterization of ω-regular languages, we obtain by Lemma 4.59:

Corollary 4.60. Intersection of ω-Regular Languages

If $\mathcal{L}_1$ and $\mathcal{L}_2$ are ω-regular languages over the alphabet Σ, then so is $\mathcal{L}_1 \cap \mathcal{L}_2$.

4.4 Model-Checking ω-Regular Properties

The examples provided in Section 4.3.2 (see page 176 ff.) illustrated that NBAs yield a simple formalism for ω-regular properties. We now address the question how the automata-based approach for verifying regular safety properties can be generalized for the verification of ω-regular properties.

The starting point is a finite transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states and an ω-regular property P. The aim is to check algorithmically whether $TS \models P$. As we did for regular safety properties, the verification algorithm we present now attempts to show that $TS \not\models P$ by providing a counterexample, i.e., a path π in TS with $trace(\pi) \notin P$. (If no such path exists, then P holds for TS.) For this, we assume that we are given an automata-representation of the "bad traces" by means of an NBA $\mathcal{A}$ for the complement property $\overline{P} = (2^{AP})^\omega \setminus P$. The goal is then to check whether $Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. Note that:

$$Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$$

if and only if $Traces(TS) \cap \overline{P} \neq \varnothing$

if and only if $Traces(TS) \cap (2^{AP})^\omega \setminus P \neq \varnothing$

if and only if $Traces(TS) \not\subseteq P$

if and only if $TS \not\models P$.

The reader should notice the similarities with regular safety property checking. In that case, we started with an NFA for the bad prefixes for the given safety property P_{safe} (the bad behaviors). This can be seen as an automata-representation of the complement property $\overline{P_{safe}}$, see Remark 4.31 on page 177. The goal was then to find a finite, initial path fragment in TS that yields a trace accepted by the NFA, i.e., a bad prefix for P_{safe}.

Let us now address the problem to check whether $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. For this, we can follow the same pattern as for regular safety properties and construct the product $TS \otimes \mathcal{A}$ which combines paths in TS with the runs in $\mathcal{A}$. We then perform a graph analysis in $TS \otimes \mathcal{A}$ to check whether there is a path that visits an accept state of $\mathcal{A}$ infinitely often, which then yields a counterexample and proves $TS \not\models P$. If no such path in the product exists, i.e., if accept states can be visited at most finitely many times on all paths in the product, then all runs for the traces in TS are nonaccepting, and hence $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$ and thus $TS \models P$

In the sequel, we will explain these ideas in more detail. For doing so, we first introduce the notion of a persistence property. This is a simple type of LT property that will serve to formalize the condition stating that *accept states are only visited finitely many times*. The problem of verifying ω-regular properties is then shown to be reducible to the persistence checking problem. Recall that the problem of verifying regular safety properties is reducible to the invariant checking problem, see Section 4.2.

4.4.1 Persistence Properties and Product

Persistence properties are special types of liveness properties that assert that from some moment on a certain state condition Φ holds continuously. Stated in other words, $\neg\Phi$ is required to hold at most finitely many times. As for invariants, we assume a representation of Φ by a propositional logic formula over AP.

Definition 4.61. Persistence Property

A *persistence property* over AP is an LT property $P_{pers} \subseteq (2^{AP})^\omega$ "eventually forever Φ" for some propositional logic formula Φ over AP. Formally,

$$P_{pers} \;=\; \Big\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \overset{\infty}{\forall}\, j.\; A_j \models \Phi \Big\}$$

where $\overset{\infty}{\forall}\, j$ is short for $\exists i \geqslant 0.\; \forall j \geqslant i$. Formula Φ is called a persistence (or state) condition of P_{pers}. ■

Intuitively, a persistence property "eventually forever Φ" ensures the tenacity of the state

property given by the persistence condition Φ. One may say that Φ is an invariant after a while; i.e., from a certain point on all states satisfy Φ. The formula "eventually forever Φ" is true for a path if and only if almost all, i.e., all except for finitely many, states satisfy the proposition Φ.

Our goal is now to show that the question whether $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$ holds can be reduced to the question whether a certain persistence property holds in the product of TS and $\mathcal{A}$. The formal definition of the product $TS \otimes \mathcal{A}$ is exactly the same as for an NFA. For completeness, we recall the definition here:

Definition 4.62. Product of Transition System and NBA

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ a nonblocking NBA. Then, $TS \otimes \mathcal{A}$ is the following transition system:

$$TS \otimes \mathcal{A} \;=\; (S \times Q, Act, \rightarrow', I', AP', L')$$

where $\rightarrow'$ is the smallest relation defined by the rule

$$\frac{s \xrightarrow{\alpha} t \;\wedge\; q \xrightarrow{L(t)} p}{\langle s, q\rangle \xrightarrow{\alpha}{}' \langle t, p\rangle}$$

and where

- $I' = \{\, \langle s_0, q\rangle \mid s_0 \in I \;\wedge\; \exists q_0 \in Q_0.\; q_0 \xrightarrow{L(s_0)} q \,\}$,
- $AP' = Q$ and $L' : S \times Q \to 2^Q$ is given by $L'(\langle s, q\rangle) \;=\; \{\, q \,\}$.

Furthermore, let $P_{pers(\mathcal{A})}$ be the persistence property over $AP' = Q$ given by

"eventually forever $\neg F$"

where $\neg F$ denotes the propositional formula $\bigwedge\limits_{q \in Q} \neg q$ over $AP' = Q$. ■

We now turn to the formal proof that the automata-based approach for checking ω-regular properties relies on checking a persistence property for the product transition system:

Theorem 4.63. Verification of ω-Regular Properties

Let TS be a finite transition system without terminal states over AP and let P be an ω-regular property over AP. Furthermore, let $\mathcal{A}$ be a nonblocking NBA with the alphabet 2^{AP} and $\mathcal{L}_\omega(\mathcal{A}) = (2^{AP})^\omega \setminus P$. Then, the following statements are equivalent:

(a) $TS \models P$

(b) $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing$

(c) $TS \otimes \mathcal{A} \models P_{pers(\mathcal{A})}$

Proof: Let $TS = (S, Act, \to, I, AP, L)$ and $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$. The equivalence of (a) and (b) was shown on page 198. Let us now check the equivalence of (b) and (c). For this we show

$$\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing \text{ if and only if } TS \otimes \mathcal{A} \not\models P_{pers(\mathcal{A})}.$$

"$\Leftarrow$": Assume $TS \otimes \mathcal{A} \not\models P_{pers(\mathcal{A})}$. Let $\pi' = \langle s_0, q_1\rangle \langle s_1, q_2\rangle \ldots$ be a path in $TS \otimes \mathcal{A}$ such that

$$\pi' \not\models P_{pers(\mathcal{A})}.$$

Then there are infinitely many indices i with $q_i \in F$. The projection of π' to the states in TS yields a path $\pi = s_0\, s_1\, s_2 \ldots$ in TS.

Let $q_0 \in Q_0$ be an initial state of $\mathcal{A}$ such that $q_0 \xrightarrow{L(s_0)} q_1$. Such a state q_0 exists, since $\langle s_0, q_1\rangle$ is an initial state of $TS \otimes \mathcal{A}$. The state sequence $q_0\, q_1\, q_2 \ldots$ is a run in $\mathcal{A}$ for the word

$$\mathit{trace}(\pi) = L(s_0)\, L(s_1)\, L(s_2) \ldots \in \mathit{Traces}(TS).$$

Since there are infinitely many i with $q_i \in F$, the run $q_0\, q_1\, q_2 \ldots$ is accepting. Hence,

$$\mathit{trace}(\pi) \in \mathcal{L}_\omega(\mathcal{A}).$$

This yields $\mathit{trace}(\pi) \in \mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A})$, and thus $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$.

"$\Rightarrow$": Assume that $\mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$. Then there exists a path in TS, say $\pi = s_0\, s_1\, s_2 \ldots$ with

$$\mathit{trace}(\pi) = L(s_0)\, L(s_1)\, L(s_2) \ldots \in \mathcal{L}_\omega(\mathcal{A}).$$

Let $q_0\, q_1\, q_2 \ldots$ be an accepting run in $\mathcal{A}$ for $\mathit{trace}(\pi)$. Then

$$q_0 \in Q_0 \quad \text{and} \quad q_i \xrightarrow{L(s_i)} q_{i+1} \quad \text{for all } i \geqslant 0.$$

Furthermore, $q_i \in F$ for infinitely many indices i. Thus, we can combine π and the run $q_0\, q_1 \ldots$ to obtain a path in the product

$$\pi' = \langle s_0, q_1\rangle \langle s_1, q_2\rangle \ldots \in \mathit{Paths}(TS \otimes \mathcal{A}).$$

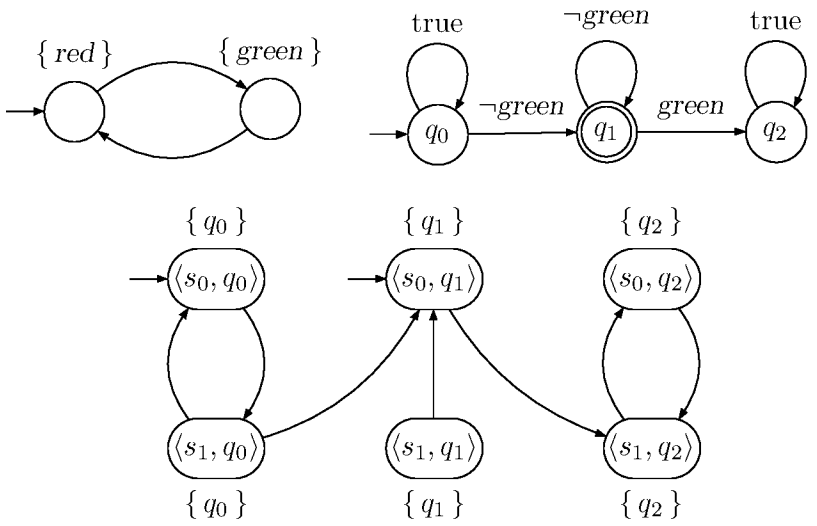


Figure 4.22: A simple traffic light (upper left), an NBA corresponding to P_{pers} (upper right), and their product (below).

Since $q_i \in F$ for infinitely many i, we have $\pi' \not\models P_{pers(\mathcal{A})}$. This yields $TS \otimes \mathcal{A} \not\models P_{pers(\mathcal{A})}$. ■

Example 4.64. Checking a Persistence Property

Consider a simple traffic light as one typically encounters at pedestrian crossings. It only has two possible modes: red or green. Assume that the traffic light is initially red, and alternates between red and green, see the transition system *PedTrLight* depicted in the upper left part of Figure 4.22. The ω-regular property P to be checked is "infinitely often green". The complement property $\overline{P}$ thus is "eventually always not green". The NBA depicted in the upper right part of Figure 4.22 accepts $\overline{P}$.

To check the validity of P, we first construct the product automaton $PedTrLight \otimes \mathcal{A}$, see the lower part of Figure 4.22. Note that the state $\langle s_1, q_1 \rangle$ is unreachable and could be omitted. Let $P_{pers(\mathcal{A})}$ = "eventually forever $\neg q_1$". From the lower part of Figure 4.22 one can immediately infer that there is no run of the product automaton that goes infinitely often through a state of the form $\langle \cdot, q_1 \rangle$. That is, transition system *PedTrLight* and NBA $\mathcal{A}$ do not have any trace in common. Thus we conclude:

$$PedTrLight \otimes \mathcal{A} \models \text{"eventually forever"}\ \neg q_1$$

and consequently (as expected):

$$PedTrLight \models \text{"infinitely often green"}.$$

As a slight alternative, we now consider a pedestrian traffic light that may automatically switch off to save energy. Assume, for simplicity, that the light may only switch off when it is red for some undefined amount of time. Clearly, this traffic light cannot guarantee the validity of $\overline{P}$ = "eventually forever $\neg$*green*" as it exhibits a run that (possibly after a while) alternates between red and off infinitely often. This can be formally shown as follows. First, we construct the product automaton, see Figure 4.23 (lower part). For instance, the path $\langle s_0, q_0\rangle\,(\langle s_2, q_1\rangle\,\langle s_0, q_1\rangle)^\omega$ goes infinitely often through the accept state q_1 of $\mathcal{A}$ and generates the trace

$$\{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\,\ldots\ldots$$

That is, *Traces*(*PedTrLight'*) $\cap\ \mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$, and thus

$$\textit{PedTrLight'} \otimes \mathcal{A} \not\models \text{ "eventually forever" } \neg q_1$$

and thus

$$\textit{PedTrLight}' \not\models \text{"infinitely often green"}.$$

More concretely, the path $\pi = s_0\,s_1\,s_0\,s_1\ldots$ generating the trace

$$\textit{trace}(\pi) = \{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\,\{\,red\,\}\,\varnothing\ldots$$

has an accepting run $q_0\,(q_1)^\omega$ in $\mathcal{A}$. ■

According to Theorem 4.63, the problem of checking an arbitrary ω-regular property can be solved with algorithms that check a simple type of ω-regular liveness property, namely persistence properties. An algorithm for the latter will be provided in the following section where the transition system under consideration results from the product of the original transition system and an NBA for the undesired behaviors.

4.4.2 Nested Depth-First Search

The next problem that we need to tackle is how to establish whether for a given finite transition system *TS*:

$$TS \not\models P_{pers}$$

where P_{pers} is a persistence property. Let Φ be the underlying propositional formula that specifies the state condition which has to hold "eventually forever".

The following result shows that answering the question "does $TS \not\models P_{pers}$ hold?" amounts to checking whether *TS* contains a reachable state violating Φ that is on a cycle in *TS*.

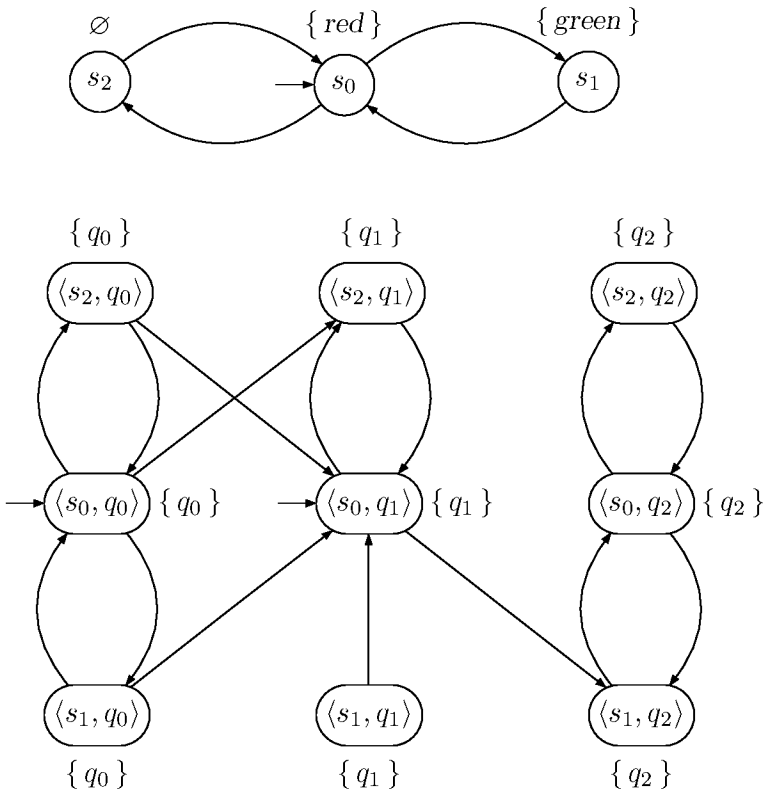


Figure 4.23: A simple traffic light that can switch off (upper part) and its product (lower part).

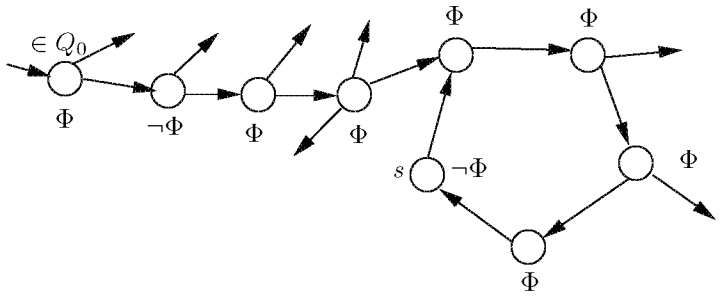


Figure 4.24: An example of a run violating "eventually always" Φ.

This can be justified intuitively as follows. Suppose s is a state that is reachable from an initial state in TS and $s \not\models \Phi$. As s is reachable, TS has an initial path fragment that ends in s. If s is on a cycle, then this path fragment can be continued by an infinite path that is obtained by traversing the cycle containing s infinitely often. In this way, we obtain a path in TS that visits the $\neg\Phi$-state s infinitely often. But then, $TS \not\models P_{pers}$. This is exemplified in Figure 4.24 where a fragment of a transition system is shown; for simplicity the action labels have been omitted. (Note that – in contrast to invariants – a state violating Φ which is not on a cycle does not cause the violation of P_{pers}.)

The reduction of checking whether $TS \models P_{pers}$ to a cycle detection problem is formalized by the following theorem.

Theorem 4.65. Persistence Checking and Cycle Detection

Let TS be a finite transition system without terminal states over AP, Φ a propositional formula over AP, and P_{pers} the persistence property "eventually forever Φ". Then, the following statements are equivalent:

(a) $TS \not\models P_{pers}$,

(b) There exists a reachable $\neg\Phi$-state s which belongs to a cycle. Formally:

$$\exists s \in Reach(TS).\, s \not\models \Phi \wedge s \text{ is on a cycle in } G(TS)\ .^5$$

Before providing the proof, let us first explain how to obtain an error indication whenever $TS \not\models P_{pers}$. Let $\widehat{\pi} = u_0\, u_1\, u_2\, \ldots\, u_k$ be a path in the graph induced by TS, i.e., $G(TS)$, such that $k > 0$ and $s = u_0 = u_k$. Assume $s \not\models \Phi$. That is, $\widehat{\pi}$ is a cycle in $G(TS)$ containing

[5] Recall that $G(TS)$ denotes the underlying directed graph of TS.

a state violating Φ. Let $s_0\, s_1\, s_2 \ldots s_n$ be an initial path fragment of TS such that $s_n = s$. Then the concatenation of this initial path fragment and the unfolding of the cycle

$$\pi = s_0\, s_1\, s_2 \ldots \underbrace{s_n}_{=s}\, u_1\, u_2 \ldots \underbrace{u_k}_{=s}\, u_1\, u_2 \ldots \underbrace{u_k}_{=s} \ldots$$

is a path in TS. As state $s \not\models \Phi$ is visited by π infinitely often, it follows that π does not satisfy "eventually always Φ". The prefix

$$s_0\, s_1\, s_2 \ldots \underbrace{s_n}_{=s}\, u_1\, u_2 \ldots \underbrace{u_k}_{=s}$$

can be used as diagnostic feedback as it shows that s may be visited infinitely often.

Proof: Let $TS = (S, Act, \rightarrow, I, AP)$.

(a) $\Longrightarrow$ (b): Assume $TS \not\models P_{pers}$, i.e., there exists a path $\pi = s_0\, s_1\, s_2 \ldots$ in TS such that $trace(\pi) \notin P_{pers}$. Thus, there are infinitely many indices i such that $s_i \not\models \Phi$. Since TS is finite, there is a state s with $s = s_i \not\models \Phi$ for infinitely many i. As s appears on a path starting in an initial state we have $s \in Reach(TS)$. A cycle $\widehat{\pi}$ is obtained by any fragment $s_i\, s_{i+1}\, s_{i+2} \ldots s_{i+k}$ of π where $s_i = s_{i+k} = s$ and $k > 0$.

(b) $\Longrightarrow$ (a): Let s and $\widehat{\pi} = u_0 u_1 \ldots u_k$ be as indicated above, i.e., $s \in Reach(TS)$ and $\widehat{\pi}$ is a cycle in TS with $s = u_0 = u_k$. Since $s \in Reach(TS)$, there in an initial state $s_0 \in I$ and a path fragment $s_0\, s_1 \ldots s_n$ with $s_n = s$. Then:

$$\pi = s_0\, s_1\, s_2 \ldots \underbrace{s_n}_{=s}\, u_1\, u_2 \ldots \underbrace{u_k}_{=s}\, u_1\, u_2\, \underbrace{u_k}_{=s} \ldots$$

is a path in TS. Since $s \not\models \Phi$, it follows that π does not satisfy "eventually forever Φ", and thus $TS \not\models P_{pers}$. ∎

Example 4.66. Pedestrian Traffic Lights Revisited

Consider the transition system model of the simple traffic light that one typically encounters at pedestrian crossings (see Figure 4.22) and the persistence property "eventually forever" $\neg q_1$ where q_1 is the accept state of the NBA $\mathcal{A}$. As there is no reachable cycle in the product transition system that contains a state violating $\neg q_1$, i.e., a cycle that contains a state labeled with q_1, it follows that

$$PedTrLight \otimes \mathcal{A} \models \text{"eventually forever"}\ \neg q_1.$$

For the traffic light that has the possibility to automatically switch off, it can be inferred directly from the product transition system (see the lower part of Figure 4.23) that there

is a reachable state, e.g., $\langle s_2, q_1 \rangle \not\models \neg q_1$, that lies on a cycle. Thus:

$$\mathit{PedTrLight'} \otimes \mathcal{A} \not\models \text{“eventually forever” } \neg q_1.$$

■

Thus, persistence checking for a finite transition system requires the same techniques as checking emptiness of an NBA, see page 184. In fact, the algorithm we suggest below can also be used for checking emptiness in an NBA.

A Naive Depth-First Search Theorem 4.65 entails that in order to check the validity of a persistence property, it suffices to check whether there exists a reachable cycle containing a $\neg\Phi$-state. How to check for such reachable cycles? A possibility is to compute the strongly connected components (SCCs, for short) in $G(TS)$ – this can be done in a worst-case time complexity that is linear in the number of states and transitions – and to check whether one such SCC is reachable from an initial state, contains at least one edge, and, moreover, contains a $\neg\Phi$-state. If indeed such a SCC does exist, P_{pers} is refuted; otherwise the answer is affirmative.

Although this SCC-based technique is optimal with respect to the asymptotic worst-case time complexity, it is more complex and less adequate for an on-the-fly implementation. In that respect, pure cycle-check algorithms are more appropriate. Therefore, in the sequel we will detail the standard DFS-based cycle detection algorithm.

Let us first recall how for a finite directed graph G and node v, it can be checked with a DFS-based approach whether v belongs to a cycle. For this, one may simply start a depth-first search in node v and check for any visited node w whether there is an edge from w to v. If so, a cycle has been found: it starts in v and follows the path to node w given by the current stack content and then takes the edge from w to v. Vice versa, if no such edge is found, then v does not belong to a cycle. To determine whether G has a cycle, a similar technique can be exploited: we perform a depth-first search to visit all nodes in G. Moreover, on investigating the edge from w to v it is checked whether v has already been visited and whether v is still on the DFS stack. If so, then a so-called *backward edge* has been found which closes a cycle. Otherwise, if no backward edge has been found during the DFS in G then G is acyclic. (A detailed description of this DFS-based cycle detection technique can be found in textbooks on algorithms and data structures, e.g., [100].)

We now DFS-based cycle checks (by searching for backward edges) for persistence checking. The naive approach works in two phases, as illustrated in Algorithm 6:

1. In the first step, all states satisfying $\neg\Phi$ that are reachable from some initial state are determined. This is performed by a standard depth-first search.

2. In the second step, for each reachable $\neg\Phi$-state s, it is checked whether it belongs to a cycle. This algorithm (called cycle_check, see Algorithm 7) relies on the technique sketched above: we start a depth-first search in s (with initially empty DFS stack V and initially empty set T of visited states) and check for all states reachable from s whether there is an outgoing backward edge leading to s.

This yields an algorithm for checking the validity of the persistence property with quadratic worst-case running time. More precisely, its time complexity is in $\mathcal{O}(N{\cdot}(|\Phi| + N{+}M))$ where N is the number of *reachable* states in TS and M the number of transitions between these reachable states. This can be seen as follows. Visiting all states that are reachable from some initial state takes $\mathcal{O}(N{+}M{+}N{\cdot}|\Phi|)$ as a depth-first search over all states suffices and in each reachable state the validity of Φ is checked (which is assumed to be linear in the size of Φ). In the worst case, all states refute Φ, and a depth-first search takes place (procedure cycle_check) for all these states. This takes $\mathcal{O}(N{\cdot}(N{+}M))$. Together this yields $\mathcal{O}(N{\cdot}(|\Phi| + N{+}M))$.

Several simple modifications of the suggested technique are possible to increase efficiency. For instance, cycle_check(s') can be invoked inside visit(s) immediately before or after s' is inserted into $R_{\neg\Phi}$, in which case the whole persistence checking algorithm can abort with the answer "no" if cycle_check(s') returns true. However, the quadratic worst-case running time cannot be avoided if the cycle check algorithm for the $\neg\Phi$-states relies on separate depth-first searches. The problem is that certain fragments of TS might be reachable from different $\neg\Phi$-states. These fragments are (re-)explored in the depth-first searches (cycle_check) invoked by several $\neg\Phi$-states. To obtain linear running time, we aim at a cycle detection algorithm that searches for backward edges leading to one of the $\neg\Phi$-states and ensures that any state is visited *at most once* in the depth-first searches for the cycle detection. This will be explained in the following subsection.

A Nested Depth-First Search Algorithm The rough idea of the linear-time cycle detection-based persistence checking algorithm is to perform two depth-first searches (DFSs) in TS in an interleaved way. The first (outer) depth-first search serves to encounter all reachable $\neg\Phi$-states. The second (inner) depth-first search seeks backward edges leading to a $\neg\Phi$-state. The inner depth-first search is *nested* in the outer one in the following sense: whenever a $\neg\Phi$-state s has been fully expanded by the outer depth-first search, then the inner depth-first search continues with state s and visits all states s' that are reachable from s and that have not yet been visited in the inner depth-first search before. If no backward edge has been found when treating s in the inner DFS, then the outer DFS continues until the next $\neg\Phi$-state t has been fully expanded, in which case the inner DFS proceeds with t.

Algorithm 6 Naive persistence checking

Input: finite transition system TS without terminal states, and proposition Φ
Output: "yes" if $TS \models$ "eventually forever Φ", otherwise "no".

```
set of states R := ∅; R_¬Φ := ∅;          (* set of reachable states resp. ¬Φ-states *)
stack of states U := ε;                    (* DFS stack for first DFS, initial empty *)
set of states T := ∅;                      (* set of visited states for the cycle check *)
stack of states V := ε;                    (* DFS stack for the cycle check *)

for all s ∈ I \ R do visit(s); od          (* a DFS for each unvisited initial state *)
for all s ∈ R_¬Φ do
  T := ∅; V := ε;                          (* initialize set T and stack V *)
  if cycle_check(s) then return "no"       (* s belongs to a cycle *)
od
return "yes"                               (* none of the ¬Φ-states belongs to a cycle *)
```

```
procedure visit (state s)
  push(s, U);                              (* push s on the stack *)
  R := R ∪ { s };                          (* mark s as reachable *)
  repeat
    s' := top(U);
    if Post(s') ⊆ R then
      pop(U);
      if s' ⊭ Φ then R_¬Φ := R_¬Φ ∪ { s' }; fi
    else
      let s'' ∈ Post(s') \ R
      push(s'', U);
      R := R ∪ { s''};                     (* state s'' is a new reachable state *)
    fi
  until (U = ε)
endproc
```

Algorithm 7 Cycle detection

Input: finite transition system TS and state s in TS with $s \not\models \Phi$
Output: true if s lies on a cycle in TS, otherwise false

```
(* T organizes the set of states that have been visited, V serves as DFS stack.   *)
(* In the standard approach to check whether there is a backward edge to s,      *)
(* T and V are initially empty.                                                  *)

procedure boolean cycle_check(state s)
  boolean cycle_found := false;                          (* no cycle found yet *)
  push(s, V);                                            (* push s on the stack *)
  T := T ∪ { s };
  repeat
    s' := top(V);                                        (* take top element of V *)
    if s ∈ Post(s') then
      cycle_found := true;                               (* if s ∈ Post(s'), a cycle is found *)
      push(s, V);                                        (* push s on the stack *)
    else
      if Post(s') \ T ≠ ∅ then
        let s'' ∈ Post(s') \ T;
        push(s'', V);                                    (* push an unvisited successor of s' *)
        T := T ∪ { s'' };                                (* and mark it as reachable *)
      else
        pop(V);                                          (* unsuccessful cycle search for s' *)
      fi
    fi
  until ((V = ε) ∨ cycle_found)
  return cycle_found
endproc
```

Algorithm 8 Persistence checking by nested depth-first search

Input: transition system TS without terminal states, and proposition Φ
Output: "yes" if $TS \models$ "eventually forever Φ", otherwise "no" plus counterexample

```
set of states R := ∅;                          (* set of visited states in the outer DFS *)
stack of states U := ε;                        (* stack for the outer DFS *)
set of states I := ∅;                          (* set of visited states in the inner DFS *)
stack of states V := ε;                        (* stack for the inner DFS *)
boolean cycle_found := false;

while (I \ R ≠ ∅ ∧ ¬cycle_found) do
  let s ∈ I \ R;                               (* explore the reachable *)
  reachable_cycle(s);                          (* fragment with outer DFS *)
od
if ¬cycle_found then
  return ("yes")                               (* TS ⊨ "eventually forever Φ" *)
else
  return ("no", reverse(V.U))                  (* stack contents yield a counterexample *)
fi
```

```
procedure reachable_cycle (state s)
  push(s, U);                                  (* push s on the stack *)
  R := R ∪ { s };
  repeat
    s' := top(U);
    if Post(s') \ R ≠ ∅ then
      let s'' ∈ Post(s') \ R;
      push(s'', U);                            (* push the unvisited successor of s' *)
      R := R ∪ { s'' };                        (* and mark it reachable *)
    else
      pop(U);                                  (* outer DFS finished for s' *)
      if s' ⊭ Φ then
        cycle_found := cycle_check(s');        (* proceed with the inner *)
                                               (* DFS in state s' *)
      fi
    fi
  until ((U = ε) ∨ cycle_found)                (* stop when stack for the outer *)
                                               (* DFS is empty or cycle found *)
endproc
```

This algorithm is called *nested depth-first search*. Algorithm 8 shows the pseudocode for the outer depth-first search (called reachable_cycle) which invokes the inner depth-first search (cycle_check, cf. Algorithm 7 on page 210). Later we will explain that it is important that cycle_check(s) is called immediately after the outer depth-first search has visited all successors of s. The major difference from the naive approach is that cycle_check(s) reuses the information of previous calls of cycle_check($\cdot$) and ignores all states in T. When cycle_check(s) is invoked then T consists of all states that have been visited in the inner DFS before, i.e., during the execution of cycle_check(u) called prior to cycle_check(s).

An interesting aspect of this nested depth-first strategy is that once a cycle is determined containing a $\neg\Phi$-state s, then a path to s can be computed easily: stack U for the outer DFS contains a path fragment from the initial state $s_0 \in I$ to s (in reversed order), while stack V for the inner DFS — as maintained by the procedure cycle_check($\cdot$) — contains a cycle from state s to s (in reversed order). Concatenating these path fragments thus provides an error indication in the same vein as described before in the proof of Theorem 4.65 on page 205.

Example 4.67. Running the Nested DFS Algorithm

Consider the transition system depicted in Figure 4.25 and assume that $s_0 \models \Phi$, $s_3 \models \Phi$, whereas $s_1 \not\models \Phi$ and $s_2 \not\models \Phi$. Consider the scenario in which s_2 is considered as a first successor of s_0, i.e., s_2 is considered in the outer depth-first search prior to considering state s_1. This means that the order in which the states are put by Algorithm 8 on the stack U equals $\langle s_1, s_3, s_2, s_0 \rangle$ where we write the stack content from the top to the bottom, i.e., s_1 is the top element (the last element that has been pushed on the stack). Besides, $R = S$. Thus, the outer DFS takes s_1 as the top element of U and verifies $Post(s_1) \subseteq R$. As $s_1 \not\models \Phi$, the invocation cycle_check(s_1) takes place and s_1 is deleted from stack U for the outer DFS. This yields that s_1 is put on the stack V for the inner DFS, followed by its only successor s_3. Then the cycle $s_1 \to s_3 \to s_1$ is detected, cycle_check(s_1) yields true, and as a result, reachable_cycle(s_0) terminates with the conclusion that P_{pers} on Φ is refuted. An error indication is obtained by first concatenating the content of $V = \langle s_1, s_3, s_1 \rangle$ and $U = \langle s_3, s_2, s_0 \rangle$ and then reversing the order. This yields the path $s_0\, s_2\, s_3\, s_1\, s_3\, s_1$, which is indeed a prefix of a run refuting "eventually always Φ". ■

The soundness of the nested depth-first search is no longer trivial because – by the treatment of T as a global variable – not all states reachable from s are explored in cycle_check(s), but only those states that have not been visited before in the inner depth-first search. Thus, we could imagine that there is a cycle with $\neg\Phi$-states which will not be found with the nested depth-first search: this cycle might have been explored during cycle_check(u) where u does not belong to this or any other cycle. Then, none of the following procedure calls of cycle_check($\cdot$) will find this cycle. The goal is now to show

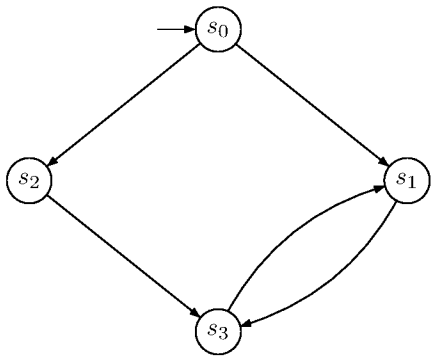


Figure 4.25: An example of a transition system for nested DFS.

that this cannot happen, i.e., if there is a reachable cycle with $\neg\Phi$-states, then the nested depth-first search will find such a cycle.

In fact, things are more tricky than it seems. In Algorithm 8 it is important that the invocations to the procedure cycle_check(s) for $s \in \mathit{Reach}(TS)$ and $s \not\models \Phi$ occur in appropriate order. Let $s \not\models \Phi$. Then cycle_check(s) is only invoked if $\mathit{Post}(s) \subseteq R$, i.e., when all states reachable from s have been encountered and visited.[6] So, the invocation cycle_check(s) is made immediately once all states that are reachable from s have been visited and expanded in the outer depth-first search. As a result, Algorithm 8 satisfies the following property: if s' is a successor of state s in the visit-order of the outer depth-first search and $s \not\models \Phi$ and $s' \not\models \Phi$, then the invocation cycle_check(s') occurs prior to the invocation cycle_check(s).

Example 4.68. Modifying the Nested DFS

Let us illustrate by an example that the nested depth-first search algorithm would be wrong if the outer and inner DFS are interleaved in an arbitrary way. We consider again the transition system in Figure 4.25 on page 213. We start the outer DFS with the initial state s_0 and assume that state s_2 is visited prior to s_1. Let us see what happens if we do not wait until s_2 has been fully expanded in the outer DFS and start cycle_check(s_2) immediately. Then, cycle_check(s_2) visits s_3 and s_1 and returns false, since there is no backward edge leading to state s_2. Thus, cycle_check(s_2) yields $T = \{s_2, s_3, s_1\}$ (and $V = \varepsilon$). Now the outer DFS proceeds and visits s_3 and s_1. It calls cycle_check(s_1) which fails to find the cycle $s_1 s_3 s_1$. Note that cycle_check(s_1) immediately halts and returns false since $s_1 \in T = \{s_2, s_3, s_1\}$. Thus, the nested depth-first search would return the wrong answer "yes". ■

[6] For a recursive formulation of the outer depth-first search, this is the moment where the depth-first search call for s terminates.

Theorem 4.69. Correctness of the Nested DFS

Let TS be a finite transition system over AP without terminal states, Φ a propositional formula over AP, and P_{pers} the persistence property "eventually forever Φ". Then:

$$\textit{Algorithm 8 returns the answer "no" if and only if } TS \not\models P_{pers}.$$

Proof:

$\Rightarrow$: This follows directly from the fact that the answer "false" is only obtained when an initial path fragment of the form $s_0 \ldots s \ldots s$ has been encountered for some $s_0 \in I$ and $s \not\models \Phi$. But then, $TS \not\models P_{pers}$.

$\Leftarrow$: To establish this direction, we first prove that the following condition holds:

(*) On invoking cycle_check(s), there is no cycle $s'_0\, s'_1 \ldots s'_k$ in TS such that:

$$\{ s'_0, s'_1, \ldots, s'_k \} \cap T \neq \varnothing \quad \text{and} \quad s \in \{ s'_0, \ldots, s'_k \}.$$

So, on invoking cycle_check(s), there is no cycle containing s and a state in T. Statement (*) ensures that in the inner DFS, all already visited states in T—the states that were visited during an earlier cycle detection—can be safely ignored during the next cycle search. In other words, if s belongs to a cycle then cycle_check(s) has to find one such cycle. We prove the statement (*) by contradiction. Consider the invocation cycle_check(s). Assume there exists a cycle $s'_0\, s'_1 \ldots s'_k$ such that

$$s'_0 = s'_k = s \quad \text{and} \quad s \not\models \Phi \quad \text{and} \quad \{ s'_0, s'_1, \ldots, s'_k \} \cap T \neq \varnothing.$$

Without loss of generality we assume that s is the *first* state for which this condition holds on invoking cycle_check(s). More precisely, we assume:

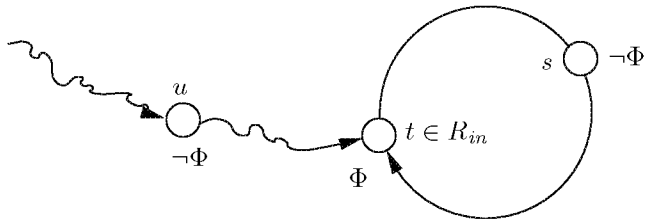
(+) For all states $\widehat{s}$ with $\widehat{s} \not\models \Phi$ where cycle_check($\widehat{s}$) has been invoked prior to cycle_check(s), there is no cycle containing $\widehat{s}$ and a state in T, on invoking cycle_check(s).

Let $t \in \{ s'_0, \ldots, s'_k \} \cap T$, i.e., t is a state on a cycle with state s. Since $t \in T$ on the invocation cycle_check(s), there must be a state, u say, for which cycle_check($\cdot$) has been invoked earlier and during this search t has been encountered (and added to T). Thus, we have $u \not\models \Phi$ and the following conditions (4.1) and (4.2):

$$\text{cycle\_check}(u) \text{ was invoked prior to cycle\_check}(s) \tag{4.1}$$

as a result of cycle_check(u), t was visited and added to T (4.2)

Obviously, from (4.2), it follows that state t is reachable from u. As the states s and t are on a cycle, and t is reachable from u, we have that s is reachable from u. This situation is sketched in the following figure:



Now consider the outer depth-first search, i.e., reachable_cycle($\cdot$), and discuss the following cases 1 and 2:

1. u has been visited prior to s in the outer DFS, i.e., s has been pushed on stack U after u.

   Since s is reachable from u, state s is visited during the expansion of u in the outer DFS and taken from stack U before u. Hence, cycle_check(s) is invoked prior to cycle_check(u) which contradicts (4.1).

2. u has been visited after s in the outer DFS, i.e., s has been pushed on stack U before u.

   By (4.1), s is still in stack U when cycle_check(u) is invoked. This yields that u is reachable from s. But as s is reachable from u, this means that s and u are on a cycle. This cycle or another cycle with state u would have been encountered during cycle_check(u) because of (+) and Algorithm 8 would have terminated without invoking cycle_check(s).

■

It remains to discuss the complexity of the nested depth-first search algorithm. Since T is increasing during the execution of the nested depth-first search (i.e., we insert states in T, but never take them out) any state in TS is visited at most once in the inner depth-first search, when ranging over all procedure calls of cycle_check($\cdot$). The same holds for the outer depth-first search since it is roughly a standard depth-first search. Thus, each reachable state s' in TS is taken

- at most once[7] as the top element of stack U in the outer depth-first search;
- at most once as the top element of stack V in the inner depth-first search (when ranging over all calls of cycle_check).

In particular, this observation yields the termination of Algorithm 8. The cost caused by the top-element s' for the body of the repeat loop in cycle_check (inner depth-first search) and in reachable_cycle (outer depth-first search) is $\mathcal{O}(|Post(s')|)$. Thus, the worst-case time complexity of Algorithm 8 is linear in the size of the reachable fragment of TS, but has the chance to terminate earlier when a cycle has been found. In the following theorem we also take into account that Φ might be a complex formula and evaluating the truth value of Φ for a given state (by means of its label) requires $\mathcal{O}(|\Phi|)$ steps.

Theorem 4.70. Time Complexity of Persistence Checking

The worst-case time complexity of Algorithm 8 is in $\mathcal{O}((N+M) + N \cdot |\, \Phi \,|)$ *where* N *is the number of reachable states, and* M *the number of transitions between the reachable states.*

The space complexity is bounded above by $\mathcal{O}(|S| + |\rightarrow|)$ where S is the state space of TS and $|\rightarrow|$ the number of transitions in TS. This is an adequate bound if we assume a representation of TS by adjacency lists. However, in the context of model checking the starting point is typically not an explicit representation of the composite transition system, but a syntactic description of the concurrent processes, e.g., by high-level modeling languages with a program graph or channel system semantics. Such syntactic descriptions are typically much smaller than the resulting transition system. (Recall the state explosion problem which appears through parallel composition and the unfolding of program graphs into transition systems, see page 77 ff.) In the persistence checking algorithm, we may assume that the elements in $Post(s')$ are generated on the fly by means of the semantic rules for the transition relation. Ignoring the space required for the syntactic descriptions of the processes, the additional space requirements for the presented persistence checking algorithm is $\mathcal{O}(N)$ for the sets T and R and the stacks U and V, where N is the number of reachable states in TS. In fact, the representation of T and R is the most (space-)critical aspect when implementing the nested depth-first search and running it on large examples. Typically, T and R are organized using appropriate hash techniques. In fact, T and R can even be represented by a single hash table where the entries are pairs $\langle s, b\rangle$ with $b \in \{0, 1\}$. The meaning of $\langle s, 0\rangle$ is that s is in R, but not in T (i.e., s has been visited in the outer, but not yet in the inner DFS). The pair $\langle s, 1\rangle$ means that s has been visited in both the outer and the inner DFS. The single bit b is sufficient to cover all possible cases, since T is always a subset of R.

[7] exactly once, if Algorithm 8 does not abort with the answer "no".

Another simple observation can speed up the nested depth-first search in case the persistence property is violated: whenever the inner DFS cycle_check(s) reaches a state t which is on the stack U (the DFS stack for the outer DFS), then the upper part of U yields a path fragment from t to s, while the content of the DFS stack V for the inner DFS describes a path fragment from s to t. Thus, a cycle has been found which visits s infinitely often and the nested DFS may abort with a counterexample. To support checking whether state t is contained in stack U, a further bit can be added to the entries in the hash table for representing T and R. I.e., we then have to deal with a hash table for triples $\langle s, b, c \rangle$ with $s \in R$ and $b, c \in \{0, 1\}$ depending on whether s is in T (in which case $b = 1$) and whether s is in U (in which case $c = 1$). This leads to a slight variant of Algorithm 8 which is often faster and generates smaller counterexamples than the original version.

4.5 Summary

- NFAs and DFAs are equivalent automata models for regular languages and can serve to represent the bad prefixes of regular safety properties.
- Checking a regular safety property on a finite transition system is solvable by checking an invariant on a product automaton, and thus amounts to solving a reachability problem.
- ω-Regular languages are languages of infinite words that can be described by ω-regular expressions.
- NBAs are acceptors for infinite words. The syntax is as for NFAs. The accepted language of an NBA is the set of all infinite words that have a run where an accept state is visited infinitely often.
- NBAs can serve to represent ω-regular properties.
- The class of languages that are recognized by NBAs agrees with the class of ω-regular languages.
- DBAs are less powerful than NBAs and fail, for instance, to represent the persistence property "eventually forever a".
- Generalized NBAs are defined as NBAs, except that they require repeated visits for several acceptance sets. Their expressiveness is the same as for NBAs.
- Checking an ω-regular property P on a finite transition system TS can be reduced to checking the persistence property "eventually forever no accept state" in the product of TS and an NBA for the undesired behaviors (i.e., the complement property $\overline{P}$).

- Persistence checking requires checking the existence of a reachable cycle containing a state violating the persistence condition. This is solvable in linear time by a nested depth-first search (or by analyzing the strongly connected components). The same holds for the nonemptiness problem for NBAs which boils down to checking the existence of a reachable cycle containing an accept state.
- The nested depth-first search approach consists of the interleaving of two depth-first searches: one for encountering the reachable states, and one for cycle detection.

4.6 Bibliographic Notes

Finite automata and regular languages. The first papers on finite automata were published in the nineteen fifties by Huffman [217], Mealy [291], and Moore [303] who used deterministic finite automata for representing sequential circuits. Regula r expressions and their equivalence to finite automata goes back to Kleene [240]. Rabin and Scott [350] presented various algorithms on finite automata, including the powerset construction. The existence of minimal DFAs relies on results stated by Myhill [309] and Nerode [313]. The $\mathcal{O}(N \log N)$ minimization algorithm we mentioned at the end of Section 4.1 has been suggested by Hopcroft [213]. For other algorithms on finite automata, a detailed description of the techniques sketched here and other aspects of regular languages, we refer to the text books [272, 363, 214, 383] and the literature mentioned therein.

Automata over infinite words. Research on automata over infinite words (and trees) started in the nineteen sixties with the work of Büchi [73], Trakhtenbrot [392], and Rabin [351] on decision problems for mathematical logics. At the same time, Muller [307] studied a special type of deterministic ω-automata (today called Muller automata) in the context of asynchronous circuits. The equivalence of nondeterministic Büchi automata and ω-regular expressions has been shown by McNaughton [290]. He also established a link between NBAs and deterministic Muller automata [307] by introducing another acceptance condition that has been later formalized by Rabin [351]. (The resulting ω-automata type is today called Rabin automata.) An alternative transformation from NBA to deterministic Rabin automata has been presented by Safra [361]. Unlike Büchi automata, the nondeterministic and deterministic versions of Muller and Rabin automata are equally expressive and yield automata-characterizations of ω-regular languages. The same holds for several other types of ω-automata that have been introduced later, e.g., Streett automata [382] or automata with the parity condition [305].

The fact that ω-regular languages are closed under complementation (we stated this result without proof) can be derived easily from deterministic automata representations. The proof of Theorem 4.50 follows the presentation given in the book by Peled [327]. Various

decision problems for ω-automata have been addressed by Landweber [261] and later by Emerson and Lei [143] and Sistla, Vardi, and Wolper [373]. For a survey of automata on infinite words, transformations between the several classes of ω-automata, complementation operators and other algorithms on ω-automata, we refer to the articles by Choueka [81], Kaminsky [229], Staiger [376], and Thomas [390, 391]. An excellent overview of the main concepts of and recent results on ω-automata is provided by the tutorial proceedings [174].

Automata and linear-time properties. The use of Büchi automata for the representation and verification of linear-time properties goes back to Vardi and Wolper [411, 412] who studied the connection of Büchi automata with linear temporal logic. Approaches with similar automata models have been developed independently by Lichtenstein, Pnueli, and Zuck [274] and Kurshan [250]. The verification of (regular) safety properties has been described by Kupferman and Vardi [249]. The notion of persistence property has been introduced by Manna and Pnueli [282] who provided a hierarchy of temporal properties. The nested depth-first algorithm (see Algorithm 8) originates from Courcoubetis et al. [102] and its implementation in the model checker SPIN has been reported by Holzmann, Peled, and Yannakakis [212]. The Murφ verifier developed by Dill [132] focuses on verifying safety properties. Variants of the nested depth-first search have been proposed by several authors, see, e.g., [106, 368, 161, 163]. Approaches that treat generalized Büchi conditions (i.e., conjunctions of Büchi conditions) are discussed in [102, 388, 184, 107]. Further implementation details of the nested depth-first search approach can be found in the book by Holzman [209].

4.7 Exercises

EXERCISE 4.1. Let $AP = \{a, b, c\}$. Consider the following LT properties:

(a) If a becomes valid, afterward b stays valid ad infinitum or until c holds.

(b) Between two neighboring occurrences of a, b always holds.

(c) Between two neighboring occurrences of a, b occurs more often than c.

(d) $a \wedge \neg b$ and $b \wedge \neg a$ are valid in alternation or until c becomes valid.

For each property P_i $(1 \leqslant i \leqslant 4)$, decide if it is a regular safety property (justify your answers) and if so, define the NFA $\mathcal{A}_i$ with $\mathcal{L}(\mathcal{A}_i) = \mathit{BadPref}(P_i)$. (*Hint: You may use propositional formulae over the set AP as transition labels.*)

EXERCISE 4.2. Let $n \geqslant 1$. Consider the language $\mathcal{L}_n \subseteq \Sigma^*$ over the alphabet $\Sigma = \{ A, B \}$ that consists of all finite words where the symbol B is on position n from the right, i.e., $\mathcal{L}$ contains exactly the words $A_1 A_2 \ldots A_k \in \{A, B\}^*$ where $k \geqslant n$ and $A_{k-n+1} = B$. For instance, the word $ABBAABAB$ is in $\mathcal{L}_3$.

(a) Construct an NFA $\mathcal{A}_n$ with at most $n+1$ states such that $\mathcal{L}(\mathcal{A}_n) = \mathcal{L}_n$.

(b) Determinize this NFA $\mathcal{A}_n$ using the powerset construction algorithm.

EXERCISE 4.3. Consider the transition system TS_{Sem} for the two-process mutual exclusion with a semaphore (see Example 2.24 on page 43) and TS_{Pet} for Peterson's algorithm (see Example 2.25 on page 45).

(a) Let P_{safe} be the regular safety property "process 1 never enters its critical section from its noncritical section (i.e., process 1 must be in its waiting location before entering the critical section)" and $AP = \{ wait_1, crit_1 \}$.

  (i) Depict an NFA for the minimal bad prefixes for P_{safe}.

  (ii) Apply the algorithm in Section 4.2 to verify $TS_{Sem} \models P_{safe}$.

(b) Let P_{safe} be the safety property "process 1 never enters its critical section from a state where $x = 2$" and $AP = \{ crit_1, x = 2 \}$.

  (i) Depict an NFA for the minimal bad prefixes for P_{safe}.

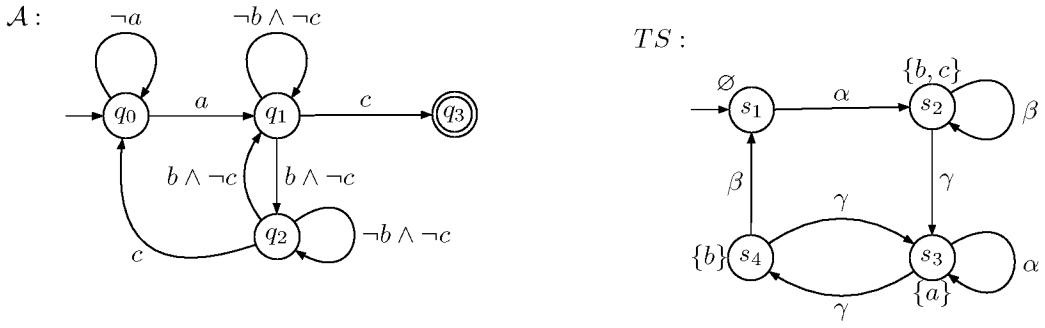
  (ii) Apply the algorithm in Section 4.2 to verify $TS_{Pet} \not\models P_{safe}$. Which counterexample is returned by the algorithm?

EXERCISE 4.4. Let P_{safe} be a safety property. Prove or disprove the following statements:

(a) If $\mathcal{L}$ is a regular language with $\mathit{MinBadPref}(P_{safe}) \subseteq \mathcal{L} \subseteq \mathit{BadPref}(P_{safe})$, then P_{safe} is regular.

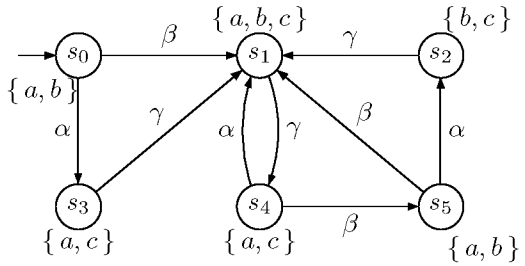
(b) If P_{safe} is regular, then any $\mathcal{L}$ for which $\mathit{MinBadPref}(P_{safe}) \subseteq \mathcal{L} \subseteq \mathit{BadPref}(P_{safe})$ is regular.

EXERCISE 4.5. Let $AP = \{ a, b, c \}$. Consider the following NFA $\mathcal{A}$ (over the alphabet 2^{AP}) and the following transition system TS:



Construct the product $TS \otimes \mathcal{A}$ of the transition system and the NFA.

EXERCISE 4.6. Consider the following transition system TS



and the regular safety property

$$P_{safe} = \begin{array}{l}\text{“always if } a \text{ is valid and } b \wedge \neg c \text{ was valid somewhere before,}\\ \text{then } a \text{ and } b \text{ do not hold thereafter at least until } c \text{ holds”}\end{array}$$

As an example, it holds:

$$\begin{array}{ll}\{b\}\varnothing\{a, b\}\{a, b, c\} & \in \mathit{pref}(P_{safe})\\ \{a, b\}\{a, b\}\varnothing\{b, c\} & \in \mathit{pref}(P_{safe})\\ \{b\}\{a, c\}\{a\}\{a, b, c\} & \in \mathit{BadPref}(P_{safe})\\ \{b\}\{a, c\}\{a, c\}\{a\} & \in \mathit{BadPref}(P_{safe})\end{array}$$

Questions:

(a) Define an NFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathit{MinBadPref}(P_{safe})$.

(b) Decide whether $TS \models P_{safe}$ using the $TS \otimes \mathcal{A}$ construction. Provide a counterexample if $TS \not\models P_{safe}$.

EXERCISE 4.7. Prove or disprove the following equivalences for ω-regular expressions:

(a) $(\mathsf{E}_1 + \mathsf{E}_2).\mathsf{F}^\omega \equiv \mathsf{E}_1.\mathsf{F}^\omega + \mathsf{E}_2.\mathsf{F}^\omega$

(b) $\mathsf{E}.(\mathsf{F}_1 + \mathsf{F}_2)^\omega \equiv \mathsf{E}.\mathsf{F}_1^\omega + \mathsf{E}.\mathsf{F}_2^\omega$

(c) $\mathsf{E}.(\mathsf{F}.\mathsf{F}^*)^\omega \equiv \mathsf{E}.\mathsf{F}^\omega$

(d) $(\mathsf{E}^*.\mathsf{F})^\omega \equiv \mathsf{E}^*.\mathsf{F}^\omega$

where $\mathsf{E}, \mathsf{E}_1, \mathsf{E}_2, \mathsf{F}, \mathsf{F}_1, \mathsf{F}_2$ are arbitrary regular expressions with $\varepsilon \notin \mathcal{L}(\mathsf{F}) \cup \mathcal{L}(\mathsf{F}_1) \cup \mathcal{L}(\mathsf{F}_2)$.

EXERCISE 4.8. Generalized ω-regular expressions are built from the symbols $\underline{\varnothing}$ (to denote the empty language), $\underline{\varepsilon}$ (to denote the language $\{\varepsilon\}$ consisting of the empty word), the symbols $\underline{A}$ for $A \in \Sigma$ (for the singleton sets $\{A\}$) and the language operators "+" (union), "." (concatenation), "$*$" (Kleene star, finite repetition), and "ω"(infinite repetition). The semantics of a generalized ω-regular expression G is a language $\mathcal{L}_g(\mathsf{G}) \subseteq \Sigma^* \cup \Sigma^\omega$, which is defined by

- $\mathcal{L}_g(\underline{\varnothing}) = \varnothing, \quad \mathcal{L}_g(\underline{\varepsilon}) = \{\varepsilon\}, \quad \mathcal{L}_g(\underline{A}) = \{A\},$
- $\mathcal{L}_g(\mathsf{G}_1 + \mathsf{G}_2) = \mathcal{L}_g(\mathsf{G}_1) \cup \mathcal{L}_g(\mathsf{G}_2)$ and $\mathcal{L}_g(\mathsf{G}_1.\mathsf{G}_2) = \mathcal{L}_g(\mathsf{G}_1).\mathcal{L}_g(\mathsf{G}_2),$
- $\mathcal{L}_g(\mathsf{G}^*) = \mathcal{L}_g(\mathsf{G})^*$, and $\mathcal{L}_g(\mathsf{G}^\omega) = \mathcal{L}_g(\mathsf{G})^\omega$.

Two generalized ω-regular expressions G and G' are called equivalent iff $\mathcal{L}_g(\mathsf{G}) = \mathcal{L}_g(\mathsf{G}')$.

Show that for each generalized ω-regular expression G there exists an equivalent generalized ω-regular expression G' of the form

$$\mathsf{G}' = \mathsf{E} + \mathsf{E}_1.\mathsf{F}_1^\omega + \ldots \mathsf{E}_n.\mathsf{F}_n^\omega$$

where $\mathsf{E}, \mathsf{E}_1, \ldots, \mathsf{E}_n, \mathsf{F}_1, \ldots, \mathsf{F}_n$ are regular expressions and $\varepsilon \notin \mathcal{L}(\mathsf{F}_i)$, $i = 1, \ldots, n$.

EXERCISE 4.9. Let $\Sigma = \{A, B\}$. Construct an NBA $\mathcal{A}$ that accepts the set of infinite words σ over Σ such that A occurs infinitely many times in σ and between any two successive A's an odd number of B's occur.

EXERCISE 4.10. Let $\Sigma = \{A, B, C\}$ be an alphabet.

(a) Construct an NBA $\mathcal{A}$ that accepts exactly the infinite words σ over Σ such that A occurs infinitely many times in σ and between any two successive A's an odd number of B's or an odd number of C's occur. Moreover, between any two successive A's either only B's or only C's are allowed. That is, the accepted words should have the form

$$wAv_1Av_2Av_3\ldots$$

where $w \in \{B, C\}^*$, $v_i \in \{B^{2k+1} \mid k \geqslant 0\} \cup \{C^{2k+1} \mid k \geqslant 0\}$ for all $i > 0$. Give also an ω-regular expression for this language.

(b) Repeat the previous exercise such that any accepting word contains only finitely many C's.

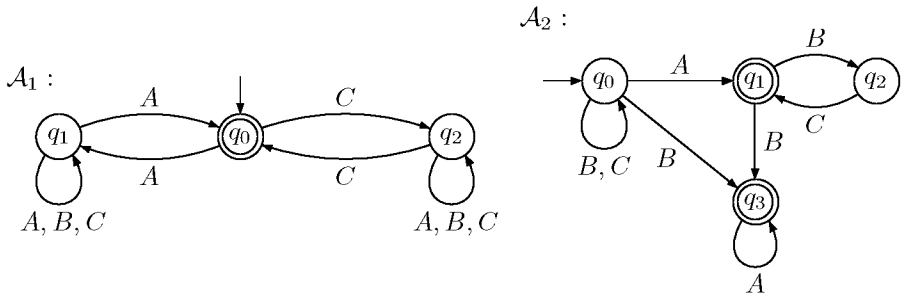
(c) Change your automaton from part (a) such that between any two successive A's an odd number of symbols from the set $\{B, C\}$ may occur.

(d) Same exercise as in (c), except that now an odd number of B's *and* an odd number of C's must occur between any two successive A symbols.

EXERCISE 4.11. Depict an NBA for the language described by the ω-regular expression

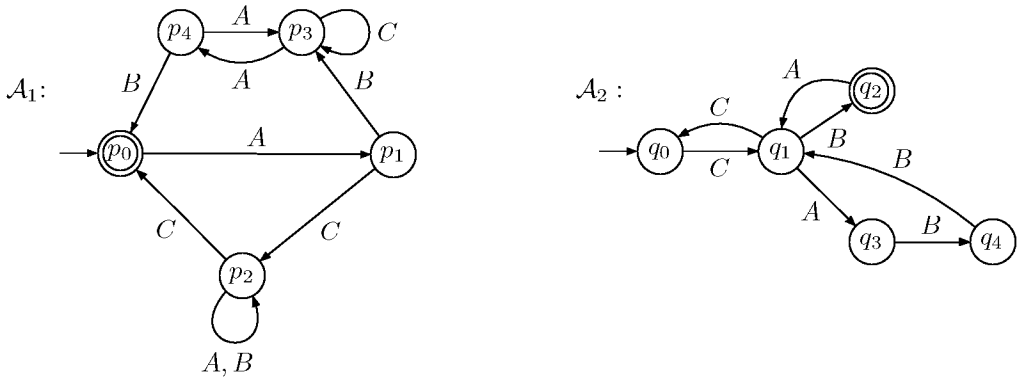
$$(AB + C)^*((AA + B)C)^\omega + (A^*C)^\omega.$$

EXERCISE 4.12. Consider the following NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ over the alphabet $\{A, B, C\}$:



Find ω-regular expressions for the languages accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$.

EXERCISE 4.13. Consider the NFA $\mathcal{A}_1$ and $\mathcal{A}_2$:



Construct an NBA for the language $\mathcal{L}(\mathcal{A}_1).\mathcal{L}(\mathcal{A}_2)^\omega$.

EXERCISE 4.14. Let $AP = \{a, b\}$. Give an NBA for the LT property consisting of the infinite words $A_0A_1A_2\ldots \in (2^{AP})^\omega$ such that

$$\overset{\infty}{\exists}\, j \geqslant 0.\, (a \in A_j \wedge b \in A_j) \quad \text{and} \quad \exists j \geqslant 0.\, (a \in A_j \wedge b \notin A_j).$$

Provide an ω-regular expression for $\mathcal{L}_\omega(\mathcal{A})$.

EXERCISE 4.15. Let $AP = \{a, b, c\}$. Depict an NBA for the LT property consisting of the infinite words $A_0A_1A_2\ldots \in (2^{AP})^\omega$ such that

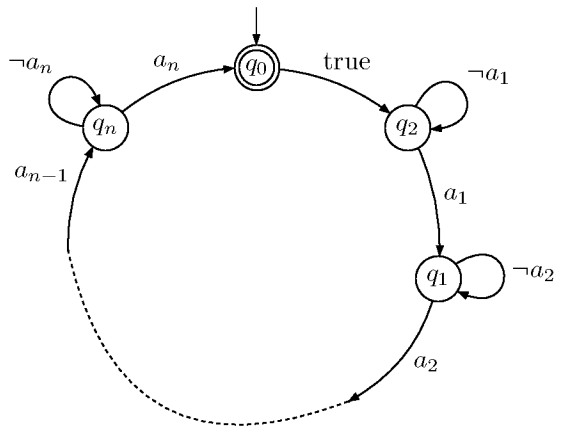
$$\forall j \geqslant 0.\, A_{2j} \models (a \ \vee \ (b \wedge c))$$

Recall that $A \models (a \vee (b \wedge c))$ means $a \in A$ or $\{b, c\} \subseteq A$, i.e., $A \in \{\{a\}, \{b, c\}, \{a, b, c\}\}$.

EXERCISE 4.16. Consider NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Figure 4.26. Show that the powerset construction applied to $\mathcal{A}_1$ and $\mathcal{A}_2$ (viewed as NFA) yields the same deterministic automaton, while $\mathcal{L}_\omega(\mathcal{A}_1) \neq \mathcal{L}_\omega(\mathcal{A}_2)$. (This exercise is taken from [408].)

Figure 4.26: NBA $\mathcal{A}_1$ (a) and $\mathcal{A}_2$ (b).

EXERCISE 4.17. Consider the following NBA $\mathcal{A}$ with the alphabet $\Sigma = 2^{AP}$ where $AP = \{a_1, \ldots, a_n\}$ for $n > 0$.



(a) Determine the accepted language $\mathcal{L}_\omega(\mathcal{A})$.

(b) Show that there is no NBA $\mathcal{A}'$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}')$ and less than n states.

(This exercise is inspired by [149].)

EXERCISE 4.18. Provide an example for a regular safety property P_{safe} over AP and an NFA $\mathcal{A}$ for its minimal bad prefixes such that

$$\mathcal{L}_\omega(\mathcal{A}) \;\neq\; \left(2^{AP}\right)^\omega \setminus P_{safe}$$

when $\mathcal{A}$ is viewed as an NBA.

EXERCISE 4.19. Provide an example for a liveness property that is not ω-regular. Justify your answer.

EXERCISE 4.20. Is there a DBA that accepts the language described by the ω-regular expression $(A+B)^*(AB+BA)^\omega$? Justify your answer.

EXERCISE 4.21. Provide an example for an ω-regular language $\mathcal{L} = \mathcal{L}_k$ that is recognizable for a DBA such that the following two conditions are satisfied:

(a) There exists an NBA $\mathcal{A}$ with $|\mathcal{A}| = \mathcal{O}(k)$ and $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}$.

(b) Each DBA $\mathcal{A}'$ for $\mathcal{L}$ is of the size $|\mathcal{A}'| = \Omega(2^k)$.

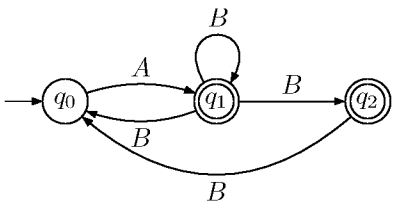
Hint: There is a simple answer to this question that uses the result that the regular language for the expression $(A+B)^*B(A+B)^k$ is recognizable by an NFA of size $\mathcal{O}(k)$, while any DFA has $\Omega(2^k)$ states.

EXERCISE 4.22. Show that the class of languages that are accepted by DBAs is not closed under complementation.

EXERCISE 4.23. Show that the class of languages that are accepted by DBAs is closed under union. To do so, prove the following stronger statement:

Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two DBAs both over the alphabet Σ. Show that there exists a DBA $\mathcal{A}$ with $|\mathcal{A}| = \mathcal{O}(|\mathcal{A}_1| \cdot |\mathcal{A}_2|)$ and $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{A}_1) \cup \mathcal{L}_\omega(\mathcal{A}_2)$.

EXERCISE 4.24.
Consider the GNBA outlined on the right with acceptance sets $F_1 = \{ q_1 \}$ and $F_2 = \{ q_2 \}$. Construct an equivalent NBA.

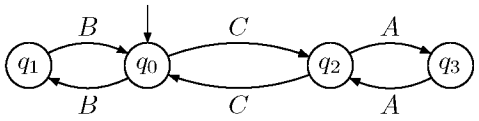


EXERCISE 4.25. Provide NBA $\mathcal{A}_1$ and $\mathcal{A}_2$ for the languages given by the expressions $(AC+B)^*B^\omega$ and $(B^*AC)^\omega$ and apply the product construction to obtain a GNBA $\mathcal{G}$ with $\mathcal{L}_\omega(\mathcal{G}) = \mathcal{L}_\omega(\mathcal{A}_1) \cap \mathcal{L}_\omega(\mathcal{A}_2)$. Justify that $\mathcal{L}_\omega(\mathcal{G}) = \varnothing$.

EXERCISE 4.26. A nondeterministic Muller automaton is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ where Q, Σ, δ, Q_0 are as for NBA and $\mathcal{F} \subseteq 2^Q$. For an infinite run ρ of $\mathcal{A}$, let $\lim(\rho) := \Big\{ q \in Q \mid \overset{\infty}{\exists}\, i \geq 0.\ \rho[i] = q \Big\}$. Let $\alpha \in \Sigma^\omega$.

$$\mathcal{A} \text{ accepts } \alpha \iff \text{ex. infinite run } \rho \text{ of } \mathcal{A} \text{ on } \alpha \text{ s.t. } \lim(\rho) \in \mathcal{F}$$

(a) Consider the following Muller automaton $\mathcal{A}$ with $\mathcal{F} = \{\{q_2, q_3\}, \{q_1, q_3\}, \{q_0, q_2\}\}$:



Define the language accepted by $\mathcal{A}$ by means of an ω-regular expression.

(b) Show that every GNBA $\mathcal{G}$ can be transformed into a nondeterministic Muller automaton $\mathcal{A}$ such that $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{G})$ by defining the corresponding transformation.

EXERCISE 4.27. Consider the transition systems TS_{Sem} and TS_{Pet} for mutual exclusion with a semaphore and the Peterson algorithm, respectively. Let P_{live} be the following ω-regular property over $AP = \{ wait_1, crit_1 \}$:

"whenever process 1 is in its waiting location then it will eventually enter its critical section"

(a) Depict an NBA for P_{live} and an NBA $\bar{\mathcal{A}}$ for the complement property $\bar{P}_{live} = \big(2^{AP}\big)^\omega \setminus P_{live}$.

(b) Show that $TS_{Sem} \not\models P_{live}$ by applying the techniques explained in Section 4.4:

(i) Depict the reachable fragment of the product $TS_{Sem} \otimes \bar{\mathcal{A}}$

(ii) Sketch the main steps of the nested depth-first search applied to $TS_{Sem} \otimes \bar{\mathcal{A}}$ for the persistence property "eventually forever $\neg F$'" where F' is the acceptance set of $\bar{\mathcal{A}}$. Which counterexample is returned by Algorithm ?8

(c) Apply now the same techniques (product construction, nested DFS) to show that $TS_{Pet} \models P_{live}$.

EXERCISE 4.28. The nested depth-first search approach can also be reformulated for an emptiness check for NBA. The path fragment returned by Algorithm 8 in case of a negative answer then yields a prefix of an accepting run.

Consider the automaton shown in Exercise 4.24 as an NBA, i.e., the acceptance set is $F' = \{ q_1, q_2 \}$. Apply the nested depth-first search approach to verify that $\mathcal{L}_\omega(\mathcal{A}) \neq \varnothing$.

Chapter 5

Linear Temporal Logic

This chapter introduces (propositional) linear temporal logic (LTL), a logical formalism that is suited for specifying LT properties. The syntax and semantics of linear temporal logic are defined. Various examples are provided that show how linear temporal logic can be used to specify important system properties. The second part of the chapter is concerned with a model-checking algorithm—based on Büchi automata—for LTL. This algorithm can be used to answer the question: given a transition system TS and LTL-formula φ, how to check whether φ holds in TS?

5.1 Linear Temporal Logic

For reactive systems, correctness depends on the executions of the system—not only on the input and output of a computation—and on fairness issues. Temporal logic is a formalism par excellence for treating these aspects. Temporal logic extends propositional or predicate logic by modalities that permit to referral to the infinite behavior of a reactive system. They provide a very intuitive but mathematically precise notation for expressing properties about the relation between the state labels in executions, i.e., LT properties. Temporal logics and related modal logics have been studied in ancient times in different areas such as philosophy. Their application to verifying complex computer systems was proposed by Pnueli in the late seventies.

In this monograph, we will focus our attention on *propositional temporal logics*, i.e., extensions of propositional logic by temporal modalities. These logics should be distinguished from first- (or higher-) order temporal logics that impose temporal modalities on top of

predicate logic. Throughout this monograph we assume some familiarity with the basic principles of propositional logic. A brief introduction and summary of our notations can be found in Appendix A.3. The elementary temporal modalities that are present in most temporal logics include the operators:

$\Diamond$ "eventually" (eventually in the future)
$\Box$ "always" (now and forever in the future)

The underlying nature of time in temporal logics can be either *linear* or *branching*. In the linear view, at each moment in time there is a single successor moment, whereas in the branching view it has a branching, tree-like structure, where time may split into alternative courses. This chapter considers LTL (Linear Temporal Logic), a temporal logic that is based on a linear-time perspective. Chapter 6 introduces CTL (Computation Tree Logic), a logic that that is based on a branching-time view. Several model-checking tools use LTL (or a slight variant thereof) as a property specification language. The model checker SPIN is a prominent example of such an automated verification tool. One of the main advantages of LTL is that imposing fairness assumptions (such as strong and weak fairness) does not require the use of any new machinery: the typical fairness assumptions can all be specified in LTL. Verifying LTL-formulae under fairness constraints can be done using the algorithm for LTL. This does not apply to CTL.

Before introducing LTL in more detail, a short comment on the adjective "temporal" is in order to avoid any possible confusion. Although the term *temporal* suggests a relationship with the real-time behavior of a reactive system, this is only true in an abstract sense. A temporal logic allows for the specification of the relative *order* of events. Some examples are "the car stops once the driver pushes the brake", or "the message is received after it has been sent". It does however *not* support any means to refer to the precise timing of events. The fact that there is a minimal delay of at least 3 μs between braking and the actual halting of the car cannot be specified. In terms of transition systems, neither the duration of taking a transition nor state residence times can be specified using the elementary modalities of temporal logics. Instead, these modalities do allow for specifying the order in which state labels occur during an execution, or to assess that certain state labels occur infinitely often in a (or all) system execution. One might thus say that the modalities in temporal logic are *time-abstract*.

As will be discussed in this chapter, LTL may be used to express the timing for the class of synchronous systems in which all components proceed in a lock-step fashion. In this setting, a transition corresponds to the advance of a single time-unit. The underlying time domain is thus *discrete*, i.e., the present moment refers to the current state and the next moment corresponds to the immediate successor state. Stated differently, the system behavior is assumed to be observable at the time points $0, 1, 2, \ldots$. The treatment

of real-time constraints in asynchronous systems by means of a *continuous-time* domain will be discussed in Chapter 9 where a timed version of CTL, called Timed CTL, will be introduced. Table 5.1 summarizes the distinguishing features of the main temporal logics considered in this monograph.

logic	linear-time (path-based)	branching-time (state-based)	real-time requirements (continuous-time domain)
LTL	√		
CTL		√	
Timed CTL		√	√

Table 5.1: Classification of the temporal logics in this monograph.

5.1.1 Syntax

This subsection describes the syntactic rules according to which formulae in LTL can be constructed. The basic ingredients of LTL-formulae are atomic propositions (state labels $a \in AP$), the Boolean connectors like conjunction $\wedge$, and negation $\neg$, and two basic temporal modalities $\bigcirc$ (pronounced "next") and U (pronounced "until"). The atomic proposition $a \in AP$ stands for the state label a in a transition system. Typically, the atoms are assertions about the values of control variables (e.g., locations in program graphs) or the values of program variables such as "$x > 5$" or "$x \leqslant y$". The $\bigcirc$-modality is a unary prefix operator and requires a single LTL formula as argument. Formula $\bigcirc \varphi$ holds at the current moment, if φ holds in the next "step". The U-modality is a binary infix operator and requires two LTL formulae as argument. Formula $\varphi_1 \,\mathsf{U}\, \varphi_2$ holds at the current moment, if there is some future moment for which φ_2 holds and φ_1 holds at all moments until that future moment.

Definition 5.1. Syntax of LTL

LTL formulae over the set AP of atomic proposition are formed according to the following grammar:[1]

$$\varphi ::= \text{true} \;\Big|\; a \;\Big|\; \varphi_1 \wedge \varphi_2 \;\Big|\; \neg\varphi \;\Big|\; \bigcirc\, \varphi \;\Big|\; \varphi_1 \,\mathsf{U}\, \varphi_2$$

where $a \in AP$. ■

[1] The Backus Naur form (BNF) is used in a somewhat liberal way. More concretely, nonterminals are identified with derived words (formulae) and indices in the rules. Moreover, brackets will be used, e.g. in $a \wedge (b \,\mathsf{U}\, c)$, which are not shown in the grammar. Such simplified notations for grammars to determine the syntax of formulae of some logic (or terms of other calculi) are often called *abstract* syntax.

We mostly abstain from explicitly indicating the set AP of propositions as this follows either from the context or can be defined as the set of atomic propositions occurring in the LTL formula at hand.

The precedence order on the operators is as follows. The unary operators bind stronger than the binary ones. $\neg$ and $\bigcirc$ bind equally strong. The temporal operator U takes precedence over $\wedge$, $\vee$, and $\rightarrow$. Parentheses are omitted whenever appropriate, e.g., we write $\neg\varphi_1 \,\mathsf{U}\, \bigcirc\, \varphi_2$ instead of $(\neg\varphi_1)\,\mathsf{U}\,(\bigcirc\,\varphi_2)$. Operator U is right-associative, e.g., $\varphi_1 \,\mathsf{U}\, \varphi_2 \,\mathsf{U}\, \varphi_3$ stands for $\varphi_1 \,\mathsf{U}\, (\varphi_2 \,\mathsf{U}\, \varphi_3)$.

Using the Boolean connectives $\wedge$ and $\neg$, the full power of propositional logic is obtained. Other Boolean connectives such as disjunction $\vee$, implication $\rightarrow$, equivalence $\leftrightarrow$, and the parity (or: exclusive or) operator $\oplus$ can be derived as follows:

$$\begin{array}{lcl}
\varphi_1 \vee \varphi_2 & \stackrel{\text{def}}{=} & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
\varphi_1 \rightarrow \varphi_2 & \stackrel{\text{def}}{=} & \neg\varphi_1 \;\vee\; \varphi_2 \\
\varphi_1 \leftrightarrow \varphi_2 & \stackrel{\text{def}}{=} & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
\varphi_1 \oplus \varphi_2 & \stackrel{\text{def}}{=} & (\varphi_1 \wedge \neg\varphi_2) \;\vee\; (\varphi_2 \wedge \neg\varphi_1) \\
& \vdots &
\end{array}$$

The until operator allows to derive the temporal modalities $\Diamond$ ("eventually", sometimes in the future) and $\Box$ ("always", from now on forever) as follows:

$$\Diamond\varphi \stackrel{\text{def}}{=} \text{true}\,\mathsf{U}\,\varphi \qquad \Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$$

As a result, the following intuitive meaning of $\Diamond$ and $\Box$ is obtained. $\Diamond\varphi$ ensures that φ will be true eventually in the future. $\Box\varphi$ is satisfied if and only if it is not the case that eventually $\neg\varphi$ holds. This is equivalent to the fact that φ holds from now on forever.

Figure 5.1 sketches the intuitive meaning of temporal modalities for the simple case in which the arguments of the modalities are just atomic propositions from $\{a, b\}$. On the left-hand side, some LTL formulae are indicated, whereas on the right hand side sequences of states (i.e., paths) are depicted.

By combining the temporal modalities $\Diamond$ and $\Box$, new temporal modalities are obtained. For instance, $\Box\Diamond a$ ("always eventually a") describes the (path) property stating that at any moment j there is a moment $i \geqslant j$ at which an a-state is visited. This thus amounts to assert that an a-state is visited infinitely often. The dual modality $\Diamond\Box a$ expresses that from some moment j on, only a-states are visited. So:

$\Box\Diamond\varphi$ "infinitely often φ"

$\Diamond\Box\varphi$ "eventually forever φ"

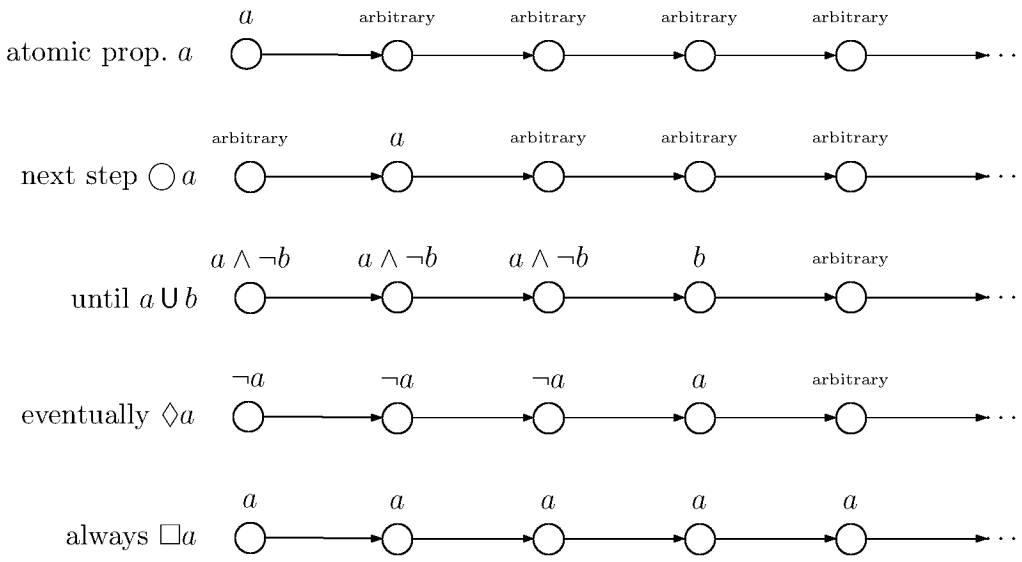


Figure 5.1: Intuitive semantics of temporal modalities.

Before proceeding with the formal semantics of LTL, we present some examples.

Example 5.2. Properties for the Mutual Exclusion Problem

Consider the mutual exclusion problem for two concurrent processes P_1 and P_2, say. Process P_i is modeled by three locations: (1) the noncritical section, (2) the waiting phase which is entered when the process intends to enter the critical section, and (3) the critical section. Let the propositions $wait_i$ and $crit_i$ denote that process P_i is in its waiting phase and critical section, respectively.

The safety property stating that P_1 and P_2 never simultaneously have access to their critical sections can be described by the LTL-formula:

$$\Box(\neg\, crit_1 \ \vee \ \neg\, crit_2).$$

This formula expresses that always ($\Box$) at least one of the two processes is not in its critical section ($\neg crit_i$).

The liveness requirement stating that each process P_i is infinitely often in its critical

section is described by the LTL formula:

$$(\Box\Diamond \mathit{crit}_1) \ \wedge \ (\Box\Diamond \mathit{crit}_2).$$

The weakened form that every waiting process will eventually enter its critical section (i.e., starvation freedom) can—by using the additional proposition wait_i—be formulated as follows:

$$(\Box\Diamond \mathit{wait}_1 \rightarrow \Box\Diamond \mathit{crit}_1) \ \wedge \ (\Box\Diamond \mathit{wait}_2 \rightarrow \Box\Diamond \mathit{crit}_2).$$

These formulae only refer to the locations (i.e., values of the program counters) by the atomic propositions wait_i and crit_i. Propositions can however also refer to program variables. For instance, for the solution of the mutual exclusion problem using a binary semaphore y, the formula:

$$\Box((y = 0) \rightarrow \mathit{crit}_1 \ \vee \ \mathit{crit}_2)$$

states that whenever the semaphore y has the value 0, one of the processes is in its critical section. ■

Example 5.3. Properties for the dining philosophers

For the dining philosophers (see Example 3.2 on page 90) deadlock freedom can be described by the LTL formula

$$\Box\neg\Big(\bigwedge_{0\leqslant i<n} \mathit{wait}_i \ \wedge \ \bigwedge_{0\leqslant i<n} \mathit{occupied}_i\Big).$$

We assume here that there are n philosophers and chop sticks, indexed from 0 to $n-1$. The atom wait_i means that philosopher i waits for one of the sticks on his left or right, but keeps the other one in his hand. Similarly, $\mathit{occupied}_i$ indicates that stick i is in use. ■

Example 5.4. Properties for a Traffic Light

For a traffic light with the phases "green", "red" and "yellow", the liveness property $\Box\Diamond \mathit{green}$ expresses that the traffic light is infinitely often green. A specification of the traffic light cycles and their chronological order can be provided by means of a conjunction of LTL-formulae stating the predecessor phase of any phase. For instance, the requirement "once red, the light cannot become green immediately" can be expressed by the LTL formula

$$\Box(\mathit{red} \rightarrow \neg \bigcirc \mathit{green}).$$

The requirement "once red, the light always becomes green eventually after being yellow for some time" is expressed by

$$\Box(\mathit{red} \rightarrow \bigcirc(\mathit{red} \,\mathsf{U}\, (\mathit{yellow} \ \wedge \ \bigcirc \ (\mathit{yellow} \,\mathsf{U}\, \mathit{green})))).$$

A progress property like "every request will eventually lead to a response" can be described by the following formula of the type

$$\Box(\mathit{request} \rightarrow \Diamond \mathit{response}).$$

■

Remark 5.5. Length of a Formula

Let $|\varphi|$ denote the length of LTL formula φ in terms of the number of operators in φ. This can easily be defined by induction on the structure of φ. For instance, the length of the formula true and $a \in AP$ is 0. Formulae $\bigcirc a \vee b$ and $a \vee \neg b$ have length 2, and $(\bigcirc a) \,\mathsf{U}\, (a \wedge \neg b)$ has length 4. Throughout this monograph, mostly the asymptotic size $\Theta(|\varphi|)$ is needed. For this purpose, it is irrelevant whether or not the derived Boolean operators $\vee$, $\rightarrow$, and so on, and the derived temporal modalities $\Diamond$ and $\Box$ are taken into account in determining the length. ■

5.1.2 Semantics

LTL formulae stand for properties of paths (or in fact their trace). This means that a path can either fulfill an LTL-formula or not. To precisely formulate when a path satisfies an LTL formula, we proceed as follows. First, the semantics of LTL formula φ is defined as a language $\mathit{Words}(\varphi)$ that contains all infinite words over the alphabet 2^{AP} that satisfy φ. That is, to every LTL formula a single LT property is associated. Then, the semantics is extended to an interpretation over paths and states of a transition system.

Definition 5.6. Semantics of LTL (Interpretation over Words)

Let φ be an LTL formula over AP. The LT property induced by φ is

$$\mathit{Words}(\varphi) \;=\; \Big\{\sigma \in (2^{AP})^{\omega} \mid \sigma \models \varphi\Big\}$$

where the satisfaction relation $\models \subseteq \; (2^{AP})^{\omega} \times \mathrm{LTL}$ is the smallest relation with the properties in Figure 5.2. ■

Here, for $\sigma = A_0\, A_1\, A_2 \ldots \in (2^{AP})^{\omega}$, $\sigma[j \ldots] = A_j\, A_{j+1}\, A_{j+2} \ldots$ is the suffix of σ starting in the $(j{+}1)$st symbol A_j.

Note that in the definition of the semantics of LTL-formulae the word fragment $\sigma[j \ldots]$ cannot be replaced with A_j. For the formula $\bigcirc (a \,\mathsf{U}\, b)$, e.g., the suffix $A_1\, A_2\, A_3 \ldots$ has to

$$
\begin{array}{lcll}
\sigma & \models & \text{true} & \\
\sigma & \models & a & \text{iff} \quad a \in A_0 \quad \text{(i.e., } A_0 \models a\text{)} \\
\sigma & \models & \varphi_1 \wedge \varphi_2 & \text{iff} \quad \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma & \models & \neg\, \varphi & \text{iff} \quad \sigma \not\models \varphi \\
\sigma & \models & \bigcirc \varphi & \text{iff} \quad \sigma[1\ldots] = A_1A_2A_3\ldots \models \varphi \\
\sigma & \models & \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} \quad \exists j \geqslant 0.\ \sigma[j\ldots] \models \varphi_2 \ \text{ and } \ \sigma[i\ldots] \models \varphi_1, \text{ for all } 0 \leqslant i < j
\end{array}
$$

Figure 5.2: LTL semantics (satisfaction relation $\models$) for infinite words over 2^{AP}.

be regarded in order to be able to refer to the truth-value of the subformula $a \,\mathsf{U}\, b$ in the "next step".

For the derived operators $\Diamond$ and $\Box$ the expected result is:

$$
\begin{array}{lcll}
\sigma & \models & \Diamond\varphi & \text{iff} \quad \exists j \geqslant 0.\ \sigma[j\ldots] \models \varphi \\
\sigma & \models & \Box\varphi & \text{iff} \quad \forall j \geqslant 0.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

The statement for $\Diamond$ is immediate from the definition of $\Diamond$ and the semantics of U. The statement for $\Box$ follows from:

$$
\begin{array}{lll}
\sigma \models \Box\varphi = \neg\Diamond\neg\varphi & \text{iff} & \neg\exists j \geqslant 0.\ \sigma[j\ldots] \models \neg\varphi \\
& \text{iff} & \neg\exists j \geqslant 0.\ \sigma[j\ldots] \not\models \varphi \\
& \text{iff} & \forall j \geqslant 0.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

The semantics of the combinations of $\Box$ and $\Diamond$ can now be derived:

$$
\begin{array}{lcll}
\sigma & \models & \Box\Diamond\varphi & \text{iff} \quad \overset{\infty}{\exists}\, j.\ \sigma[j\ldots] \models \varphi \\
\sigma & \models & \Diamond\Box\varphi & \text{iff} \quad \overset{\infty}{\forall}\, j.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

Here, $\overset{\infty}{\exists}\, j$ means $\forall i \geqslant 0.\ \exists j \geqslant i$, "for infinitely many $j \in \mathbb{N}$", while $\overset{\infty}{\forall}\, j$ stands for $\exists i \geqslant 0.\ \forall j \geqslant i$, "for almost all $j \in \mathbb{N}$". Let us verify the first statement. The argument for the second statement is similar.

$$
\begin{array}{lll}
\sigma \models \Box\Diamond\varphi & \text{iff} & \forall i \geqslant 0.\ \sigma[i\ldots] \models \Diamond\varphi \\
& \text{iff} & \forall i \geqslant 0.\ \exists j \geqslant i.\ \sigma[j\ldots] \models \varphi \\
& \text{iff} & \overset{\infty}{\exists}\, j.\ \sigma[j\ldots] \models \varphi.
\end{array}
$$

As a subsequent step, we determine the semantics of LTL-formulae with respect to a

transition system. According to the satisfaction relation for LT properties (see Definition 3.11 on page 100), the LTL formula φ holds in state s if all paths starting in s satisfy φ. The transition system TS satisfies φ if TS satisfies the LT property $Words(\varphi)$, i.e., if *all* initial paths of TS—paths starting in an initial state $s_0 \in I$—satisfy φ.

Recall that we may assume without loss of generality that transition system TS has no terminal states (if it has such states, a trap state can be introduced. Thus, we may assume that all paths and traces are infinite. This assumption is made for the sake of simplicity; it is also possible to define the semantics of LTL for finite paths. Note that for the semantics it is irrelevant whether or not TS is finite. Only for the model-checking algorithm later on in this chapter, is the finiteness of TS required.

As for the LT properties, when defining $TS \models \varphi$ for transition system TS over AP', it is assumed that φ is an LTL-formula with atomic propositions in $AP = AP'$. (Here, one could be more liberal and allow for $AP \subseteq AP'$.)

Definition 5.7. Semantics of LTL over Paths and States

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, and let φ be an LTL-formula over AP.

- For infinite path fragment π of TS, the satisfaction relation is defined by
$$\pi \models \varphi \quad \text{iff} \quad trace(\pi) \models \varphi.$$
- For state $s \in S$, the satisfaction relation $\models$ is defined by
$$s \models \varphi \quad \text{iff} \quad (\forall \pi \in Paths(s).\ \pi \models \varphi).$$
- TS satisfies φ, denoted $TS \models \varphi$, if $Traces(TS) \subseteq Words(\varphi)$.

■

From this definition, it immediately follows that

$TS \models \varphi$
iff (* Definition 5.7 *)
$Traces(TS) \subseteq Words(\varphi)$
iff (* Definition of $\models$ for LT properties *)
$TS \models Words(\varphi)$
iff (* Definition of $Words(\varphi)$ *)
$\pi \models \varphi$ for all $\pi \in Paths(TS)$
iff (* Definition 5.7 of $\models$ for states *)
$s_0 \models \varphi$ for all $s_0 \in I$.

Thus, $TS \models \varphi$ if and only if $s_0 \models \varphi$ for all initial states s_0 of TS.

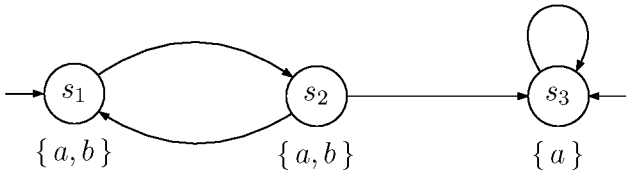


Figure 5.3: Example for semantics of LTL.

Example 5.8. Semantics of LTL

Consider the transition system TS depicted in Figure 5.3 with the set of propositions $AP = \{a, b\}$. For example, we have that $TS \models \Box a$, since all states are labeled with a, and hence, all traces of TS are words of the form $A_0 A_1 A_2 \ldots$ with $a \in A_i$ for all $i \geqslant 0$. Thus, $s_i \models \Box a$ for $i = 1, 2, 3$. Moreover:

$$s_1 \models \bigcirc (a \wedge b) \text{ since } s_2 \models a \wedge b \text{ and } s_2 \text{ is the only successor of } s_1$$

$$s_2 \not\models \bigcirc (a \wedge b) \text{ and } s_3 \not\models \bigcirc (a \wedge b) \text{ as } s_3 \in \mathit{Post}(s_2), s_3 \in \mathit{Post}(s_3) \text{ and } s_3 \not\models a \wedge b.$$

This yields $TS \not\models \bigcirc (a \wedge b)$ as s_3 is an initial state for which $s_3 \not\models \bigcirc (a \wedge b)$. As another example:

$$TS \models \Box(\neg b \rightarrow \Box(a \wedge \neg b)),$$

since s_3 is the only $\neg b$ state, s_3 cannot be left anymore, and $a \wedge \neg b$ in s_3 is true. However,

$$TS \not\models b \,\mathsf{U}\, (a \wedge \neg b),$$

since the initial path $(s_1 s_2)^\omega$ does not visit a state for which $a \wedge \neg b$ holds. Note that the initial path $(s_1 s_2)^* s_3^\omega$ satisfies $b \,\mathsf{U}\, (a \wedge \neg b)$. ■

Remark 5.9. Semantics of Negation

For paths, it holds $\pi \models \varphi$ if and only if $\pi \not\models \neg\varphi$. This is due to the fact that

$$\mathit{Words}(\neg\varphi) = (2^{AP})^\omega \setminus \mathit{Words}(\varphi).$$

However, the statements $TS \not\models \varphi$ and $TS \models \neg\varphi$ are *not* equivalent in general. Instead, we have $TS \models \neg\varphi$ implies $TS \not\models \varphi$. Note that

$$\begin{aligned} TS \not\models \varphi \quad &\text{iff} \quad \mathit{Traces}(TS) \not\subseteq \mathit{Words}(\varphi) \\ &\text{iff} \quad \mathit{Traces}(TS) \setminus \mathit{Words}(\varphi) \neq \varnothing \\ &\text{iff} \quad \mathit{Traces}(TS) \cap \mathit{Words}(\neg\varphi) \neq \varnothing. \end{aligned}$$

Thus, it is possible that a transition system (or a state) satisfies neither φ nor $\neg\varphi$. This is caused by the fact that there might be paths π_1 and π_2 in TS such that $\pi_1 \models \varphi$ and $\pi_2 \models \neg\varphi$ (and therefore $\pi_2 \not\models \varphi$). In this case, $TS \not\models \varphi$ and $TS \not\models \neg\varphi$ holds.

To illustrate this effect, consider the transition system depicted in Figure 5.4. Let $AP = \{ a \}$. It follows that $TS \not\models \Diamond a$, since the initial path $s_0(s_2)^\omega \not\models \Diamond a$. On the other hand, $TS \not\models \neg\Diamond a$ also holds, since the initial path $s_0(s_1)^\omega \models \Diamond a$, and thus, $s_0(s_1)^\omega \not\models \neg\Diamond a$. ■

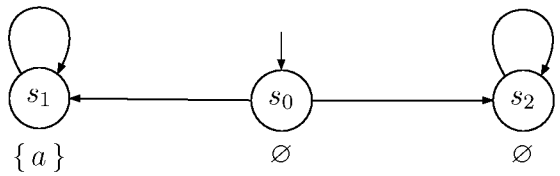


Figure 5.4: A transition system for which $TS \not\models \Diamond a$ and $TS \not\models \neg\Diamond a$.

5.1.3 Specifying Properties

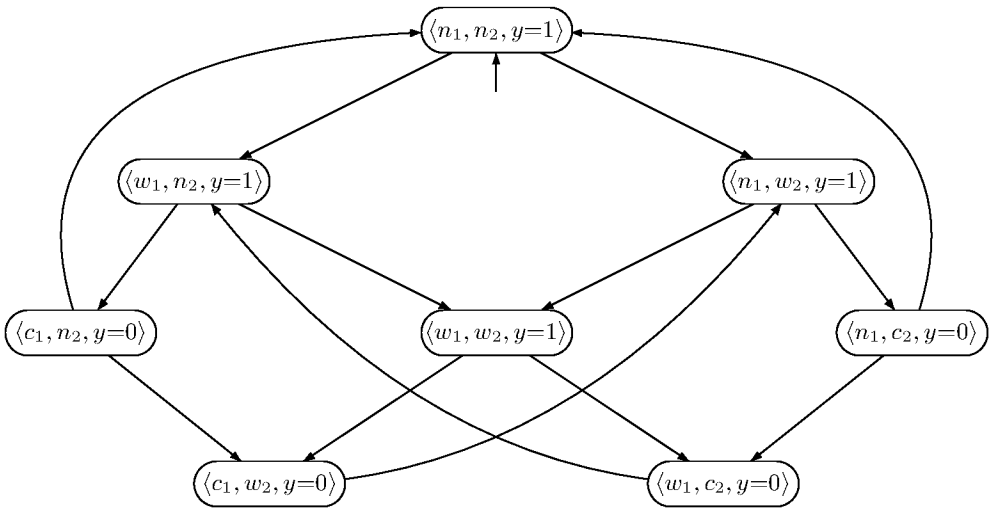


Figure 5.5: Transition system of semaphore-based mutual exclusion algorithm.

Example 5.10. Semaphore-Based Mutual Exclusion Revisited

Consider the transition system TS_{Sem} depicted in Figure 5.5 which represents a semaphore-based solution to the mutual exclusion problem; see also Example 3.9 on page 98. Each

state of the form $\langle c_1, \cdot, \cdot \rangle$ is labeled with proposition $crit_1$ and each state of the form $\langle \cdot, c_2, \cdot \rangle$ is labeled with $crit_2$. It follows that

$$TS_{Sem} \models \Box(\neg crit_1 \vee \neg crit_2) \quad \text{and} \quad TS_{Sem} \models \Box\Diamond crit_1 \vee \Box\Diamond crit_2,$$

where the first LTL-formula stands for the mutual exclusion property and the second LTL-formula for the fact that at least one of the two processes enters its critical section infinitely often. However,

$$TS_{Sem} \not\models \Box\Diamond crit_1 \wedge \Box\Diamond crit_2,$$

since—in the absence of any fairness assumption—it is not ensured that process P_1 is enabled infinitely often. It may not be able to acquire access to its critical section once. (A similar argument applies to process P_2.) The same argument applies to show that

$$TS_{Sem} \not\models \Box\Diamond wait_1 \rightarrow \Box\Diamond crit_1$$

as in principle process P_1 may not get its turn once it starts to wait. ■

Example 5.11. Modulo 4 Counter

A modulo 4 counter can be represented by a sequential circuit C, which outputs 1 in every fourth cycle, otherwise 0. C has no input bits, one output bit y, and two registers r_1 and r_2. The register evaluation $[r_1 = c_1, r_2 = c_2]$ can be identified with the number $i = 2 \cdot r_1 + r_2$. In every cycle, the value of i is increased by 1 (modulo 4). We construct C in a way such that the output bit y is set exactly for $i = 0$ (hence, $r_1 = r_2 = 0$). The transition relation and output function are given by

$$\delta_{r_1} = r_1 \oplus r_2, \quad \delta_{r_2} = \neg r_1, \quad \lambda_y = \neg r_1 \wedge \neg r_2.$$

Figure 5.6 illustrates the diagram (on the left) and the transition system TS_C (on the right). Let $AP = \{ r_1, r_2, y \}$. The following statement can be directly inferred from TS_C:

$$\begin{aligned} TS_C &\models \Box (y \leftrightarrow \neg r_1 \wedge \neg r_2) \\ TS_C &\models \Box (r_1 \rightarrow (\bigcirc y \vee \bigcirc \bigcirc y)) \\ TS_C &\models \Box (y \rightarrow (\bigcirc \neg y \wedge \bigcirc \bigcirc \neg y)) \end{aligned}$$

If it is assumed that only the output variable y (and not the register evaluations) can be perceived by an observer, then an appropriate choice for AP is $AP = \{ y \}$. The property that at least during every four cycles the output 1 is obtained holds for TS_C, i.e., we have

$$TS_C \models \Box (y \vee \bigcirc y \vee \bigcirc \bigcirc y \vee \bigcirc \bigcirc \bigcirc y).$$

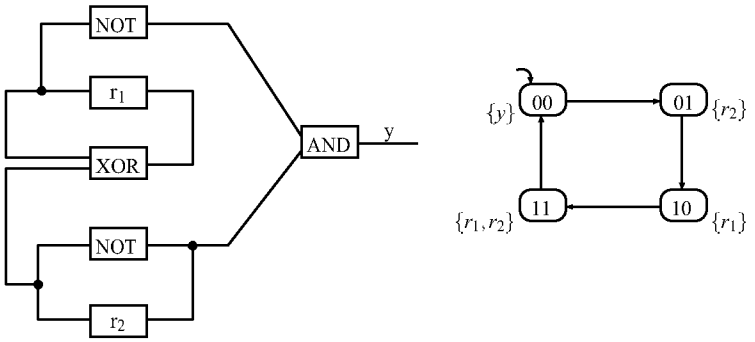


Figure 5.6: A modulo 4 counter.

The fact that these outputs are produced in a periodic manner where every fourth cycle yields the output 1 is expressed as

$$TS_C \models \Box\,(\,y \longrightarrow (\bigcirc\neg y \wedge \bigcirc\bigcirc\neg y \wedge \bigcirc\bigcirc\bigcirc\neg y)\,).$$

■

Example 5.12. A Communication Channel

Consider an unidirectional channel between two communicating processes, a sender S and a receiver R. Sender S is equipped with an output buffer $S.out$ and recipient R with an input buffer $R.in$. If sender S sends a message m to R it inserts the message into its output buffer $S.out$. The output buffer $S.out$ and the input buffer $R.in$ are connected via an unidirectional channel. The receiver R receives messages by deleting messages from its input buffer $R.in$. The capacity of the buffers is not of importance here.

A schematic view of the system under consideration is:

In the following LTL-specifications, we use the atoms "$m \in S.out$" and "$m \in R.in$" where m is an arbitrary message. We formalize the following informal requirements by LTL formulae:

- "Whenever message m is in the out-buffer of S, then m will eventually be consumed by the receiver."

$$\Box(m \in S.out \longrightarrow \Diamond(m \in R.in))$$

The above property is still satisfied for paths $s_1 s_2 s_3 \ldots$ where $s_1 \models m \in S.out$, $s_2 \models m \notin S.out$, $s_2 \models m \notin R.in$, and $s_3 \models m \in R.in$. However, such paths stand for a mysterious behavior where message m in the output buffer for S (state s_1) gets lost (state s_2), but still arrives in the input buffer of R (state s_3). In fact, such a behavior is impossible for a reliable FIFO channel which satisfies the following stronger condition

$$\Box(m \in S.out \longrightarrow (m \in S.out \,\mathsf{U}\, m \in R.in))$$

stating that message m stays in $S.out$ until the receiver R consumes m. Since writing and reading in a FIFO channel cannot happen at the same moment, we can even use the formula

$$\Box(m \in S.out \longrightarrow \bigcirc (m \in S.out \,\mathsf{U}\, m \in R.in)).$$

- If we assume that no message occurs twice in $S.out$ then the asynchronous behavior of a FIFO channel ensures that the property "message m cannot be in both buffers at the same time". This is formalized by the LTL formula:

$$\Box \neg (m \in S.out \ \wedge \ m \in R.in).$$

- The characteristic of FIFO-channels is that they are "order-preserving" according to the "first in, first out principle" stating that if message m is offered first by S to its output buffer $S.out$ and subsequently m', then m will be received by R before m':

$$\Box\Big(m \in S.out \ \wedge \ \neg\, m' \in S.out \ \wedge \ \Diamond(m' \in S.out) \\ \longrightarrow \Diamond\,(m \in R.in \ \wedge \ \neg\, m' \in R.in \ \wedge \ \Diamond(m' \in R.in))\Big).$$

  Note that in the premise the conjunct $\neg\, m' \in S.out$ is needed in order to specify that m' is put in $S.out$ after m. $\Diamond\,(m' \in S.out)$ on its own does not exclude that m' is already in the sender's buffer when message m is in $S.out$.

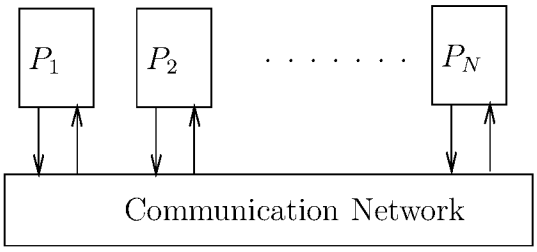
The above formulae refer to fixed messages m and m'. In order to state the above properties for all messages, we have to take the conjunction over all messages m, m'. As long as the message alphabet is finite we still obtain an LTL formula. ■

Example 5.13. Dynamic Leader Election

(This example has been taken from [69].) In current distributed systems several services are offered by some dedicated process(es) in the system. Consider, for example, address assignment and registration, query coordination in a distributed database system,

clock distribution, token regeneration after token loss in a token ring network, initiation of topology updates in a mobile network, load balancing, and so forth. Usually many processes in the system are potentially capable of providing these services. However, for consistency reasons it is usually the case that at any time only one process is allowed to actually provide a given service. This process – called the "leader" – is in fact elected. Sometimes it suffices to elect an arbitrary process, but for other services it is important to elect the process with the best capabilities for performing that service. Here we abstract from specific capabilities and use ranking on the basis of process identities. The idea is therefore that the higher the process' identity, the better its capabilities.

Assume we have a finite number $N > 0$ of processes connected via some communication means. The communication between processes is asynchronous, as in the previous example. Pictorially,



Each process has a unique identity, and it is assumed that a total ordering exists on these identities. Processes behave dynamically in the sense that they are initially inactive, i.e., not participating in the election, and may become active, i.e., participating in the election, at arbitrary moments. In order to have some progress we assume that a process cannot be inactive indefinitely; that is, each process becomes active at some time. (This corresponds to a fairness condition.) Once a process participates it continues to do so, i.e., it does not become inactive anymore. For a given set of active processes a leader will be elected; if an inactive process becomes active, a new election takes place if this process has a higher identity than the current leader.

To give an idea of using LTL as specification formalism we formulate several properties by LTL formulae. We will use i, j as process identities. Let the set of atomic propositions be $\{\, \mathit{leader}_i, \mathit{active}_i \mid 1 \leqslant i, j \leqslant N \,\}$, where leader_i means that process i is a leader, active_i means that process i is active. An inactive process cannot be a leader.

- The property "There is always one leader" can be formalized by

$$\Box\Big(\bigvee_{1 \leqslant i \leqslant N} \mathit{leader}_i \wedge \bigwedge_{\substack{1 \leqslant j \leqslant N \\ j \neq i}} \neg\, \mathit{leader}_j \Big).$$

Although this formula expresses the informally stated property, it will not be satisfied by any realistic protocol. One reason is that processes may be initially inactive, and thus no leader is guaranteed to exist initially. Besides, in a distributed system with asynchronous communication, switching from one leader to another can hardly be made atomic. So, it is more realistic to allow the temporary absence of a leader. As a first attempt to do so, one could modify the above formula into

$$\varphi \;=\; \Box\Diamond\Big(\bigvee_{1\leqslant i\leqslant N} \mathit{leader}_i \wedge \bigwedge_{\substack{1\leqslant j\leqslant N\\ j\neq i}} \neg\, \mathit{leader}_j\Big).$$

Problematic, though, is that this allows there to be more than one leader at a time temporarily – it is only stated that infinitely often there should be exactly one leader, but no statement is made about the moments at which this is not the case. For consistency reasons this is not desired. We therefore replace the above formula φ with $\varphi_1 \wedge \varphi_2$ where φ_1 and φ_2 correspond to the following two properties.

- "There must always be at most one leader":

$$\varphi_1 \;=\; \Box \bigwedge_{1\leqslant i\leqslant N} \Big(\mathit{leader}_i \rightarrow \bigwedge_{\substack{1\leqslant j\leqslant N\\ j\neq i}} \neg\, \mathit{leader}_j\Big)$$

- "There will be enough leaders in due time":

$$\varphi_2 \;=\; \Box\Diamond \bigvee_{1\leqslant i\leqslant N} \mathit{leader}_i$$

  φ_2 does not imply that there will be infinitely many leaders. It only states that there are infinitely many states at which a leader exists. This requirement classifies a leader election protocol that never elects a leader to be wrong. In fact, such a protocol would fulfill the previous requirement, but is not desired for obvious reasons.

- "In the presence of an active process with a higher identity the leader will resign at some time":

$$\Box\Big(\bigwedge_{\substack{1\leqslant i,j\leqslant N\\ i<j}} ((\mathit{leader}_i \;\wedge\; \neg\, \mathit{leader}_j \;\wedge\; \mathit{active}_j) \rightarrow \Diamond\, \neg\, \mathit{leader}_i)\Big)$$

  For reasons of efficiency it is assumed not to be desirable that a leader eventually resigns in the presence of an inactive process that may participate at some unknown time in the future. Therefore we require j to be an active process.

- "A new leader will be an improvement over the previous one". This property requires that successive leaders have an increasing identity. In particular, a process that resigns once will not become a leader anymore.

$$\Box\Big(\bigwedge_{1\leqslant i,j\leqslant N} (leader_i \ \wedge \ \neg \bigcirc \ leader_i \ \wedge \ \bigcirc \Diamond leader_j) \rightarrow (i < j)\Big)$$

  Here, we use "$i < j$" as an atomic proposition that compares the identifiers of processes P_i and P_j and evaluates to true if and only if the process identifier of P_i is smaller than that of P_j. Assuming that the identity of P_i is i (for $i = 1, \ldots, N$), the above property can also be specified by the LTL formula

$$\Box \neg \Big(\bigwedge_{\substack{1\leqslant i,j\leqslant N \\ i\geqslant j}} (leader_i \ \wedge \ \neg \bigcirc \ leader_i \ \wedge \ \bigcirc \Diamond leader_j)\Big).$$

■

Example 5.14. Specifying the Input/Output Behavior of Sequential Programs

The typical requirements on sequential programs such as partial correctness and termination can "in principle" be represented in LTL. Let us briefly describe what termination and partial correctness mean. Assume that a sequential program *Prog* computes a function of the type $f : Inp \rightarrow Outp$, i.e., *Prog* takes as input a value $i \in Inp$ and terminates either by reporting an output value $o \in Outp$ or does not terminate. *Prog* is called *terminating* if the computation of *Prog* halts for each input value $i \in Inp$. *Prog* is *partially correct* if for any input value $i \in Inp$, whenever *Prog* terminates then the output value o equals $f(i)$. How can termination and partial correctness be expressed by means of LTL formulae?

Termination can be specified by a formula of the form $init \ \rightarrow \ \Diamond halt$ where *init* is the labeling for the initial states and *halt* is an atomic proposition characterizing exactly those states that stand for termination. (Without loss of generality it can be assumed that transition systems have no terminal states, i.e., this means terminating states either are equipped with a self-loop, or have a transition leading to a *trap*-state with a self-loop and no other outgoing transitions.)

Partial correctness can be represented by a formula of the form

$$\Box(\ halt \ \longrightarrow \ \Diamond(y = f(x)))$$

where y is the output variable and x the input variable which is assumed not to change during program execution. Additional initial conditions such as expressed by the formula *init* can be added as premise as follows:

$$init \ \longrightarrow \ \Box(halt \ \longrightarrow \ \Diamond(y = f(x))).$$

It should be stressed that this is an extremely simplified representation. In practice, predicate logic concepts are needed to precisely formulate partial correctness. And even in cases where propositional logic formulae of the above form can be used to exactly describe termination and partial correctness, the algorithmic proof of the LTL formulae is very difficult or even impossible. (Recall the undecidability of the halting problem.) ■

Remark 5.15. Specifying Timed Properties with LTL for Synchronous Systems

For *synchronous* systems, LTL can be used as a formalism to specify "real-time" properties that refer to a discrete time scale. Recall that in synchronous systems, the involved processes proceed in a lock step fashion, i.e., at each discrete time instance each process performs a (sometimes idle) step. In this kind of system, the next-step operator $\bigcirc$ has a "timed" interpretation: $\bigcirc\varphi$ states that "at the next time instant φ holds". By putting applications of $\bigcirc$ in sequence, we obtain, e.g.:

$$\bigcirc^k \varphi \stackrel{\text{def}}{=} \underbrace{\bigcirc \bigcirc \ldots \bigcirc}_{k\text{-times}} \varphi \qquad \text{"}\varphi \text{ holds after (exactly) } k \text{ time instants"}.$$

Assertions like "φ will hold within at most k time instants" are obtained by

$$\Diamond^{\leqslant k} \varphi \;=\; \bigvee_{0 \leqslant i \leqslant k} \bigcirc^i \varphi.$$

Statements like "φ holds now and will hold during the next k instants" can be represented as follows:

$$\Box^{\leqslant k}\varphi \;=\; \neg \Diamond^{\leqslant k} \neg \varphi \;=\; \neg \bigvee_{0 \leqslant i \leqslant k} \bigcirc^i \neg \varphi.$$

For the modulo 4 counter of Example 5.11 (page 240) we in fact already implicitly used LTL-formulae as real-time specifications. For example, the formula expressing that once the output is $y{=}1$, the next three steps the output is $y{=}0$:

$$\Box(\, y \;\longrightarrow\; (\bigcirc \neg y \;\wedge\; \bigcirc \bigcirc\; \neg y \;\wedge\; \bigcirc \bigcirc \bigcirc \neg y)\,)$$

can be abbreviated as $\Box(y \;\longrightarrow\; \bigcirc \Box^{\leqslant 2}\; \neg y)$.

It should, however, be noted that the temporal interpretation of the next-step operator is only appropriate for synchronous systems. Every transition in these systems represents the cumulative effect of the actions possible within a single time instant. For asynchronous systems (for which the transition system representation is time-abstract), the next-step operator cannot be interpreted as a real-time modality. In fact, for asynchronous systems the next-step operator should be used with care. The phase changes of a traffic light, for example, can be described by

$$\varphi \;=\; \Box(\mathit{green} \rightarrow \bigcirc \mathit{yellow}) \;\wedge\; \Box(\mathit{yellow} \rightarrow \bigcirc \mathit{red}) \;\wedge\; \ldots$$

For the interleaving of two independent traffic lights (Example 2.17 on page 36) and the formulae φ_1, φ_2 (where the indexed atomic propositions $green_i$, $yellow_i$, etc., are used),

$$TrLight_1 \,|||\, TrLight_2 \not\models \varphi_1 \wedge \varphi_2.$$

This stems from the fact that, e.g., the first traffic light does not change its location when the second traffic light changes its phase. To avoid this problem, the until operator can be used instead, e.g.,

$$\varphi' \;=\; \Box(green \rightarrow (green \,\mathsf{U}\, yellow)) \;\wedge\; \Box(yellow \rightarrow (yellow \,\mathsf{U}\, red)) \;\wedge\; \ldots$$

This differs from the synchronous product operator $\otimes$, where

$$TrLight_1 \otimes TrLight_2 \models \varphi.$$

■

Remark 5.16. Other Notations and Variants of LTL

Many variants and notations have been introduced for LTL. Alternative notations for the temporal modalities are X for $\bigcirc$ (ne$X$t), F for $\Diamond$ (Finally), and G for $\Box$ (Globally). All operators from LTL refer to the future (including the current state). Consequently, operators are known as future operators. LTL can, however, also be extended with *past* operators. This can be useful for specifying some properties more easily (and succinctly) in terms of the past than in terms of the future. For instance, $\Box^{-1}\, a$ ("always in the past") means that a is valid now and in any state in the past. $\Diamond^{-1}\, a$ ("sometime in the past") means that either a is valid in the current state or in some state in the past and $\bigcirc^{-1}\, a$ means that a holds in the previous state, provided such state exists. For example, the property "every red light phase is preceded by a yellow one" can be described by

$$\Box(red \;\rightarrow\; \bigcirc^{-1}\, yellow).$$

The main reason for introducing past operators is to simplify the specification of several properties. The expressive power of the logic is, however, not affected by the addition of past operators when a discrete notion of time is taken (as we do). Thus, for any property which contains one or more past operators, an LTL-formula with only future temporal operators exists expressing the same thing. More information is described in Section 5.4.

■

5.1.4 Equivalence of LTL Formulae

For any type of logic, a clear separation between syntax and semantics is an essential aspect. On the other hand, two formulae are intuitively identified whenever they have the

duality law	*idempotency law*
$\neg \bigcirc \varphi \equiv \bigcirc \neg\varphi$	$\Diamond\Diamond\varphi \equiv \Diamond\varphi$
$\neg\Diamond\varphi \equiv \Box\neg\varphi$	$\Box\Box\varphi \equiv \Box\varphi$
$\neg\Box\varphi \equiv \Diamond\neg\varphi$	$\varphi \,\mathsf{U}\, (\varphi \,\mathsf{U}\, \psi) \equiv \varphi \,\mathsf{U}\, \psi$
	$(\varphi \,\mathsf{U}\, \psi) \,\mathsf{U}\, \psi \equiv \varphi \,\mathsf{U}\, \psi$
absorption law	*expansion law*
$\Diamond\Box\Diamond\varphi \equiv \Box\Diamond\varphi$	$\varphi \,\mathsf{U}\, \psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{U}\, \psi))$
$\Box\Diamond\Box\varphi \equiv \Diamond\Box\varphi$	$\Diamond\psi \equiv \psi \vee \bigcirc\Diamond\psi$
	$\Box\psi \equiv \psi \wedge \bigcirc\Box\psi$
distributive law	
$\bigcirc(\varphi \,\mathsf{U}\, \psi) \equiv (\bigcirc\varphi) \,\mathsf{U}\, (\bigcirc\psi)$	
$\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$	
$\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Box\psi$	

Figure 5.7: Some equivalence rules for LTL.

same truth-value under all interpretations. For example, it seems useless to distinguish between $\neg\neg a$ and a, although both formulae are syntactically different.

Definition 5.17. Equivalence of LTL Formulae

LTL formulae φ_1, φ_2 are *equivalent*, denoted $\varphi_1 \equiv \varphi_2$, if $\mathit{Words}(\varphi_1) = \mathit{Words}(\varphi_2)$. ■

As LTL subsumes propositional logic, equivalences of propositional logic also hold for LTL, e.g., $\neg\neg\varphi \equiv \varphi$ and $\varphi \wedge \varphi \equiv \varphi$. In addition, there exist a number of equivalence rules for temporal modalities. They include the equivalence laws indicated in Figure 5.7. We explain some of these equivalence laws. The *duality rule* $\neg \bigcirc \varphi \equiv \bigcirc \neg\varphi$ shows that the next-step operator $\bigcirc$ is dual to itself. It results from the observation that

$$
\begin{array}{ll}
 & A_0\, A_1\, A_2 \ldots \models \neg \bigcirc \varphi \\
\text{iff} & A_0\, A_1\, A_2 \ldots \not\models \bigcirc \varphi \\
\text{iff} & A_1\, A_2 \ldots \not\models \varphi \\
\text{iff} & A_1\, A_2 \ldots \models \neg\varphi \\
\text{iff} & A_0\, A_1\, A_2 \ldots \models \bigcirc \neg\varphi.
\end{array}
$$

The first *absorption law* is explained by the fact that "infinitely often φ" is equal to "from

a certain point of time on, φ is true infinitely often".

The distributive laws for $\Diamond$ and disjunction, or $\Box$ and conjunction, respectively, are dual to each other. They can be regarded as the temporal logic analogon to the distributive laws for $\exists$ and $\vee$ or $\forall$ and $\wedge$ in predicate logic. It should be noted, however, that $\Diamond$ does not distribute over conjunction (as existential quantification), and $\Box$ does not distribute over disjunction (like universal quantification):

$$\Diamond(a \wedge b) \not\equiv \Diamond a \wedge \Diamond b \quad \text{and} \quad \Box(a \vee b) \not\equiv \Box a \vee \Box b.$$

The formula $\Diamond(a \wedge b)$ ensures that a state will be reached for which a and b hold, while $\Diamond a \wedge \Diamond b$ ensures that eventually an a-state and eventually a b-state will be reached. According to the latter formula, a and b need not to be satisfied at the same time. Figure 5.8 depicts a transition system that satisfies $\Diamond a \wedge \Diamond b$, but not $\Diamond(a \wedge b)$.

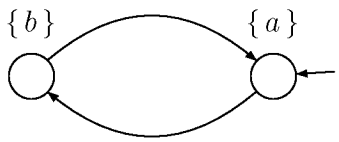


Figure 5.8: $TS \not\models \Diamond(a \wedge b)$ and $TS \models \Diamond a \wedge \Diamond b$.

The expansion laws play an important role. They describe the temporal modalities U, $\Diamond$, and $\Box$ by means of a *recursive* equivalence. These equivalences all have the same global structure: they assert something about the current state, and about the direct successor state. The assertion about the current state is done without the need to use temporal modalities whereas the assertion about the next state is done using the $\bigcirc$ operator. The expansion law for until, for instance, can be considered as follows. Formula $\phi \,\mathsf{U}\, \psi$ is a solution of the equivalence

$$\kappa \equiv \psi \vee (\varphi \wedge \bigcirc \kappa).$$

(current state: ψ, φ; first suffix: $\bigcirc \kappa$)

Let us explain the expansion law for the until operator in more detail. Let $A_0\, A_1\, A_2 \ldots$ be an infinite word over the alphabet 2^{AP}, such that $A_0\, A_1\, A_2 \ldots \models \varphi \,\mathsf{U}\, \psi$. By the definition of "until", there exists a $k \geqslant 0$, such that

$$A_i\, A_{i+1}\, A_{i+2} \ldots \models \varphi, \quad \text{for all } 0 \leqslant i < k \quad \text{and} \quad A_k\, A_{k+1}\, A_{k+2} \ldots \models \psi.$$

Distinguish between $k = 0$ and $k > 0$. If $k = 0$, then $A_0\, A_1\, A_2 \ldots \models \psi$ and thus $A_0\, A_1\, A_2 \ldots \models \psi \vee \ldots$. If $k > 0$, then

$$A_0\,A_1\,A_2\ldots \models \varphi \quad \text{and} \quad A_1\,A_2\ldots \models \varphi\,\mathsf{U}\,\psi.$$

From this it immediately follows that

$$A_0\,A_1\,A_2\ldots \models \varphi \;\wedge\; \bigcirc\,(\varphi\,\mathsf{U}\,\psi).$$

Gathering the results for $k = 0$ and $k > 0$ yields

$$A_0\,A_1\,A_2\ldots \models \psi \;\vee (\varphi \;\wedge\; \bigcirc\,(\varphi\,\mathsf{U}\,\psi)).$$

For the reverse direction, a similar argument can be provided.

The expansion law for $\Diamond\psi$ is a special case of the expansion law for until:

$$\Diamond\psi \;=\; \text{true}\,\mathsf{U}\,\psi \;\equiv\; \psi \;\vee\; \underbrace{(\text{true} \;\wedge\; \bigcirc\,(\text{true}\,\mathsf{U}\,\psi))}_{\equiv\,\bigcirc\,(\text{true}\,\mathsf{U}\,\psi)\,=\,\bigcirc\,\Diamond\psi} \;\equiv\; \psi \;\vee\; \bigcirc\,\Diamond\psi.$$

The expansion law for $\Box\psi$ now results from the duality of $\Diamond$ and $\Box$, the duality of $\vee$ and $\wedge$ (i.e., de Morgan's law) and from the duality law for $\bigcirc$

$$\begin{array}{lll}
 & \Box\psi & \\
= & & \text{(* definition of } \Box \text{ *)} \\
 & \neg\Diamond\neg\psi & \\
\equiv & & \text{(* expansion law for } \Diamond\text{*)} \\
 & \neg(\neg\psi \;\vee\; \bigcirc\,\Diamond\neg\psi) & \\
\equiv & & \text{(* de Morgan's law *)} \\
 & \neg\neg\psi \;\wedge\; \neg\bigcirc\,\Diamond\neg\psi & \\
\equiv & & \text{(* self-duality of } \bigcirc\text{*)} \\
 & \psi \;\wedge\; \bigcirc\neg\Diamond\neg\psi & \\
\equiv & & \text{(* definition of } \Box \text{ *)} \\
 & \psi \;\wedge\; \bigcirc\Box\psi &
\end{array}$$

It is important to realize that none of the indicated expansion laws represents a complete recursive characterization of the temporal operator at hand. For example, the formulae $\varphi = \text{false}$ and $\varphi = \Box a$ both satisfy the recursive "equation" $\varphi \;\equiv\; a \wedge \bigcirc\,\varphi$ since false $\equiv$ $a \wedge \bigcirc\,\text{false}$ and $\Box a \;\equiv\; a \wedge \bigcirc\,\Box a$. However, $\varphi\,\mathsf{U}\,\psi$ and $\Diamond\varphi$ are the *least* solutions of the expansion law indicated for "until" and"eventually", respectively. Similarly, $\Box\varphi$ is the *greatest* solution of the expansion law for "always". The precise meaning of these statements will be explained by considering the until operator as example:

Lemma 5.18. Until is the Least Solution of the Expansion Law

For LTL formulae φ and ψ, $Words(\varphi \,\mathsf{U}\, \psi)$ is the least LT property $P \subseteq (2^{AP})^\omega$ such that:

$$Words(\psi) \cup \{A_0\, A_1\, A_2 \ldots \in Words(\varphi) \;\mid\; A_1\, A_2 \ldots \in P\} \;\subseteq\; P \qquad (*)$$

Moreover, $Words(\varphi \,\mathsf{U}\, \psi)$ agrees with the set

$$Words(\psi) \cup \{A_0\, A_1\, A_2 \ldots \in Words(\varphi) \;\mid\; A_1\, A_2 \ldots \in Words(\varphi \,\mathsf{U}\, \psi)\}.$$

The formulation "least LT property satisfying condition (*)" means that the following conditions hold:

(1) $P = Words(\varphi \,\mathsf{U}\, \psi)$ satisfies (*).

(2) $Words(\varphi \,\mathsf{U}\, \psi) \subseteq P$ for all LT properties P satisfying condition (*).

Proof: Condition (1) follows immediately from the expansion law $\varphi \,\mathsf{U}\, \psi \;\equiv\; \psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{U}\, \psi))$. In fact, the expansion law even yields that $\subseteq$ in (*) can be replaced by equality, i.e., $Words(\varphi \,\mathsf{U}\, \psi)$ agrees with

$$Words(\psi) \cup \{A_0\, A_1\, A_2 \ldots \in Words(\varphi) \;\mid\; A_1\, A_2 \ldots \in Words(\varphi \,\mathsf{U}\, \psi)\}.$$

To prove condition (2), P is assumed to be an LT property that satisfies (*). We show that $Words(\varphi \,\mathsf{U}\, \psi) \subseteq P$. Since P fulfills (*), we have:

(i) $Words(\psi) \subseteq P$,

(ii) If $B_0\, B_1\, B_2 \ldots \in Words(\varphi)$ and $B_1\, B_2 \ldots \in P$ then $B_0\, B_1\, B_2 \ldots \in P$.

Let $A_0\, A_1\, A_2 \ldots \in Words(\varphi \,\mathsf{U}\, \psi)$. Then, there exists an index $k \geqslant 0$ such that

(iii) $A_i\, A_{i+1}\, A_{i+2} \ldots \in Words(\varphi)$, for all $0 \leqslant i < k$,

(iv) $A_k\, A_{k+1}\, A_{k+2} \ldots \in Words(\psi)$.

We now derive

$$
\begin{array}{lll}
 & A_k\, A_{k+1}\, A_{k+2}\, A_{k+3} \ldots \in P & \text{due to (iv) and (i)} \\
\Longrightarrow & A_{k-1}\, A_k\, A_{k+1}\, A_{k+2} \ldots \in P & \text{due to (ii) and (iii)} \\
\Longrightarrow & A_{k-2}\, A_{k-1}\, A_k\, A_{k+1} \ldots \in P & \text{due to (ii) and (iii)} \\
 & \vdots & \\
\Longrightarrow & A_0\, A_1\, A_2\, A_3 \ldots \in P & \text{due to (ii) and (iii).}
\end{array}
$$

■

5.1.5 Weak Until, Release, and Positive Normal Form

Any LTL formula can be transformed into a canonical form, the so-called *positive normal form* (PNF). This canonical form is characterized by the fact that negations only occur adjacent to atomic propositions. PNF formulae in propositional logic are constructed from the constants true and false, the literals a and $\neg a$, and the operators $\wedge$ and $\vee$. For instance, $\neg a \wedge ((\neg b \wedge c) \vee \neg a)$ is in PNF, while $\neg(a \wedge \neg b)$ is not. The well-known disjunctive and conjunctive normal forms are special cases of the PNF for propositional logic.

In order to transform any LTL formula into PNF, for each operator a dual operator needs to be incorporated into the syntax of PNF formulae. The propositional logical primitives of the positive normal form for LTL are the constant true and its dual constant false $= \neg$true, as well as conjunction $\wedge$ and its dual, $\vee$. De Morgan's rules $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ and $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ yield the duality of conjunction and disjunction. According to the duality rule $\neg \bigcirc \varphi \equiv \bigcirc \neg\varphi$, the next-step operator is a dual of itself. Therefore, no extra operator is necessary for $\bigcirc$. Now consider the until operator. First we observe that

$$\neg(\varphi \,\mathsf{U}\, \psi) \equiv ((\varphi \wedge \neg\psi) \,\mathsf{U}\, (\neg\varphi \wedge \neg\psi)) \vee \Box(\varphi \wedge \neg\psi).$$

The first disjunct on the right-hand side asserts that φ stops to hold "too early", i.e., before ψ becomes valid. The second disjunct states that φ always holds but never ψ. Clearly, in both cases, $\neg(\varphi \,\mathsf{U}\, \psi)$ holds.

This observation provides the motivation to introduce the operator W (called *weak until* or *unless*) as the dual of U. It is defined by:

$$\varphi \,\mathsf{W}\, \psi \stackrel{\text{def}}{=} (\varphi \,\mathsf{U}\, \psi) \vee \Box\varphi.$$

The semantics of $\varphi \,\mathsf{W}\, \psi$ is similar to that of $\varphi \,\mathsf{U}\, \psi$, except that $\varphi \,\mathsf{U}\, \psi$ requires a state to be reached for which ψ holds, whereas this is not required for $\varphi \,\mathsf{W}\, \psi$. Until U and weak until W are dual in the following sense:

$$
\begin{aligned}
\neg(\varphi \,\mathsf{U}\, \psi) &\equiv (\varphi \wedge \neg\psi) \,\mathsf{W}\, (\neg\varphi \wedge \neg\psi) \\
\neg(\varphi \,\mathsf{W}\, \psi) &\equiv (\varphi \wedge \neg\psi) \,\mathsf{U}\, (\neg\varphi \wedge \neg\psi)
\end{aligned}
$$

The reason that weak until is not a standard operator for LTL is that U and W have the same expressiveness, since

$$\begin{aligned} \Box\psi &\equiv \psi \,\mathsf{W}\, \text{false}, \\ \varphi \,\mathsf{U}\, \psi &\equiv (\varphi \,\mathsf{W}\, \psi) \wedge \underbrace{\Diamond\psi}_{\equiv\, \neg\Box\neg\psi}. \end{aligned}$$

That is to say, weak until is relevant only if restrictions are imposed on the occurrence of negation (as in PNF). It is interesting to observe that W and U satisfy the same expansion law:

$$\varphi \,\mathsf{W}\, \psi \;\equiv\; \psi \;\vee\; (\varphi \;\wedge\; \bigcirc(\varphi \,\mathsf{W}\, \psi)).$$

For $\psi = \text{false}$ this results in the expansion law for $\Box\varphi$:

$$\Box\varphi \;=\; \varphi \,\mathsf{W}\, \text{false} \;\equiv\; \text{false} \;\vee\; (\varphi \;\wedge\; \bigcirc(\varphi \,\mathsf{W}\, \text{false})) \;\equiv\; \varphi \;\wedge\; \bigcirc\Box\varphi.$$

The semantic difference between U and W is shown by the fact that $\varphi \,\mathsf{W}\, \psi$ is the *greatest* solution of

$$\kappa \;\equiv\; \psi \;\vee\; (\varphi \;\wedge\; \bigcirc\kappa).$$

This result is proven below. Recall $\varphi \,\mathsf{U}\, \psi$ is the least solution of this equivalence, see Lemma 5.18 on page 251.

Lemma 5.19. Weak-Until is the Greatest Solution of the Expansion Law

For LTL formulae φ and ψ, $Words(\varphi \,\mathsf{W}\, \psi)$ is the greatest LT property $P \subseteq (2^{AP})^\omega$ such that:

$$Words(\psi) \cup \{A_0A_1A_2\ldots \in Words(\varphi) \;\mid\; A_1A_2\ldots \in P\} \;\supseteq\; P \qquad (*)$$

Moreover, $Words(\varphi \,\mathsf{W}\, \psi)$ agrees with the LT property

$$Words(\psi) \cup \{A_0A_1A_2\ldots \in Words(\varphi) \;\mid\; A_1A_2\ldots \in Words(\varphi \,\mathsf{W}\, \psi)\}.$$

The formulation "greatest LT property with the indicated condition (*) is to be understood in the following sense:

(1) $P \supseteq Words(\varphi \,\mathsf{W}\, \psi)$ satisfies (*).

(2) $Words(\varphi \,\mathsf{W}\, \psi) \supseteq P$ for all LT properties P satisfying condition (*).

Proof: The fact that condition (1) is satisfied, even with equality rather than $\supseteq$, i.e., $\mathit{Words}(\varphi \,\mathsf{W}\, \psi)$ agrees with

$$\mathit{Words}(\psi) \cup \{A_0 A_1 A_2 \ldots \in \mathit{Words}(\varphi) \;\mid\; A_1 A_2 \ldots \in \mathit{Words}(\varphi \,\mathsf{W}\, \psi)\},$$

is an immediate conclusion from the expansion law $\varphi \,\mathsf{W}\, \psi \;\equiv\; \psi \vee (\varphi \wedge \bigcirc(\varphi \,\mathsf{W}\, \psi))$.

For proving condition (2), P is assumed to be an LT property satisfying (*). In particular, for all words $B_0\, B_1\, B_2\, B_3 \ldots \in (2^{AP})^\omega \setminus \mathit{Words}(\psi)$ we have:

(i) If $B_0\, B_1\, B_2\, B_3 \ldots \notin \mathit{Words}(\varphi)$, then $B_0\, B_1\, B_2\, B_3 \ldots \notin P$.

(ii) If $B_1\, B_2\, B_3 \ldots \notin P$, then $B_0\, B_1\, B_2\, B_3 \ldots \notin P$.

Now we demonstrate that

$$(2^{AP})^\omega \setminus \mathit{Words}(\varphi \,\mathsf{W}\, \psi) \;\subseteq\; (2^{AP})^\omega \setminus P.$$

Let $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega \setminus \mathit{Words}(\varphi \,\mathsf{W}\, \psi)$. Then $A_0\, A_1\, A_2 \ldots \not\models \varphi \,\mathsf{W}\, \psi$ and thus

$$A_0\, A_1\, A_2 \ldots \models \neg(\varphi \,\mathsf{W}\, \psi) \;\equiv\; (\varphi \wedge \neg\psi)\,\mathsf{U}\,(\neg\varphi \wedge \neg\psi).$$

Thus there exists $k \geqslant 0$, such that:

(iii) $A_i\, A_{i+1}\, A_{i+2} \ldots \models \varphi \wedge \neg\psi, \quad$ for all $0 \leqslant i < k$,

(iv) $A_k\, A_{k+1}\, A_{k+2} \ldots \models \neg\varphi \wedge \neg\psi$.

In particular, none of the words $A_i\, A_{i+1}\, A_{i+2} \ldots$ belongs to $\mathit{Words}(\psi)$ for $0 \leqslant i \leqslant k$. We obtain:

$$\begin{array}{lll}
 & A_k\, A_{k+1}\, A_{k+2}\, A_{k+3} \ldots \notin P & \text{due to (i) and (iv)} \\
\Longrightarrow & A_{k-1}\, A_k\, A_{k+1}\, A_{k+2} \ldots \notin P & \text{due to (ii) and (iii)} \\
\Longrightarrow & A_{k-2}\, A_{k-1}\, A_k\, A_{k+1} \ldots \notin P & \text{due to (ii) and (iii)} \\
 & \vdots & \\
\Longrightarrow & A_0\, A_1\, A_2\, A_3 \ldots \notin P & \text{due to (ii) and (iii)}\ .
\end{array}$$

Thus, $(2^{AP})^\omega \setminus \mathit{Words}(\varphi \,\mathsf{W}\, \psi) \subseteq (2^{AP})^\omega \setminus P$, or equivalently, $\mathit{Words}(\varphi \,\mathsf{W}\, \psi) \supseteq P$. ∎

We are now ready to introduce the positive normal form for LTL which permits negation only on the level of literals and – to ensure the full expressiveness of LTL – uses the dual Boolean connectors $\wedge$ and $\vee$, the self-dual next-step operator $\bigcirc$, and U and W:

Definition 5.20. Positive Normal Form for LTL (Weak-Until PNF)

For $a \in AP$, the set of LTL formulae in *weak-until positive normal form* (weak-until PNF, for short, or simply PNF) is given by:

$$\varphi \ ::= \ \text{true} \ \Big| \ \text{false} \ \Big| \ a \ \Big| \ \neg a \ \Big| \ \varphi_1 \wedge \varphi_2 \ \Big| \ \varphi_1 \vee \varphi_2 \ \Big| \ \bigcirc \varphi \ \Big| \ \varphi_1 \,\mathsf{U}\, \varphi_2 \ \Big| \ \varphi_1 \,\mathsf{W}\, \varphi_2.$$

■

Due to the law $\Box\varphi \equiv \varphi \,\mathsf{W}\, \text{false}$, $\Box$ can also be considered as permitted operator of the W-positive normal form. As before, $\Diamond\varphi = \text{true}\,\mathsf{U}\,\varphi$. An example LTL formula in PNF is

$$\Diamond(a \,\mathsf{U}\, \Box b) \ \vee \ (a \wedge \neg c) \,\mathsf{W}\, (\Diamond(\neg a \,\mathsf{U}\, b)).$$

The LTL formulae $\neg(a \,\mathsf{U}\, b)$ and $c \vee \neg(a \wedge \Diamond b)$ are not in weak-until PNF.

The previous considerations were aimed to rewrite any LTL formula into weak-until PNF. This is done by successively "pushing" negations "inside" the formula at hand. This is facilitated by the following transformations:

$$\begin{array}{lcl}
\neg\text{true} & \rightsquigarrow & \text{false} \\
\neg\text{false} & \rightsquigarrow & \text{true} \\
\neg\neg\varphi & \rightsquigarrow & \varphi \\
\neg(\varphi \wedge \psi) & \rightsquigarrow & \neg\varphi \vee \neg\psi \\
\neg \bigcirc \varphi & \rightsquigarrow & \bigcirc \neg\varphi \\
\neg(\varphi \,\mathsf{U}\, \psi) & \rightsquigarrow & (\varphi \wedge \neg\psi) \,\mathsf{W}\, (\neg\varphi \wedge \neg\psi).
\end{array}$$

These rewrite rules are lifted to the derived operators as follows:

$$\neg(\varphi \ \vee \ \psi) \ \rightsquigarrow \ \neg\varphi \wedge \neg\psi \quad \text{and} \quad \neg\Diamond\varphi \ \rightsquigarrow \Box\neg\varphi \quad \text{and} \quad \neg\Box\varphi \ \rightsquigarrow \ \Diamond\neg\varphi.$$

Example 5.21. Positive Normal Form

Consider the LTL formula $\neg\Box((a \,\mathsf{U}\, b) \ \vee \ \bigcirc \, c)$. This formula is not in PNF, but can be transformed into an equivalent LTL formula in weak-until PNF as follows:

$$\begin{array}{ll}
 & \neg\Box((a \,\mathsf{U}\, b) \ \vee \ \bigcirc \, c) \\
\equiv & \Diamond\neg((a \,\mathsf{U}\, b) \ \vee \ \bigcirc \, c) \\
\equiv & \Diamond(\neg(a \,\mathsf{U}\, b) \wedge \neg \bigcirc \, c) \\
\equiv & \Diamond((a \wedge \neg b) \,\mathsf{W}\, (\neg a \wedge \neg b) \ \wedge \ \bigcirc \, \neg c)
\end{array}$$

■

Theorem 5.22. Existence of Equivalent Weak-Until PNF Formulae

For each LTL formula there exists an equivalent LTL formula in weak-until PNF.

The main drawback of the rewrite rules introduced above is that the length of the resulting LTL formula (in weak-until PNF) may be exponential in the length of the original nonPNF LTL formula. This is due to the rewrite rule for until where φ and ψ are duplicated. Although this can be slightly improved by adopting the law:

$$\neg(\varphi \,\mathsf{U}\, \psi) \;\equiv\; (\neg\psi)\,\mathsf{W}\,(\neg\varphi \wedge \neg\psi)$$

an exponential growth in length is not avoided here too due to the duplication of ψ in the right-hand side.

To avoid this exponential blowup in transforming an LTL formula in PNF, another temporal modality is used that is dual to the until operator: the so-called binary *release*-operator, denoted R. It is defined by

$$\varphi \,\mathsf{R}\, \psi \;\stackrel{\text{def}}{=}\; \neg(\neg\varphi \,\mathsf{U}\, \neg\psi).$$

Its intuitive interpretation is as follows. Formula $\varphi \,\mathsf{R}\, \psi$ holds for a word if ψ always holds, a requirement that is released as soon as φ becomes valid. Thus, the formula $\mathrm{false}\,\mathsf{R}\,\varphi$ is valid if φ always holds, since the release condition (false) is a contradiction. Formally, for a given word $\sigma = A_0\, A_1 \ldots \in (2^{AP})^\omega$:

$$\begin{array}{ll}
 & \sigma \models \varphi \,\mathsf{R}\, \psi \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* definition of } \mathsf{R} \text{ *)} \\
 & \neg\exists j \geqslant 0.\ (\sigma[j..] \models \neg\psi \wedge \forall i < j.\ \sigma[i..] \models \neg\varphi) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* semantics of negation *)} \\
 & \neg\exists j \geqslant 0.\ (\sigma[j..] \not\models \psi \;\wedge\; \forall i < j.\ \sigma[i..] \not\models \varphi) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* duality of } \exists \text{ and } \forall \text{ *)} \\
 & \forall j \geqslant 0.\ \neg\Big(\sigma[j..] \not\models \psi \;\wedge\; \forall i < j.\ \sigma[i..] \not\models \varphi\Big) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* de Morgan's law *)} \\
 & \forall j \geqslant 0.\ \Big(\neg(\sigma[j..] \not\models \psi) \;\vee\; \neg\forall i < j.\ \sigma[i..] \not\models \varphi\Big) \\
\text{iff} & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(* semantics of negation *)} \\
 & \forall j \geqslant 0.\ \Big(\sigma[j..] \models \psi \;\vee\; \exists i < j.\ \sigma[i..] \models \varphi\Big) \\
\text{iff} & \\
 & \forall j \geqslant 0.\ \sigma[j..] \models \psi \quad \text{or} \quad \exists i \geqslant 0.\ (\sigma[i..] \models \varphi) \wedge \forall k \leqslant i.\ \sigma[k..] \models \psi).
\end{array}$$

The always operator is obtained from the release operator by:

$$\Box\varphi \;\equiv\; \mathrm{false}\ \mathsf{R}\ \varphi.$$

The weak-until and the until operator are obtained by:

$$\varphi \,\mathsf{W}\, \psi \;\equiv\; (\neg\varphi \;\vee\; \psi)\,\mathsf{R}\,(\varphi \;\vee\; \psi), \qquad \varphi \,\mathsf{U}\, \psi \;\equiv\; \neg(\neg\varphi \,\mathsf{R}\, \neg\psi).$$

Vice versa, $\varphi \,\mathsf{R}\, \psi \;\equiv\; (\neg\varphi \wedge \psi)\,\mathsf{W}\,(\varphi \wedge \psi)$. The expansion law (see Exercise 5.8) for release reads as follows:

$$\varphi \,\mathsf{R}\, \psi \;\equiv\; \psi \wedge (\varphi \;\vee\; \bigcirc\, (\varphi \,\mathsf{R}\, \psi)).$$

We now revisit the notion of PNF, which is defined using the R-operator rather than W:

Definition 5.23. Positive Normal Form (release PNF)

For $a \in AP$, LTL formulae in release positive normal form (release PNF, or simply PNF) are given by

$$\varphi \;::=\; \text{true} \;\Big|\; \text{false} \;\Big|\; a \;\Big|\; \neg a \;\Big|\; \varphi_1 \wedge \varphi_2 \;\Big|\; \varphi_1 \vee \varphi_2 \;\Big|\; \bigcirc\, \varphi \;\Big|\; \varphi_1 \,\mathsf{U}\, \varphi_2 \;\Big|\; \varphi_1 \,\mathsf{R}\, \varphi_2.$$

■

The following transformation rules push negations inside and serve to transform a given LTL formula into an equivalent LTL formula in PNF:

$$\begin{array}{lcl}
\neg\text{true} & \rightsquigarrow & \text{false} \\
\neg\neg\varphi & \rightsquigarrow & \varphi \\
\neg(\varphi \wedge \psi) & \rightsquigarrow & \neg\varphi \vee \neg\psi \\
\neg \bigcirc\, \varphi & \rightsquigarrow & \bigcirc\, \neg\varphi \\
\neg(\varphi \,\mathsf{U}\, \psi) & \rightsquigarrow & \neg\varphi \,\mathsf{R}\, \neg\psi
\end{array}$$

In each rewrite rule the size of the resulting formula increases at most by an additive constant.

Theorem 5.24. Existence of Equivalent Release PNF Formulae

For any LTL formula φ there exists an equivalent LTL formula φ' in release PNF with $|\varphi'| = \mathcal{O}(|\varphi|)$.

5.1.6 Fairness in LTL

In Chapter 3, we have seen that typically some fairness constraints are needed to verify liveness properties. Three types of fairness constraints (for sets of actions) have been distinguished, namely unconditional, strong, and weak fairness. Accordingly, the satisfaction

relation for LT properties (denoted $\models$) has been adapted to the fairness assumption $\mathcal{F}$ (denoted $\models_{\mathcal{F}}$), where $\mathcal{F}$ is a triple of (sets of) fairness constraints. This entails that only fair paths are considered while determining the satisfaction of a property. In this section, this approach is adopted in the context of LTL. That is to say, rather than determining for transition system TS and LTL formula φ whether $TS \models \varphi$, we focus on the fair executions of TS. The main difference with the action-based fairness assumptions (and constraints) is that we now focus on *state-based* fairness.

Definition 5.25. LTL Fairness Constraints and Assumptions

Let Φ and Ψ be propositional logic formulae over AP.

1. An *unconditional LTL fairness constraint* is an LTL formula of the form

$$ufair \;=\; \Box\Diamond\Psi.$$

2. A *strong LTL fairness condition* is an LTL formula of the form

$$sfair \;=\; \Box\Diamond\Phi \;\longrightarrow\; \Box\Diamond\Psi.$$

3. A *weak LTL fairness constraint* is an LTL formula of the form

$$wfair \;=\; \Diamond\Box\Phi \;\longrightarrow\; \Box\Diamond\Psi.$$

An *LTL fairness assumption* is a conjunction of LTL fairness constraints (of any arbitrary type). ■

For instance, a strong LTL fairness assumption denotes a conjunction of strong LTL fairness constraints, i.e., a formula of the form

$$sfair \;=\; \bigwedge_{0<i\leqslant k} (\Box\Diamond\Phi_i \;\longrightarrow\; \Box\Diamond\Psi_i)$$

for propositional logical formulae Φ_i and Ψ_i over AP. Weak and unconditional LTL fairness assumptions are defined in a similar way.

In their most general form, LTL fairness assumptions are (as in Definition 3.46, page 133) a conjunction of unconditional, strong, and weak fairness assumptions:

$$\mathit{fair} \;=\; \mathit{ufair} \;\wedge\; \mathit{sfair} \;\wedge\; \mathit{wfair}.$$

where *ufair*, *sfair*, and *wfair* are unconditional, strong, and weak LTL fairness assumptions, respectively. As in the case of action-based fairness assumptions, the rule of thumb for imposing fairness assumptions is: strong (or unconditional) fairness assumptions are useful for solving contentions, and weak fairness is often sufficient for resolving the nondeterminism that results from interleaving.

In the sequel, we adopt the same notations as for action-based fairness assumptions. Let $\mathit{FairPaths}(s)$ denote the set of all fair paths starting in s and $\mathit{FairTraces}(s)$ the set of all traces induced by fair paths starting in s. Formally, for fixed formula *fair*,

$$\begin{aligned} \mathit{FairPaths}(s) &= \{\, \pi \in \mathit{Paths}(s) \mid \pi \models \mathit{fair} \,\}, \\ \mathit{FairTraces}(s) &= \{\, \mathit{trace}(\pi) \mid \pi \in \mathit{FairPaths}(s) \,\}. \end{aligned}$$

These notions can be lifted to transition systems in the obvious way yielding $\mathit{FairPaths}(TS)$ and $\mathit{FairTraces}(TS)$. To identify the fairness assumption *fair*, we may write $\mathit{FairPaths}_{\mathit{fair}}(\cdot)$ or $\mathit{FairTraces}_{\mathit{fair}}(\cdot)$.

Definition 5.26. Satisfaction Relation for LTL with Fairness

For state s in transition system TS (over AP) without terminal states, LTL formula φ, and LTL fairness assumption *fair* let

$$\begin{aligned} s \models_{\mathit{fair}} \varphi \quad &\text{iff} \quad \forall \pi \in \mathit{FairPaths}(s).\, \pi \models \varphi \quad \text{and} \\ TS \models_{\mathit{fair}} \varphi \quad &\text{iff} \quad \forall s_0 \in I.\, s_0 \models_{\mathit{fair}} \varphi. \end{aligned}$$

■

TS satisfies φ under the LTL fairness assumption *fair* if φ holds for all *fair* paths that originate from some initial state.

Example 5.27. Mutual Exclusion with Randomized Arbiter (Fairness)

Consider the following approach to two-process mutual exclusion. A randomized arbiter, see the program graphs in Figure 5.9, decides which process is acquiring access to the critical section. It does so by tossing coins. We abstract from the probabilities, and model the coin tossing by a nondeterministic choice between the alternatives "heads" and "tails". It is assumed that the two contending processes communicate with the arbiter via the actions enter_1 and enter_2. The critical section is released by synchronizing over

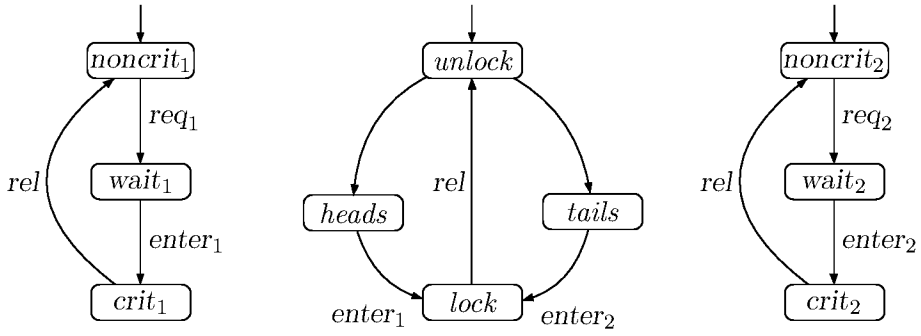


Figure 5.9: Mutual exclusion with a randomized arbiter.

the action *release*. For the sake of simplicity, we refrain from indicating which process is releasing the critical section.

The property "process P_1 is in its critical section infinitely often" *cannot* be established, since, for instance, the underlying transition system representation does not exclude an execution in which only the second process may perform an action while P_1 is entirely ignored. Thus:

$$TS_1 \,\|\, Arbiter \,\|\, TS_2 \;\not\models\; \Box\Diamond\, crit_1.$$

If a coin is assumed to be fair enough such that both events "heads" and "tails" occur with positive probability, the unrealistic case of one of the two alternatives never happening can be ignored by means of the unconditional LTL fairness assumption:

$$fair \;=\; \Box\Diamond heads \;\wedge \Box\Diamond tails.$$

It is not difficult to check that now:

$$TS_1 \,\|\, Arbiter \,\|\, TS_2 \;\models_{fair}\; \Box\Diamond crit_1 \;\wedge\; \Box\Diamond crit_2.$$

■

Example 5.28. Communication Protocol (Fairness)

Consider the alternating bit protocol as described in Example 2.32 (page 57). For the sake of convenience, the program graph of the sender of the alternating bit protocol is repeated in Figure 5.10. The liveness property $\Box\Diamond start$ states that the protocol returns infinitely often to its initial state. In this initial state the action *snd_msg(0)* is enabled. It follows that

$$ABP \;\not\models\; \Box\Diamond start$$

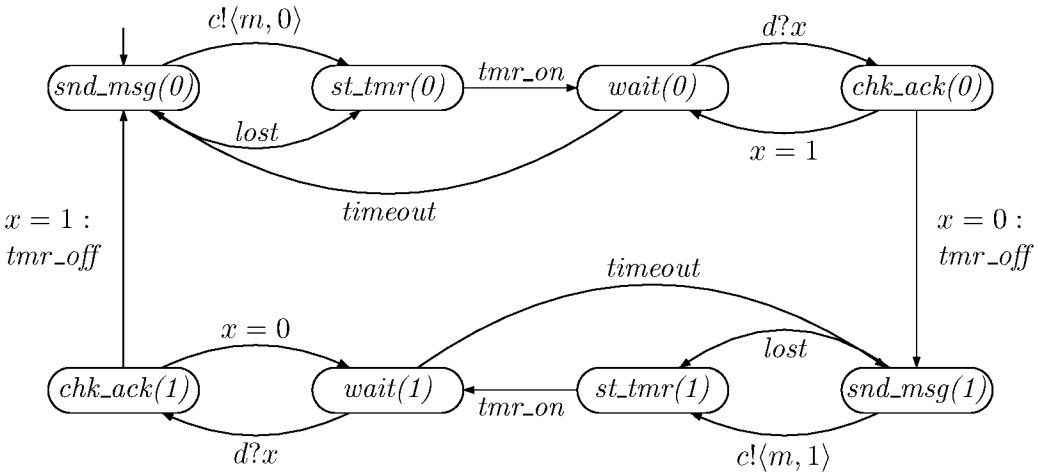


Figure 5.10: Program graph of ABP sender S.

since the unrealistic scenario in which (after some finite time) each message with alternating bit 1 is lost cannot be excluded. This corresponds to the path

$$\ldots\ldots\, s_i \xrightarrow{lost} s_{i+1} \xrightarrow{tmr\_on} s_{i+2} \xrightarrow{timeout} s_{i+3} \xrightarrow{lost} \ldots\ldots$$

Suppose we impose the strong LTL fairness assumption

$$sfair \;=\; \bigwedge_{b=0,1} \; \bigwedge_{k<cap(c)} \; (\Box\Diamond\,(send(b) \wedge |c| = k) \;\rightarrow\; \Box\Diamond\,|c| = k+1).$$

Here, $|c| = n$ stands for the atomic proposition that holds in the states $\langle \ell, \eta, \xi\rangle$ in which channel c contains exactly n elements, i.e., the length of the word $\xi(c)$ equals n. Thus, *sfair* describes (from the state-based point of view) that the loss of a transmitted message is not continuously possible. We now obtain

$$ABP \;\models_{sfair}\; \Box\Diamond start.$$

Note that it is essential to impose a strong fairness assumption on $send(b)$; as this action is not continuously enabled, a weak fairness assumption is insufficient. ■

In Section 3.5 (page 126 ff.), fairness was introduced using sets of actions; e.g., an execution is unconditionally A-fair for a set of actions A, whenever each action $\alpha \in A$ occurs infinitely often. LTL-fairness, however, is defined on atomic propositions, i.e., from a state-based perspective. Is there any relationship between these two—at first sight, rather different—approaches toward fairness?

The advantage of the action-based formulation of fairness assumptions is that many useful (and realizable) fairness assumptions can easily be expressed. Using the state-based

perspective, this may be less intuitive. For instance, the enabling of a process (or, more generally, of a certain action) is not necessarily a property that can be determined from the (atomic propositions in a) state. This discrepancy is, however, not always present.

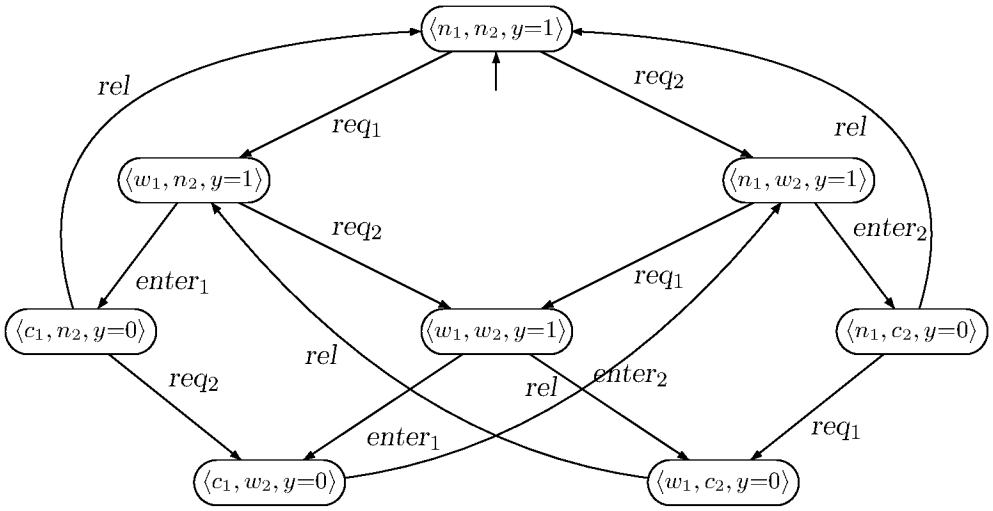


Figure 5.11: Semaphore-based mutual exclusion algorithm.

Example 5.29. State-Based vs. Action-Based Fairness

To exemplify this, consider the semaphore-based two-process mutual exclusion protocol (see Figure 5.11) together with the action-based strong-fairness assumption

$$\mathcal{F}_{strong} \;=\; \{\,\{\,enter_1\,\},\{\,enter_2\,\}\,\}.$$

Let us try to state the same constraint by means of a (state-based) LTL fairness assumption. Observe first that the action $enter_1$ is executable if and only if process P_1 is in the local state $wait_1$ and process P_2 is not in its critical section. Besides, on executing action $enter_1$, process P_1 moves to its critical section. Thus, strong fairness for $\{\,enter_1\,\}$ can be described by the LTL fairness assumption:

$$sfair_1 \;=\; \Box\Diamond(wait_1 \wedge \neg crit_2) \rightarrow \Box\Diamond crit_1.$$

The assumption $sfair_2$ is defined analogously. It now follows that $sfair \;=\; sfair_1 \wedge sfair_2$ describes $\mathcal{F}_{strong}$.

$\mathcal{F}_{strong}$ requires each process to enter its critical section infinitely often when it infinitely often gets the opportunity to do so. This does not forbid a process to never leave its noncritical section. To avoid this unrealistic scenario, the weak fairness constraint

$$\mathcal{F}_{weak} \;=\; \{\{\,req_1\,\},\{\,req_2\,\}\}$$

requires that any process infinitely often requests to enter the critical section when it continuously is able to do so. This (action-based) weak fairness constraint can be formulated as (state-based) LTL fairness assumption in a similar way as above. Observe that the request action of P_i is executable if and only if process P_i is in the local state $noncrit_i$. Weak fairness for $\{ req_i \}$ thus correponds to the LTL fairness assumption:

$$wfair_i \;=\; \Diamond\Box noncrit_i \rightarrow \Box\Diamond wait_i.$$

Let $fair \;=\; sfair \wedge wfair$ where $wfair \;=\; wfair_1 \wedge wfair_2$. It then follows that

$$TS_{Sem} \;\models_{fair}\; \Box\Diamond crit_1 \;\wedge\; \Box\Diamond crit_2.$$

■

In many cases, it is possible to replace action-based fairness by state-based LTL fairness assumptions. This, however, requires the possibility to deduce from the state label the possible enabled actions and the last executed action. It turns out that action-based fairness assumptions can always be "translated" into analogous LTL fairness assumptions. This works as follows. The intuition is to make a copy of each noninitial state s such that it is recorded which action was executed to enter state s. Such copy is made for every possible action via which the state can be entered. The copied state $\langle s, \alpha \rangle$ indicates that state s has been reached by performing α as last action.

Formally, this works as follows. For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ let

$$TS' \;=\; (S', Act', \rightarrow', I', AP', L')$$

where $Act' = Act \uplus \{ begin \}$, $I' = I \times \{ begin \}$ and $S' \;=\; I' \;\cup\; (S \times Act)$. The transition relation in TS' is defined by the rules:

$$\frac{s \xrightarrow{\alpha} s'}{\langle s, \beta \rangle \xrightarrow{\alpha}{}' \langle s', \alpha \rangle} \qquad \text{and} \qquad \frac{s_0 \xrightarrow{\alpha} s \quad s_0 \in I}{\langle s_0, begin \rangle \xrightarrow{\alpha}{}' \langle s, \alpha \rangle}$$

The state labeling is defined as follows. Let

$$AP' \;=\; AP \cup \{ enabled(\alpha), taken(\alpha) \mid \alpha \in Act \}$$

with the labeling function

$$L'(\langle s, \alpha \rangle) \;=\; L(s) \;\cup\; \{ taken(\alpha) \} \;\cup\; \{ enabled(\beta) \mid \beta \in Act(s) \}$$

for $\langle s, \alpha \rangle \in S \times Act$ and

$$L'(\langle s_0, begin \rangle) \;=\; L(s_0) \cup \{ enabled(\beta) \mid \beta \in Act(s_0) \}.$$

It can easily be established that

$$Traces_{AP}(TS) \;=\; Traces_{AP}(TS').$$

Strong fairness for a set of actions $A \subseteq Act$ can now be described by the strong LTL fairness assumption:

$$sfair_A \;=\; \Box\Diamond enabled(A) \;\rightarrow\; \Box\Diamond taken(A)$$

where

$$enabled(A) = \bigvee_{\alpha\in A} enabled(\alpha) \quad \text{and} \quad taken(A) = \bigvee_{\alpha\in A} taken(\alpha).$$

Unconditional and weak action-based fairness assumptions for TS can be transformed into LTL fairness assumptions for TS' in a similar way. For action-based fairness assumption $\mathcal{F}$ for TS and *fair* the corresponding LTL fairness assumption for TS', it follows that the set of fair traces coincides:

$$\begin{aligned} &\{trace_{AP}(\pi) \mid \pi \in Paths(TS), \pi \text{ is } \mathcal{F}\text{-fair}\} \\ =\; &\{trace_{AP}(\pi') \mid \pi' \in Paths(TS'), \pi' \models fair\}. \end{aligned}$$

Stated differently, $FairTraces_{\mathcal{F}}(TS) = FairTraces_{fair}(TS')$ where in TS' only atomic propositions in AP are considered. In particular, for every LT property P over AP,

$$TS \models_{\mathcal{F}} P \;\text{ iff }\; TS' \models_{fair} P.$$

Conversely, a (state-based) LTL fairness assumption cannot always be represented as action-based fairness assumption. This fact follows from the fact that strong or weak LTL fairness assumptions need not be realizable, while any action-based strong or weak fairness assumptions can be realized by a scheduler. In this sense, *state-based LTL fairness assumptions are more general than action-based fairness assumptions.*

The following theorem shows that the satisfaction relation $\models_{fair}$ as defined in Definition 5.26 has a strong relation to the usual satisfaction relation $\models$.

Theorem 5.30. Reduction of $\models_{fair}$ to $\models$

For transition system TS without terminal states, LTL formula φ, and LTL fairness assumption fair:

$$TS \models_{fair} \varphi \qquad \textit{if and only if} \qquad TS \models (fair \rightarrow \varphi).$$

Proof: $\Rightarrow$: Assume $TS \models_{fair} \varphi$. Then, for any path $\pi \in Paths(TS)$ either $\pi \models fair \wedge \varphi$ or $\pi \models \neg fair$. Thus, $\pi \models (fair \rightarrow \varphi)$, and consequently $TS \models (fair \rightarrow \varphi)$.

$\Leftarrow$: by a similar reasoning. ■

Example 5.31. On-The-Fly Garbage Collection

An important characteristic of pointer-manipulating programs is that certain parts of the dynamically changing data structure, such as a list or a tree, may become inaccessible. That is to say, this part of the data structure is not "reachable" by dereferencing one of the program variables. In order to be able to recycle these inaccessible memory cells, so-called *garbage collecting* algorithms are employed. These algorithms are key parts of operating systems, and play a major role in compilers. In this example, we focus on *on-the-fly* garbage collection algorithms. These algorithms attempt to identify and recycle inaccessible memory cells *concurrently* with the running programs that may manipulate the dynamically changing data structure. Typical requirements for such garbage collection algorithms are

Safety: An accessible (i.e., reachable) memory cell is never collected.

Liveness: Any unreachable storage cell is eventually collected.

The memory cells. The memory is considered to consist of a fixed number N of memory cells. The number of memory cells is static, i.e., the dynamic allocation and deallocation of cells (as in the C-statement `malloc`) is not considered. The memory cells are organized in a directed graph, whose structure may change during the computation; this happens, e.g., when carrying out an assignment $v := w$ where v and w point to a memory cell. For the sake of simplicity, it is assumed that each cell (= vertex) has at most one pointer (= edge) to another cell. Let $son(i)$ denote the immediate successor of cell i. Cells that do not have an outgoing reference are equipped with a self-reference. Thus, each cell has exactly one outgoing edge in the graph representation. Vertices are numbered. There is a fixed set of *root* vertices. Root cells are never considered as garbage. A memory cell is reachable if it is reachable in the graph from a root vertex. Cells that are not reachable from a root vertex are *garbage* cells. The memory is partitioned into three fragments: the accessible cells (i.e., cells that are reachable by the running processes), the free cells (i.e., cells that are unused and that can be assigned to running processes), and the unreachable cells. As for the garbage collection algorithm, it is only of importance which cells are unreachable. The distinction between free and accessible cells is of no importance, and will be neglected in the remainder.

Modeling the garbage collector. The entire system is modeled as the parallel composition of the garbage collector (process *Collector*), and a process that models the behaviour of the running processes manipulating the shared data structure. Thus:

$$Mutator \parallel Collector$$

For simplicity, we abstain from a detailed transition system representation and describe the mutator and the collector by pseudocode algorithms. As the garbage collecting algorithm should work correctly for any arbitrary set of concurrently running processes, the mutator is modeled by means of a single process that can nondeterministically choose two arbitrary reachable cells i and k and change the current reference $i \to j$ into $i \to k$. This corresponds to the assignment "$son(i) := k$". Note that if i was the only cell pointing to j, then after the assignment $son(i) := k$, the memory cell j has become garbage.

Algorithm 9 Mutator and garbage collector (naive version)

```
                                                        (* mutator *)
while true do
  Nondeterministically choose two reachable nodes i and k;
  son(i) := k
od
```

```
                                                        (* garbage collector *)
while true do
  label all reachable cells;                  (* e.g., by depth-first search *)
  for all cells i do
    if i is not labeled then collect i
  od
  add the collected cells to the list of free memory cells
od
```

Now let us consider the garbage collector. A naive technique could be based on a DFS or BFS which labels cells and, subsequently, collects all unlabeled cells as garbage (see Algorithm 9). The collected cells are added to the list of free memory cells, and thus are turned into reachable cells. This simple idea works correctly when the actions of the mutator are not interlocked with those of the garbage collector. As an on-the-fly algorithm, however, this naive idea fails due to the possible interleavings between the mutator and the collector. This is illustrated in Figure 5.12 where six memory configurations are depicted. Circles denote memory cells and edges represent the pointer structure. Nonshaded cells are not visited, gray ones are visited, and cells with a thick border have been completely processed. Due to the changes made by the mutator during the garbage collection phase, one of the memory cells remains unlabeled although it is accessible. The collector would now collect the unlabeled but accessible cell.

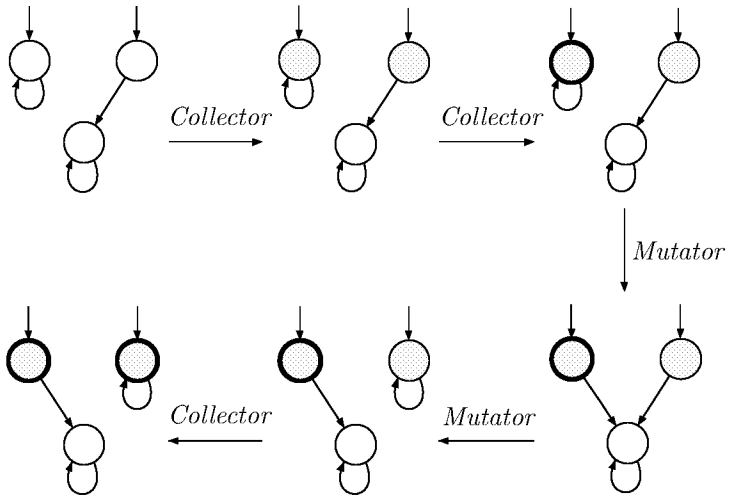


Figure 5.12: Failure of the naive garbage collector.

An alternative is to use a labeling algorithm that successively passes through all cells and labels their sons if the appropriate cell is labeled itself. Initially, only root cells are labeled. This technique is iterated as long as the number of labeled cells does not change anymore. The auxiliary variables M and M_{old} are used to this purpose. M counts the number of labeled cells in the current stage of labeling. M_{old} stands for the number of labeled cells in the previous stage of labeling. Additionally, the mutator supports the labeling process by labeling cell k as soon as a reference is redirected to k (see Algorithm 10). It is not difficult to show that the participation of the mutator is necessary to obtain a correct garbage collection algorithm. If it does not participate, there can always be found some interference pattern between the mutator and the collector such that reachable cells are made inaccessible by the mutator, causing a flaw in the garbage collection. Figure 5.13 demonstrates the functioning of the collector for four linearly linked reachable cells, where it is assumed that the labeling steps of the collector are not interrupted by the mutator.

The shading represents the labeling by the collector. The thick border of a node indicates that this node was considered in the for loop. Thus, in the last step, all four nodes were processed in the for loop. Therefore, the result of the first iteration is $M_{old} = 1$ (number of root cells) and $M = 3$, since exactly three cells were labeled. The labels arising at the end of the subsequent iterations are indicated in Figure 5.14.

Figure 5.15 shows an example that indicates that the mutator has to label cells, as otherwise it cannot be ensured that the collector identifies all reachable cells. The leftmost figure indicates the starting memory configuration. The situation after the collector has labeled the root cell and has processed the top left node in the FOR loop is depicted in

Algorithm 10 Ben Ari's on-the-fly garbage collection algorithm

```
                                                          (* mutator *)
while true do
  let  i and k be reachable memory cells;
  label k;
  son(i) := k
od
                                                          (* garbage collector *)
while true do
  label all root cells;
  M := number of root cells;
  repeat
    M_old := M;
    for all node i do
      if i is labeled then
        if son(i) is unlabeled then label son(i); M := M+1;  fi
      fi
    od
  until M = M_old;

  for all cell i do
    if i is labeled  then  delete label for cell i
    else collect cell i                                   (* cell i is garbage *)
    fi
  od
  add the collected cells to the list of free cells
od
```

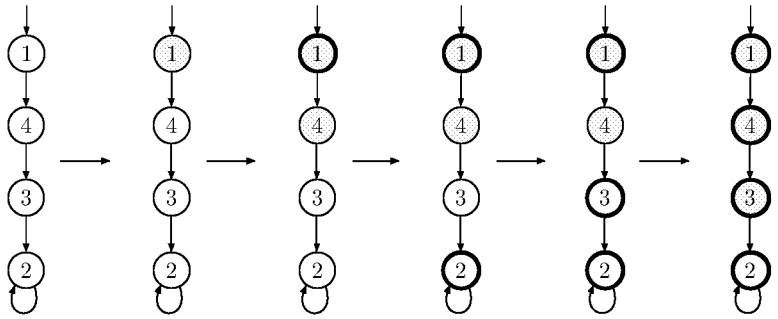


Figure 5.13: Example of Ben Ari's on-the-fly garbage collector (one iteration).

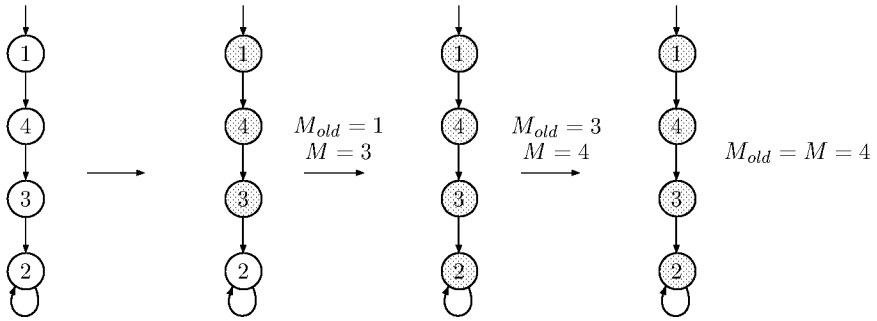


Figure 5.14: Example of Ben-Ari's on-the-fly garbage collector (all iterations).

the next figure. The third and fourth figures show a possible modification of the pointer structure by the mutator. The collector then would investigate the upper cell on the right and label it. Since this cell does not have an unlabeled successor, the first round of the collector is finished. As the obtained memory configuration is symmetric to the initial one, the whole procedure might be repeated, which leads to the initial configuration of Figure 5.15. In the second round, the collector also just labels the two upper cells. Since the number of labeled cells has not increased and assuming that the mutator does not label the lower cell, this cell would be wrongly regarded as garbage (and thus be collected).

The examples illustrate the functioning of the algorithm but are by no means meant as proof of correctness. Due to the enormous size of the state space and the extremely high degree of nondeterminism, the analysis of the garbage collection algorithm is extremely difficult. Note that the states of the transition system, which results from the parallel composition of mutator and collector, consist of control components plus a representation of the current labels and of another component that gives information about the current

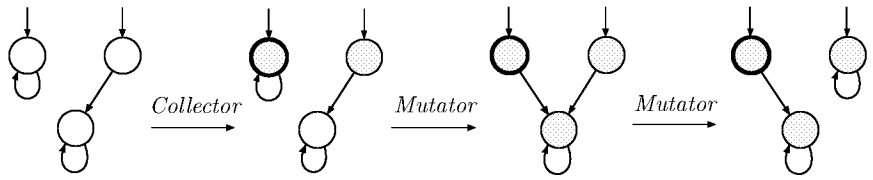


Figure 5.15: Labeling memory cells by the mutator is necessary.

memory configuration. For N nodes, there exist 2^N possible labels and N^N possible graphs. The high degree of nondeterminism results from the mutator that can change an arbitrary pointer. It is not useful to restrict the mutator (and thus reducing the degree of nondeterminism), since this would impose restrictions on the applicability of the garbage collecting algorithm.

To conclude, we will formalize the requirements on the concurrent garbage collection algorithm. To that end, let atomic proposition $collect(i)$ hold for cell i in a state if and only if cell i has been collected in that state. Similarly, $accessible(i)$ holds in those states in which cell i is reachable from a root cell. Given these atomic propositions, the safety property "an accessible memory cell is never collected" can be specified as

$$\bigwedge_{0<i\leqslant N} \Box(\ collect(i) \rightarrow \neg accessible(i)\ ).$$

The liveness property "any unreachable storage cell is eventually collected" is described as LTL formula

$$\bigwedge_{0<i\leqslant N} \Box(\ \neg accessible(i)\ \rightarrow\ \Diamond collect(i)\ ).$$

It turns out that Ben Ari's concurrent garbage collection algorithm satisfies the safety property, but refutes the liveness property. The latter happens, e.g., in the pathological case where only the mutator acts infinitely often. To rule out this unrealistic behavior, weak process fairness can be imposed. ■

5.2 Automata-Based LTL Model Checking

In this section we address the model-checking problem for LTL. The starting point is a finite transition system TS and an LTL formula φ that formalizes a requirement on TS. The problem is to check whether $TS \models \varphi$. If φ is refuted, an error trace needs to be provided for debugging purposes. The considerations in Section 2.1 show that transition

systems are typically huge. Therefore, a manual proof of $TS \models \varphi$ is extremely difficult. Instead, verification tools are desirable, which enable a fully automated analysis of the transition system.

In general, not just a single requirement but several requirements are relevant. These requirements can be represented by separate formulae, such as $\varphi_1, \ldots, \varphi_k$, and be combined into $\varphi_1 \wedge \ldots \wedge \varphi_k$ to obtain a specification of all requirements. Alternatively, the requirements φ_i can be treated separately. This is often more efficient than considering them together. Moreover, the decomposition of the entire requirement specification into several requirements is advised if errors are to be expected or if the validity of φ_i is known by a prior analysis.

An LTL model-checking algorithm is a decision procedure which for a transition system TS and LTL formula φ returns the answers "yes" if $TS \models \varphi$, and "no" (plus a counterexample) if $TS \not\models \varphi$. The counterexample consists of an appropriate finite prefix of an infinite path in TS where φ does not hold. Theorem 5.30 shows that special measures to handle fairness assumptions are unnecessary as fairness assumptions can be encoded in the LTL formula to be checked. (For reasons of efficiency, however, it is advised to use special algorithms to treat fairness assumptions.)

Throughout this section, TS is assumed to be finite and to have no terminal states. The model-checking algorithm presented in the following is based on the *automata-based approach* as originally suggested by Vardi and Wolper (1986). This approach is based on the fact that each LTL formula φ can be represented by a nondeterministic Büchi automaton (NBA). The basic idea is to try to disprove $TS \models \varphi$ by "looking" for a path π in TS with $\pi \models \neg\varphi$. If such a path is found, a prefix of π is returned as error trace. If no such path is encountered, it is concluded that $TS \models \varphi$.

The essential steps of the model-checking algorithm as summarized in Algorithm 11 and Figure 5.16, rely on the following observations:

$$\begin{aligned} TS \models \varphi \quad & \text{iff} \quad \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi) \\ & \text{iff} \quad \mathit{Traces}(TS) \cap ((2^{AP})^\omega \setminus \mathit{Words}(\varphi)) = \varnothing \\ & \text{iff} \quad \mathit{Traces}(TS) \cap \mathit{Words}(\neg\varphi) = \varnothing. \end{aligned}$$

Hence, for NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathit{Words}(\neg\varphi)$ we have

$$TS \models \varphi \quad \text{if and only if} \quad \mathit{Traces}(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \varnothing.$$

Thus, to check whether φ holds for TS one first constructs an NBA for the negation of the input formula φ (representing the "bad behaviors") and then applies the techniques explained in Chapter 4 for the intersection problem.

Algorithm 11 Automaton-based LTL model checking

Input: finite transition system TS and LTL formula φ (both over AP)
Output: "yes" if $TS \models \varphi$; otherwise, "no" plus a counterexample

Construct an NBA $\mathcal{A}_{\neg\varphi}$ such that $\mathcal{L}_\omega(\mathcal{A}_{\neg\varphi}) = \mathit{Words}(\neg\varphi)$
Construct the product transition system $TS \otimes \mathcal{A}$

if there exists a path π in $TS \otimes \mathcal{A}$ satisfying the accepting condition of $\mathcal{A}$ **then**
  return "no" and an expressive prefix of π
else
  return "yes"
fi

It remains to explain how a given LTL formula can be represented by an NBA and how such an NBA can be constructed algorithmically. First observe that for the LTL formula φ, the LTL semantics provided in Definition 5.6 on page 235 yields a language $\mathit{Words}(\varphi) \subseteq (2^{AP})^\omega$. Thus, the alphabet of NBA for LTL formulae is $\Sigma = 2^{AP}$. The next step is to show that $\mathit{Words}(\varphi)$ is ω-regular, and hence, representable by a nondeterministic Büchi automaton.

Example 5.32. NBA for LTL Formulae

Before treating the details of transforming an LTL formula into an NBA, we provide some examples. As in Chapter 4, propositional logic formulae are used (instead of the set notations) for a symbolic representation of the edges of an NBA. These formulae are built by the symbols $a \in AP$, the constant true, and the Boolean connectors, and thus they can be interpreted over *sets* of atomic propositions, i.e., the elements $A \in \Sigma = 2^{AP}$. For instance, if $AP = \{a, b\}$ then $q \xrightarrow{a \vee b} q'$ is a short notation for the three transitions:

$$q \xrightarrow{\{a\}} q', \; q \xrightarrow{\{b\}} q', \text{ and } q \xrightarrow{\{a,b\}} q'.$$

The language of all words $\sigma = A_0 A_1 \ldots \in 2^{AP}$ satisfying the LTL formula $\Box\Diamond \mathit{green}$ ("infinitely often *green*") is accepted by the NBA $\mathcal{A}$ shown in Figure 5.17. Here, AP is a set of atomic propositions containing *green*. Note that $\mathcal{A}$ is in the accept state q_1 if and only if the last consumed symbol (the last set A_i of the input word $A_0 A_1 A_2 \ldots \in (2^{AP})^\omega$) contains the propositional symbol *green*. Therefore, the accepted language $\mathcal{L}_\omega(\mathcal{A})$ is exactly the set of all infinite words $A_0 A_1 A_2 \ldots$ with infinitely many indices i where $\mathit{green} \in A_i$. Thus,

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \mathit{Words}(\Box\Diamond \mathit{green}).$$

The accepting run that generates the word $\sigma = \{\mathit{green}\}\varnothing\{\mathit{green}\}\varnothing\ldots$ is $(q_0\, q_1)^\omega$.

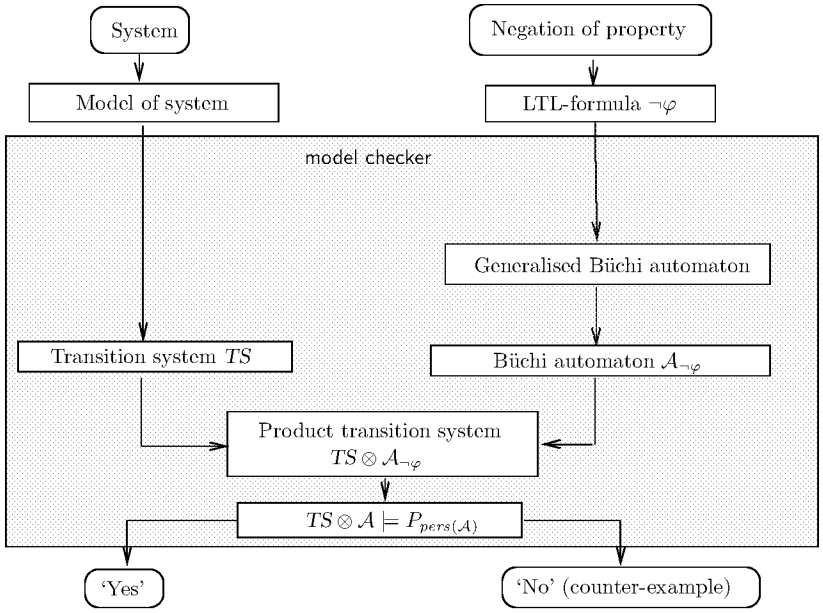


Figure 5.16: Overview of LTL model checking.

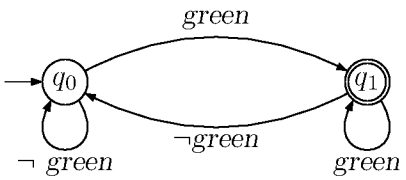


Figure 5.17: NBA for $\Box\Diamond green$.

As a second example consider the liveness property: "whenever event a occurs, event b will eventually occur". For example, the property given by the LTL formula $\Box(request \rightarrow \Diamond response)$ is of this form. An associated NBA over the alphabet $2^{\{a,b\}}$ where $a = request$ and $b = response$ is shown in Figure 5.18.

The automata in Figures 5.17 and 5.18 are *deterministic*, i.e., they have exactly one run for each input word. To represent temporal properties like "eventually forever (from some moment on)", the concept of nondeterminism is, however, necessary.

The NBA $\mathcal{A}$ shown in Figure 5.19 accepts the language $Words(\Diamond\Box a)$. Here, $AP \supseteq \{ a \}$ and $\Sigma = 2^{AP}$; see also Example 4.51 (page 191). Intuitively, the NBA $\mathcal{A}$ nondeterministically decides (by means of an omniscient oracle) when a continuously holds. Note that state q_2 may be omitted, as there is no accepting run beginning in q_2. (The reader should bear in mind that DBA and NBA are not equally expressive; see Section 4.3.3 on page 188.) ■

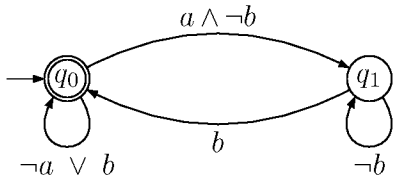


Figure 5.18: NBA for $\Box(a \rightarrow \Diamond b)$.

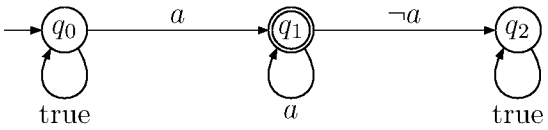


Figure 5.19: NBA for $\Diamond\Box a$.

A key ingredient to the model-checking algorithm for LTL is the construction of an NBA $\mathcal{A}$ satisfying

$$\mathcal{L}_\omega(\mathcal{A}) \;=\; \mathit{Words}(\varphi)$$

for the LTL formula φ. In order to do so, first a generalized NBA is constructed for φ, which subsequently is transformed into an equivalent NBA. For the latter step we employ the recipe as provided in Theorem 4.56 on page 195. For the sake of convenience we recall the definition of generalized NBA; see also Definition 4.52 on page 193.

Definition 5.33. Generalized NBA (GNBA)

A generalized NBA is a tuple $\mathcal{G} = (Q, \Sigma, \delta, Q_0, \mathcal{F})$ where Q, Σ, δ, Q_0 are defined as for NBA (i.e., Q is a finite state space, Σ an alphabet, $Q_0 \subseteq Q$ the set of initial states, and $\delta : Q \times \Sigma \rightarrow 2^Q$ the transition relation) and $\mathcal{F}$ is a (possibly empty) subset of 2^Q. The elements of $\mathcal{F}$ are called *acceptance sets*. The accepted language $\mathcal{L}_\omega(\mathcal{G})$ consists of all infinite words in $(2^{AP})^\omega$ that have at least one infinite run $q_0\, q_1\, q_2 \ldots$ in $\mathcal{G}$ such that for each acceptance set $F \in \mathcal{F}$ there are infinitely many indices i with $q_i \in F$. ■

A GNBA for which $\mathcal{F}$ is a singleton set can be regarded as an NBA. If the set $\mathcal{F}$ of acceptance sets in $\mathcal{G}$ is empty, the language $\mathcal{L}_\omega(\mathcal{G})$ consists of all infinite words that have an infinite run in $\mathcal{G}$. Hence, if $\mathcal{F} = \varnothing$, then $\mathcal{G}$ can be viewed as an NBA for which all states are accepting.

Let us consider how to construct a GNBA over the alphabet 2^{AP} for a given LTL formula φ (over AP), i.e., a GNBA $\mathcal{G}_\varphi$ with $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. Assume φ only contains the operators $\wedge$, $\neg$, $\bigcirc$ and U, i.e., the derived operators $\vee$, $\rightarrow$, $\Diamond$, $\Box$, W, and so on are assumed to be expressed in terms of the basic operators. Since the special case $\varphi = \text{true}$ is trivial, it may be assumed that $\varphi \neq \text{true}$.

The basic idea for the construction of $\mathcal{G}_\varphi$ is as follows. Let $\sigma = A_0\, A_1\, A_2 \ldots \in \mathit{Words}(\varphi)$. The sets $A_i \subseteq AP$ are expanded by subformulae ψ of φ such that an infinite word $\bar{\sigma} = B_0\, B_1\, B_2 \ldots$ with the following property arises:

$$\psi \in B_i \qquad \text{if and only if} \qquad \underbrace{A_i\, A_{i+1}\, A_{i+2} \ldots}_{\sigma^i} \models \psi.$$

For technical reasons, the subformulae ψ of φ are considered as well as their negation $\neg\psi$. Consider, e.g.,

$$\varphi = a \,\mathsf{U}\, (\neg a \wedge b) \quad \text{and} \quad \sigma = \{\, a \,\}\, \{\, a, b \,\}\, \{\, b \,\} \ldots .$$

In this case, B_i is a subset of the set of formulae

$$\underbrace{\{\, a, b, \neg a, \neg a \wedge b, \varphi \,\}}_{\text{subformulae of } \varphi} \cup \underbrace{\{\, \neg b, \neg(\neg a \wedge b), \neg\varphi \,\}}_{\text{their negation}}.$$

The set $A_0 = \{\, a \,\}$ is extended with the formulae $\neg b$, $\neg(\neg a \wedge b)$, and φ, since all these formulae hold in $\sigma^0 = \sigma$, and all other subformulae in the above set are refuted by σ. We thus obtain

$$B_0 \;=\; \{\, a, \neg b, \neg(\neg a \wedge b), \varphi \,\}.$$

The set $A_1 = \{\, a, b \,\}$ is extended with $\neg(\neg a \wedge b)$ and φ, as these are the only subformulae in the above set that hold in $\sigma^1 = \{\, a, b \,\}\, \{\, b \,\} \ldots$. The $A_2 = \{\, b \,\}$ is extended with $\neg a$, $\neg a \wedge b$ and φ as they hold in $\sigma^2 = \{\, b \,\} \ldots$. This yields a word of the form:

$$\bar{\sigma} \;=\; \{\, a, \neg b, \neg(\neg a \wedge b), \varphi \,\}\, \{\, a, b, \neg(\neg a \wedge b), \varphi \,\}\, \{\, \neg a, b, \neg a \wedge b, \varphi \,\} \ldots$$

As σ is infinite, this procedure is of course not effective. The example is just meant to explain the intuition behind the construction of the states in the GNBA.

The GNBA $\mathcal{G}_\varphi$ is constructed such that the sets B_i constitute its *states*. Moreover, the construction ensures that $\bar{\sigma} = B_0\, B_1\, B_2 \ldots$ is a *run* for $\sigma = A_0\, A_1\, A_2 \ldots$ in $\mathcal{G}_\varphi$. The accepting conditions for $\mathcal{G}_\varphi$ are chosen such that the run $\bar{\sigma}$ is accepting if and only if $\sigma \models \varphi$. Thus, we have to encode the meaning of the logical operators into the states, transitions, and acceptance sets of $\mathcal{G}_\varphi$. The meaning of propositional logic operators $\wedge$, $\neg$, and the constant true will be encoded in the states by requiring consistent formula sets B_i. The semantics of the next-step operator relies on a nonlocal condition and will be encoded in the transition relation. The meaning of the until operator is split according to the expansion law into local conditions (encoded in the states) and a next-step condition (encoded in the transitions). Since the expansion law does not provide a full characterization of until, a further condition is imposed which expresses the fact that the meaning of until yields the least solution of the expansion law (see Lemma 5.18 on page 251). This will be encoded by the acceptance sets of $\mathcal{G}_\varphi$.

As explained above, the formula sets are subsets of subformulae of φ and their negation.

Definition 5.34. Closure of φ

The *closure* of LTL formula φ is the set $\mathit{closure}(\varphi)$ consisting of all subformulae ψ of φ and their negation $\neg\psi$ (where ψ and $\neg\neg\psi$ are identified). ■

For instance, for $\varphi = a \,\mathsf{U}\, (\neg a \wedge b)$, the set $\mathit{closure}(\varphi)$ consists of the formulae

$$a, b, \neg a, \neg b, \neg a \wedge b, \neg(\neg a \wedge b), \varphi, \neg\varphi.$$

It is not difficult to assess that $|\mathit{closure}(\varphi)| \in \mathcal{O}(|\varphi|)$. A set of formulae $B \subseteq \mathit{closure}(\varphi)$ is called *elementary* if B is the set of all formulae $\psi \in \mathit{closure}(\varphi)$ with $\pi \models \psi$ for a path π. For this, B should not contain propositional logic contradictions and it must be *locally consistent* with respect to the until operator. Since for any path π and formula ψ, either $\pi \models \psi$ or $\pi \models \neg\psi$, it is additionally required that elementary sets of formulae are *maximal*. The precise definition of these three conditions is provided in Figure 5.20 on page 277.

Definition 5.35. Elementary Sets of Formulae

$B \subseteq \mathit{closure}(\varphi)$ is *elementary* if it is consistent with respect to propositional logic, maximal, and locally consistent with respect to the until operator. ■

The requirements for local consistency result from the expansion law

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \;\equiv\; \varphi_2 \;\vee\; (\varphi_1 \;\wedge\; \bigcirc\, (\varphi_1 \,\mathsf{U}\, \varphi_2)).$$

Due to the required maximality and propositional logic consistency, we have

$$\psi \in B \text{if and only if} \quad \neg\psi \notin B$$

for all elementary sets B and subformulae ψ of φ. Further, due to maximality and local consistency, we have

$$\varphi_1, \varphi_2 \notin B \quad \text{implies} \quad \varphi_1 \,\mathsf{U}\, \varphi_2 \notin B.$$

Hence, if $\varphi_1, \varphi_2 \notin B$ then $\{\, \neg\varphi_1, \neg\varphi_2, \neg(\varphi_1 \,\mathsf{U}\, \varphi_2) \,\} \subseteq B$; here, it is assumed that $\varphi_1 \,\mathsf{U}\, \varphi_2$ is a subformula of φ.

Example 5.36. Elementary Sets of Formulae

Let $\varphi = a \,\mathsf{U}\, (\neg a \wedge b)$. The set $B = \{\, a, b, \varphi \,\} \subseteq \mathit{closure}(\varphi)$ is consistent with respect to propositional logic and locally consistent with respect to the until operator. It is, however, not maximal, since for $\neg a \wedge b \in \mathit{closure}(\varphi)$:

$$\neg a \wedge b \notin B \qquad \text{and} \qquad \neg(\neg a \wedge b) \notin B.$$

1. B is *consistent* with respect to propositional logic, i.e., for all $\varphi_1 \wedge \varphi_2$, $\psi \in \mathit{closure}(\varphi)$:
   - $\varphi_1 \wedge \varphi_2 \in B \;\Leftrightarrow\; \varphi_1 \in B$ and $\varphi_2 \in B$
   - $\psi \in B \;\Rightarrow\; \neg\psi \notin B$
   - $\mathrm{true} \in \mathit{closure}(\varphi) \;\Rightarrow\; \mathrm{true} \in B$.
2. B is *locally consistent* with respect to the until operator, i.e., for all $\varphi_1 \mathsf{U} \varphi_2 \in \mathit{closure}(\varphi)$:
   - $\varphi_2 \in B \;\Rightarrow\; \varphi_1 \mathsf{U} \varphi_2 \in B$
   - $\varphi_1 \mathsf{U} \varphi_2 \in B$ and $\varphi_2 \notin B \;\Rightarrow\; \varphi_1 \in B$.
3. B is *maximal*, i.e., for all $\psi \in \mathit{closure}(\varphi)$:
   - $\psi \notin B \;\Rightarrow\; \neg\psi \in B$.

Figure 5.20: Properties of elementary sets of formulae.

The set of formulae $\{ a, b, \neg a \wedge b, \varphi \}$ contains the propositional logic "contradiction" a and $\neg a \wedge b$ and therefore is not elementary. The set

$$\{\neg a, \neg b, \neg(\neg a \wedge b), \varphi\}$$

is consistent with respect to propositional logic but contains a local inconsistency with respect to the until operator U, since $a \mathsf{U} (\neg a \wedge b) \in B$ and $\neg a \wedge b \notin B$, but $a \notin B$. This means that

$$\pi \models \neg a, \quad \pi \models \neg(\neg a \wedge b), \quad \text{and} \quad \pi \models \varphi$$

are impossible for any path π.

The following sets are elementary:

$$\begin{array}{lcl}
B_1 & = & \{ \quad a, \quad b, \quad \neg(\neg a \wedge b), \quad \varphi \quad \}, \\
B_2 & = & \{ \quad a, \quad b, \quad \neg(\neg a \wedge b), \quad \neg\varphi \quad \}, \\
B_3 & = & \{ \quad a, \quad \neg b, \quad \neg(\neg a \wedge b), \quad \varphi \quad \}, \\
B_4 & = & \{ \quad a, \quad \neg b, \quad \neg(\neg a \wedge b), \quad \neg\varphi \quad \}, \\
B_5 & = & \{ \quad \neg a, \quad \neg b, \quad \neg(\neg a \wedge b), \quad \neg\varphi \quad \}, \\
B_6 & = & \{ \quad \neg a, \quad b, \quad \neg a \wedge b, \quad \varphi \quad \}.
\end{array}$$

■

The proof of the following theorem shows how to construct for an arbitrary LTL formula φ a GNBA $\mathcal{G}_\varphi$ with $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. This construction is one of the initial steps of the

LTL model checking algorithm; see Figure 5.16 (page 273). Subsequently, the resulting GNBA $\mathcal{G}_\varphi$ is transformed into an NBA $\mathcal{A}_\varphi$ by means of the technique indicated in the proof of Theorem 4.56 (page 195).

Theorem 5.37. GNBA for LTL Formula

For any LTL formula φ (over AP) there exists a GNBA $\mathcal{G}_\varphi$ over the alphabet 2^{AP} such that

(a) $\mathit{Words}(\varphi) = \mathcal{L}_\omega(\mathcal{G}_\varphi)$.

(b) $\mathcal{G}_\varphi$ *can be constructed in time and space* $2^{\mathcal{O}(|\varphi|)}$.

(c) The number of accepting sets of $\mathcal{G}_\varphi$ *is bounded above by* $\mathcal{O}(|\varphi|)$.

Proof: Let φ be an LTL formula over AP. Let $\mathcal{G}_\varphi = (Q, 2^{AP}, \delta, Q_0, \mathcal{F})$ where

- Q is the set of all elementary sets of formulae $B \subseteq \mathit{closure}(\varphi)$,
- $Q_0 \;=\; \{\, B \in Q \mid \varphi \in B \,\}$,
- $\mathcal{F} \;=\; \{\, F_{\varphi_1 \,\mathsf{U}\, \varphi_2} \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \in \mathit{closure}(\varphi) \}$ where

$$F_{\varphi_1 \,\mathsf{U}\, \varphi_2} \;=\; \{\, B \in Q \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \notin B \text{ or } \varphi_2 \in B \,\}.$$

The transition relation $\delta : Q \times 2^{AP} \to 2^Q$ is given by:

- If $A \neq B \cap AP$, then $\delta(B, A) \;=\; \varnothing$.
- If $A = B \cap AP$, then $\delta(B, A)$ is the set of all elementary sets of formulae B' satisfying
  - (i) for every $\bigcirc \psi \in \mathit{closure}(\varphi)$: $\bigcirc \psi \in B \;\Leftrightarrow\; \psi \in B'$, and
  - (ii) for every $\varphi_1 \,\mathsf{U}\, \varphi_2 \in \mathit{closure}(\varphi)$:

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \in B \;\Leftrightarrow\; (\varphi_2 \in B \;\vee\; (\varphi_1 \in B \wedge \varphi_1 \,\mathsf{U}\, \varphi_2 \in B')).$$

The constraints (i) and (ii) reflect the semantics of the next-step and the until operator, respectively. Rule (ii) is justified by the expansion law:

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \;\equiv\; \varphi_2 \vee (\varphi_1 \wedge \bigcirc\, (\varphi_1 \,\mathsf{U}\, \varphi_2)).$$

To model the semantics of U, an acceptance set F_ψ is introduced for every subformula $\psi = \varphi_1 \,\mathsf{U}\, \varphi_2$ of φ. The underlying idea is to ensure that in every run $B_0\, B_1\, B_2 \ldots$ for which $\psi \in B_0$, we have $\varphi_2 \in B_j$ (for some $j \geqslant 0$) and $\varphi_1 \in B_i$ for all $i < j$. The requirement that a word σ satisfies $\varphi_1 \,\mathsf{U}\, \varphi_2$ only if φ_2 will actually eventually become true is ensured by the accepting set $F_{\varphi_1 \,\mathsf{U}\, \varphi_2}$.

Let us first consider the claim (b). States in the GNBA $\mathcal{G}_\varphi$ are elementary sets of formulae in $\mathit{closure}(\varphi)$. Let $\mathrm{subf}(\varphi)$ denote the set of all subformulae of φ. The number of states in $\mathcal{G}_\varphi$ is bounded by $2^{|\mathrm{subf}(\varphi)|}$, the number of possible formula sets. (The elementary formula sets B can be represented by bit vectors containing a single bit per subformula ψ of φ which indicates whether ψ or $\neg\psi$ belongs to B.) As $|\mathrm{subf}(\varphi)| \leqslant 2{\cdot}|\varphi|$, the number of states in the GNBA $\mathcal{G}_\varphi$ is bounded by $2^{\mathcal{O}(|\varphi|)}$. Claim (c) follows directly from the fact that the number of accept sets is equal to the number of until-subformulae in φ.

It remains to show that (a) $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. We prove set inclusion in both directions.

$\supseteq$: Let $\sigma = A_0\, A_1\, A_2 \ldots \in \mathit{Words}(\varphi)$. Then, $\sigma \in (2^{AP})^\omega$ and $\sigma \models \varphi$. The elementary set B_i of formulae is defined as follows:

$$B_i \;=\; \{\, \psi \in \mathit{closure}(\varphi) \mid A_i A_{i+1} \ldots \models \psi \,\} \tag{5.1}$$

Obviously, B_i is an elementary set of formulae, i.e., $B_i \in Q$. We now prove that $B_0\, B_1\, B_2 \ldots$ is an accepting run for σ. Observe that $B_{i+1} \in \delta(B_i, A_i)$ for all $i \geqslant 0$, since for all i:

- $A_i = B_i \cap AP$
- for $\bigcirc \psi \in \mathit{closure}(\varphi)$:

$$\begin{array}{ll} & \bigcirc \psi \in B_i \\ \text{iff} & \qquad\qquad \text{(* equation (5.1) for } B_i \text{ *)} \\ & A_i\, A_{i+1} \ldots \models \bigcirc \psi \\ \text{iff} & \qquad\qquad \text{(* semantics of } \bigcirc \text{ *)} \\ & A_{i+1}\, A_{i+2} \ldots \models \psi \\ \text{iff} & \qquad\qquad \text{(* equation (5.1) for } B_{i+1} \text{ *)} \\ & \psi \in B_{i+1} \end{array}$$

- for $\varphi_1 \mathsf{U} \varphi_2 \in \mathit{closure}(\varphi)$:

$$
\begin{array}{ll}
 & \varphi_1 \mathsf{U} \varphi_2 \in B_i \\
\text{iff} & \qquad\qquad (* \text{ equation (5.1) for } B_i \ *) \\
 & A_i\, A_{i+1} \ldots \models \varphi_1 \mathsf{U} \varphi_2 \\
\text{iff} & \qquad\qquad (* \text{ semantics of until } *) \\
 & A_i\, A_{i+1} \ldots \models \varphi_2 \ \text{ or} \\
 & A_i\, A_{i+1} \ldots \models \varphi_1 \text{ and } (A_{i+1}\, A_{i+2} \ldots \models \varphi_1 \mathsf{U} \varphi_2 \qquad) \\
\text{iff} & \qquad\qquad (* \text{ equation (5.1) for } B_i \text{ and } B_{i+1} \ *) \\
 & \varphi_2 \in B_i \text{ or } (\varphi_1 \in B_i \text{ and } \varphi_1 \mathsf{U} \varphi_2 \in B_{i+1}) \qquad .
\end{array}
$$

This shows that $B_0\, B_1\, B_2 \ldots$ is a run of $\mathcal{G}_\varphi$. It remains to prove that this run is accepting, i.e., for each subformula $\varphi_{1,j} \mathsf{U} \varphi_{2,j}$ in $\mathit{closure}(\varphi)$, $B_i \in F_j$ for infinitely many i. By contraposition. Assume there are finitely many i such that $B_i \in F_j$. We have:

$$B_i \notin F_j = F_{\varphi_{1,j} \mathsf{U} \varphi_{2,j}} \;\Rightarrow\; \varphi_{1,j} \mathsf{U} \varphi_{2,j} \in B_i \text{ and } \varphi_{2,j} \notin B_i.$$

As $B_i = \{\, \psi \in \mathit{closure}(\varphi) \mid A_i\, A_{i+1} \ldots \models \psi \,\}$, it follows that if $B_i \notin F_j$, then:

$$A_i\, A_{i+1} \ldots \models \varphi_{1,j} \mathsf{U} \varphi_{2,j} \quad \text{and} \quad A_i\, A_{i+1} \ldots \not\models \varphi_{2,j}.$$

Thus, $A_k\, A_{k+1} \ldots \models \varphi_{2,j}$ for some $k > i$. By definition of the formula sets B_i, it then follows that $\varphi_{2,j} \in B_k$, and by definition of F_j, $B_k \in F_j$. Thus, $B_i \in F_j$ for finitely many i, then $B_k \in F_j$ for infinitely many k. Contradiction.

Thus, $B_0\, B_1\, B_2 \ldots$ is an accepting run of $\mathcal{G}_\varphi$, and hence, $A_0\, A_1\, A_2 \ldots \in \mathcal{L}_\omega(\mathcal{G}_\varphi)$.

$\subseteq$: Let $\sigma = A_0\, A_1\, A_2 \ldots \in \mathcal{L}_\omega(\mathcal{G}_\varphi)$, i.e., there is an accepting run $B_0\, B_1\, B_2 \ldots$, say, for σ in $\mathcal{G}_\varphi$. Since

$$\delta(B, A) = \varnothing \quad \text{for all pairs } (B, A) \text{ with } A \neq B \cap AP,$$

it follows that $A_i = B_i \cap AP$ for $i \geqslant 0$. Thus

$$\sigma \;=\; (B_0 \cap AP)\ (B_1 \cap AP)\ (B_2 \cap AP) \ldots$$

Our proof obligation now becomes $(B_0 \cap AP)\ (B_1 \cap AP)\ (B_2 \cap AP) \ldots \models \varphi$. We prove the following more general proposition:

For $B_0 B_1 B_2 \ldots$ a sequence with $B_i \in Q$ satisfying

(i) for all $i \geqslant 0 : B_{i+1} \in \delta(B_i, A_i)$, and

(ii) for all $F \in \mathcal{F} : \overset{\infty}{\exists}\, j \geqslant 0.\ B_j \in F$,

we have for all $\psi \in \mathit{closure}(\varphi)$:

$$\psi \in B_0 \quad \Leftrightarrow \quad A_0\, A_1\, A_2 \ldots \models \psi.$$

The proof of this claim is by structural induction on the structure of ψ.

Base case: The statement for $\psi = \text{true}$ or $\psi = a$ with $a \in AP$ follows directly from equation (5.1) and the definition of closure.

Induction step: Based on the induction hypothesis that the claim holds for $\psi', \varphi_1, \varphi_2 \in \mathit{closure}(\varphi)$, it is proven that for the formulae

$$\psi = \bigcirc \psi',\ \psi = \neg\psi',\ \psi = \varphi_1 \wedge \varphi_2 \text{ and } \psi = \varphi_1\, \mathsf{U}\, \varphi_2$$

the claim also holds. We provide the detailed proof for $\psi = \varphi_1\, \mathsf{U}\, \varphi_2$. Let $A_0\, A_1\, A_2 \ldots \in (2^{AP})^\omega$ and $B_0\, B_1\, B_2 \ldots \in Q^\omega$ satisfying the constraints (i) and (ii). It is now shown that:

$$\psi \in B_0 \quad \text{iff} \quad A_0\, A_1\, A_2 \ldots \models \psi.$$

This goes as follows.

$\Leftarrow$: Assume $A_0\, A_1\, A_2 \ldots \models \psi$ where $\psi = \varphi_1\, \mathsf{U}\, \varphi_2$. Then, there exists $j \geqslant 0$ such that

$$A_j\, A_{j+1} \ldots \models \varphi_2 \quad \text{and} \quad A_i\, A_{i+1} \ldots \models \varphi_1 \text{ for } 0 \leqslant i < j.$$

From the induction hypothesis (applied to φ_1 and φ_2) it follows that

$$\varphi_2 \in B_j \quad \text{and} \quad \varphi_1 \in B_i \quad \text{for } 0 \leqslant i < j.$$

By induction on j we obtain: $\varphi_1\, \mathsf{U}\, \varphi_2 \in B_j, B_{j-1} \ldots, B_0$.

$\Rightarrow$: Assume $\varphi_1\, \mathsf{U}\, \varphi_2 \in B_0$. Since B_0 is elementary, $\varphi_1 \in B_0$ or $\varphi_2 \in B_0$. Distinguish between $\varphi_2 \in B_0$ and $\varphi_2 \notin B_0$. If $\varphi_2 \in B_0$, it follows from the induction hypothesis $A_0\, A_1 \ldots \models \varphi_2$, and thus $A_0\, A_1 \ldots \models \varphi_1\, \mathsf{U}\, \varphi_2$. This remains the case $\varphi_2 \notin B_0$. Then $\varphi_1 \in B_0$ and $\varphi_1\, \mathsf{U}\, \varphi_2 \in B_0$. Assume $\varphi_2 \notin B_j$ for all $j \geqslant 0$. From the definition of the transition relation δ, we obtain using an inductive argument (successively applied to $\varphi_1 \in B_j$, $\varphi_2 \notin B_j$ and $\varphi_1\, \mathsf{U}\, \varphi_2 \in B_j$ for $j \geqslant 0$):

$$\varphi_1 \in B_j \quad \text{and} \quad \varphi_1\, \mathsf{U}\, \varphi_2 \in B_j \quad \text{for all } j \geqslant 0.$$

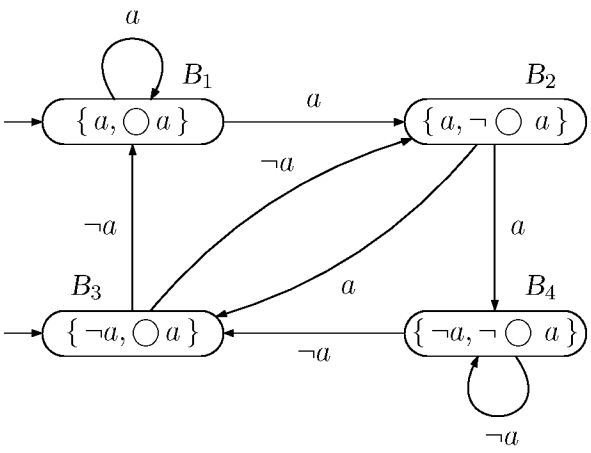


Figure 5.21: A generalised Büchi automaton for the LTL formula $\bigcirc a$.

As $B_0\, B_1\, B_2 \ldots$ satisfies constraint (ii), it follows that

$$B_j \in F_{\varphi_1 \mathsf{U} \varphi_2} \text{for infinitely many } j \geqslant 0.$$

On the other hand, we have

$$\underbrace{\varphi_2 \notin B_j \quad \text{and} \quad \varphi_1 \mathsf{U} \varphi_2 \in B_j}_{\text{iff} \quad B_j \notin F_{\varphi_1 \mathsf{U} \varphi_2}}$$

for all j. Contradiction! Thus, $\varphi_2 \in B_j$ for some $j \geqslant 0$. Without loss of generality, assume $\varphi_2 \notin B_0, \ldots, B_{j-1}$, i.e., let j be the smallest index such that $\varphi_2 \in B_j$. The induction hypothesis for $0 \leqslant i < j$ yields

$$\varphi_1 \in B_i \quad \text{and} \quad \varphi_1 \mathsf{U} \varphi_2 \in B_i \quad \text{for all } 0 \leqslant i < j.$$

From the induction hypothesis applied to φ_1 and φ_2 it follows that

$$A_j\, A_{j+1} \ldots \models \varphi_2 \quad \text{and} \quad A_i\, A_{i+1} \ldots \models \varphi_1 \text{ for } 0 \leqslant i < j.$$

We conclude that $A_0\, A_1\, A_2 \ldots \models \varphi_1 \mathsf{U} \varphi_2$. ■

Example 5.38. Construction of a GNBA (Next Step)

Consider $\varphi = \bigcirc a$. The GNBA $\mathcal{G}_\varphi$ (see Figure 5.21) is obtained as indicated in the proof of Theorem 5.37, . The states of the automaton are the elementary sets of formulae contained in

$$\mathit{closure}(\varphi) \;=\; \{\, a, \bigcirc a, \neg a, \neg \bigcirc a \,\}.$$

The state space Q consists of the following elementary sets

$$\begin{array}{lcllcl} B_1 & = & \{\, a, \bigcirc a \,\}, & B_2 & = & \{\, a, \neg\bigcirc a \,\}, \\ B_3 & = & \{\, \neg a, \bigcirc a \,\}, & B_4 & = & \{\, \neg a, \neg\bigcirc a \,\}. \end{array}$$

The initial states of $\mathcal{G}_\varphi$ are the elementary sets $B \in Q$ with $\varphi = \bigcirc a \in B$. Thus $Q_0 = \{\, B_1, B_3 \,\}$. Let us justify some of the transitions. For state B_1, $B_1 \cap \{\, a \,\} = \{\, a \,\}$, so $\delta(B_1, \varnothing) = \varnothing$. In addition, $\delta(B_1, \{\, a \,\}) = \{\, B_1, B_2 \,\}$ since $\bigcirc a \in B_1$ and B_1 and B_2 are the only states that contain a. As $B_2 \cap \{\, a \,\} = \{\, a \,\}$, we get $\delta(B_2, \varnothing) = \varnothing$. Moreover, $\delta(B_2, \{\, a \,\}) = \{\, B_3, B_4 \,\}$. This follows from the fact that for $\neg\bigcirc\ \psi \in \mathit{closure}(\varphi)$, and any direct successor B' of B we have

$$\neq \bigcirc \psi \in B \text{ if and only if } \psi \notin B'.$$

(This follows by the definition of δ, local consistency and maximality.) Since $\neg\bigcirc\ a \in B_2$, and B_3 and B_4 are the only states that do not contain a, we have $\delta(B_2, \{\, a \,\}) = \{\, B_3, B_4 \,\}$. Hence, $\delta(B_4, \{\, a \,\}) = \varnothing$ since $B_4 \cap \{\, a \,\} = \varnothing \neq \{\, a \,\}$. Using a similar reasoning as above, we obtain $\delta(B_4, \varnothing) = \{\, B_3, B_4 \,\}$. The outgoing transitions of B_3 are determined analogously. The set $\mathcal{F}$ is empty as $\varphi = \bigcirc a$ does not contain an until operator. Since $\mathcal{F} = \varnothing$, every infinite run in the GNBA $\mathcal{G}_\varphi$ is accepting. As each infinite run is either of the form $B_1\, B_1 \ldots$, $B_1\, B_2 \ldots$, $B_3\, B_1 \ldots$, or $B_3\, B_2 \ldots$, and $\varphi = \bigcirc a \in B_1, B_2$, it follows that indeed all runs satisfy $\bigcirc a$. ■

Example 5.39. Construction of a GNBA (Until)

Consider $\varphi = a \,\mathsf{U}\, b$ and let $AP = \{\, a, b \,\}$. Then

$$\mathit{closure}(\varphi) = \{\, a, b, \neg a, \neg b,\ a \,\mathsf{U}\, b, \neg(a \,\mathsf{U}\, b) \,\}.$$

The construction in the proof of Theorem 5.37 yields the GNBA $\mathcal{G}_\varphi$ illustrated in Figure 5.22. In order not to blur the figure, transition labels have been omitted. (The label of transition $B \to B'$ equals the propositional logic formula characterising the set $B \cap AP$.) The states correspond to the elementary sets of $\mathit{closure}(\varphi)$

$$\begin{array}{lcl} B_1 & = & \{\, a, \quad b, \quad \varphi \,\}, \\ B_2 & = & \{\, \neg a, \quad b, \quad \varphi \,\}, \\ B_3 & = & \{\, a, \quad \neg b, \quad \varphi \,\}, \\ B_4 & = & \{\, \neg a, \quad \neg b, \quad \neg\varphi \,\}, \\ B_5 & = & \{\, a, \quad \neg b, \quad \neg\varphi \,\}. \end{array}$$

The initial states are the sets $B_i \in Q$ with $\varphi \in B_i$; thus, $Q_0 = \{\, B_1, B_2, B_3 \,\}$. The set $\mathcal{F} = \{\, F_\varphi \,\}$ of the accepting sets is a singleton, since φ contains a single until operator.

The set F_φ is given by

$$F_\varphi \;=\; \{\, B \in Q \mid \varphi \notin B \;\vee\; b \in B \,\} \;=\; \{\, B_1, B_2, B_4, B_5 \,\}.$$

Since the accepting set is a singleton set, the GNBA $\mathcal{G}_\varphi$ can be understood as an NBA with the accepting set F_φ.

Let us justify some of the transitions. We have $B_1 \cap AP = \{\, a, b \,\}$ and $a \,\mathsf{U}\, b \in B_1$. As:

$$b \in B_1 \vee (a \in B_1 \wedge a \,\mathsf{U}\, b \in B')$$

holds for any elementary set $B' \subseteq \mathit{closure}(a \,\mathsf{U}\, b)$, we have that $\delta(B_1, \{\, a, b \,\})$ contains all states. These are the only outgoing transitions of state B_1. A similar reasoning applies to $\delta(B_2, \{\, b \,\})$. Consider state B_3. Then $B_3 \cap AP = \{\, a \,\}$. The states B' that are possible direct successors of B_3 under $\{\, a \,\}$ are those satisfying

$$a \in B_3 \wedge a \,\mathsf{U}\, b \in B'.$$

Thus, $\delta(B_3, \{\, a \,\}) = \{\, B_1, B_2, B_3 \,\}$. Finally, consider state B_5. This state only has successors for input symbol $B_5 \cap AP = \{\, a \,\}$. For any $\varphi_1 \,\mathsf{U}\, \varphi_2 \in \mathit{closure}(\varphi)$ it can be shown that for any successor B' of B:

$$\varphi_1 \,\mathsf{U}\, \varphi_2 \notin B \quad \text{iff} \quad \varphi_2 \notin B \;\wedge\; (\varphi_1 \notin B \;\vee\; \varphi_1 \,\mathsf{U}\, \varphi_2 \notin B').$$

(The proof of this fact is left as an exercise.) Applying this to the state B_5 yields that all states not containing φ are possible successors of B_5. For example, the run $B_3\, B_3\, B_1\, B_4^\omega$ is accepting. This run corresponds to the word $\{\, a \,\}\{\, a \,\}\{\, a, b \,\}\varnothing^\omega$ which indeed satisfies $a \,\mathsf{U}\, b$. The word $\{\, a \,\}^\omega$ does not satisfy $a \,\mathsf{U}\, b$. It has exactly one run in $\mathcal{G}_\varphi$, namely B_3^ω. As B_3 is not a final state, this run is not accepting, i.e., $\{\, a \,\}^\omega \notin \mathcal{L}_\omega(\mathcal{G}_\varphi)$. ■

Remark 5.40. Simplified Representation of the Automata States

Any state of the GNBA for an LTL formula φ contains either ψ or its negation $\neg\psi$ for every subformula ψ of φ. This is somewhat redundant. It suffices to represent state $B \in \mathit{closure}(\varphi)$ by the propositional symbols $a \in B \cap AP$, and the formulae $\bigcirc\psi$ or $\varphi_1 \,\mathsf{U}\, \varphi_2 \in B$. ■

Having constructed a GNBA $\mathcal{G}_\varphi$ for a given LTL formula φ, an NBA for φ can be obtained by the transformation "GNBA $\rightsquigarrow$ NBA" described in Theorem 4.56 on page 195. Recall that this transformation for GNBA with two or more acceptance sets generates a copy of $\mathcal{G}_\varphi$ for each acceptance set of $\mathcal{G}_\varphi$. In our case, the number of copies that we need is given by the number of until subformulae of φ. We obtain the following result:

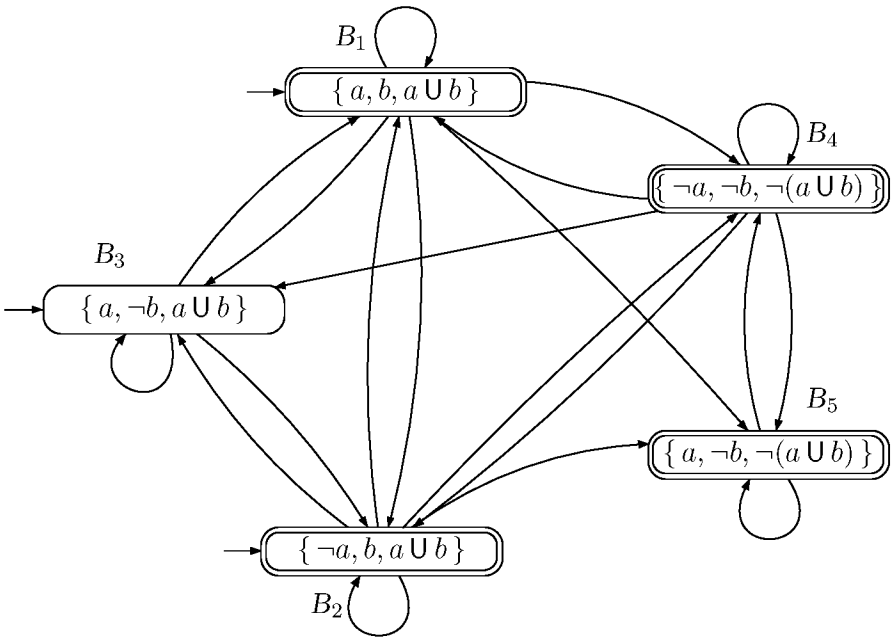


Figure 5.22: A generalised Büchi automaton for $a \mathsf{U} b$.

Theorem 5.41. Constructing an NBA for an LTL Formula

For any LTL formula φ (over AP) there exists an NBA $\mathcal{A}_\varphi$ with Words(φ) = $\mathcal{L}_\omega(\mathcal{A}_\varphi)$ which can be constructed in time and space $2^{\mathcal{O}(|\varphi|)}$.

Proof: From Theorem 5.37 (page 278), it follows that a GNBA $\mathcal{G}_\varphi$ can be constructed which has at most $2^{|\varphi|}$ states. As the number of accepting states in $\mathcal{G}_\varphi$ equals the number of until-subformulas in φ, GNBA $\mathcal{G}_\varphi$ has at most $|\varphi|$ accepting states. Transforming the GNBA into an equivalent NBA (as described in the proof of Theorem 4.56, page 195), yields an NBA with at most $|\varphi|$ copies of the state space of $\mathcal{G}_\varphi$. Thus, the number of states in the NBA is at most $2^{|\varphi|} \cdot |\varphi| = 2^{|\varphi| + \log |\varphi|}$ states. This yields the claim. ■

There are various algorithms in the literature for associating an automaton for infinite words to an LTL formula. The presented algorithm is one of the conceptually simplest algorithms, but often yields unnecessarily large GNBAs. For example, for the LTL formulae $\bigcirc a$ and $a \mathsf{U} b$, an NBA with two states suffices. (It is left to the reader to provide these NBAs.) Several optimizations are possible to improve the size of the resulting GNBA, but the exponential blowup cannot be avoided. This is formally stated in the following theorem:

Theorem 5.42. Lower Bound for NBA from LTL Formulae

There exists a family of LTL formulae φ_n with $|\varphi_n| = \mathcal{O}(poly(n))$ such that every NBA for φ_n has at least 2^n states.

Proof: Let AP be an arbitrary nonempty set of atomic propositions, that is, $|2^{AP}| \geqslant 2$. Consider the family of languages:

$$\mathcal{L}_n \;=\; \{\, A_1 \ldots A_n\, A_1 \ldots A_n\, \sigma \mid A_i \subseteq AP \wedge \sigma \in (2^{AP})^\omega \,\}, \qquad \text{for } n \geqslant 0.$$

It is not difficult to check that $\mathcal{L}_n \;=\; \mathit{Words}(\varphi_n)$ where

$$\varphi_n \;=\; \bigwedge_{a \in AP} \; \bigwedge_{0 \leqslant i < n} (\bigcirc^i a \longleftrightarrow \bigcirc^{n+i} a).$$

Here, $\bigcirc^j$ stands for the j-fold application of the next-step operator $\bigcirc$, i.e., $\bigcirc^1 \varphi = \bigcirc \varphi$ and $\bigcirc^{n+1} \varphi = \bigcirc \bigcirc^n \varphi$. It follows that φ_n is an LTL formula of polynomial length. More precisely, $|\varphi_n| \;\in\; \mathcal{O}(|AP| \cdot n)$.

However, any NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_n$ has at least 2^n states. Essentially, this is justified by the following consideration. Since the words

$$A_1 \ldots A_n\, A_1 \ldots A_n\, \varnothing\, \varnothing\, \varnothing \ldots$$

are accepted by $\mathcal{A}$, $\mathcal{A}$ contains for every word $A_1 \ldots A_n$ of length n, a state $q(A_1 \ldots A_n)$, which can be reached from an initial state by consuming the prefix $A_1 \ldots A_n$. Starting from $q(A_1 \ldots A_n)$ it is possible to visit an accept state infinitely often by accepting the suffix $A_1 \ldots A_n\, \varnothing\, \varnothing\, \varnothing \ldots$. If $A_1 \ldots A_n \;\neq\; A'_1 \ldots A'_n$ then

$$A_1 \ldots A_n\, A'_1 \ldots A'_n\, \varnothing\, \varnothing\, \varnothing \ldots \notin \mathcal{L}_n \;=\; \mathcal{L}_\omega(\mathcal{A}).$$

Therefore, the states $q(A_1 \ldots A_n)$ are all pairwise different. As there are $|2^{AP}|$ possible combinations for $A_1 \ldots A_n$, $\mathcal{A}$ has at least $(|2^{AP}|)^n \;\geqslant\; 2^n$ states. ∎

Remark 5.43. Büchi Automata are More Expressive Than LTL

The results so far show that for every LTL formula φ an NBA can be constructed that accepts exactly the infinite sequences satisfying φ. We state without proof that the reverse, however, is *not* true. It can be shown that for, e.g., the LT property

$$P \;=\; \Big\{ A_0\, A_1\, A_2 \ldots \in (2^{\{\,a\,\}})^\omega \mid a \in A_{2i} \text{ for } i \geqslant 0 \Big\},$$

which requires a to hold in every even position, there is no LTL formula φ with $\mathit{Words}(\varphi) = P$. On the other hand, there exists an NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = P$. (It is left to the reader to provide such an NBA.) ∎

5.2.1 Complexity of the LTL Model-Checking Problem

Let us summarize the results of the previous sections in order to provide an overview of LTL model checking. Subsequently, we discuss the complexity of the LTL model-checking problem.

As explained before, the essential idea behind the automata-based model-checking algorithm for LTL is based upon the following relations:

$$\begin{aligned} TS \models \varphi \quad &\text{iff} \quad \mathit{Traces}(TS) \subseteq \mathit{Words}(\varphi) \\ &\text{iff} \quad \mathit{Traces}(TS) \subseteq (2^{AP})^{\omega} \setminus \mathit{Words}(\neg\varphi) \\ &\text{iff} \quad \mathit{Traces}(TS) \cap \underbrace{\mathit{Words}(\neg\varphi)}_{\mathcal{L}_{\omega}(\mathcal{A}_{\neg\varphi})} = \varnothing \\ &\text{iff} \quad TS \otimes \mathcal{A}_{\neg\varphi} \models \Diamond\Box\, \neg F. \end{aligned}$$

Here, NBA $\mathcal{A}_{\neg\varphi}$ accepts $\mathit{Words}(\neg\varphi)$ and F is its set of accept states. The algorithm to transform an LTL formula φ into an NBA may give rise to an NBA whose state space size is exponential in the length of φ. The NBA $\mathcal{A}_{\neg\varphi}$ can thus be constructed in exponential time:

$$\mathcal{O}(2^{|\varphi|} \cdot |\varphi|) \;=\; \mathcal{O}(2^{|\varphi| + \log|\varphi|}).$$

This complexity bound, together with the fact that the state space of $\mathcal{A}$ is exponential in $|\varphi|$, yields an upper bound for the time- and space-complexity of LTL model checking (see Algorithm 11, page 272):

$$\mathcal{O}(\,|TS| \cdot 2^{|\varphi|}\,).$$

Remark 5.44. LTL Model Checking with Fairness

As a consequence of Theorem 5.30 (see page 264), the model-checking problem for LTL with fairness assumptions can be reduced to the model-checking problem for plain LTL. So, in order to check the formula φ under fairness assumption *fair*, it suffices to verify the formula $\mathit{fair} \rightarrow \varphi$ with an LTL model-checking algorithm. This approach, however, has as its main drawback that the length $|\mathit{fair}|$ can have an exponential influence on the run-time of the algorithm. This is due to the construction of an NBA for the negated formula, i.e., $\neg(\mathit{fair} \rightarrow \varphi)$, whose size is exponential in $|\neg(\mathit{fair} \rightarrow \varphi)| \;=\; |\mathit{fair}| + |\varphi|$. To avoid this additional exponential blowup, a modified persistence check (see Algorithm 8 on page 211) can be exploited to analyze the product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$ (instead of $TS \otimes \mathcal{A}_{\neg(\mathit{fair} \rightarrow \varphi)}$). This can be done using standard graph algorithms. The reader is referred to Exercise 5.22 (on page 308) for more details. ■

An interesting aspect of the LTL model-checking algorithm is that it can be executed *on-the-fly*, i.e., while constructing the NBA $\mathcal{A}_{\neg\varphi}$. This may avoid the need for constructing the entire automaton $\mathcal{A}_{\neg\varphi}$. This on-the-fly procedure works as follows. Suppose we are given a high-level description of the transition system TS, e.g., by means of a syntactic description of the concurrent processes (as in SPIN's input language PROMELA). The generation of the reachable states of TS can proceed in parallel with the construction of the relevant fragment of $\mathcal{A}_{\neg\varphi}$. Simultaneously, the reachable fragment of the product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$ is constructed in a DFS-manner. (This, in fact, yields the outermost DFS in the nested DFS for checking persistence in $TS \otimes \mathcal{A}_{\neg\varphi}$.) So the entire LTL model-checking procedure can be interleaved with the generation of the relevant fragments of TS and $\mathcal{A}_{\neg\varphi}$. In this way, the product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$ is constructed "on demand", so to speak. A new vertex is only considered if no accepting cycle has been encountered yet in the partially constructed product transition system $TS \otimes \mathcal{A}_{\neg\varphi}$. When generating the successors of a state in $\mathcal{A}_{\neg\varphi}$, it suffices to only consider the successors matching the current state TS (rather than all possible successors). It is thus possible that an accepting cycle is found, i.e., a violation of φ (with corresponding counterexample), without the need for generating the entire automaton $\mathcal{A}_{\neg\varphi}$.

This on-the-fly generation of $Reach(TS)$, $\mathcal{A}_{\neg\varphi}$, and $TS \otimes \mathcal{A}_{\neg\varphi}$ is adopted in practical LTL model checkers (such as SPIN) and for many examples yields an efficient verification procedure. From a theoretical point of view, though, the LTL model-checking problem remains computationally hard and is "probably" not efficiently solvable. It is shown in the sequel of this section that the LTL model-checking problem is PSPACE-complete. We assume some familiarity with basic notions of complexity theory and the complexity classes coNP and PSPACE; see, e.g., the textbooks [160, 320] and the Appendix.

Before proving the PSPACE-hardness of the LTL model-checking problem, a weaker result is provided. To that end, let us recall the so-called *Hamiltonian path problem*. Consider a finite directed graph $G = (V, E)$ with set V of vertices and set $E \subseteq V \times V$ of edges. The Hamiltonian path problem is a decision problem that yields an affirmative answer when G has a Hamiltonian path, i.e., a path which passes through every vertex in V exactly once.

The next ingredient needed for the following result is the *complement* of the LTL model-checking problem. This decision problem takes as input a finite transition system TS and an LTL formula φ and asks whether $TS \not\models \varphi$. Thus, whenever the LTL model-checking problem provides an affirmative answer, its complement yields "no", and whenever the result of the LTL model-checking problem is negative, its complement yields "yes".

Lemma 5.45.

The Hamiltonian path problem is polynomially reducible to the complement of the LTL model-checking problem.

Proof: To establish a polynomial reduction from the Hamiltonian path problem to the complement of the LTL model-checking problem, a mapping is needed from instances of the Hamiltonian path problem onto instances for the complement model-checking problem. That is, we need to map a finite directed graph G onto a pair (TS, φ) where TS is a finite transition system and φ is an LTL formula, such that

(i) G has a Hamiltonian path if and only if $TS \not\models \varphi$, and

(ii) the transformation $G \rightsquigarrow (TS, \varphi)$ can be performed in polynomial time.

The basic concept of the mapping is as follows. Essentially, TS corresponds to G, i.e., the states of TS are the vertices in G and the transitions in TS correspond to the edges in G. For technical reasons, G is slightly modified to ensure that TS has no terminal states. More precisely, the state graph of TS arises from G by inserting a new vertex b to G, which is reachable from every vertex v via an edge and which is only equipped with a self-loop $b \to b$ (i.e., b has no further outgoing edges). Formally, for $G = (V, E)$ the associated transition system is

$$TS = (V \uplus \{ b \}, \{ \tau \}, \to, V, V, L)$$

where $L(v) = \{ v \}$ for any vertex $v \in V$ and $L(b) = \varnothing$. The atomic propositions for TS are thus obtained by taking the identities of the vertices in G. The transition relation $\to$ is defined as follows:

$$\frac{(v, w) \in E}{v \xrightarrow{\tau} w} \quad \text{and} \quad \frac{v \in V \cup \{ b \}}{v \xrightarrow{\tau} b}.$$

This completes the mapping of directed graph G onto transition system TS. It remains to formalize the negation of the existence (i.e., the absence) of a Hamiltonian path by means of an LTL formula. Let

$$\varphi \;=\; \neg \bigwedge_{v \in V} \big(\Diamond v \;\wedge\; \Box(v \to \bigcirc \Box \neg v) \big).$$

Stated in words, φ asserts that it is not the case that each vertex $v \in V$ is eventually visited and never visited again. Note that the conjunction does not quantify over $b \notin V$.

It is evident that TS and φ can be constructed in polynomial time for a given directed graph G. As a last step, we need to show that G has a Hamiltonian path if and only if $TS \not\models \varphi$.

$\Leftarrow$: Assume $TS \not\models \varphi$. Then there exists a path π in TS such that

$$\pi \;\models\; \bigwedge_{v \in V} \big(\Diamond v \;\wedge\; \Box(v \to \bigcirc \Box \neg v) \big).$$

As $\pi \models \bigwedge_{v \in V} \Diamond v$, each vertex $v \in V$ occurs at least once in the path π. Since

$$\pi \models \bigwedge_{v \in V} \Box(v \rightarrow \bigcirc \Box \neg v),$$

there is no vertex that occurs more than once in π. Thus, π is of the form $v_1 \ldots v_n b\, b\, b \ldots$ where $V = \{ v_1, \ldots, v_n \}$ and $|V| = n$. In particular, $v_1 \ldots v_n$ is a Hamiltonian path in G.

$\Rightarrow$: Each Hamiltonian path $v_1 \ldots v_n$ in G can be extended to a path $\pi = v_1 \ldots v_n b\, b\, b \ldots$ in TS. Thus, $\pi \not\models \varphi$ and, hence, $TS \not\models \varphi$. ■

Since the Hamiltonian path problem is known to be NP-complete, it follows from the previous lemma that the LTL model-checking problem is coNP-hard. The following complexity result states that the LTL model-checking problem is PSPACE-hard.

Theorem 5.46. Lower Bound for LTL Model Checking

The LTL model-checking problem is PSPACE*–hard.*

Proof: Let TS be a finite transition system and φ an LTL formula. As a first observation, note that it suffices to show the PSPACE-hardness of the *existential* variant of the LTL model-checking problem. This existential decision problem takes as input TS and φ and yields "yes" if $\pi \models \varphi$ for *some* (initial, infinite) path π in TS, and "no" otherwise. Given that

$$\begin{aligned} TS \models \varphi \quad & \text{if and only if} \quad \pi \models \varphi \text{ for all paths } \pi \\ & \text{if and only if} \quad \text{not } (\pi \models \neg\varphi \text{ for } \textit{some} \text{ path } \pi) \end{aligned}$$

it follows that the output for the instance (TS, φ) of the LTL model-checking problem yields "yes" if and only if the output for the instance $(TS, \neg\varphi)$ of the existential variant of the LTL model-checking problem yields "no". Thus, the PSPACE-hardness of the existential variant of the LTL model-checking problem yields the PSPACE-hardness of the complement of the existential variant of the LTL model checking problem which again induces the PSPACE-hardness of the LTL model-checking problem. Recall that PSPACE = coPSPACE, thus, the complement of any PSPACE-hard problem is PSPACE-hard.

In the remainder of the proof we concentrate on showing the PSPACE–hardness of the existential LTL model-checking problem. This is established by providing a polynomial reduction from any decision problem $K \in$ PSPACE to the existential LTL model-checking problem. Let $\mathcal{M}$ be a polynomial space-bounded deterministic Turing machine that accepts exactly the words $w \in K$. The goal is now to transform $(\mathcal{M}, w)$ by a deterministic

polynomial-time bounded procedure into a pair (TS, φ) such that $\mathcal{M}$ accepts w if and only if TS contains a path π with $\pi \models \varphi$.

The following proof, adopted from [372], has some similarities with Cook's theorem stating the NP-completeness of the satisfiability problem for propositional logic. The rough idea is to encode the initial configurations, possible transitions, and accepting configurations of a given Turing machine $\mathcal{M}$ by LTL formulae. In the sequel, let $\mathcal{M}$ be a deterministic single-tape Turing machine with state-space Q, starting state $q_0 \in Q$, the set F of accept states, the tape alphabet Σ, and the transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{ L, R, N \}$. The intuitive meaning of δ is as follows. Suppose $\delta(q, A) = (p, B, L)$. Then, whenever the current state is q and the current symbol in cell i under the cursor is A, then $\mathcal{M}$ changes to state p, overwrites the symbol A by B in cell i, and moves the cursor one position to the left, i.e., it positions the cursor under cell $i{-}1$. For R or N, the cursor moves one position to the right, or does not move at all, respectively. (For our purposes, there is no need to distinguish between the input and the tape alphabet.)

For technical reasons, it is required that the accept states are absorbing, i.e., $\delta(q, A) = (q, A, N)$ for all $q \in F$. Moreover, it is assumed that $\mathcal{M}$ is *polynomially space-bounded*, i.e., there is a polynomial P such that the computation for an input word $A_1 \ldots A_n \in \Sigma^*$ of length n visits at most the first $P(n)$ cells on the tape. Without loss of generality, the coefficients of P are assumed to be natural numbers and $P(n) \geqslant n$.

To encode $\mathcal{M}$'s possible behaviors by LTL formulae for an input word of length n, the Turing machine is transformed into a transition system $TS = TS(\mathcal{M}, n)$ with the following state space:

$$S \;=\; \{0, 1, \ldots, P(n)\} \cup \{(q, A, i) \mid q \in Q \cup \{ * \},\ A \in \Sigma,\ 0 < i \leqslant P(n)\}.$$

The structure of TS is shown in Figure 5.23. TS consists of $P(n)$ copies of "diamonds". The ith copy starts in state $i{-}1$, ends in state i, and contains the states (q, A, i) where $q \in Q \cup \{ * \}$ and $A \in \Sigma$ "between" state $i{-}1$ and state i. Intuitively, the first component $q \in Q \cup \{ * \}$ of state (q, A, i) in the ith copy indicates whether the cursor points to cell i (in which case $q \in Q$ is the current state) or to some other tape cell (in which case $q = *$). The symbol $A \in \Sigma$ in state (q, A, i) stands for the current symbol in cell i. The path fragments from state 0 to state $P(n)$ serve to represent the possible configurations of $\mathcal{M}$. More precisely, the configuration in which the current content of the tape is $A_1\, A_2 \ldots A_{P(n)}$, the current state is q, and the cursor points to cell i is encoded by the path fragment:

$$0\,(*, A_1, 1)\,1\,(*, A_2, 2)\,2\;\ldots\;i{-}1\,(q, A_i, i)\,i\,(*, A_{i+1}, i{+}1)\,i{+}1\;\ldots\;P(n)$$

Accordingly, the computation of $\mathcal{M}$ for an input word of length n can be described by the concatenation of such path fragments.

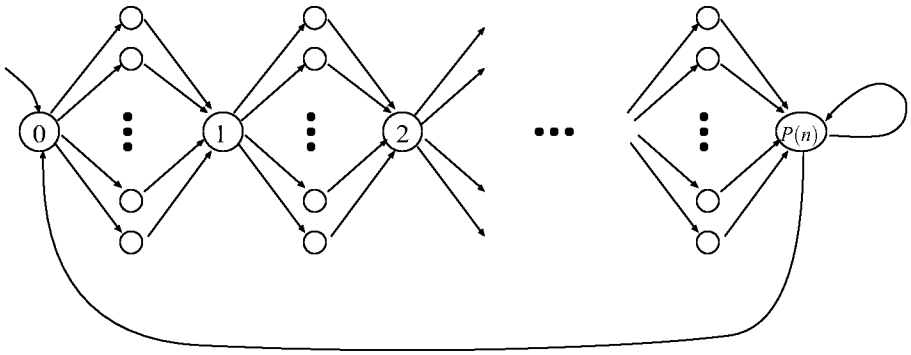


Figure 5.23: Transition system $TS(\mathcal{M}, n)$ for Turing machine $\mathcal{M}$ and input length n.

We use the state identities as atomic propositions. In addition, proposition *begin* is used to identify state 0, while proposition *end* is used for state $P(n)$. That is, $AP = S \cup \{ \mathit{begin}, \mathit{end} \}$ with the obvious labeling function. Let Φ_Q denote the disjunction over all atoms (q, A, i) where $q \in Q$ (i.e., $q \neq *$) and $A \in \Sigma$, $0 < i \leqslant P(n)$. The LTL formulae:

$$\varphi_{Conf} = \Box\Big(\mathit{begin} \longrightarrow \varphi^1_{Conf} \wedge \varphi^2_{Conf}\Big) \quad \text{where}$$

$$\varphi^1_{Conf} = \bigvee_{1 \leqslant i \leqslant P(n)} \bigcirc^{2i-1} \Phi_Q$$

$$\varphi^2_{Conf} = \bigwedge_{1 \leqslant i \leqslant P(n)} \Big(\bigcirc^{2i-1} \Phi_Q \longrightarrow \bigwedge_{\substack{1 \leqslant j \leqslant P(n) \\ j \neq i}} \bigcirc^{2j-1} \neg \Phi_Q \Big)$$

characterize any path π in TS such that all path fragments of π that lead from 0 via $1, \ldots, P(n-1)$ to $P(n)$ encode a configuration of $\mathcal{M}$. Note that $\varphi^1_{Conf} \wedge \varphi^2_{Conf}$ ensures that the cursor points exactly to one of the positions $1, \ldots, P(n)$. Thus, any path π in TS such that $\pi \models \varphi_{Conf}$ can be viewed as a sequence of configurations in $\mathcal{M}$. However, this sequence need not be a computation of $\mathcal{M}$, since the configurations might not be consecutive according to $\mathcal{M}$'s operational behavior. For this, we need additional constraints that formalize $\mathcal{M}$'s stepwise behavior.

The transition relation δ of $\mathcal{M}$ can be encoded by an LTL formula φ_δ that arises through a conjunction of formulae $\varphi_{q,A}$ describing the semantics of $\delta(q, A)$. Here, q ranges over all states in Q and A over the tape-symbols in Σ. For instance, if $\delta(q, A) = (p, B, L)$ then

$$\varphi_{q,A} = \Box \bigwedge_{1 \leqslant i \leqslant P(n)} \Big(\bigcirc^{2i-1} (q, A, i) \longrightarrow \psi_{(q,A,i,p,B,L)} \Big)$$

where $\psi_{(q,A,i,p,B,L)}$ is defined as

$$\bigwedge_{\substack{1\leqslant j\leqslant P(n)\\ j\neq i, C\in\Sigma}} \underbrace{(\bigcirc^{2j-1} C \leftrightarrow \bigcirc^{2j-1+2P(n)+1} C)}_{\text{content of all cells } j\neq i \text{ unchanged}} \wedge \underbrace{\bigcirc^{2i-1+2P(n)+1} B}_{\text{overwrite } A \text{ by } B \text{ in cell } i} \wedge \underbrace{\bigcirc^{2i-1+2P(n)+1-2} p}_{\substack{\text{move to state } p\\ \text{and cursor to cell } i-1}}.$$

Here, C denotes the disjunction of the atoms (r, C, j) where $r \in Q \cup \{*\}$ and $1 \leqslant j \leqslant P(n)$, and p for the disjunction of all atoms (p, D, j) where $D \in \Sigma$ and $1 \leqslant j \leqslant P(n)$.

The starting configuration of $\mathcal{M}$ for a given input word $w = A_1 \ldots A_n \in \Sigma^*$ is given by the formula

$$\varphi^w_{start} \;=\; \bigcirc q_0 \;\wedge \bigwedge_{1\leqslant i\leqslant n} \bigcirc^{2i-1} A_i \;\wedge \bigwedge_{n<i\leqslant P(n)} \bigcirc^{2i-1} \sqcup .$$

The first conjunct $\bigcirc q_0$ asserts that $\mathcal{M}$ starts in its starting state; the other conjuncts assert that $A_1 \ldots A_n \sqcup^{P(n)-n}$ is the content of the tape where $\sqcup$ denotes the blank symbol. The accepting configurations are formalized by the formula

$$\varphi_{acccpt} = \Diamond \bigvee_{q\in F} q.$$

For a given input word $w = A_1 \ldots A_n$ of length n for $\mathcal{M}$, let

$$\varphi_w \;=\; \varphi^w_{start} \;\wedge\; \varphi_{Conf} \;\wedge\; \varphi_\delta \;\wedge\; \varphi_{acccpt}.$$

Note that the length of φ_w is polynomial in the size of $\mathcal{M}$ and the length n of w. Thus, $TS = TS(\mathcal{M}, n)$ and φ_w can be constructed from $(\mathcal{M}, w)$ in polynomial time. Moreover, it also follows that there exists a path $\pi \models \varphi_w$ in TS if and only if $\mathcal{M}$ accepts the input word w. ■

For real applications, the aforementioned theoretical complexity result is less dramatic than it seems at first sight, since the complexity is *linear* in the size of the transition system and exponential in formula length. In practice, typical requirement specifications yield short LTL formulae. In fact, the exponential growth in the length of the formula is not decisive for the practical application of LTL model checking. Instead, the linear dependency on the size of the transition system is the critical factor. As has been discussed at the end of Chapter 2 (page 77), the size of transition systems may be huge even for relatively simple systems—the state-space explosion problem. In later chapters of this monograph, various techniques will be treated to combat this state-space explosion problem.

We mentioned before that the LTL model-checking problem is PSPACE-complete. The previous result showed PSPACE-hardness. It remains to show that it belongs to the complexity class PSPACE.

To prove that the LTL model-checking problem is in PSPACE, we resort (again) to the existential variant of the LTL model-checking problem. This is justified by the fact that PSPACE – as any other deterministic complexity class – is closed under complementation. That is, the LTL model-checking problem is in PSPACE iff its complement is in PSPACE. This is equivalent to the statement that the existential variant of the LTL model-checking problem is solvable by a polynomial space-bounded algorithm. By Savitch's theorem (PSPACE agrees with NPSPACE), it suffices to provide a *nondeterministic polynomial space-bounded* algorithm that solves the existential LTL model-checking problem.

Lemma 5.47.

The existential LTL model-checking problem is solvable by a nondeterministic space-bounded algorithm.

Proof: In the sequel, let φ be an LTL formula and $TS = (S, Act, \rightarrow, I, AP, L)$ a finite transition system. The goal is to check nondeterministically whether TS has a path π with $\pi \models \varphi$, while the memory requirements are bounded by $\mathcal{O}(\,\mathrm{poly}(\mathit{size}(TS), |\varphi|)\,)$. The techniques discussed in Section 5.2 (page 270 ff) suggest to build an NBA $\mathcal{A}_\varphi$ for φ, construct the product transition system $TS \otimes \mathcal{A}_\varphi$ and check whether this product contains a reachable cycle containing an accept state of $\mathcal{A}_\varphi$. We now modify this approach to obtain an NPSPACE algorithm. Instead of an NBA for φ, we deal here with the GNBA $\mathcal{G}_\varphi$ for φ. Recall that states in this automaton are elementary subsets of the closure of φ (see Definition 5.34 on page 276). The goal is to guess nondeterministically a finite path $u_0\, u_1\, \ldots\, u_{n-1}\, v_0\, v_1\, \ldots\, v_{m-1}$ in $TS \otimes \mathcal{G}_\varphi$ and to check whether the components of $\mathcal{G}_\varphi$ in the infinite path

$$u_0\, u_1\, \ldots\, u_{n-1}\, (v_0\, v_1\, \ldots\, v_{m-1})^\omega$$

constitute an accepting run in $\mathcal{G}_\varphi$. This, of course, requires that v_0 is a successor of v_{m-1}. The states u_i, v_j in the infinite path in $TS \otimes \mathcal{G}_\varphi$ are pairs consisting of a state in TS and an elementary set of formulae. For the lengths of the prefix $u_0 \ldots u_{n-1}$ and the cycle $v_0\, v_1 \ldots v_{m-1}\, v_m$ we can deal with $n \leqslant k$ and $m \leqslant k \cdot |\varphi|$ where k is the number of reachable states in $TS \otimes \mathcal{G}_\varphi$. (Note that $|\varphi|$ is an upper bound for the number of acceptance sets in $\mathcal{G}_varphi$.) An upper bound for the value k is given by

$$K \;=\; N_{TS} \cdot 2^{N_\varphi}$$

where N_{TS} denotes the number of states in TS and $N_\varphi = |\mathit{closure}(\varphi)|$. Note that

$$K \;=\; \mathcal{O}(\,\mathit{size}(TS) \cdot \exp(|\varphi|)\,).$$

The algorithm now works as follows. We first nondeterministically choose two natural numbers n, m with $n \leqslant K$ and $m \leqslant K \cdot |\varphi|$ (by guessing $\lceil \log K \rceil = \mathcal{O}(\log(\mathit{size}(TS)) \cdot |\varphi|)$

bits for n and $\lceil \log K \rceil + \lceil \log |\varphi| \rceil = \mathcal{O}(\log(size(TS)) \cdot |\varphi|)$ bits for m). Then the algorithm guesses nondeterministically a sequence $u_0 \ldots u_{n-1}$, $u_n \ldots u_{n+m}$ where the u_i's are pairs $\langle s_i, B_i \rangle$ consisting of a state s_i in TS and a subset B_i of $closure(\varphi)$. For each such state $u_i = \langle s_i, B_i \rangle$, the algorithm checks whether

1. s_i is a successor of s_{i-1}, provided that $i \geqslant 1$,
2. B_i is elementary,
3. $B_i \cap AP = L(s_i)$,
4. $B_i \in \delta(B_{i-1}, L(s_i))$, provided that $i \geqslant 1$.

For $i = 0$, the algorithm checks whether $s_0 \in I$ is an initial state of TS and whether $B_0 \in \delta(B, L(s_0))$ for some elementary set B which contains φ. Here, δ denotes the transition relation of the GNBA $\mathcal{G}_\varphi$. (Recall that the sets B where $\varphi \in B$ are the initial states in the GNBA for φ.) Conditions 1-4 are local and simple to check. If one of these four conditions is violated for some i, the algorithm rejects and halts. Otherwise $u_0 \ldots u_{n+m}$ is a finite path in $TS \otimes \mathcal{G}_\varphi$. We finally check whether u_n agrees with the last state u_{n+m}. Again, the algorithm rejects and halts if this condition does not hold. Otherwise, $u_0 \ldots u_{n-1}(u_n \ldots u_{n+m-1})^\omega$ is an infinite path in $TS \otimes \mathcal{G}_\varphi$, and it finally amounts to check whether the acceptance condition of $\mathcal{G}_\varphi$ is fulfilled. This means we have to verify that whenever $\psi_1 \,\mathsf{U}\, \psi_2 \in B_i$ for some $i \in \{n, \ldots, n+m-1\}$, then there is some $j \in \{n, \ldots, n+m-1\}$ such that $\psi_2 \in B_j$. If this condition holds, then the algorithm terminates with the answer "yes".

This algorithm is correct since if TS has a path where φ holds, then there is a computation of the above algorithm that returns "yes". Otherwise, i.e., if TS does not have a path where φ holds, then all computations of the algorithm are rejecting.

It remains to explain how the sketched algorithm can be realized such that the memory requirements are polynomial in the size of TS and length of φ. Although the length $n+m$ of the path $u_0 \ldots u_{n+m}$ might be exponentially in the length of φ (note that, e.g., $n = K$ is possible and that K grows exponentially in the length of φ), this procedure can be realized with only polynomial space requirements. This is due to the observation that for checking the above conditions 1-4 for $u_i = \langle s_i, B_i \rangle$, we only need state $u_{i-1} = \langle s_{i-1}, B_{i-1} \rangle$. Thus, there is no need to store all states u_j for $0 \leqslant j \leqslant n+m$. Instead, the actual and previous states are sufficient. Moreover, to verify that $\mathcal{G}_\varphi$'s acceptance condition holds, we only need to remember the subformulae $\psi_1 \,\mathsf{U}\, \psi_2$ of φ that are contained in some of the sets $B_n, \ldots B_{n+m-1}$, and the subformulae ψ_2 that appear on the right hand side of an until subformula of φ and are contained in some of the sets $B_n, \ldots B_{n+m-1}$. This additional

information requires $\mathcal{O}(|\varphi|)$ space. Thus, the above nondeterministic algorithm can be realized in a polynomially space-bounded way. ■

By Lemma 5.47 and Theorem 5.46 we get:

Theorem 5.48.

The LTL model-checking problem is PSPACE*-complete.*

5.2.2 LTL Satisfiability and Validity Checking

The last section of this chapter considers the satisfiability problem and the validity problem for LTL. The satisfiability problem is: for a given LTL formula φ, does there exist a model for which φ holds? That is, do we have $\mathit{Words}(\varphi) \neq \varnothing$? Satisfiability can be solved by constructing an NBA $\mathcal{A}_\varphi$ for LTL formula φ. In this way, the existence of an infinite word $\sigma \in \mathit{Words}(\varphi) = \mathcal{L}_\omega(\mathcal{A}_\varphi)$ can be established. The emptiness problem for NBA $\mathcal{A}$, i.e., whether $\mathcal{L}_\omega(\mathcal{A}) = \varnothing$ or not, can be solved by means of a technique similar to persistence checking, see Algorithm 12. In addition to an affirmative response, a prefix of a word $\sigma \in \mathcal{L}_\omega(\mathcal{A}) = \mathit{Words}(\varphi)$ can be provided similar to a counterexample for model checking LTL.

Algorithm 12 Satisfiability checking for LTL

Input: LTL formula φ over AP
Output: "yes" if φ is satisfiable. Otherwise "no".

Construct an NBA $\mathcal{A} = (Q, 2^{AP}, \delta, Q_0, F)$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathit{Words}(\varphi)$

(* Check whether $\mathcal{L}_\omega(\mathcal{A}) = \varnothing$. *)

Perform a nested DFS to determine whether there exists a state $q \in F$ reachable from $q_0 \in Q_0$ and that lies on a cycle

If so, then return "yes". Otherwise, "no".

Given that satisfiability for LTL can be tackled using similar techniques as for model checking LTL, let us now consider the validity problem. Formula φ is *valid* whenever φ holds under all interpretations, i.e., $\varphi \equiv \text{true}$. For LTL formula φ over AP we have φ is valid if and only if $\mathit{Words}(\varphi) = (2^{AP})^\omega$. The validity of φ can be established by using the

observation that φ is valid if and only if $\neg\varphi$ is not satisfiable. Hence, to algorithmically check whether φ is obtained, one constructs an NBA for $\neg\varphi$ and applies the satisfiablity algorithm (see Algorithm 12) to $\neg\varphi$.

The outlined LTL satisfiability algorithm has a runtime that is exponential in the length of φ. The following result shows that an essentially more efficient technique cannot be achieved as both the validity and satisfiability problems are PSPACE-hard. In fact, both problems are even PSPACE-complete. Membership to PSPACE for the LTL satisfiability problem can be shown by providing a nondeterministic polynomially space-bounded algorithm that guesses a finite run in the GNBA $\mathcal{G}_\varphi$ for the given formula φ and checks whether this finite run is a prefix of an accepting run in $\mathcal{G}_\varphi$. The details are skipped here since they are very similar to the algorithm we provided for the existential LTL model checking problem (see Lemma 5.47 on page 294). The fact that the LTL validity problem belongs to PSPACE can be derived from the observations that φ is valid iff $\neg\varphi$ is not satisfiable and that PSPACE is closed under complementation. We now focus on the proof for the PSPACE-hardness.

Theorem 5.49. LTL Satisfiability and Validity (Lower Bound)

The satisfiability and validity problems for LTL are PSPACE*-hard.*

Proof: Since satisfiability and validity are complementary in the sense that φ is satisfiable if and only if $\neg\varphi$ is not valid, and vice versa, it suffices to show the PSPACE-hardness of the satisfiability problem. This is done by providing a polynomial reduction from the existential variant of the LTL model-checking problem (see the proof of Theorem 5.46 on page 290) to the satisfiability problem for LTL.

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite transition system and φ an LTL formula over AP. The goal is to construct an LTL formula ψ such that ψ is satisfiable if and only if there is a path π in TS with $\pi \models \varphi$. Besides, ψ should be constructed in polynomial time.

The atomic propositions in ψ are elements in $AP' = AP \uplus S$. For any state $s \in S$ let

$$\Phi_s = \bigwedge_{a \in L(s)} a \ \wedge \bigwedge_{a \notin L(s)} \neg a.$$

The formula Φ_s can be viewed as a characteristic formula for the labeling of s, since $s' \models \Phi_s$ if and only if $L(s) = L(s')$, for any $s' \in S$. However, for the LTL satisfiability problem, there is no fixed transition system and Φ_s can hold also for other states (in another transition system). For $s \in S$ and $T \subseteq S$, let $\Psi_T = \bigvee_{t \in T} t$ be the characteristic formula for the set T. Let

$$\psi_s \ = \ s \rightarrow (\Phi_s \ \wedge \ \bigcirc \ \Psi_{Post(s)})$$

assert that in state s, the labels $L(s)$ hold, and that transitions exist to any of its immediate successors $Post(s)$. Let

$$\Xi \;=\; \bigvee_{s \in S} (s \;\wedge \bigwedge_{t \in S \setminus \{\, s \,\}} \neg t).$$

Stated in words, Ξ asserts that exactly one of the atomic propositions $s \in AP'$ holds. These definitions constitute the ingredients for the definition of ψ. For set I of initial states, let

$$\psi \;=\; \Psi_I \;\wedge\; \Box\,\Xi \;\wedge\; \Box\,\Psi_S \;\wedge \bigwedge_{s \in S} \Box \psi_s \;\wedge\; \varphi.$$

It follows directly that ψ can be derived from TS and φ in polynomial time. It remains to show that

$$\exists \pi \in Paths(TS).\, \pi \models \varphi \quad \text{if and only if} \quad \psi \text{ is satisfiable.}$$

$\Rightarrow$: Let $\pi = s_0\, s_1\, s_2 \ldots$ be an initial, infinite path in TS with $\pi \models \varphi$. Now consider π as a path in the transition system TS' that agrees with TS but uses the extended labeling function $L'(s) = L(s) \cup \{\, s \,\}$. Then, $\pi \models \Psi_I$, since $s_0 \in I$. In addition, $\pi \models \Box\Xi$ and $\pi \models \Box\psi_s$ since Ξ and ψ_s hold in all states of TS'. Thus, $\pi \models \psi$. Hence, π is a witness for the satisfiability of ψ.

$\Leftarrow$: Assume ψ is satisfiable. Let $A_0\, A_1\, A_2 \ldots$ be an infinite word over the alphabet $2^{AP'}$ with $A_0\, A_1\, A_2 \ldots \in Words(\psi)$. Since

$$A_0\, A_1\, A_2 \ldots \models \Box\Xi$$

there is a unique state sequence $\pi = s_0\, s_1\, s_2 \ldots$ in TS with $s_i \in A_i$ for all $i \geqslant 0$. Since $A_0\, A_1\, A_2 \ldots \models \Psi_I$, we get $s_0 \in I$. As

$$A_0\, A_1\, A_2 \ldots \models \Box \bigwedge_{s \in S} \psi_s,$$

we have $A_i \cap AP = L(s_i)$ and $s_{i+1} \in Post(s_i)$ for all $i \geqslant 0$. This yields that π is a path in TS and $\pi \models \varphi$. ■

5.3 Summary

- Linear Temporal Logic (LTL) is a logic for formalizing path-based properties.
- LTL formulae can be transformed algorithmically into nondeterministic Büchi automata (NBA). This transformation can cause an exponential blowup.

- The presented algorithm for the construction of an NBA for a given LTL formula φ relies on first constructing a GNBA for φ, which is then transformed into an equivalent NBA.
- The GNBA for φ encodes the semantics of propositional logic and the semantics of the next-step operator in its transitions. Based on the expansion law, the meaning of until is split into local requirements (encoded by the states of a GNBA), next-step requirements (encoded by the transitions of the GNBA), and a fairness condition (encoded by the acceptance sets of the GNBA).
- LTL formulae describe ω-regular LT properties, but do not have the same expressiveness as ω-regular languages.
- The LTL model-checking problem can be solved by a nested depth-first search in the product of the given transition system and an NBA for the negated formula.
- The time complexity of the automata-based model-checking algorithm for LTL is linear in the size of the transition system and exponential in the length of the formula.
- Fairness assumptions can be described by LTL formulae. The model-checking problem for LTL with fairness assumptions is reducible to the standard LTL model-checking problem.
- The LTL model-checking problem is PSPACE-complete.
- Satisfiability and validity of LTL formulae can be solved via checking emptiness of nondeterministic Büchi automata. The emptiness check can be determined by a nested DFS that checks the existence of a reachable cycle containing an accept state. Both problems are PSPACE-complete.

5.4 Bibliographic Notes

Linear temporal logic. Based on prior work on modal logics and temporal modalities [270, 345, 244, 230], Pnueli introduced (linear) temporal logics for reasoning about reactive systems in the late seventies in his seminal paper [337]. Since then, a variety of variants and extensions of LTL have been investigated, such as Lamport's Temporal Logic of Actions (TLA) [260] and LTL with past operators [159, 274, 262]. LTL forms the basis for the recently standardized industrial property specification language PSL [136]. The past extension of LTL does not change the expressiveness of LTL, but can be helpful for specification convenience and modular reasoning. For several properties, the use of past operators may lead to (exponentially) more succinct formulae than in ordinary LTL. To

cover the full class of ω-regular LT properties, Vardi and Wolper introduced an extension of LTL by automata formulae [424, 425, 411, 412].

LTL model checking. Vardi and Wolper also developed the automata-based model checking algorithm for LTL presented in this chapter. The presented algorithm to construct an NBA from a given LTL formula is, in our opinion, the simplest and most intuitive one. In the meantime, various alternative techniques have been developed that generate more compact NBAs or that attempt to minimize a given NBA, see e.g. [166, 110, 148, 375, 162, 149, 167, 157, 389, 369]. Alternative LTL model-checking algorithms that do not use Büchi automata, but a so-called tableau for the LTL formula, were presented by Lichtenstein and Pnueli [273] and Clarke, Grumberg, and Hamaguchi [88]. The results about the complexity of LTL model checking and the satisfiability problem are due to Sistla and Clarke [372].

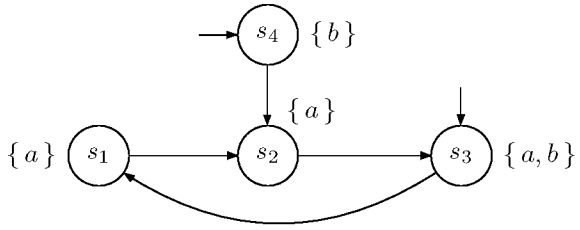
There is a variety of surveys and textbooks; see, e.g.,[245, 138, 173, 283, 158, 284, 92, 219, 379, 365], where several other aspects of LTL and related logics, such as deductive proof systems, alternative model-checking algorithms, or more details about the expressiveness, are treated.

Examples. The garbage collection algorithm presented in Example 5.31 is due to Ben-Ari [41]. Several leader election protocols that fit into the shape of Example 5.13 have been suggested; see, e.g., [280].

LTL model checkers. SPIN is the most well-known LTL model checker and has been developed by Holzmann [209]. Transition systems are described in the modeling language Promela, and LTL formulae are checked using the algorithm advocated by Gerth et al. [166]. LTL model checking using a tableau construction is supported by NuSMV [83].

5.5 Exercises

EXERCISE 5.1. Consider the following transition system over the set of atomic propositions $\{a, b\}$:



Indicate for each of the following LTL formulae the set of states for which these formulae are

fulfilled:

(a) $\bigcirc\, a$

(b) $\bigcirc\, \bigcirc\, \bigcirc\, a$

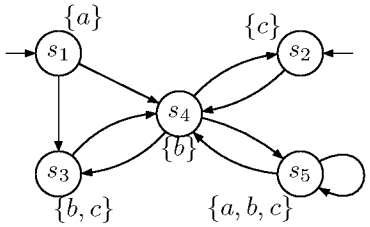
(c) $\Box\, b$

(d) $\Box \Diamond\, a$

(e) $\Box\, (b \,\mathsf{U}\, a)$

(f) $\Diamond\, (a \,\mathsf{U}\, b)$

EXERCISE 5.2. Consider the transition system TS over the set of atomic propositions $AP = \{\, a, b, c \,\}$:



Decide for each of the LTL formulae φ_i below, whether $TS \models \varphi_i$ holds. Justify your answers! If $TS \not\models \varphi_i$, provide a path $\pi \in Paths(TS)$ such that $\pi \not\models \varphi_i$.

$$
\begin{aligned}
\varphi_1 &= \Diamond \Box\, c \\
\varphi_2 &= \Box \Diamond\, c \\
\varphi_3 &= \bigcirc \neg c \rightarrow \bigcirc\, \bigcirc\, c \\
\varphi_4 &= \Box\, a \\
\varphi_5 &= a \,\mathsf{U}\, \Box\, (b \vee c) \\
\varphi_6 &= (\bigcirc\, \bigcirc\, b) \,\mathsf{U}\, (b \vee c)
\end{aligned}
$$

EXERCISE 5.3. Consider the sequential circuit in Figure 5.24 and let $AP = \{\, x, y, r_1, r_2 \,\}$. Provide LTL formulae for the following properties:

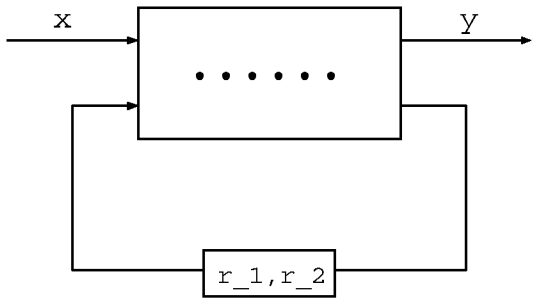


Figure 5.24: Circuit for Exercise 5.3.

(a) "It is impossible that the circuit outputs two successive 1s."

(b) "Whenever the input bit is 1, in at most two steps the output bit will be 1."

(c) "Whenever the input bit is 1, the register bits do not change in the next step."

(d) "Register r_1 has infinitely often the value 1."

Determine which of these properties are satisfied for the initial register evaluation where $r_1 = 0$ and $r_2 = 0$? Justify your answers.

EXERCISE 5.4. Suppose we have two users, *Peter* and *Betsy*, and a single printer device *Printer*. Both users perform several tasks, and every now and then they want to print their results on the *Printer*. Since there is only a single printer, only one user can print a job at a time. Suppose we have the following atomic propositions for *Peter* at our disposal:

- *Peter.request* ::= indicates that *Peter* requests usage of the printer;
- *Peter.use* ::= indicates that *Peter* uses the printer;
- *Peter.release* ::= indicates that *Peter* releases the printer.

For *Betsy*, similar predicates are defined. Specify in LTL the following properties:

(a) Mutual exclusion, i.e., only one user at a time can use the printer.

(b) Finite time of usage, i.e., a user can print only for a finite amount of time.

(c) Absence of individual starvation, i.e., if a user wants to print something, he/she eventually is able to do so.

(d) Absence of blocking, i.e., a user can always request to use the printer

(e) Alternating access, i.e., users must strictly alternate in printing.

EXERCISE 5.5. Consider an elevator system that services $N > 0$ floors numbered 0 through $N-1$. There is an elevator door at each floor with a call-button and an indicator light that signals whether or not the elevator has been called. For simplicity consider $N = 4$. Present a set of atomic propositions – try to minimize the number of propositions – that are needed to describe the following properties of the elevator system as LTL formulae and give the corresponding LTL formulae:

(a) The doors are "safe", i.e., a floor door is never open if the elevator is not present at the given floor.

(b) A requested floor will be served sometime.

(c) Again and again the elevator returns to floor 0.

(d) When the top floor is requested, the elevator serves it immediately and does not stop on the way there.

EXERCISE 5.6. Which of the following equivalences are correct? Prove the equivalence or provide a counterexample that illustrates that the formula on the left and the formula on the right are not equivalent.

(a) $\Box\varphi \rightarrow \Diamond\psi \quad \equiv \quad \varphi \,\mathsf{U}\, (\psi \vee \neg\varphi)$

(b) $\Diamond\Box\varphi \rightarrow \Box\Diamond\psi \quad \equiv \quad \Box\big(\varphi \,\mathsf{U}\, (\psi \vee \neg\varphi)\big)$

(c) $\Box\Box(\varphi \vee \neg\psi) \;\equiv\; \neg\Diamond(\neg\varphi \wedge \psi)$

(d) $\Diamond(\varphi \wedge \psi) \;\equiv\; \Diamond\varphi \;\wedge\; \Diamond\psi$

(e) $\Box\varphi \;\wedge\; \bigcirc\Diamond\varphi \;\equiv\; \Box\varphi$

(f) $\Diamond\varphi \;\wedge\; \bigcirc\Box\varphi \;\equiv\; \Diamond\varphi$

(g) $\Box\Diamond\varphi \rightarrow \Box\Diamond\psi \;\equiv\; \Box\big(\varphi \rightarrow \Diamond\psi\big)$

(h) $\neg(\varphi_1 \,\mathsf{U}\, \varphi_2) \;\equiv\; \neg\varphi_2 \,\mathsf{W}\, (\neg\varphi_1 \wedge \neg\varphi_2)$

(i) $\bigcirc\Diamond\varphi_1 \equiv \Diamond\bigcirc\varphi_2$

(j) $(\Diamond\Box\varphi_1) \wedge (\Diamond\Box\varphi_2) \;\equiv\; \Diamond(\Box\varphi_1 \wedge \Box\varphi_2)$

(k) $(\varphi_1 \,\mathsf{U}\, \varphi_2) \,\mathsf{U}\, \varphi_2 \;\equiv\; \varphi_1 \,\mathsf{U}\, \varphi_2$

EXERCISE 5.7. Let φ and ψ be LTL formulae. Consider the following new operators:

(a) "At next" $\varphi \,\mathsf{N}\, \psi$: at the next time where ψ holds, φ also holds.

(b) "While" $\varphi \,\mathsf{W}\, \psi$: φ holds as least as long as ψ does.

(c) "Before" $\varphi \,\mathsf{B}\, \psi$: if ψ holds sometime, φ does so before.

Make the definitions of these informally explained operators precise by providing LTL formulae that formalize their intuitive meanings.

EXERCISE 5.8. We consider the release operator R which was defined by $\varphi \,\mathsf{R}\, \psi \stackrel{\text{def}}{=} \neg(\neg\varphi \,\mathsf{U}\, \neg\psi)$; see Section 5.1.5 on page 252 ff.

(a) Prove the expansion law $\varphi_1 \,\mathsf{R}\, \varphi_2 \;\equiv\; \varphi_2 \wedge (\varphi_1 \vee \bigcirc(\varphi_1 \wedge \varphi_2))$.

(b) Prove that $\varphi \,\mathsf{R}\, \psi \equiv (\neg\varphi \wedge \psi)\,\mathsf{W}\,(\varphi \wedge \psi)$.

(c) Prove that $\varphi_1 \,\mathsf{W}\, \varphi_2 \;\equiv\; (\neg\varphi_1 \vee \varphi_2)\,\mathsf{R}\,(\varphi_1 \vee \varphi_2)$.

(d) Prove that $\varphi_1 \,\mathsf{U}\, \varphi_2 \;\equiv\; \neg(\neg\varphi_1 \,\mathsf{R}\, \neg\varphi_2)$.

EXERCISE 5.9. Consider the LTL formula

$$\varphi \;=\; \neg\Big((\Box a) \rightarrow \big((a \wedge \neg c)\,\mathsf{U}\,\neg(\bigcirc b)\big)\Big) \wedge \neg(\neg a \vee \bigcirc \Diamond c).$$

Transform φ into an equivalent LTL formula in PNF

(a) using the weak-until operator W,

(b) using the release operator R.

EXERCISE 5.10. Provide an example for a sequence (φ_n) of LTL formulae such that the LTL formula ψ_n is in weak-until PNF, $\varphi_n \equiv \psi_n$, and ψ_n is exponentially longer than φ_n. Use the transformation rules in Section 5.1.5

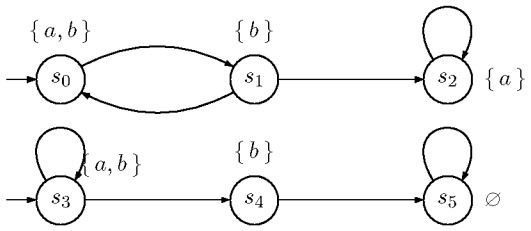


Figure 5.25: Transition system for Exercise 5.11.

EXERCISE 5.11. Consider the transition system TS in Figure 5.25 with the set $AP = \{a, b, c\}$ of atomic propositions. Note that this is a single transition system with two initial states. Consider the LTL fairness assumption

$$\mathit{fair} \;=\; \big(\Box\Diamond(a \wedge b) \;\rightarrow\; \Box\Diamond\neg c\big) \;\wedge\; \big(\Diamond\Box(a \wedge b) \rightarrow \Box\Diamond\neg b\big).$$

Questions:

(a) Determine the fair paths in TS, i.e., the initial, infinite paths satisfying *fair*

(b) For each of the following LTL formulae:

$$\begin{aligned}
\varphi_1 &= \Diamond\Box a \\
\varphi_2 &= \bigcirc\neg a \;\longrightarrow\; \Diamond\Box a \\
\varphi_3 &= \Box a \\
\varphi_4 &= b\,\mathsf{U}\,\Box\neg b \\
\varphi_5 &= b\,\mathsf{W}\,\Box\neg b \\
\varphi_6 &= \bigcirc\bigcirc\, b\,\mathsf{U}\,\Box\neg b
\end{aligned}$$

determine whether $TS \models_{fair} \varphi_i$. In case $TS \not\models_{fair} \varphi_i$, indicate a path $\pi \in \mathit{Paths}(TS)$ for which $\pi \not\models \varphi_i$.

EXERCISE 5.12. Let $\varphi = (a \rightarrow \bigcirc \neg b)\, \mathsf{W}\, (a \wedge b)$ and $P = \mathit{Words}(\varphi)$ where $AP = \{ a, b \}$.

(a) Show that P is a safety property.

(b) Define an NFA $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = \mathit{BadPref}(P)$.

(c) Now consider $P' = \mathit{Words}\big((a \rightarrow \bigcirc \neg b)\, \mathsf{U}\, (a \wedge b)\big)$. Decompose P' into a safety property P_{safe} and a liveness property P_{live} such that

$$P' = P_{safe} \cap P_{live}.$$

Show that P_{safe} is a safety and that P_{live} is a liveness property.

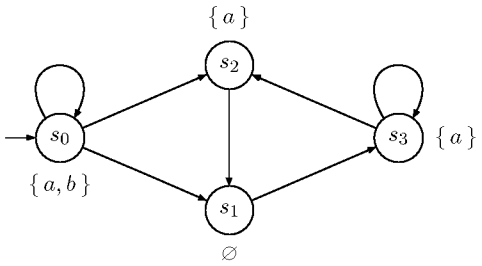


Figure 5.26: Transition system for Exercise 5.14.

EXERCISE 5.13. Provide an NBA for each of the following LTL formulae:

$$\Box(a \vee \neg \bigcirc\, b) \quad \text{and} \quad \Diamond a \;\vee\; \Box\Diamond(a \leftrightarrow b) \quad \text{and} \quad \bigcirc\, \bigcirc\, (a \vee \Diamond\Box b).$$

EXERCISE 5.14. Consider the transition system TS in Figure 5.26 with the atomic propositions $\{ a, b \}$. Sketch the main steps of the LTL model-checking algorithm applied to TS and the LTL formulae

$$\varphi_1 \;=\; \Box\Diamond a \rightarrow \Box\Diamond b \quad \text{and} \quad \varphi_2 \;=\; \Diamond(a \wedge \bigcirc\, a).$$

To that end, carry out the following steps:

(a) Depict an NBA A_i for $\neg\varphi_i$.

(b) Depict the reachable fragment of the product transition system $TS \otimes \mathcal{A}_i$.

(c) Explain the main steps of the nested DFS in $TS \otimes \mathcal{A}_i$ by illustrating the order in which the states are visited during the "outer" and "inner" DFS.

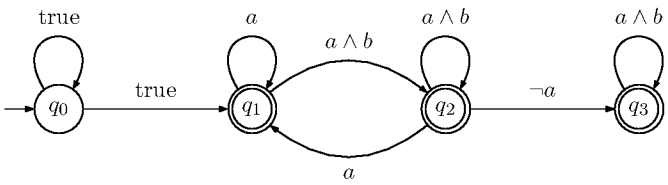


Figure 5.27: GNBA for Exercise 5.15.

(d) If $TS \not\models \varphi_i$, provide the counterexample resulting from the nested DFS.

EXERCISE 5.15. Consider the GNBA $\mathcal{G}$ in Figure 5.27 with the alphabet $\Sigma = 2^{\{a,b\}}$ and the set $\mathcal{F} = \{\{q_1, q_3\}, \{q_2\}\}$ of accepting sets.

(a) Provide an LTL formula φ with $\mathit{Words}(\varphi) = \mathcal{L}_\omega(\mathcal{G})$. Justify your answer.

(b) Depict the NBA $\mathcal{A}$ with $\mathcal{L}_\omega(\mathcal{A}) = \mathcal{L}_\omega(\mathcal{G})$.

EXERCISE 5.16. Depict a GNBA $\mathcal{G}$ over the alphabet $\Sigma = 2^{\{a,b,c\}}$ such that

$$\mathcal{L}_\omega(\mathcal{G}) \;=\; \mathit{Words}\big(\; (\Box\Diamond a \rightarrow \Box\Diamond b) \;\wedge\; \neg a \;\wedge\; (\neg a \,\mathsf{W}\, c) \;\big).$$

EXERCISE 5.17. Let $\psi = \Box\,(a \leftrightarrow \bigcirc \neg a)$ and $AP = \{a\}$.

(a) Show that ψ can be transformed into the following equivalent basic LTL formula

$$\varphi = \neg\,\big[\,\mathrm{true}\,\mathsf{U}\,\big(\neg\,(a \wedge \bigcirc \neg a) \wedge \neg\,(\neg a \wedge \neg \bigcirc\, \neg a)\,\big)\big]\,.$$

The basic LTL syntax is given by the following context-free grammar:

$$\varphi ::= \mathrm{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\, \varphi \mid \varphi_1\,\mathsf{U}\,\varphi_2.$$

(b) Compute all elementary sets with respect to $\mathit{closure}(\varphi)$ (*Hint: There are six elementary sets.*)

(c) Construct the GNBA $\mathcal{G}_\varphi$ with $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$. To that end:

(i) Define its set of initial states and its acceptance component.

(ii) For each elementary set B, define $\delta(B, B \cap AP)$.

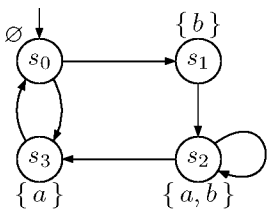
EXERCISE 5.18. Let $AP = \{a\}$ and $\varphi = (a \wedge \bigcirc\, a)\,\mathsf{U}\,\neg a$ an LTL formula over AP.

(a) Compute all elementary sets with respect to φ.
(*Hint: There are five elementary sets.*).

(b) Construct the GNBA $\mathcal{G}_\varphi$ such that $\mathcal{L}_\omega(\mathcal{G}_\varphi) = \mathit{Words}(\varphi)$.

EXERCISE 5.19. Consider the formula $\varphi = a \,\mathsf{U}\, (\neg a \wedge b)$ and let $\mathcal{G}$ be the GNBA for φ that is obtained through the construction explained in the proof of Theorem 4.56. What are the initial states in $\mathcal{G}$? What are the accept states in $\mathcal{G}$? Provide an accepting run for the word $\{a\}\{a\}\{a,b\}\{b\}^\omega$. Explain why there are no accepting runs for the words $\{a\}^\omega$ and $\{a\}\{a\}\{a,b\}^\omega$. (*Hint: The answers to these questions can be given without depicting* $\mathcal{G}$.)

EXERCISE 5.20.
We consider the LTL formula $\varphi = \Box\,(a \rightarrow (\neg b \,\mathsf{U}\, (a \wedge b)))$ over the set $AP = \{\,a, b\,\}$ of atomic propositions and we want to check $TS \models \varphi$ for TS outlined on the right.



(a) To check $TS \models \varphi$, convert $\neg\varphi$ into an equivalent LTL formula ψ which is constructed according to the following grammar:

$$\Phi ::= \text{true} \mid \text{false} \mid a \mid b \mid \Phi \wedge \Phi \mid \neg\Phi \mid \bigcirc\, \Phi \mid \Phi \,\mathsf{U}\, \Phi.$$

Then construct $\mathit{closure}(\psi)$.

(b) Give the elementary sets w.r.t. $\mathit{closure}(\psi)$!

(c) Construct the GNBA $\mathcal{G}_\psi$.

(d) Construct an NBA $\mathcal{A}_{\neg\varphi}$ *directly* from $\neg\varphi$, i.e., without relying on $\mathcal{G}_\psi$. (*Hint: Four states suffice.*)

(e) Construct $TS \otimes \mathcal{A}_{\neg\varphi}$.

(f) Use the nested DFS algorithm to check $TS \models \varphi$. Therefore, sketch the algorithm's main steps and interpret its outcome!

EXERCISE 5.21. The construction of a $\mathcal{G}$ from a given LTL formula in the proof of Theorem 4.56 assumes an LTL formula that only uses the basic temporal modalities $\bigcirc$ and U. The derived operators $\Diamond$, $\Box$, W and R can be treated by syntactic replacements of their definitions. Alternatively, and more efficient, is to treat them as basic modalities and to allow for formulae $\Diamond\psi$, $\Box\psi$, $\varphi_1 \,\mathsf{W}\, \varphi_2$ and $\varphi_1 \,\mathsf{R}\, \varphi_2$ as elements of elementary sets of formulae and redefine the components of the constructed GNBA.

Explain which modifications are necessary for such a "direct" treatment of $\Diamond$ (eventually), $\Box$ (always), W (weak-until), and R (release). That is, which additional conditions do the elementary sets, the transition function δ, and the set $\mathcal{F}$ of accepting sets have to fulfill?

EXERCISE 5.22. Let $TS = (S, Act, \rightarrow, S_0, AP, L)$ be a finite transition system without terminal states and let $wfair = \Diamond\Box b_1 \rightarrow \Box\Diamond b_2$ be a weak LTL fairness assumption with $b_1, b_2 \in AP$. Explain how the nested DFS can be modified to check directly whether $TS \models_{wfair} \Diamond\Box a$ (where $a \in AP$), that is, without using the transformation $TS \models_{wfair} \Diamond\Box a$ iff $TS \models (wfair \rightarrow \Diamond\Box a)$.

EXERCISE 5.23. Which of the following LTL formulae φ_i are representable by a deterministic Büchi automaton?

$$\varphi_1 \;=\; \Box(a \rightarrow \Diamond b), \quad \varphi_2 \;=\; \neg\varphi_1.$$

Explain your answer.

EXERCISE 5.24. Check for the following LTL formula whether they are (i) satisfiable, and/or (ii) valid:

(a) $\bigcirc\,\bigcirc\, a \;\Rightarrow\; \bigcirc\, a$

(b) $\bigcirc\,(a \vee \Diamond\, a) \;\Rightarrow\; \Diamond\, a$

(c) $\Box\, a \;\Rightarrow\; \neg\,\bigcirc\,(\neg\, a \wedge \Box\,\neg\, a)$

(d) $(\Box\, a)\,\mathsf{U}\,(\Diamond\, b) \;\Rightarrow\; \Box\,(a\,\mathsf{U}\,\Diamond\, b)$

(e) $\Diamond\, b \;\Rightarrow\; (a\,\mathsf{U}\, b)$

Practical Exercises

EXERCISE 5.25. Consider an arbitrary, but finite, number of identical processes[2], that execute in parallel. Each process consists of a noncritical part and a critical part, usually called the *critical section*. In this exercise we are concerned with the verification of a mutual exclusion protocol, that is, a protocol that should ensure that at any moment of time at most one process (among the N processes in our configuration) is in its critical section. There are many different mutual exclusion protocols developed in the literature. In this exercise we are concerned with Szymanski's protocol [384]. Assume there are N processes for some fixed $N > 0$. There is a global variable, referred to as *flag*, which is an array of length N, such that *flag[i]* is a value between 0 and 4 (for $0 \leqslant i < N$). The idea is that *flag[i]* indicates the status of process i. The protocol executed by process i looks as follows:

l0: **loop forever do**
begin
l1: Noncritical section
l2: *flag[i]* := 1;

[2] Only the identity of a process is unique.

```
l3: wait until (flag[0] < 3 and flag[1] < 3 and ... and flag[N-1] < 3)
l4: flag[i] := 3;
l5: if (flag[0] = 1 or flag[1] = 1 or ... or flag[N-1] = 1)
    then begin
            l6: flag[i] := 2;
            l7: wait until (flag[0] = 4 or flag[1] = 4 or ... or flag[N-1] = 4)
    end
l8: flag[i] := 4;
l9: wait until (flag[0] < 2 and flag[1] < 2 and ... and flag[i-1] < 2)
l10: Critical section
l11: wait until (flag[i+1] ∈ {0, 1, 4}) and ... and (flag[N-1] ∈ {0, 1, 4})
l12: flag[i] := 0;
end.
```

Before doing any of the exercises listed below, try first to informally understand what the protocol is doing and why it could be correct in the sense that mutual exclusion is ensured. If you are convinced of the fact that the correctness of this protocol is not easy to see — otherwise please inform me — then start with the following questions.

1. Model Szymanski's protocol in Promela. Assume that all tests on the global variable *flag* (such as the one in statement l3) are *atomic*. Look carefully at the indices of the variable *flag* used in the tests. Make the protocol description modular such that the number of processes can be changed easily.
2. Check for several values of N ($N \geqslant 2$) that the protocol indeed ensures mutual exclusion. Report your results for N equal to 4.
3. The code that a process has to go through before reaching the critical section can be divided into several segments. We refer to statement l4 as the *doorway*, to segments l5, l6, and l7, as the *waiting room* and to segments l8 through l12 (which contains the critical section) as the *inner sanctum*. You are requested to check the following basic claims using assertions. Give for each case the changes to your original Promela specification for Szymanski's protocol and present the verification results. In case of negative results, simulate the counterexample by means of guided simulation.
   (a) Whenever some process is in the inner sanctum, the doorway is locked, that is, no process is at location l4.
   (b) If a process i is at l10, l11 or l12, then it has the least index of all the processes in the waiting room and the inner sanctum.
   (c) If some process is at l12, then all processes in the waiting room and in the inner sanctum must have flag value 4.

EXERCISE 5.26. We assume N processes in a ring topology, connected by unbounded queues. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier *ident* (which is assumed to be a natural number). A process can be either active or

relaying. Initially a process is active. In Peterson's leader election algorithm (1982) each process in the ring carries out the following task:

active:
$d := ident$;
do forever
begin
  /* start phase */
  $send(d)$;
  $receive(e)$;
  if $e = ident$ **then** announce elected;
  if $d > e$ **then** $send(d)$ **else** $send(e)$;
  $receive(f)$;
  if $f = ident$ **then** announce elected;
  if $e \geqslant max(d, f)$ **then** $d := e$ **else goto** *relay*;
end

relay:
do forever
begin
  $receive(d)$;
  if $d = ident$ **then** announce elected;
  $send(d)$
end

Solve the following questions concerning the leader election protocol:

1. Model Peterson's leader election protocol in Promela (avoid invalid end states).
2. Verify the following properties:
   (a) There is always at most one leader.
   (b) Eventually always a leader will be elected.
   (c) The elected leader will be the process with the highest number.
   (d) The maximum total amount of messages sent in order to elect a leader is at most $2N\lfloor \log_2 N \rfloor + N$.

EXERCISE 5.27. This exercise deals with a simple fault-tolerant communication protocol in which processes can fail. A failed process is still able to communicate, i.e., it is able to send and receive messages, but the content of its transmitted messages is unreliable. More precisely, a failed process can send messages with arbitrary content. A failed process is therefore also called *unreliable*.

We are given N reliable processes (i.e., processes that have not failed and that are working as expected) and K unreliable processes, where N is larger than $3 \cdot K$ and K is at least 0. There is no way, a priori, to distinguish the reliable and the unreliable processes. All processes communicate

by means of exchanging messages. Each process has a local variable, which initially has a value, 0 or 1. The following informally described protocol is aimed to be followed by the reliable processes, such that at the end of round $K+1$ we have

- eventually every reliable process has the same value in its local variable, and
- if all reliable processes have the same initial value, then their final value is the same as their common initial value.

The difficulty of this protocol is to establish these constraints in the presence of the K unreliable processes!

Informal description of the protocol The following protocol is due to Berman and Garay [47]. Let the processes be numbered 1 through $N+K$. Processes communicate with each other in "rounds". Each round consists of two phases of message transmissions: In round i, $i > 0$, in the first phase, every process sends its value to all processes (including itself); in the second phase, process i sends the majority value it received in the first phase (for majority to be well-defined we assume that $N+K$ is odd) to all processes. If a process receives N, or more, instances of the same value in its first phase of the round, it sets its local variable to this value; otherwise, it sets its local variable to the value received (from process i) in the second phase of this round.

1. Model this protocol in Promela. Make the protocol description modular such that the number of reliable and unreliable processes can be changed easily. As the state space of your protocol model could be very large, instantiate your model with a single unreliable process and four reliable processes.

   First hint: One of the main causes for the large state space is the model for the unreliable process, so try to keep this model as simple as possible. This can be achieved by, for instance, assuming that an unreliable process can only transmit arbitrary 0 or 1 values (and not any other value) and that a process always starts with a fixed initial value (and not with a randomly selected one). In addition, use atomic broadcast for message transmissions.

   Second hint: It might be convenient (though not necessary) to use a matrix of size $(N+K) \cdot (N+K)$ of channels for the communication structure. As Promela does not support multi-dimensional arrays, you could use the following construct (where M equals $N+K$):

   ```
   typedef Arraychan {
   chan ch[M] = [1] of {bit};    /* M channels of size 1 */
   }

   Arraychan A[M];            /* a matrix A of MxM channels of size 1 */
   ```

   Statement `A[i].ch[j]!0` denotes an output of value `0` over the channel directed process from `i` to `j`. Similarly, statement `A[i].ch[j]?b` denotes the receipt of a bit value stored in variable `b` via the channel directed from process `i` to `j`.

2. Formalize the two constraints of the protocol in LTL and convert these into never claims.

3. Check the two temporal logic properties by SPIN and hand in the verification output generated by SPIN.

4. Show that the requirement $N > 3 \cdot K$ is essential, by, for instance, changing the configuration of your system such that $N \leqslant 3 \cdot K$ and checking that for this configuration the first aforementioned constraint is violated. Create the shortest counterexample (select the shortest trail in the advanced verification options) and perform a guided simulation of this undesired scenario. Hand in the counterexample you found and give an explanation of it.

Chapter 6

Computation Tree Logic

This chapter introduces Computation Tree Logic (CTL), a prominent branching temporal logic for specifying system properties. In particular, the syntax and semantics of CTL are presented, a comparison to Linear Temporal Logic (LTL) is provided, and the issue of fairness in CTL is treated. CTL model checking is explained in detail. First, the core recursive algorithm is presented that is based on a bottom-up traversal of the parse tree of the formula at hand. The foundations of this algorithm are discussed and the adaptations needed to incorporate fairness are detailed. This is followed by an algorithm for the generation of counterexamples. The chapter is concluded by presenting a model-checking algorithm for CTL*, a branching-time logic that subsumes both LTL and CTL.

6.1 Introduction

Pnueli [337] has introduced linear temporal logic for the specification and verification of reactive systems. LTL is called linear, because the qualitative notion of time is path-based and viewed to be linear: at each moment of time there is only one possible successor state and thus each time moment has a unique possible future. Technically speaking, this follows from the fact that the interpretation of LTL formulae is defined in terms of paths, i.e., sequences of states.

Paths themselves, though, are obtained from a transition system that might be branching: a state may have several, distinct direct successor states, and thus several computations may start in a state. The interpretation of LTL-formulae in a state requires that a formula φ holds in state s if *all* possible computations that start in s satisfy φ. The universal

quantification over all computations that is implicit in the LTL semantics can also be made explicit in the formula, e.g.:

$s \models \forall\varphi$ if and only if $\pi \models \varphi$ for all paths π starting in s

In LTL, we thus can state properties over all possible computations that start in a state, but not easily about *some* of such computations. To some extent this may be overcome by exploiting the duality between universal and existential quantification. For instance, to check whether there exists some computation starting in s that satisfies φ we may check whether $s \models \forall \neg \varphi$; if this formula is not satisfied by s, then there must be a computation that meets φ, otherwise they should all refute φ.

For more complicated properties, like "for every computation it is always possible to return to the initial state", this is, however, not possible. A naive attempt would be to require $\Box \Diamond \mathit{start}$ for every computation, i.e., $s \models \forall\Box \Diamond \mathit{start}$, where the proposition *start* uniquely identifies the initial state. This is, however, too strong as it requires a computation to *always* return to the initial state, not just *possibly*. Other attempts to specify the intended property also fail, and it even turns out to be the case that the property *cannot* be specified in LTL.

To overcome these problems, in the early eighties another strand of temporal logics for specification and verification purposes was introduced by Clarke and Emerson [86]. The semantics of this kind of temporal logic is not based on a linear notion of time—an infinite sequence of states—but on a *branching* notion of time—an infinite *tree* of states. Branching time refers to the fact that at each moment there may be several different possible futures. Each moment of time may thus split into several possible futures. Due to this branching notion of time, this class of temporal logic is known as *branching* temporal logic. The semantics of a branching temporal logic is defined in terms of an infinite, directed *tree* of states rather than an infinite sequence. Each traversal of the tree starting in its root represents a single path. The tree itself thus represents all possible paths, and is directly obtained from a transition system by "unfolding" at the state of interest. The tree rooted at state s thus represents all possible infinite computations in the transition system that start in s. Figure 6.1 depicts a transition system and its unfolding. (For convenience, each node in the tree consists of a pair indicating the state and the level of the node in the tree.)

The temporal operators in branching temporal logic allow the expression of properties of *some* or *all* computations that start in a state. To that end, it supports an existential path quantifier (denoted $\exists$) and a universal path quantifier (denoted $\forall$). For instance, the property $\exists\Diamond \Phi$ denotes that there exists a computation along which $\Diamond \Phi$ holds. That is, it states that there is at least one possible computation in which a state that satisfies Φ

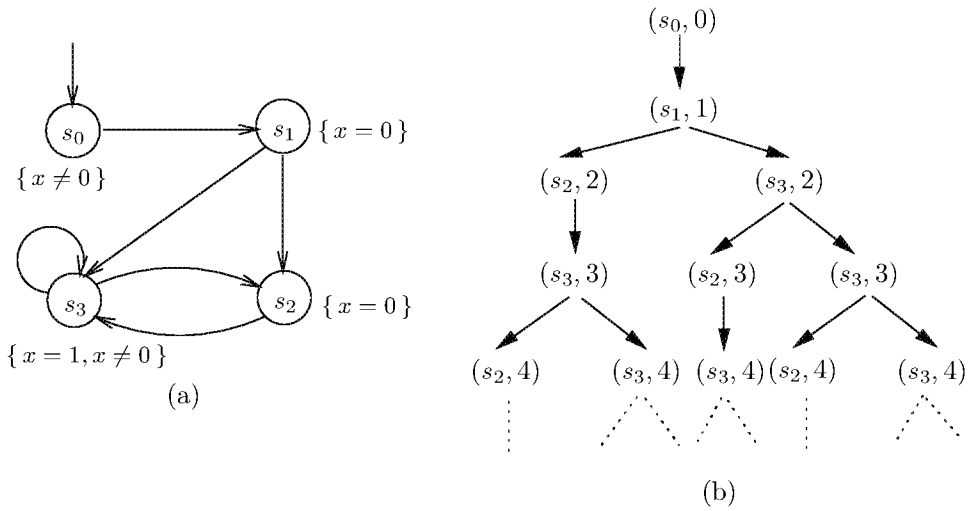


Figure 6.1: (a) A transition system and (b) a prefix of its infinite computation tree

is eventually reached. This does not, however, exclude the fact that there can also be computations for which this property does not hold, for instance, computations for which Φ is always refuted. The property $\forall\Diamond\,\Phi$, in contrast, states that all computations satisfy the property $\Diamond\,\Phi$. More complicated properties can be expressed by nesting universal and existential path quantifiers. For instance, the aforementioned property "for every computation it is always possible to return to the initial state" can be faithfully expressed by $\forall\Box\,\exists\Diamond\,\mathit{start}$: in any state ($\Box$) of any possible computation ($\forall$), there is a possibility ($\exists$) to eventually return to the start state ($\Diamond\,\mathit{start}$).

This chapter considers *Computation Tree Logic (CTL)*, a temporal logic based on propositional logic with a discrete notion of time, and only future modalities. CTL is an important branching temporal logic that is sufficiently expressive for the formulation of an important set of system properties. It was originally used by Clarke and Emerson [86] and (in a slightly different form) by Queille and Sifakis [347] for model checking. More importantly, it is a logic for which efficient and—as we will see—rather simple model-checking algorithms do exist.

Anticipatory to the results presented in this chapter, we summarize the major aspects of the linear-vs-branching-time debate and provide arguments that justify the treatment of model checking based on linear or branching time logics:

- The expressiveness of many linear and branching temporal logics is incomparable. This means that some properties that are expressible in a linear temporal logic cannot be expressed in certain branching temporal logics, and vice versa.

Aspect	*Linear time*	*Branching time*
"behavior" in a state s	path-based: $trace(s)$	state-based: computation tree of s
temporal logic	LTL: path formulae φ $s \models \varphi$ iff $\forall \pi \in Paths(s).\, \pi \models \varphi$	CTL: state formulae existential path quantification $\exists\varphi$ universal path quantification: $\forall\varphi$
complexity of the model checking problems	PSPACE–complete $\mathcal{O}\left(\lvert TS\rvert \cdot \exp(\lvert\varphi\rvert)\right)$	*PTIME* $\mathcal{O}(\lvert TS\rvert \cdot \lvert\Phi\rvert)$
implementation-relation	trace inclusion and the like (proof is PSPACE-complete)	simulation and bisimulation (proof in polynomial time)
fairness	no special techniques needed	special techniques needed

Table 6.1: Linear-time vs. branching-time in a nutshell.

- The model-checking algorithms for linear and branching temporal logics are quite different. This results, for instance, in significantly different time and space complexity results.
- The notion of fairness can be treated in linear temporal logic without the need for any additional machinery since fairness assumptions can be expressed in the logic. For various branching temporal logics this is not the case.
- The equivalences and preorders between transition systems that "correspond" to linear temporal logic are based on traces, i.e., trace inclusion and equality, whereas for branching temporal logic such relations are based on simulation and bisimulation relations (see Chapter 7).

Table 6.1 summarizes the main differences between the linear-time and branching-time perspective in a succinct way.

6.2 Computation Tree Logic

This section presents the syntax and the semantics of CTL. The following sections will discuss the relation and differences between CTL and LTL, present a model-checking algorithm for CTL, and introduce some extensions of CTL.

6.2.1 Syntax

CTL has a two-stage syntax where formulae in CTL are classified into state and path formulae. The former are assertions about the atomic propositions in the states and their branching structure, while path formulae express temporal properties of paths. Compared to LTL formulae, path formulae in CTL are simpler: as in LTL they are built by the next-step and until operators, but they must not be combined with Boolean connectives and no nesting of temporal modalities is allowed.

Definition 6.1. Syntax of CTL

CTL *state formulae* over the set AP of atomic proposition are formed according to the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula. CTL *path formulae* are formed according to the following grammar:

$$\varphi ::= \bigcirc \Phi \mid \Phi_1 \,\mathsf{U}\, \Phi_2$$

where Φ, Φ_1 and Φ_2 are state formulae. ■

Greek capital letters will denote CTL state formulae (CTL formulae, for short), whereas lowercase Greek letters will denote CTL path formulae.

CTL distinguishes between state formulae and path formulae. Intuitively, state formulae express a property of a state, while path formulae express a property of a path, i.e., an infinite sequence of states. The temporal operators $\bigcirc$ and U have the same meaning as in LTL and are path operators. Formula $\bigcirc\Phi$ holds for a path if Φ holds at the next state in the path, and $\Phi\,\mathsf{U}\,\Psi$ holds for a path if there is some state along the path for which Ψ holds, and Φ holds in all states prior to that state. Path formulae can be turned into state formulae by prefixing them with either the path quantifier $\exists$ (pronounced "for some path") or the path quantifier $\forall$ (pronounced "for all paths"). Note that the linear

temporal operators $\bigcirc$ and U are required to be immediately preceded by $\exists$ or $\forall$ to obtain a legal state formula. Formula $\exists\varphi$ holds in a state if there exists *some* path satisfying φ that starts in that state. Dually, $\forall\varphi$ holds in a state if *all* paths that start in that state satisfy φ.

Example 6.2. Legal CTL Formulae

Let $AP = \{ x = 1, x < 2, x \geqslant 3 \}$ be a set of atomic propositions. Examples of syntactically correct CTL formulae are

$$\exists\bigcirc(x = 1), \forall\bigcirc(x = 1), \text{ and } x < 2 \ \vee \ x = 1$$

and $\exists((x < 2)\,\mathsf{U}\,(x \geqslant 3))$ and $\forall(\text{true}\,\mathsf{U}\,(x < 2))$. Some examples of formulae that are syntactically incorrect are

$$\exists(x = 1 \ \wedge \ \forall\bigcirc(x \geqslant 3)) \text{ and } \exists\bigcirc(\text{true}\,\mathsf{U}\,(x = 1)).$$

The first is not a CTL formula since $x = 1 \ \wedge \ \forall\bigcirc(x \geqslant 3)$ is not a path formula and thus must not be preceded by $\exists$ The second formula is not a CTL formula since $\text{true}\,\mathsf{U}\,(x = 1)$ is a path formula rather than a state formula, and thus cannot be preceded by $\bigcirc$. Note that

$$\exists\bigcirc(x = 1 \ \wedge \ \forall\bigcirc(x \geqslant 3)) \text{ and } \exists\bigcirc\forall(\text{true}\,\mathsf{U}\,(x = 1))$$

are, however, syntactically correct CTL formulae. ■

The Boolean operators true, false, $\wedge$, $\rightarrow$ and $\Leftrightarrow$ are defined in the usual way. The temporal modalities "eventually", "always", and "weak until" can be derived—similarly as for LTL—as follows:

$$\begin{array}{lll}
\text{eventually:} & \exists\Diamond\Phi & = \ \exists(\text{true}\,\mathsf{U}\,\Phi) \\
& \forall\Diamond\Phi & = \ \forall(\text{true}\,\mathsf{U}\,\Phi) \\
\text{always:} & \exists\Box\Phi & = \ \neg\forall\Diamond\neg\Phi \\
& \forall\Box\Phi & = \ \neg\exists\Diamond\neg\Phi
\end{array}$$

$\exists\Diamond\,\Phi$ is pronounced "Φ holds potentially" and $\forall\Diamond\,\Phi$ is pronounced "Φ is inevitable". $\exists\Box\,\Phi$ is pronounced "potentially always Φ", $\forall\Box\,\Phi$ is pronounced "invariantly Φ", and $\forall\bigcirc\,\Phi$ is pronounced "for all paths next Φ".

Note that "always" Φ cannot be obtained from the "equation" $\Box\Phi = \neg\Diamond\neg\Phi$ (as in LTL), since propositional logic operators cannot be applied to path formulae. In particular, $\exists\neg\Diamond\neg\Phi$ is *not* a CTL formula. Instead, we exploit the duality of existential and universal quantification:

- there exists a path with the property E if and only if the state property "not all paths violate property E" is satisfied, and
- all paths satisfy property E if and only if the state property "there is a path without property E" is violated.

Accordingly, $\exists\Box\,\Phi$ is not defined as $\exists\neg\Diamond\,\neg\Phi$, but rather as $\neg\forall\Diamond\,\neg\Phi$.

Example 6.3. CTL Formulae

To give a feeling about how simple properties can be formalized in CTL we treat some intuitive examples. The mutual exclusion property can be described in CTL by the formula

$$\forall\Box(\neg crit_1 \vee \neg crit_2).$$

CTL formulae of the form $\forall\Box\forall\Diamond\Phi$ express that Φ is infinitely often true on all paths. (This fact will be formally proven later; see Remark 6.8 on page 326.) The CTL formula

$$(\forall\Box\forall\Diamond crit_1) \;\wedge\; (\forall\Box\forall\Diamond crit_2)$$

thus requires each process to have access to the critical section infinitely often. In case of a traffic light, the safety property "each red light phase is preceded by a yellow light phase" can be formulated in CTL by

$$\forall\Box(yellow \;\vee\; \forall\bigcirc\neg red)$$

depending on the precise meaning of a "phase". The liveness property "the traffic light is infinitely often green" can be formulated as

$$\forall\Box\forall\Diamond green \quad .$$

Progress properties such as "every request will eventually be granted" can be described by

$$\forall\Box(request \;\longrightarrow\; \forall\Diamond response).$$

Finally, the CTL formula

$$\forall\Box\;\exists\Diamond\; start$$

expresses that in every reachable system state it is possible to return (via 0 or more transitions) to (one of) the starting state(s). ■

6.2.2 Semantics

CTL formulae are interpreted over the states and paths of a transition system TS. Formally, given a transition system TS, the semantics of CTL formulae is defined by two satisfaction relations (both denoted by $\models_{TS}$, or briefly $\models$): one for the state formulae and one for the path formulae. For the state formulae, $\models$ is a relation between the states in TS and state formulae. We write $s \models \Phi$ rather than $(s, \Phi) \in \models$. The intended interpretation is: $s \models \Phi$ if and only if state formula Φ holds in state s. For the path formulae, $\models$ is a relation between maximal path fragments in TS and path formulae. We write $\pi \models \Phi$ rather than $(\pi, \Phi) \in \models$. The intended interpretation is: $\pi \models \varphi$ if and only if path π satisfies path formula φ.

Definition 6.4. Satisfaction Relation for CTL

Let $a \in AP$ be an atomic proposition, $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, state $s \in S$, Φ, Ψ be CTL state formulae, and φ be a CTL path formula. The satisfaction relation $\models$ is defined for state formulae by

$$
\begin{array}{lll}
s \models a & \text{iff} & a \in L(s) \\
s \models \neg \Phi & \text{iff} & \text{not } s \models \Phi \\
s \models \Phi \wedge \Psi & \text{iff} & (s \models \Phi) \text{ and } (s \models \Psi) \\
s \models \exists \varphi & \text{iff} & \pi \models \varphi \text{ for some } \pi \in \mathit{Paths}(s) \\
s \models \forall \varphi & \text{iff} & \pi \models \varphi \text{ for all } \pi \in \mathit{Paths}(s)
\end{array}
$$

For path π, the satisfaction relation $\models$ for path formulae is defined by

$$
\begin{array}{lll}
\pi \models \bigcirc \Phi & \text{iff} & \pi[1] \models \Phi \\
\pi \models \Phi \,\mathsf{U}\, \Psi & \text{iff} & \exists\, j \geqslant 0.\ (\pi[j] \models \Psi \ \wedge\ (\forall\, 0 \leqslant k < j.\, \pi[k] \models \Phi)) \quad .
\end{array}
$$

where for path $\pi = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$, $\pi[i]$ denotes the $(i{+}1)$th state of π, i.e., $\pi[i] = s_i$. ■

The interpretations for atomic propositions, negation, and conjunction are as usual, where it should be noted that in CTL they are interpreted over states, whereas in LTL they are interpreted over paths. state formula $\exists\varphi$ is valid in state s if and only if there exists some path starting in s that satisfies φ. In contrast, $\forall\varphi$ is valid in state s if and only if all paths starting in s satisfy φ. The semantics of the path formulae is identical (although formulated slightly more simply) to that for LTL.[1] For instance, $\exists\bigcirc\Phi$ is valid in state s if

[1] The semantics of the CTL path formulae is formulated more simply than for LTL, since in CTL each

and only if there exists some path π starting in s such that in the next state of this path, state $\pi[1]$, the property Φ holds. This is equivalent to the existence of a direct successor s' of s such that $s' \models \Phi$. $\forall(\Phi \,\mathsf{U}\, \Psi)$ is valid in state s if and only if every path starting in s has an initial finite prefix (possibly only containing s) such that Ψ holds in the last state of this prefix and Φ holds in all other states along the prefix. $\exists(\Phi \,\mathsf{U}\, \Psi)$ is valid in s if and only if there exists a path starting in s that satisfies $\Phi \,\mathsf{U}\, \Psi$. As for LTL, the semantics of CTL here is nonstrict in the sense that the path formula $\Phi \,\mathsf{U}\, \Psi$ is valid if the initial state of the path satisfies Ψ.

Definition 6.5. CTL Semantics for Transition Systems

Given a transition system TS as before, the *satisfaction set* $Sat_{TS}(\Phi)$, or briefly $Sat(\Phi)$, for CTL-state formula Φ is defined by:

$$Sat(\Phi) \;=\; \{\, s \in S \mid s \models \Phi \,\}.$$

The transition system TS satisfies CTL formula Φ if and only if Φ holds in all initial states of TS:

$$TS \models \Phi \quad \text{if and only if} \quad \forall s_0 \in I.\, s_0 \models \Phi \quad .$$

This is equivalent to $I \subseteq Sat(\Phi)$. ■

The semantics of the derived path operators "always" and "eventually" is similar to that in LTL. For path fragment $\pi = s_0\, s_1\, s_2 \ldots$:

$$\pi \models \Diamond\Phi \quad \text{if and only if} \quad s_j \models \Phi \text{ for some } j \geqslant 0.$$

From this it can be derived that:

$$s \models \exists\Box\Phi \quad \text{iff} \quad \exists\pi \in Paths(s).\, \pi[j] \models \Phi \text{ for all } j \geqslant 0,$$
$$s \models \forall\Box\Phi \quad \text{iff} \quad \forall\pi \in Paths(s).\, \pi[j] \models \Phi \text{ for all } j \geqslant 0.$$

Therefore, $\Box\Phi$ can be understood as CTL path formula with the semantics:

$$\pi = s_0\, s_1\, s_2 \ldots \models \Box\Phi \quad \text{if and only if} \quad s_j \models \Phi \text{ for all } j \geqslant 0.$$

In particular, $\forall\Box\Phi$ corresponds to the invariant over the invariant condition Φ.

In a similar way, one can derive that $\exists\Box\,\Phi$ is valid in state s if and only if there exists some path starting at s such that for each state on this path the formula Φ holds. The formula $\exists\Diamond\,\Phi$ is valid in state s if and only if Φ holds eventually along some path that starts in s, and $\forall\Diamond\,\Phi$ is valid if and only if this property holds for all paths that start in s. A schematic overview of the validity of $\exists\Box$, $\exists\Diamond$, $\forall\Diamond$, and $\forall\Box$ is given in Figure 6.2, where

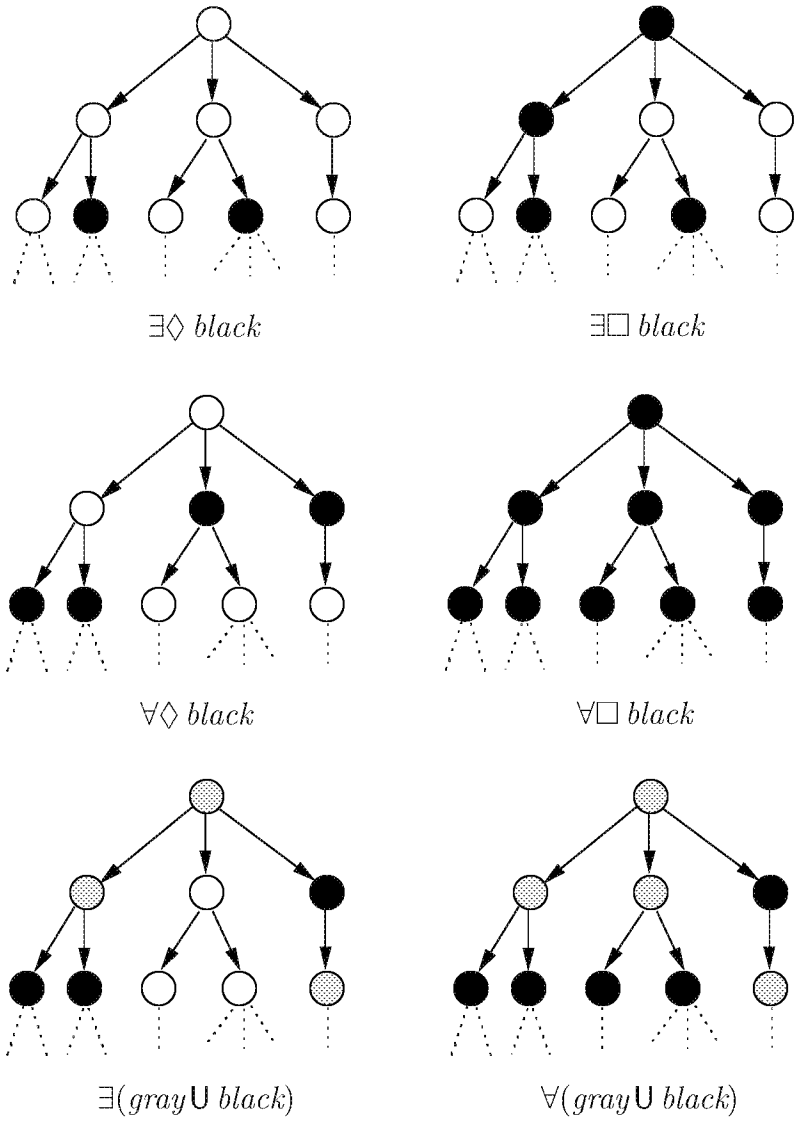


Figure 6.2: Visualization of semantics of some basic CTL formulae.

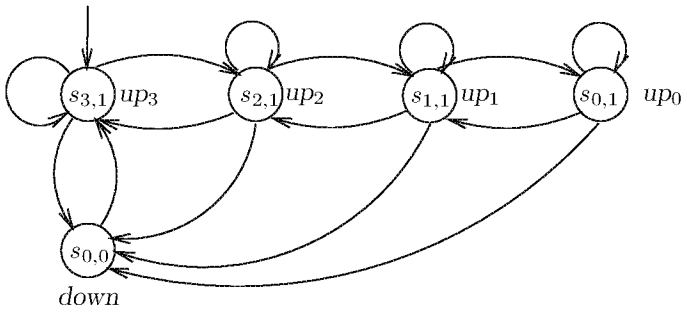


Figure 6.3: A transition system of the TMR system.

Property	*Formalization in CTL*
Possibly the system never goes down	$\exists\Box \neg \mathit{down}$
Invariantly the system never goes down	$\forall\Box \neg \mathit{down}$
It is always possible to start as new	$\forall\Box \exists\Diamond \mathit{up}_3$
The system always eventually goes down and is operational until going down	$\forall ((\mathit{up}_3 \vee \mathit{up}_2) \,\mathsf{U}\, \mathit{down})$

Table 6.2: Some properties for the TMR system and their formalization in CTL.

black-colored states satisfy the proposition *black*, gray states are labeled with *gray*, and all other states are labeled neither with *black* nor with *gray*.

Example 6.6. A Triple Modular Redundant System

Consider a triple modular redundant (TMR) system with three processors and a single voter. As each component of this system can fail, the reliability is increased by letting all processors execute the same program. The voter takes a majority vote of the outputs of the three processors. If a single processor fails, the system can still produce reliable outputs. Each component can be repaired. It is assumed that only one component at a time can fail and only one at a time can be repaired. On failure of the voter, the entire system fails. On repair of the voter, it is assumed that the system starts as being new, i.e., with three processors and a voter. The transition system of this TMR is depicted in Figure 6.3. States are of the form $s_{i,j}$ where i denotes the number of processors that is currently up ($0 < i \leqslant 3$) and j the number of operational voters ($j = 0, 1$). We consider the TMR system to be operational if at least two processors are functioning properly. Some interesting properties of this system and their formulation in CTL are listed in Table 6.2 on page 323. We consider each of the formulae in isolation:

temporal operator has to be immediately followed by a state formula.

- state formula $\exists\Box\,\neg\,down$ holds in state $s_{3,1}$, as there is a path, e.g., $(s_{3,1}\,s_{2,1})^\omega$, starting in that state and that never reaches the *down* state, i.e., $(s_{3,1}\,s_{2,1})^\omega \models \Box\,\neg\,down$.
- Formula $\forall\Box\,\neg\,down$, however, does not hold in state $s_{3,1}$, as there is a path starting from that state, such as $(s_{3,1})^+\,s_{0,0}\ldots$, that satisfies $\neg\Box\,\neg\,down$, or, equivalently, $\Diamond\,down$.
- Formula $\forall\Box\,\exists\Diamond\,up_3$ holds in state $s_{3,1}$, as in any state of any of its paths it is possible to return to the initial state, e.g., by first moving to state $s_{0,0}$ and then to $s_{3,1}$. For instance, the path $s_{3,1}\,(s_{2,1})^\omega \models \Box\exists\Diamond\,up_3$ since $s_{2,1} \models \exists\Diamond\,up_{3,1}$. This property should not be confused with the CTL formula $\forall\Diamond\,up_3$, which expresses that each path eventually will visit the initial state. (Note that this formula is trivially valid for state $s_{3,1}$ as it satisfies up_3.)
- The last property of Table 6.2 does not hold in state $s_{3,1}$ as there exists a path, such as $s_{3,1}\,s_{2,1}\,s_{1,1}\,s_{0,0}\ldots$, for which the path formula $(up_3 \vee up_2)\;\mathsf{U}\,down$ does not hold. The formula is refuted since the path visits state $s_{1,1}$, a state that satisfies neither *down* nor up_3 nor up_2.

■

Example 6.7. CTL Semantics

Consider the transition system depicted at the top (a) of Figure 6.4. Just below the transition system the validity of several CTL formulae is indicated for each state. (For simplicity, the initial states are not indicated.) A state is colored black if the formula is valid in that state; otherwise, it is white. The following formulae are considered:

- The formula $\exists\bigcirc a$ is valid for all states since all states have some direct successor state that satisfies a.
- $\forall\bigcirc a$ is not valid for state s_0, since a possible path starting at s_0 goes directly to state s_2 for which a does not hold. Since the other states have only direct successors for which a holds, $\forall\bigcirc a$ is valid for all other states.
- For all states except state s_2, it is possible to have a computation that leads to state s_3 (such as $s_0\,s_1\,s_3^\omega$ when starting in s_0) for which a is globally valid. Therefore, $\exists\Box\,a$ is valid in these states. Since $a \notin L(s_2)$ there is no path starting at s_2 for which a is globally valid.
- $\forall\Box\,a$ is only valid for s_3 since its only path, s_3^ω, always visits a state in which a holds. For all other states it is possible to have a path which contains s_2 that does not satisfy a. So for these states $\forall\Box\,a$ is not valid.

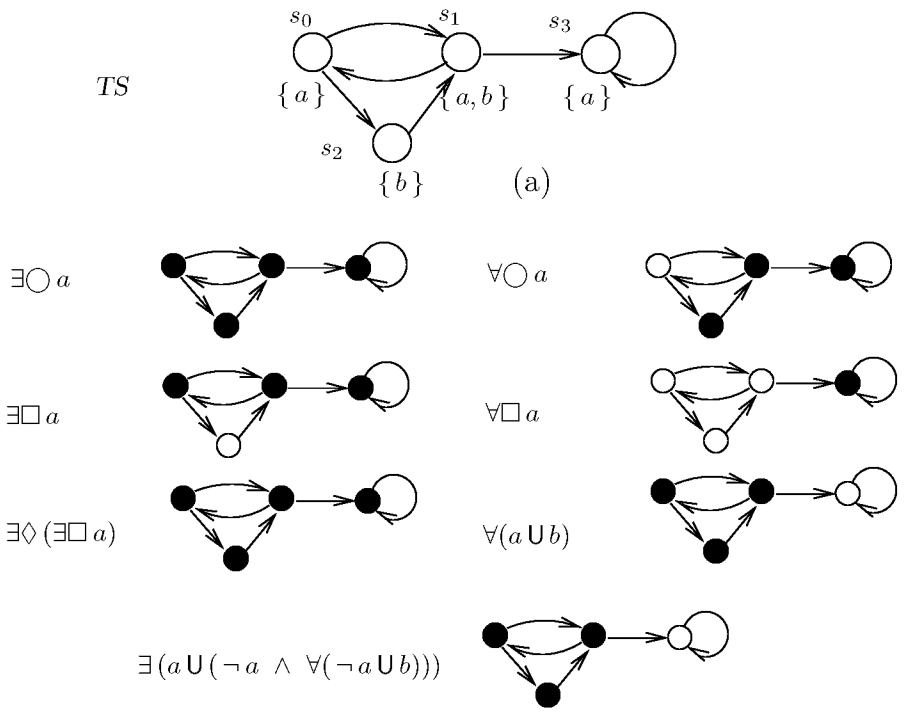


Figure 6.4: Interpretation of several CTL formulae.

- $\exists\Diamond(\exists\Box\, a)$ is valid for all states since from each state another state (either s_0, s_1, or s_3) can be eventually reached from which some computation can start along which a is globally valid.
- $\forall(a \,\mathsf{U}\, b)$ is not valid in s_3 since its only computation (s_3^ω) never reaches a state for which b holds. In state s_0 proposition a holds until b holds, and in states s_1 and s_2 proposition b holds immediately. So, for these states the formula is true.
- Finally, $\exists(a \,\mathsf{U}\, (\neg a \ \wedge\ \forall(\neg a \,\mathsf{U}\, b)))$ is not valid in s_3, since from s_3 a b-state can never be reached. For the states s_0 and s_1 the formula is valid, since state s_2 can be reached from these states via an a-path; $\neg a$ is valid in s_2, and from s_2 all possible paths satisfy $\neg a \,\mathsf{U}\, b$, since s_2 is a b-state. For instance, for state s_0 the path $(s_0\, s_2\, s_1)^\omega$ satisfies $a \,\mathsf{U}\, (\neg a \ \wedge\ \forall(\neg a \,\mathsf{U}\, b))$ since $a \in L(s_0)$, $a \notin L(s_2)$, and $b \in L(s_1)$. For state s_2 the property is valid since a is invalid in s_2 and for all paths starting at s_2 the first state is a b-state.

■

Remark 6.8. Infinitely Often

For a better comprehension of the semantics of CTL, let us prove that:

$$s \models \forall\Box\forall\Diamond a \quad \text{if and only if} \quad \forall\pi \in \mathit{Paths}(s).\, \pi[i] \models a \text{ for infinitely many } i\,.$$

$\Rightarrow$: Let s be a state, such that $s \models \forall\Box\,\forall\Diamond\, a$. The proof obligation is to show that every infinite path fragment π starting in s passes through an a-state infinitely often. Let $\pi = s_0\, s_1\, s_2 \ldots \in \mathit{Paths}(s)$ and $j \geqslant 0$. We demonstrate that there exists an index $i \geqslant j$ with $s_i \models a$. Since $s \models \forall\Box\forall\Diamond a$, we have

$$\pi \models \Box\forall\Diamond a.$$

In particular, $s_j \models \forall\Diamond a$. From $\pi[j..] = s_j\, s_{j+1} \ldots \in \mathit{Paths}(s_j)$ it follows that

$$s_j\, s_{j+1}\, s_{j+2} \ldots \models \Diamond a.$$

Thus, there exists an index $i \geqslant j$ with $s_i \models a$. As this reasoning applies to any index j, path π visits an a-state infinitely often.

$\Leftarrow$: Let s be a state such that every infinite path fragment starting in s visits infinitely many a-states. Let $\pi = s_0\, s_1\, s_2 \ldots \in \mathit{Paths}(s)$. To show that $s \models \forall\Box\forall\Diamond a$, it has to be proven that $\pi \models \Box\forall\Diamond a$, i.e.:

$$s_j \models \forall\Diamond a, \qquad \text{for any } j \geqslant 0.$$

Let $j \geqslant 0$ and $\pi' = s'_j\, s'_{j+1}\, s'_{j+2} \ldots \in \mathit{Paths}(s_j)$. To show that $s_j \models \forall\Diamond a$, it suffices to prove that π' visits at least one a-state. It is not difficult to infer that

$$\pi'' \;=\; \underbrace{s_0\, s_1\, s_2 \ldots\, s_j}_{\text{prefix of } \pi}\; \underbrace{s'_{j+1}\, s'_{j+2} \ldots}_{\pi' \in \mathit{Paths}(s_j)} \in \mathit{Paths}(s) \quad .$$

By assumption, π'' visits infinitely many a-states. In particular, there is an $i > j$ such that $s'_i \models a$. It now follows that

$$\pi' \;=\; s_j\, s'_{j+1} \ldots s'_{i-1}\, s'_i\, s'_{i+1}, \ldots \models \Diamond a$$

and, as this holds for any path $\pi' \in \mathit{Paths}(s_j)$, thus $s_j \models \forall\Diamond a$. This yields $\pi \models \Box\forall\Diamond a$ for all paths $\pi \in \mathit{Paths}(s)$. Thus, we have $s \models \forall\Box\forall\Diamond\, a$. ■

Remark 6.9. Weak Until

As for LTL (see Section 5.1.5), a slight variant of the until operator can be defined, viz. the weak-until operator, denoted W. The intuition behind this operator is that path π satisfies $\Phi\,\mathsf{W}\,\Psi$, for state formulae Φ and Ψ, if either $\Phi\,\mathsf{U}\,\Psi$ or $\Box\Phi$ holds. That is, the difference between until and weak until is that the latter does not require a Ψ-state to be reached eventually.

The weak-until operator in CTL cannot be defined directly starting from the LTL definition

$$\varphi\,\mathsf{W}\,\psi \;=\; \varphi\,\mathsf{U}\,\psi \;\vee\; \Box\varphi,$$

since $\exists(\varphi\,\mathsf{U}\,\psi \;\vee\; \Box\varphi)$ is not a syntactically correct CTL formula. However, using the LTL equivalence law $\varphi\,\mathsf{W}\,\psi \equiv \neg((\varphi \wedge \neg\psi)\,\mathsf{U}\,(\neg\varphi \wedge \neg\psi))$ and the duality between universal and existential quantification, the weak-until operator can be defined in CTL by

$$\begin{aligned} \exists(\Phi\,\mathsf{W}\,\Psi) &= \neg\forall((\Phi \wedge \neg\Psi)\,\mathsf{U}\,(\neg\Phi \wedge \neg\Psi)), \\ \forall(\Phi\,\mathsf{W}\,\Psi) &= \neg\exists((\Phi \wedge \neg\Psi)\,\mathsf{U}\,(\neg\Phi \wedge \neg\Psi)). \end{aligned}$$

Let us now check the semantics of $\exists\,\mathsf{W}$. From the above-defined duality, it follows that $s \models \exists(\Phi\,\mathsf{W}\,\Psi)$ if and only if there exists a path $\pi = s_0\, s_1\, s_2 \ldots$ that starts in s (i.e., $s_0 = s$) such that

$$\pi \not\models\; (\Phi \wedge \neg\Psi)\,\mathsf{U}\,(\neg\Phi \wedge \neg\Psi).$$

Such path exists if and only if

- either $s_j \models \Phi \wedge \neg\Psi$ for all $j \geqslant 0$, i.e., $\pi \models \Box(\Phi \wedge \neg\Psi)$, or
- there exists an index j such that

– $s_j \not\models \Phi \wedge \neg\Psi$ and $s_j \not\models \neg\Phi \wedge \neg\Psi$, i.e., $s_j \models \Psi$, and
– $s_i \models \Phi \wedge \neg\Psi$ for all $0 \leqslant i < j$.

This is equivalent to $\pi \models \Phi \,\mathsf{U}\, \Psi$.

Gathering these results yields

$$\begin{aligned} \pi \models \Phi \,\mathsf{W}\, \Psi \quad & \text{if and only if} \quad \pi \models \Phi \,\mathsf{U}\, \Psi \text{ or } \pi \models \Box(\Phi \wedge \neg\Psi), \\ & \text{if and only if} \quad \pi \models \Phi \,\mathsf{U}\, \Psi \text{ or } \pi \models \Box\Phi. \end{aligned}$$

Thus, the CTL formula $\exists(\Phi \,\mathsf{W}\, \Psi)$ is equivalent to $\exists(\Phi \,\mathsf{U}\, \Psi) \vee \exists\Box\Phi$. In the same way, one can check that the meaning of $\forall(\Phi \,\mathsf{W}\, \Psi)$ is as expected, i.e., $s \models \forall(\Phi \,\mathsf{W}\, \Psi)$ if and only if all paths starting in s fulfill $\Phi \,\mathsf{W}\, \Psi$ according to the LTL semantics of W. ■

Remark 6.10. The Semantics of Negation

For state s, we have $s \not\models \Phi$ if and only if $s \models \neg\Phi$. This, however, does not hold in general for transition systems. That is to say, it is possible that the statements $TS \not\models \Phi$ and $TS \not\models \neg\Phi$ *both hold*. This stems from the fact that there might be two initial states, s_0 and s_0', say, such that $s_0 \models \Phi$ and $s_0' \not\models \Phi$. Furthermore:

$$TS \not\models \neg\exists\varphi \;\text{ iff }\; \text{there exists a path } \pi \in \mathit{Paths}(TS) \text{ with } \pi \models \varphi.$$

This—at first glance surprising—equivalence is justified by the fact that the interpretation of CTL state formulae over transition systems is based on a universal quantification over the initial states. The statement $TS \not\models \neg\exists\varphi$ thus holds if and only if there exists an initial state $s_0 \in I$ with $s_0 \not\models \neg\exists\varphi$, i.e., $s_0 \models \exists\varphi$. On the other hand, $TS \models \exists\varphi$ requires that $s_0 \models \exists\varphi$ for *all* $s_0 \in I$. Consider the following transition system:

It follows that $s_0 \models \exists\Box\, a$, whereas $s_0' \not\models \exists\Box\, a$. Accordingly, $TS \not\models \neg\exists\Box\, a$ and $TS \not\models \exists\Box\, a$. ■

The semantics of CTL has been defined for a transition system without terminal states. This has the (technically) pleasant effect that all paths are infinite and simplifies the

definition of $\models$ for paths. In the following remark it is shown how to adapt the path semantics in case transition systems are considered with terminal states, i.e., when finite paths are possible.

Remark 6.11. CTL Semantics for Transition Systems with Terminal States

For finite maximal path fragment $\pi = s_0\, s_1\, s_2 \ldots s_n$ of length n, i.e., s_n is a terminal state, let

$$\begin{array}{lcll} \pi & \models & \bigcirc \Phi & \text{iff } n > 0 \text{ and } s_1 \models \Phi, \\ \pi & \models & \Phi \,\mathsf{U}\, \Psi & \text{iff there exists an index } j \in \mathbb{N} \text{ with } j \leqslant n, \text{ and} \\ & & & \quad s_i \models \Phi, \text{ for } i = 0, 1, \ldots, j-1, \text{ and } s_j \models \Psi. \end{array}$$

Then, $s \models \forall\bigcirc$ false if and only if s is a terminal state. For the derived operators $\Diamond$ and $\Box$ we obtain

$$\begin{array}{lcll} \pi & \models & \Diamond\Phi & \text{iff there exists an index } j \leqslant n \text{ with } s_j \models \Phi, \\ \pi & \models & \Box\Phi & \text{iff for all } j \in \mathbb{N} \text{ with } j \leqslant n \text{ we have } s_j \models \Phi. \end{array}$$

■

6.2.3 Equivalence of CTL Formulae

CTL formulae Φ and Ψ are called equivalent whenever they are semantically identical, i.e., when for any state s it holds that[2] $s \models \Phi$ if and only if $s \models \Psi$.

Definition 6.12. Equivalence of CTL Formulae

CTL formulae Φ and Ψ (over AP) are called *equivalent*, denoted $\Phi \equiv \Psi$, if $Sat(\Phi) = Sat(\Psi)$ for all transition systems TS over AP. ■

Accordingly, $\Phi \equiv \Psi$ if and only if for any transition system TS we have:

$$TS \models \Phi \quad \text{if and only if} \quad TS \models \Psi \quad .$$

Besides the standard equivalence laws for the propositional logic fragment of CTL, there exist a number of equivalence rules for temporal modalities in CTL. An important set of equivalence laws is indicated in Figure 6.5. To understand the expansion laws, let us reconsider the expansion law for the until operator in LTL:

$$\varphi \,\mathsf{U}\, \psi \;\equiv\; \psi \;\vee\; (\varphi \;\wedge\; \bigcirc\, (\varphi \,\mathsf{U}\, \psi)).$$

[2] Recall that the notion CTL formula is used for a CTL state formula.

duality laws for path quantifiers

$$\forall\bigcirc\Phi \equiv \neg\exists\bigcirc\neg\Phi \qquad \exists\bigcirc\Phi \equiv \neg\forall\bigcirc\neg\Phi$$

$$\forall\Diamond\Phi \equiv \neg\exists\Box\neg\Phi \qquad \exists\Diamond\Phi \equiv \neg\forall\Box\neg\Phi$$

$$\begin{aligned}\forall(\Phi\,\mathsf{U}\,\Psi) &\equiv \neg\exists(\neg\Psi\,\mathsf{U}\,(\neg\Phi\wedge\neg\Psi)) \;\wedge\; \neg\exists\Box\neg\Psi\\ &\equiv \neg\exists((\Phi\wedge\neg\Psi)\,\mathsf{U}\,(\neg\Phi\wedge\neg\Psi)) \;\wedge\; \neg\exists\Box(\Phi\wedge\neg\Psi)\\ &\equiv \neg\exists((\Phi\wedge\neg\Psi)\,\mathsf{W}\,(\neg\Phi\wedge\neg\Psi))\end{aligned}$$

expansion laws

$$\begin{aligned}\forall(\Phi\,\mathsf{U}\,\Psi) &\equiv \Psi \vee (\Phi \wedge \forall\bigcirc\forall(\Phi\,\mathsf{U}\,\Psi))\\ \forall\Diamond\Phi &\equiv \Phi \vee \forall\bigcirc\forall\Diamond\Phi\\ \forall\Box\Phi &\equiv \Phi \wedge \forall\bigcirc\forall\Box\Phi\end{aligned}$$

$$\begin{aligned}\exists(\Phi\,\mathsf{U}\,\Psi) &\equiv \Psi \vee (\Phi \wedge \exists\bigcirc\exists(\Phi\,\mathsf{U}\,\Psi))\\ \exists\Diamond\Phi &\equiv \Phi \vee \exists\bigcirc\exists\Diamond\Phi\\ \exists\Box\Phi &\equiv \Phi \wedge \exists\bigcirc\exists\Box\Phi\end{aligned}$$

distributive laws

$$\forall\Box(\Phi\wedge\Psi) \equiv \forall\Box\Phi \wedge \forall\Box\Psi$$

$$\exists\Diamond(\Phi\vee\Psi) \equiv \exists\Diamond\Phi \vee \exists\Diamond\Psi$$

Figure 6.5: Some equivalence rules for CTL.

In CTL, similar expansion laws for $\exists(\Phi \,\mathsf{U}\, \Psi)$ and $\forall(\Phi \,\mathsf{U}\, \Psi)$ exist. For instance, we have that $\exists(\Phi \,\mathsf{U}\, \Psi)$ is equivalent to the fact that the current state either satisfies Ψ, or it satisfies Φ, and for *some* direct successor state, $\exists(\Phi \,\mathsf{U}\, \Psi)$ holds. The expansion laws for $\exists\Diamond\,\Phi$ and $\exists\Box\,\Phi$ can be simply derived from the expansion laws for $\exists\,\mathsf{U}$. The basic idea behind these laws—as for LTL—is to express the validity of a formula by a statement about the current state (without the need to use temporal operators) and a statement about the direct successors of this state (using either $\exists\bigcirc$ or $\forall\bigcirc$ depending on whether an existential or a universally quantified formula is treated). For instance, $\exists\Box\,\Phi$ is valid in state s if Φ is valid in s (a statement about the current state) and Φ holds for all states along some path starting at s (a statement about the successor states).

Not any law in LTL can be easily lifted to CTL. Consider, for example, the following statement:

$$\Diamond\,(\varphi \;\vee\; \psi) \;\equiv\; \Diamond\,\varphi \;\vee\; \Diamond\,\psi,$$

which is valid for any path. The same is true for:

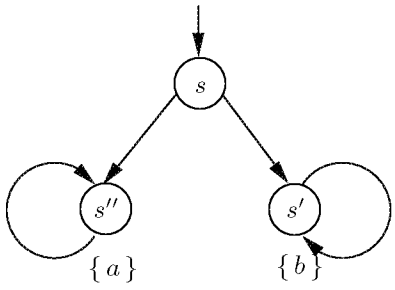
$$\exists\Diamond\,(\Phi \;\vee\; \Psi) \;\equiv\; \exists\Diamond\,\Phi \;\vee\; \exists\Diamond\,\Psi.$$

This can be seen as follows.

$\Leftarrow$: Assume that $s \models \exists\Diamond\,\Phi \;\vee\; \exists\Diamond\,\Psi$. Then, without loss of generality, we may assume that $s \models \exists\Diamond\,\Phi$. This means that there is some state s' (possibly $s = s'$), reachable from state s, such that $s' \models \Phi$. But then $s' \models \Phi \;\vee\; \Psi$. This means that there exists a reachable state from s which satisfies $\Phi \;\vee\; \Psi$. By the semantics of CTL it now follows that $s \models \exists\Diamond\,(\Phi \;\vee\; \Psi)$.

$\Rightarrow$: Let s be an arbitrary state such that $s \models \exists\Diamond\,(\Phi \;\vee\; \Psi)$. Then there exists a state s' (possibly $s = s'$) such that $s' \models \Phi \;\vee\; \Psi$. Without loss of generality we may assume that $s' \models \Phi$. But then we can conclude that $s \models \exists\Diamond\,\Phi$, as s' is reachable from s. Therefore we also have $s \models \exists\Diamond\,\Phi \;\vee\; \exists\Diamond\,\Psi$.

However, $\forall\Diamond\,(\Phi \;\vee\; \Psi) \not\equiv \forall\Diamond\,\Phi \;\vee\; \forall\Diamond\,\Psi$ since $\forall\Diamond\,(\Phi \;\vee\; \Psi) \;\Rightarrow\; \forall\Diamond\,\Phi \;\vee\; \forall\Diamond\,\Psi$ is invalid as shown by the following transition system:



For each path that starts in state s we have that $\Diamond\,(a \vee b)$ holds, so $s \models \forall\Diamond\,(a \vee b)$. This follows directly from the fact that each path visits either state s' or state s'' eventually, and $s' \models a \vee b$ and the same applies to s''. However, state s does not satisfy $\forall\Diamond\, a \vee \forall\Diamond\, b$. For instance, path $s\,(s'')^{\omega} \models \Diamond\, a$ but $s\,(s'')^{\omega} \not\models \Diamond\, b$. Thus, $s \not\models \forall\Diamond\, b$. By a similar reasoning applied to path $s\,(s')^{\omega}$ it follows that $s \not\models \forall\Diamond\, a$. Thus, $s \not\models \forall\Diamond\, a \vee \forall\Diamond\, b$. Stated in words, not all computations that start in state s eventually reach an a-state nor do they all eventually reach a b-state.

6.2.4 Normal Forms for CTL

The duality law for $\forall\bigcirc\,\Phi$ shows that $\forall\bigcirc$ can be treated as a derived operator of $\exists\bigcirc$. That is to say, the basic operators $\exists\bigcirc$, $\exists\,\mathsf{U}$, and $\forall\,\mathsf{U}$ would have been sufficient to define the syntax of CTL. The following theorem demonstrates that we can even omit the universal path quantifier and define all temporal modalities in CTL using the basic operators $\exists\bigcirc$, $\exists\,\mathsf{U}$, and $\exists\Box$.

Definition 6.13. Existential Normal Form (for CTL)

For $a \in AP$, the set of CTL state formulae in *existential normal form* (ENF, for short) is given by

$$\Phi \quad ::= \quad \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \exists\bigcirc\,\Phi \;\Big|\; \exists(\Phi_1\,\mathsf{U}\,\Phi_2) \;\Big|\; \exists\Box\,\Phi.$$

■

Theorem 6.14. Existential Normal Form for CTL

For each CTL formula there exists an equivalent CTL formula in ENF.

Proof: The following duality laws allow elimination of the universal path quantifier and thus provide a translation of CTL formulae into equivalent ENF formulae:

$$\begin{aligned} \forall \bigcirc \Phi &\equiv \neg \exists \bigcirc \neg \Phi, \\ \forall (\Phi \,\mathsf{U}\, \Psi) &\equiv \neg \exists (\neg \Psi \,\mathsf{U}\, (\neg \Phi \wedge \neg \Psi)) \;\wedge\; \neg \exists \Box \neg \Psi. \end{aligned}$$

■

Recall that the basis syntax of CTL only uses $\exists \bigcirc$, $\exists \,\mathsf{U}$ and $\forall \bigcirc$ and $\forall \,\mathsf{U}$. Thus, the two rules used in the proof of Theorem 6.14 allow the removal of all universal quantifiers from a given CTL formula. However, when implementing the translation from CTL formulae to ENF formulae one might use analogous rules for the derived operators, such as

$$\begin{aligned} \forall \Diamond \Phi &\equiv \neg \exists \Box \neg \Phi, \\ \forall \Box \Phi &\equiv \neg \exists \Diamond \neg \Phi \;=\; \neg \exists (\mathrm{true} \,\mathsf{U}\, \Phi). \end{aligned}$$

Since the rewrite rule for $\forall \,\mathsf{U}$ triples the occurrences of the right formula Ψ, the translation from CTL to ENF can cause an exponential blowup.

Another normal form of importance is the *positive normal form*. A CTL formula is said to be in positive normal form (PNF, for short) whenever negations only occur adjacent to atomic propositions. That is, e.g., $\neg \forall (a \,\mathsf{U}\, \neg b)$ is not in PNF, whereas $\exists (\neg a \wedge \neg b \,\mathsf{U}\, a)$ is in PNF. To ensure that every CTL formula is equivalent to a formula in PNF, for each operator a dual operator is necessary. We have that conjunction and disjunction are dual, and that $\bigcirc$ is dual to itself. As for LTL, we adopt the weak until operator W as a dual operator of U.

Definition 6.15. Positive Normal Form (for CTL)

The set of CTL state formulae in *positive normal form* (PNF, for short) is given by

$$\Phi \;::=\; \mathrm{true} \;\Big|\; \mathrm{false} \;\Big|\; a \;\Big|\; \neg a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \Phi_1 \;\vee\; \Phi_2 \;\Big|\; \exists \varphi \;\Big|\; \forall \varphi$$

where $a \in AP$ and the path formulae are given by

$$\varphi \;::=\; \bigcirc \Phi \;\Big|\; \Phi_1 \,\mathsf{U}\, \Phi_2 \;\Big|\; \Phi_1 \,\mathsf{W}\, \Phi_2.$$

■

Theorem 6.16. Existence of Equivalent PNF Formulae

For each CTL formula there exists an equivalent CTL formula in PNF.

Proof: Any CTL formula can be transformed into PNF by successively "pushing" negations "inside" the formula at hand. This is facilitated by the following equivalence laws:

$$\begin{array}{lcl}
\neg\text{true} & \equiv & \text{false} \\
\neg\neg\Phi & \equiv & \Phi \\
\neg(\Phi \wedge \Psi) & \equiv & \neg\Phi \ \vee \ \neg\Psi \\
\neg\forall\bigcirc \Phi & \equiv & \exists\bigcirc \neg\Phi \\
\neg\exists\bigcirc \Phi & \equiv & \forall\bigcirc \neg\Phi \\
\neg\forall(\Phi \,\mathsf{U}\, \Psi) & \equiv & \exists((\Phi \wedge \neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)) \\
\neg\exists(\Phi \,\mathsf{U}\, \Psi) & \equiv & \forall((\Phi \wedge \neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)).
\end{array}$$

■

Due to the fact that in the rules for $\forall\,\mathsf{U}$ and $\exists\,\mathsf{U}$ the number of occurrences of Ψ (and Φ) is doubled, the length of an equivalent CTL formula may be exponentially longer than the original CTL formula. [3] The same phenomenon appeared in defining the PNF for LTL when using the weak-until operator. As for LTL, this exponential blowup can be avoided by using the release operator, which in CTL can be defined by: $\exists(\Phi \,\mathsf{R}\, \Psi) = \neg\forall((\neg\Phi) \,\mathsf{U}\, (\neg\Psi))$ and $\forall(\Phi \,\mathsf{R}\, \Psi) = \neg\exists((\neg\Phi) \,\mathsf{U}\, (\neg\Psi))$.

6.3 Expressiveness of CTL vs. LTL

Although many relevant properties of reactive systems can be specified in LTL and CTL, the logics CTL and LTL are incomparable according to their expressiveness. More precisely, there are properties that one can express in CTL, but that cannot be expressed in LTL, and vice versa.

Let us first define what it means for CTL and LTL formulae to be equivalent. Intuitively speaking, equivalent means "express the same thing". More precisely:

Definition 6.17. Equivalence between CTL- and LTL Formulae

CTL formula Φ and LTL formula φ (both over AP) are *equivalent*, denoted $\Phi \equiv \varphi$, if for any transition system TS over AP:

$$TS \models \Phi \quad \text{if and only if} \quad TS \models \varphi.$$

[3] Although the rewrite rules for $\neg\forall\,\mathsf{U}$ and $\neg\exists\,\mathsf{U}$ could be simplified by $\neg\forall(\Phi \,\mathsf{U}\, \Psi) \equiv \exists\Big((\neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)\Big)$ and $\neg\exists(\Phi \,\mathsf{U}\, \Psi) \equiv \forall\Big((\neg\Psi) \,\mathsf{W}\, (\neg\Phi \wedge \neg\Psi)\Big)$, there is still a duplication of formula Ψ.

■

The LTL formula φ holds in state s of a transition system whenever all paths starting in s satisfy φ. Given this (semantical) universal quantification over paths, it seems natural that, e.g., the LTL formula $\Diamond a$ is equivalent to the CTL formula $\forall\Diamond a$. This seems to suggest that for a given CTL formula, an equivalent LTL formula is obtained by simply omitting all universal path quantifiers (as these are implicit in LTL). The following result by Clarke and Draghicescu [85] (for which the proof is omitted) shows that dropping all (universal and existential) quantifiers is a safe way to generate an equivalent LTL formula, provided there are equivalent LTL formulae:

Theorem 6.18. Criterion for Transforming CTL Formulae into Equivalent LTL Formulae

Let Φ be a CTL formula, and φ the LTL formula that is obtained by eliminating all path quantifiers in Φ. Then:

$$\Phi \equiv \varphi \text{ or there does not exist any LTL formula that is equivalent to } \Phi.$$

For the following CTL formulae, an equivalent LTL formula is obtained by simply omitting all path quantifiers: a, $\forall\bigcirc a$, $\forall(a \,\mathsf{U}\, b)$, $\forall\Diamond a$, $\forall\Box a$, and $\forall\Box\forall\Diamond a$. The fact that the CTL formula $\forall\Box\forall\Diamond a$ is equivalent to the LTL formula $\Box\Diamond a$ has been established earlier in Remark 6.8 (326). However, $\forall\Diamond\forall\Box a$ and $\Diamond\Box a$ are not equivalent. The LTL formula $\Diamond\Box a$ ensures that a will eventually forever (i.e., continuously from some point on) hold. The semantics of $\forall\Diamond\forall\Box a$ is different, however. The CTL formula $\forall\Diamond\forall\Box a$ asserts that on any computation, eventually some state, s say, is reached such that $s \models \forall\Box a$. Note that

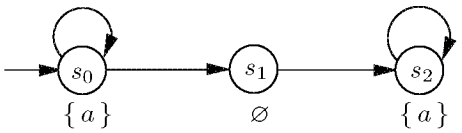
$$s \models \forall\Diamond \underbrace{\forall\Box a}_{\Phi}$$

if and only if for any path $\pi = s_0\, s_1\, s_2 \ldots \in \mathit{Paths}(s)$, $s_j \models \Phi$ for some j. For $\Phi = \forall\Box a$, this entails that for any such path π there is some state s_j such that all reachable states from s_j satisfy the atomic proposition a.

Lemma 6.19. Persistence

The CTL formula $\forall\Diamond\forall\Box a$ and the LTL formula $\Diamond\Box a$ are not *equivalent.*

Proof: Consider the following transition system TS over $AP = \{a\}$:



The initial state s_0 satisfies the LTL formula $\Diamond\Box a$, since each path starting in s_0 eventually remains forever in one of the two states s_0 or s_2, which are both labeled with a. The CTL formula $\forall\Diamond\forall\Box a$, however, does not hold in s_0, since we have $s_0^\omega \not\models \Diamond\forall\Box a$ (as $s_0 \not\models \forall\Box a$). This is due to the fact that the path $s_0^* s_1 s_2^\omega$ passes through the $\neg a$-state s_1. Thus, s_0^ω is a path starting in s_0 which will never reach a state satisfying $\forall\Box a$, i.e., $s_0^\omega \not\models \Diamond\forall\Box a$. Accordingly, it follows that

$$s_0 \not\models \forall\Diamond\forall\Box a.$$

■

Given that the CTL formulae $\forall\Diamond\forall\Box a$ and the LTL formula $\Diamond\Box a$ are not equivalent and the fact that $\Diamond\Box a$ is obtained from $\forall\Diamond\forall\Box a$ by eliminating the universal path quantifiers, it follows from Theorem 6.18 that there does not exist an LTL formula that is equivalent to $\forall\Diamond\forall\Box a$. In a similar way, it can be shown that the CTL formulae $\forall\Diamond(a \wedge \forall\bigcirc a)$ and $\Diamond(a \wedge \bigcirc a)$ are not equivalent, and thus, the requirement $\forall\Diamond(a \wedge \forall\bigcirc a)$ cannot be expressed in LTL.

Lemma 6.20. Eventually an a-State with only direct a-Successors

The CTL formula $\forall\Diamond(a \wedge \forall\bigcirc a)$ and the LTL formula $\Diamond(a \wedge \bigcirc a)$ are not *equivalent.*

Proof: Consider the transition system depicted in Figure 6.6. All paths that start in the initial state s_0 have either as prefix the path fragment $s_0 s_1$ or $s_0 s_3 s_4$. Clearly, all such paths satisfy the LTL formula $\Diamond(a \wedge \bigcirc a)$, and so, $s_0 \models \Diamond(a \wedge \bigcirc a)$. On the other hand, however, $s_0 \not\models \forall\Diamond(a \wedge \forall\bigcirc a)$ as the path $s_0 s_1 (s_2)^\omega$ does not satisfy $\Diamond(a \wedge \forall\bigcirc a)$. This follows from the fact that state s_0 has the non-a-state s_3 as direct successor, i.e., $s_0 \not\models a \wedge \forall\bigcirc a$. ■

These examples show that certain requirements that can be expressed in CTL, cannot be expressed in LTL. The following theorem provides, in addition, some examples of LTL formulae for which no equivalent CTL formula exists. This establishes that the expressiveness of the temporal logics LTL and CTL are incomparable.

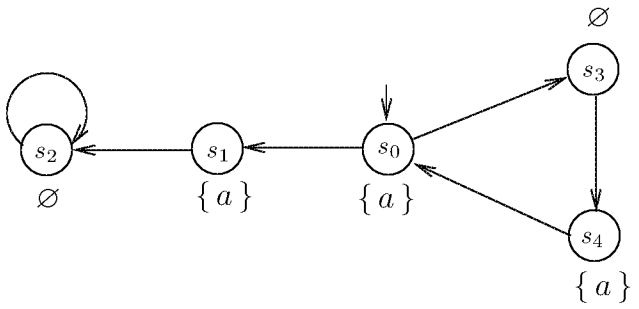


Figure 6.6: Transition system for $\forall\Diamond\,(a \wedge \forall\bigcirc\, a)$.

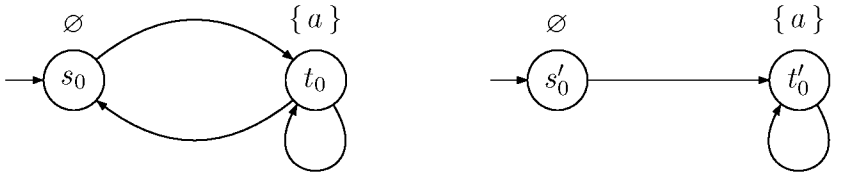


Figure 6.7: The base transition systems: TS_0 (left) and TS'_0 (right).

Theorem 6.21. Incomparable Expressiveness of CTL and LTL

(a) There exist LTL formulae for which no equivalent CTL formula exists. This holds for, for instance

$$\Diamond\Box a \qquad \textit{or} \qquad \Diamond(a \wedge \bigcirc\, a).$$

(b) There exist CTL formulae for which no equivalent LTL formula exists. This holds for, for instance

$$\forall\Diamond\,\forall\Box\, a \quad \textit{and} \quad \forall\Diamond\,(a \wedge \forall\bigcirc\, a) \quad \textit{and} \quad \forall\Box\,\exists\Diamond\, a.$$

Proof:

(a) Consider the formula $\Diamond\,\Box\, a$. The proof for $\Diamond\,(a \wedge \bigcirc\, a)$ goes along similar lines and is omitted here. Consider the two series of transition systems $TS_0, TS_1, TS_2, \ldots$ and $TS'_0, TS'_1, TS'_2, \ldots$ that are inductively defined as follows (see Figures 6.7 and 6.8).

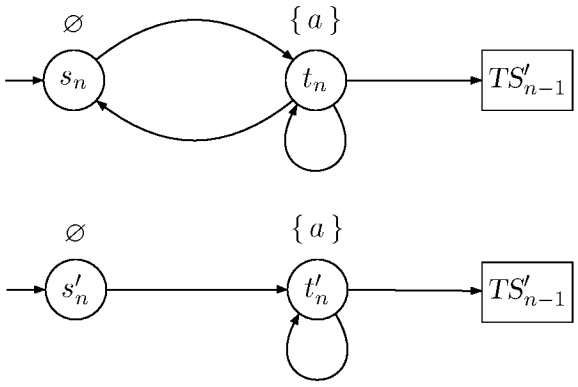


Figure 6.8: Inductive construction of TS_n (upper) and TS'_n (lower).

For all transition systems, $AP = \{ a \}$, and the action labels are not important. Let, for $n \geqslant 0$,

$$TS_n = (S_n, \{ \tau \}, \rightarrow_n, \{ s_n \}, \{ a \}, L_n)$$

and

$$TS_n = (S'_n, \{ \tau \}, \rightarrow'_n, \{ s'_n \}, \{ a \}, L'_n)$$

where $S_0 = \{ s_0, t_0 \}$, $S'_0 = \{ s'_0, t'_0 \}$, and for $n > 0$:

$$S_n \;=\; S'_{n-1} \cup \{ s_n, t_n \} \quad \text{and} \quad S'_n \;=\; S'_{n-1} \cup \{ s'_n, t'_n \}.$$

The labeling functions are defined such that all states t_i are labeled with $\{ a \}$ and all states s_i are labeled with $\varnothing$. Thus, $L_0(s_0) = \varnothing$ and $L_0(t_0) = \{ a \}$, and for $n > 0$, the labels of all states in TS'_{n-1} remain the same and are extended with

$$L_n(s_n) = L'_n(s'_n) = \varnothing \quad \text{and} \quad L_n(t_n) = L'_n(t'_n) = \{ a \}.$$

Finally, the transition relations $\rightarrow_n$ and $\rightarrow'_n$ contain $\rightarrow'_{n-1}$ (where $\rightarrow_{-1} = \varnothing$), as well as the transitions

$$\begin{aligned} TS_n: &\quad s_n \rightarrow_n t_n, \quad t_n \rightarrow_n t_n, \quad t_n \rightarrow_n s'_{n-1}, \quad t_n \rightarrow_n s_n \\ TS'_n: &\quad s'_n \rightarrow'_n t'_n, \quad t'_n \rightarrow'_n t'_n, \quad t'_n \rightarrow'_n s'_{n-1} \end{aligned}$$

where the action labels are omitted for simplicity.

Thus, the only difference between TS_n and TS'_n is the fact that TS_n includes the edge $t_n \rightarrow s_n$, whereas this edge is absent in TS'_n. Some example instantiations of TS_n and TS'_n are indicated in Figure 6.9.

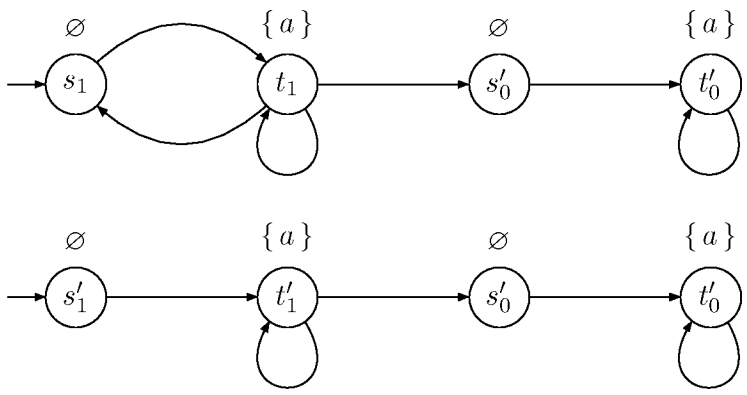


Figure 6.9: The transition systems TS_1 (upper) and TS'_1 (lower).

It follows from the construction of TS_n and TS'_n that

$$TS_n \not\models \Diamond\Box a \quad \text{and} \quad TS'_n \models \Diamond\Box a \quad \text{for all } n \geqslant 0.$$

This can be proven as follows. TS_n contains the initial path $(s_n\, t_n)^\omega$ that visits s_n and t_n in an alternating fashion. We have that

$$\mathit{trace}((s_n\, t_n)^\omega) \;=\; \varnothing\,\{\,a\,\}\,\varnothing\,\{\,a\,\}\,\varnothing\ldots \quad \text{thus} \quad \mathit{trace}((s_n\, t_n)^\omega) \not\models \Diamond\Box\, a.$$

As the considered path is initial, it follows that $TS_n \not\models \Diamond\Box\, a$. On the other hand, as TS'_n has no opportunity to infinitely often return to an $\neg a$-state, each initial path in TS'_n is of the following form:

$$\pi'_i \;=\; s'_n\, t'_n \ldots s'_i\, (t'_i)^\omega$$

for some i. Due to the fact that

$$\mathit{trace}(\pi'_i) \;=\; \varnothing\,\{\,a\,\}\,\varnothing\ldots\varnothing\,(\{\,a\,\})^\omega$$

it follows that $\mathit{trace}(\pi'_i) \models \Diamond\Box\, a$. As this applies to any initial path of TS'_n, we have $TS'_n \models \Diamond\Box\, a$.

By means of induction on n, it can be proven that TS_n and TS'_n cannot be distinguished by any CTL formula of length at most n. That is to say, for all $n \geqslant 0$,

$$\forall\text{CTL formula } \Phi \text{ with } |\Phi| \leqslant n:\; TS_n \models \Phi \quad \text{if and only if} \quad TS'_n \models \Phi.$$

(The proof of this fact is left to the interested reader.)

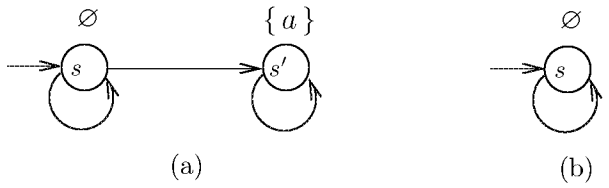


Figure 6.10: Two transition systems for $\forall\Box\,\exists\Diamond\, a$.

The final step of the proof is now as follows. Assume there is a CTL formula Φ that is equivalent to $\Diamond\Box\, a$. Let $n = |\Phi|$ be the length of the formula Φ and

$$TS \;=\; TS_n, \quad TS' \;=\; TS'_n.$$

On the one hand, it follows from $TS \not\models \Diamond\Box\, a$ and $TS' \models \Diamond\Box\, a$ that

$$TS \not\models \Phi \text{ and } TS' \models \Phi.$$

On the other hand, it results from the fact that TS_n and TS'_n cannot be distinguished by a CTL formula (of length at most n) that Φ has the same truth-value under TS and under TS'. This yields a contradiction.

(b) We concentrate on the proof that there does not exist an equivalent formulation of the CTL formula $\forall\Box\,\exists\Diamond\, a$ in LTL. The fact that there do not exist equivalent LTL formulations of the CTL formulae $\forall\Diamond\,\forall\Box\, a$ and $\forall\Diamond\,(a \wedge \forall\bigcirc\, a)$ follows directly from Lemmas 6.19 and 6.20 respectively, and Theorem 6.18. The proof for $\forall\Box\,\exists\Diamond\, a$ is by contraposition. Let φ be an LTL formula that is equivalent to $\forall\Box\,\exists\Diamond\, a$. Consider the transition system TS depicted in Figure 6.10(a). As $TS \models \forall\Box\,\exists\Diamond\, a$, and $\varphi \equiv \forall\Diamond\,\exists\Diamond\, a$, it follows that $TS \models \varphi$, and therefore

$$\mathit{Traces}(TS) \;\subseteq\; \mathit{Words}(\varphi).$$

Since $\pi = s^\omega$ is a path in TS, it follows that

$$\mathit{trace}(\pi) \;=\; \varnothing\,\varnothing\,\varnothing\,\varnothing\ldots \in \mathit{Traces}(TS) \;\subseteq\; \mathit{Words}(\varphi).$$

Now consider the transition system TS' depicted in Figure 6.10(b). Note that TS' is part of transition system TS. The paths starting in s in TS' are also paths starting from s in TS, so we have $s \models \varphi$ in TS' and thus $TS' \models \varphi$. However, $s \not\models \forall\Box\,\exists\Diamond\, a$, since $\exists\Diamond\, a$ is never valid along the only path s^ω, and thus $TS' \not\models \forall\Box\,\exists\Diamond\, a$. This is a contradiction.

■

Note that the CTL formula $\forall\Box\,\exists\Diamond\, a$ is of significant practical use, since it expresses the fact that it is possible to reach a state for which a holds irrespective of the current state. If a characterizes a state where a certain error is repaired, the formula expresses the fact that it is always possible to recover from that error.

6.4 CTL Model Checking

This section is concerned with CTL model checking, which means a decision algorithm that checks whether $TS \models \Phi$ for a given transition system TS and CTL formula Φ. Throughout this section, it is assumed that TS is finite, and has no terminal states. We will see that CTL-model checking can be performed by a recursive procedure that calculates the satisfaction set for all subformulae of Φ and finally checks whether all initial states belong to the satisfaction set of Φ.

Throughout this section, we consider CTL formulae in ENF, i.e., CTL formulae built by the basic modalities $\exists\bigcirc$, $\exists\,\mathsf{U}$, and $\exists\Box$. This requires algorithms that generate $Sat(\exists\bigcirc\,\Phi)$, $Sat(\exists(\Phi\,\mathsf{U}\,\Psi))$, and $Sat(\exists\Box\,\Phi)$ when $Sat(\Phi)$ and $Sat(\Psi)$ are given. Although each CTL formula can be transformed algorithmically into an equivalent ENF formula, for an implementation of the CTL model-checking algorithm it is recommended to use similar techniques to handle universal quantification, i.e., to design algorithms that generate $Sat(\forall\bigcirc\,\Phi)$, $Sat(\forall(\Phi\,\mathsf{U}\,\Psi))$, and $Sat(\forall\Box\,\Phi)$ directly from $Sat(\Phi)$ and $Sat(\Psi)$. (The same holds for other derived operators such as W or R.)

6.4.1 Basic Algorithm

The model-checking problem for CTL is to verify for a given transition system TS and CTL formula Φ whether $TS \models \Phi$. That is, we need to establish whether the formula Φ is valid in each initial state s of TS. The basic procedure for CTL model checking is rather straightforward:

- the set $Sat(\Phi)$ of all states satisfying Φ is computed recursively, and
- it follows that $TS \models \Phi$ if and only if $I \subseteq Sat(\Phi)$

where I is the set of initial states of TS. Note that by computing $Sat(\Phi)$ a more general problem than just checking whether $TS \models \Phi$ is solved. In fact, it checks for *any* state s in S whether $s \models \Phi$, and not just for the initial states. This is sometimes referred

to as a *global* model-checking procedure. The basic idea of the algorithm is sketched in Algorithm 13 where $Sub(\Phi)$ is the set of subformulae of Φ. In the sequel of this section it is assumed that TS is finite and has no terminal states.

Algorithm 13 CTL model checking (basic idea)

Input: finite transition system TS and CTL formula Φ (both over AP)
Output: $TS \models \Phi$

```
                                  (* compute the sets Sat(Φ) = { s ∈ S | s ⊨ Φ } *)
for all i ⩽ |Φ| do
  for all Ψ ∈ Sub(Φ) with |Ψ| = i do
    compute Sat(Ψ) from Sat(Ψ')          (* for maximal genuine Ψ' ∈ Sub(Ψ) *)
  od
od
return I ⊆ Sat(Φ)
```

The recursive computation of $Sat(\Phi)$ basically boils down to a *bottom-up traversal* of the *parse tree* of the CTL state formula Φ. The nodes of the parse tree represent the subformulae of Φ. The leaves stand for the constant true or an atomic proposition $a \in AP$. All inner nodes are labeled with an operator. For ENF formulae the labels of the inner nodes are $\neg$, $\wedge$, $\exists\bigcirc$, $\exists\,\mathsf{U}$, or $\exists\Box$.

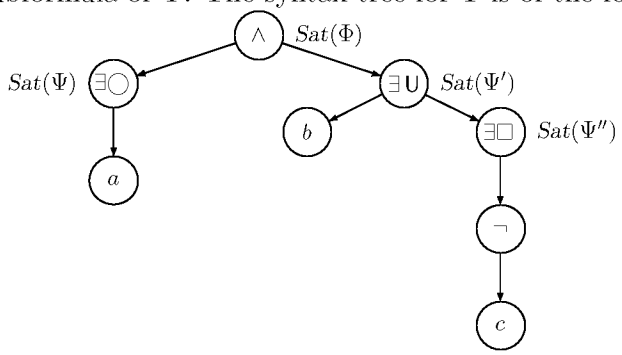
For each node of the parse tree, i.e., for each subformula Ψ of Φ, the set $Sat(\Psi)$ of states is computed for which Ψ holds. This computation is carried out in a bottom-up fashion, starting from the leaves of the parse tree and finishing at the root of the tree, the (unique) node in the parse tree that corresponds to Φ. At an intermediate node, the results of the computations of its children are used and combined in an appropriate way to establish the states of its associated subformula. The type of computation at such node, v say, depends on the operator (e.g., $\wedge$, $\exists\bigcirc$, or $\exists\,\mathsf{U}$) that is at the "top level" of the subformula treated. The children of node v stand for the *maximal genuine subformulae* of the formula Ψ_v that is represented by v. As soon as $Sat(\Psi)$ is computed, subformula Ψ is (theoretically) replaced by a new atomic proposition a_Ψ, and the labeling function L is adjusted as follows: a_Ψ is added to $L(s)$ if and only if $s \in Sat(\Psi)$. Once the bottom-up computations continue with the father, w say, of node v, $\Psi = \Psi_v$ is a maximal genuine subformula of Ψ_w, and all states that are labeled with a_Ψ are known to satisfy Ψ. In fact, one might say that Ψ is replaced in the formula by the atomic proposition a_Ψ. This technique will be of importance when treating the model checking of CTL with fairness.

Example 6.22.

Consider the following state formula over $AP = \{a, b, c\}$:

$$\Phi \;=\; \underbrace{\exists\bigcirc a}_{\Psi} \;\wedge\; \underbrace{\exists(b \,\mathsf{U}\, \underbrace{\exists\Box\, \neg c}_{\Psi''})}_{\Psi'} \quad .$$

The indicated formulae Ψ and Ψ' are the maximal proper subformulae of Φ, while Ψ'' is a maximal proper subformula of Ψ. The syntax tree for Φ is of the following form:



The satisfaction sets for the leaves result directly from the labeling function L. The treatment of subformula $\neg c$ only needs the satisfaction set for $Sat(c)$ to be complemented. Using $Sat(\neg c)$, the set $Sat(\exists\Box\, \neg c)$ can be computed. The subformula Ψ'' can now be replaced by the fresh atomic proposition a_3 where $a_3 \in L(s)$ if and only if $s \in Sat(\exists\Box\, \neg c)$. The computation now continues with determining $Sat(\exists(b \,\mathsf{U}\, a_3))$. In a similar way, $Sat(\exists\bigcirc a)$ can be computed by means of $Sat(a)$.

Once the subformulae Ψ and Ψ' are treated, they can be replaced by the atomic propositions a_1, a_2, respectively, such that

$$a_1 \in L(s) \quad \text{iff} \quad s \models \exists\bigcirc a \quad \text{and} \quad a_2 \in L(s) \quad \text{iff} \quad s \models \exists(b \,\mathsf{U}\, a_3).$$

The formula that is to be treated for the root node simply thus is: $\Phi' \;=\; a_1 \wedge a_2$. $Sat(\Phi')$ results from intersecting $Sat(a_1) = Sat(\Psi)$ and $Sat(a_2) = Sat(\Psi')$. Note that a_1, a_2, and a_3 are fresh atomic propositions, i.e., $\{a_1, a_2, a_3\} \cap AP = \varnothing$. The above procedure thus is considered over $AP' = AP \cup \{a_1, a_2, a_3\}$. ■

Theorem 6.23. Characterization of $\mathrm{Sat}(\cdot)$ ***for CTL formulae in ENF***

Let $TS = (S, Act, \to, I, AP, L)$ be a transition system without terminal states. For all CTL formulae Φ, Ψ over AP it holds that

(a) $Sat(true) = S$,

(b) $Sat(a) = \{ s \in S \mid a \in L(s) \}$, *for any* $a \in AP$,

(c) $Sat(\Phi \wedge \Psi) = Sat(\Phi) \cap Sat(\Psi)$,

(d) $Sat(\neg\Phi) = S \setminus Sat(\Phi)$,

(e) $Sat(\exists\bigcirc \Phi) = \{ s \in S \mid Post(s) \cap Sat(\Phi) \neq \varnothing \}$,

(f) $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ *is the smallest subset* T *of* S, *such that*

(1) $Sat(\Psi) \subseteq T$ *and (2)* $s \in Sat(\Phi)$ *and* $Post(s) \cap T \neq \varnothing$ *implies* $s \in T$,

(g) $Sat(\exists\Box \Phi)$ *is the largest subset* T *of* S, *such that*

(3) $T \subseteq Sat(\Phi)$ *and (4)* $s \in T$ *implies* $Post(s) \cap T \neq \varnothing$.

In the clauses (f) and (g), the terms "smallest" and "largest" should be interpreted with respect to the partial order induced by set inclusion.

Proof: The validity of the claims indicated in (a) through (e) is straightforward. We only prove the propositions (f) and (g):

Proof of (f): The proof of this claim consists of two parts:

(i) Show that $T = Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ satisfies (1) and (2). From the expansion law

$$\exists(\Phi \,\mathsf{U}\, \Psi) \equiv \Psi \vee (\Phi \wedge \exists\bigcirc \exists(\Phi \,\mathsf{U}\, \Psi)),$$

it directly follows that T satisfies the properties (1) and (2).

(ii) Show that for any T satisfying properties (1) and (2) we have

$$Sat(\exists(\Phi \,\mathsf{U}\, \Psi)) \subseteq T.$$

This is proven as follows. Let $s \in Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$. Distinguish between $s \in Sat(\Psi)$ and $s \notin Sat(\Psi)$. If $s \in Sat(\Psi)$, it follows from (1) that $s \in T$. In case $s \notin Sat(\Psi)$, there exists a path $\pi = s_0\, s_1\, s_2 \ldots$ starting in $s{=}s_0$, such that $\pi \models \Phi \,\mathsf{U}\, \Psi$. Let $n > 0$, such that $s_i \models \Phi$, $0 \leqslant i < n$, and $s_n \models \Psi$. Then:

- $s_n \in Sat(\Psi) \subseteq T$,
- $s_{n-1} \in T$, since $s_n \in Post(s_{n-1}) \cap T$ and $s_{n-1} \in Sat(\Phi)$,
- $s_{n-2} \in T$, since $s_{n-1} \in Post(s_{n-2}) \cap T$ and $s_{n-2} \in Sat(\Phi)$,

- ……,
- $s_1 \in T$, since $s_2 \in \mathit{Post}(s_1) \cap T$ and $s_1 \in \mathit{Sat}(\Phi)$, and finally
- $s_0 \in T$, since $s_1 \in \mathit{Post}(s_0) \cap T$ and $s_0 \in \mathit{Sat}(\Phi)$.

It thus follows that $s = s_0 \in T$.

Proof of (g): The proof of this claim consists of two parts:

(i) Show that $T = \mathit{Sat}(\exists\Box\Phi)$ satisfies (3) and (4). From the expansion law

$$\exists\Box\,\Phi \quad \equiv \quad \Phi \wedge \exists\bigcirc\,\exists\Box\,\Phi,$$

it directly follows that T satisfies the properties (3) and (4).

(ii) Show that for any T satisfying properties (3) and (4)

$$T \subseteq \mathit{Sat}(\exists\Box\Phi).$$

This proof goes as follows. Let $T \subseteq S$ satisfy (3) and (4) and $s \in T$. Let $\pi = s_0\, s_1\, s_2 \ldots$ be a path starting in $s{=}s_0$. (As TS has no terminal states, such a path exists.) Then we derive

- $s_0 = s$.
- Since $s_0 \in T$, there exists a state $s_1 \in \mathit{Post}(s_0) \cap T$.
- Since $s_1 \in T$, there exists a state $s_2 \in \mathit{Post}(s_1) \cap T$.
- ……

Here, property (4) is exploited in every step. From property (3), it follows that

$$s_i \in T \ \subseteq \mathit{Sat}(\Phi), \quad i \geqslant 0.$$

Thus, $\pi = s_0\, s_1\, s_2 \ldots$ satisfies $\Box\Phi$. It follows that

$$s \in \mathit{Sat}(\exists\Box\Phi).$$

As this reasoning applies to any $s \in T$, it follows that $T \subseteq \mathit{Sat}(\exists\Box\,\Phi)$.

■

Remark 6.24. Alternative Formulation of $\mathit{Sat}(\exists(\Phi\,\mathsf{U}\,\Psi)$ *and* $\mathit{Sat}(\exists\Box\,\Phi)$

The characterizations of the sets $\mathit{Sat}(\exists(\Phi\,\mathsf{U}\,\Psi))$ and $\mathit{Sat}(\exists\Box\,\Phi)$ indicated in Theorem 6.23

are based upon the fixed-point equation induced by the expansion laws for $\exists(\Phi \,\mathsf{U}\, \Psi)$ and $\exists\Box\, \Phi$, respectively. Consider, for instance, the expansion law

$$\exists(\Phi \,\mathsf{U}\, \Psi) \;\equiv\; \Psi \;\vee\; (\Phi \,\wedge\, \exists\bigcirc \exists(\Phi \,\mathsf{U}\, \Psi)).$$

The recursive nature of this law suggests to considering the CTL formula $\exists(\Phi \,\mathsf{U}\, \Psi)$ as a *fixed point* of the logical equation

$$F \;\equiv\; \Psi \;\vee\; (\Phi \;\wedge\; \exists\bigcirc F).$$

By the expansion law $F = \exists(\Phi \,\mathsf{U}\, \Psi)$ is a solution, but there are also other solutions that are not equivalent to $\exists(\Phi \,\mathsf{U}\, \Psi)$, such as $F = \exists(\Phi \,\mathsf{W}\, \Psi)$ (see Remark 6.25). However, a unique characterization of $\exists(\Phi \,\mathsf{U}\, \Psi)$ is obtained by the fact that $\exists(\Phi \,\mathsf{U}\, \Psi)$ is the *least* solution of $F \;\equiv\; \Psi \;\vee\; (\Phi \;\wedge\; \exists\bigcirc F)$. Using a set-theoretical counterpart by means of $Sat(\cdot)$, we obtain the following equivalent formulation of constraint (f) in Theorem 6.23:

(f′) $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ is the smallest set $T \subseteq S$ satisfying

$$Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing \,\} \;\subseteq\; T.$$

In fact, "$\subseteq$" may be replaced by "=".

In a similar way, $\exists\Box\, \Phi$ can be considered as the *greatest* fixed point of the logical equation

$$F \;=\; \Phi \;\wedge\; \exists\bigcirc F.$$

Using a set-theoretical counterpart of this equation we obtain the following equivalent formulation of constraint (g) in Theorem 6.23:

(g′) $Sat(\exists\Box\Phi)$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing \,\}.$$

Also in this characterization "$\subseteq$" may be replaced by "=". ■

Characterizations of the satisfaction sets for universally quantified CTL formulae can be obtained using the result in Theorem 6.23. This yields

(h) $Sat(\forall\bigcirc \Phi) \;=\; \{\, s \in S \mid Post(s) \subseteq Sat(\Phi) \,\}.$

(i) $Sat(\forall(\Phi \,\mathsf{U}\, \Psi))$ is the smallest set $T \subseteq S$ satisfying

$$Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \subseteq T \,\} \;\subseteq\; T.$$

(j) $Sat(\forall\Box\, \Phi)$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; \{\, s \in Sat(\Phi) \mid Post(s) \subseteq T \,\}.$$

Remark 6.25. Weak Until

The weak-until operator satisfies the same expansion laws as the until operator.

$$\begin{aligned}
\exists(\Phi \,\mathsf{W}\, \Psi) &\equiv \Psi \vee (\Phi \wedge \exists\bigcirc \exists(\Phi \,\mathsf{W}\, \Psi)),\\
\forall(\Phi \,\mathsf{W}\, \Psi) &\equiv \Psi \vee (\Psi \wedge \forall\bigcirc \forall(\Phi \,\mathsf{W}\, \Psi)).
\end{aligned}$$

The difference, however, is that the weak-until operator represents the largest solution (i.e., fixed point) of the expansion law, whereas the until operator denotes the smallest solution. The satisfaction sets for weak until are characterized as follows:

(k) $Sat(\exists(\Phi \,\mathsf{W}\, \Psi))$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing \,\}.$$

(l) $Sat(\forall(\Phi \,\mathsf{W}\, \Psi))$ is the largest set $T \subseteq S$ satisfying

$$T \;\subseteq\; Sat(\Psi) \;\cup\; \{\, s \in Sat(\Phi) \mid Post(s) \subseteq T \,\}.$$

■

Without loss of generality, we may assume that the CTL formula Φ to be verified is in ENF (see Theorem 6.14, page 332). That is, the model-checking algorithm is supposed to be preceded by transforming the CTL formula at hand into ENF. The characterizations of the satisfaction sets indicated in Theorem 6.23 are exploited to compute the sets $Sat(\cdot)$. The essential steps for computing the satisfaction sets are summarized in Algorithm 14 on page 348.

6.4.2 The Until and Existential Always Operator

To treat constrained reachability properties, given by CTL formulae of the form $\exists(\Phi \,\mathsf{U}\, \Psi)$, the characterization in Theorem 6.23 is exploited. Recall that $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ is character-

Algorithm 14 Computation of the satisfaction sets

Input: finite transition system TS with state set S and CTL formula Φ in ENF
Output: $Sat(\Phi) = \{ s \in S \mid s \models \Phi \}$

(* recursive computation of the sets $Sat(\Psi)$ for all subformulae Ψ of Φ *)

switch(Φ):

true	:	**return** S;
a	:	**return** $\{ s \in S \mid a \in L(s) \}$;
$\Phi_1 \wedge \Phi_2$	:	**return** $Sat(\Phi_1) \cap Sat(\Phi_2)$;
$\neg\Psi$	:	**return** $S \setminus Sat(\Psi)$;
$\exists \bigcirc \Psi$	:	**return** $\{ s \in S \mid Post(s) \cap Sat(\Psi) \neq \varnothing \}$;
$\exists(\Phi_1 \,\mathsf{U}\, \Phi_2)$	:	$T := Sat(\Phi_2)$; (* compute the smallest fixed point *) **while** $\{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \}$; $T := T \cup \{ s \}$; **od**; **return** T;
$\exists \Box \Phi$	:	$T := Sat(\Phi)$; (* compute the greatest fixed point *) **while** $\{ s \in T \mid Post(s) \cap T = \varnothing \} \neq \varnothing$ **do** **let** $s \in \{ s \in T \mid Post(s) \cap T = \varnothing \}$; $T := T \setminus \{ s \}$; **od**; **return** T;

end switch

ized as the smallest set $T \subseteq S$, where S is the set of states in the transition system under consideration, such that

$$(1)\ Sat(\Psi) \subseteq T \text{ and } (2)\ \ (s \in Sat(\Phi) \text{ and } Post(s) \cap T \neq \varnothing) \ \Rightarrow\ s \in T. \tag{6.1}$$

This characterization suggests adopting the following iterative procedure to compute $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$:

$$T_0 \ = \ Sat(\Psi) \quad \text{and} \quad T_{i+1} \ = \ T_i \ \cup \ \{ s \in Sat(\Phi) \mid Post(s) \cap T_i \neq \varnothing \}.$$

Intuitively speaking, the set T_i contains all states that can reach a Ψ-state in at most i steps via a Φ-path. This is to be understood as follows. From the fact that $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ satisfies the conditions (1) and (2) in (6.1), it can be demonstrated by induction on j that

$$T_0 \subseteq T_1 \subseteq T_2 \subseteq \ldots \subseteq T_j \subseteq T_{j+1} \subseteq \ \ldots \ \subseteq \ Sat(\exists(\Phi \,\mathsf{U}\, \Psi)).$$

Since we assume a finite transition system TS, there exists a $j \geqslant 0$ such that

$$T_j \ = \ T_{j+1} \ = \ T_{j+2} \ = \ \ldots \quad .$$

Therefore

$$T_j \;=\; T_j \;\cup\; \{\, s \in \mathit{Sat}(\Phi) \mid \mathit{Post}(s) \cap T_j \neq \varnothing \,\}$$

and, hence:

$$\{\, s \in \mathit{Sat}(\Phi) \mid \mathit{Post}(s) \cap T_j \neq \varnothing \,\} \;\subseteq\; T_j.$$

Hence, T_j satisfies property (2). Further,

$$\mathit{Sat}(\Psi) \;=\; T_0 \;\subseteq\; T_j.$$

These considerations show that T_j possesses the properties (1) and (2). Since $\mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi))$ is the *smallest* set of states satisfying the properties (1) and (2), it follows that

$$\mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi)) \;\subseteq\; T_j$$

and thus $\mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi)) = T_j$. Hence, for any $j \geqslant 0$ we have

$$T_0 \subsetneqq T_1 \subsetneqq T_2 \subsetneqq \ldots \subsetneqq T_j = T_{j+1} = \ldots = \mathit{Sat}(\exists(\Phi \,\mathsf{U}\, \Psi)).$$

Algorithm 15 (see page 351) shows a more detailed version of the backward search indicated earlier in Algorithm 14 (see page 348). As each Ψ-state obviously satisfies $\exists(\Phi \,\mathsf{U}\, \Psi)$, all states in $\mathit{Sat}(\Psi)$ are initially considered to satisfy $\exists(\Phi \,\mathsf{U}\, \Psi)$. This conforms to the initialization to the variable E. An iterative procedure is subsequently started that can be considered to systematically check the state space in a "backward" manner. In each iteration, all Φ-states are determined that can move by a single transition to (one of) the states of which we already know to satisfy $\exists(\Phi \,\mathsf{U}\, \Psi)$. Thus, in the ith iteration of the procedure, all Φ-states are considered that can move to a Ψ-state in at most i steps. This corresponds to the set T_i. Termination of the algorithm intuitively follows, as the number of states in the transition system is finite. Note that the algorithm assumes a transition system representation (or its state graph) by means of "inverse" adjacency lists, i.e., list representations for the sets of predecessors $\mathit{Pre}(s') \;=\; \{\, s \in S \mid s' \in \mathit{Post}(s) \,\}$.

Example 6.26.

Consider the transition system depicted in Figure 6.11, and suppose we are interested in checking the formula $\exists\Diamond\, \Phi$ with $\Phi = ((a = c) \wedge (a \neq b))$. Recall that $\exists\Diamond\, \Phi = \exists(\mathsf{true}\,\mathsf{U}\, \Phi)$. To check $\exists\Diamond\, \Phi$ we invoke Algorithm 14 (see page 348). This algorithm recursively computes $\mathit{Sat}(\mathsf{true})$ and $\mathit{Sat}((a = c) \wedge (a \neq b))$. This corresponds to the situation depicted in Figure 6.12(a), where all states in the set T are colored black, and white otherwise. In the first iteration, we select and delete s_5 from E, but as $\mathit{Pre}(s_5) = \varnothing$, T remains unaffected. On considering $s_4 \in E$, $\mathit{Pre}(s_4) = \{\, s_6 \,\}$ is added to T (and E), see Figure 6.12(b). During the next iteration, the only predecessor of s_6 is added, yielding the snapshot in Figure 6.12(c). After the fourth iteration, the algorithm terminates as there are no new predecessors of Φ-states encountered, i.e., $E = \varnothing$ (see Figure 6.12(d)). ■

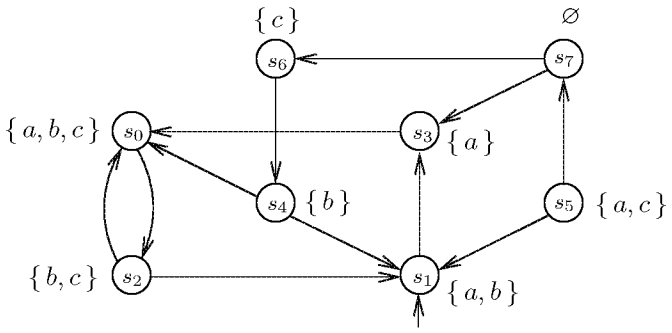


Figure 6.11: An example of a transition system.

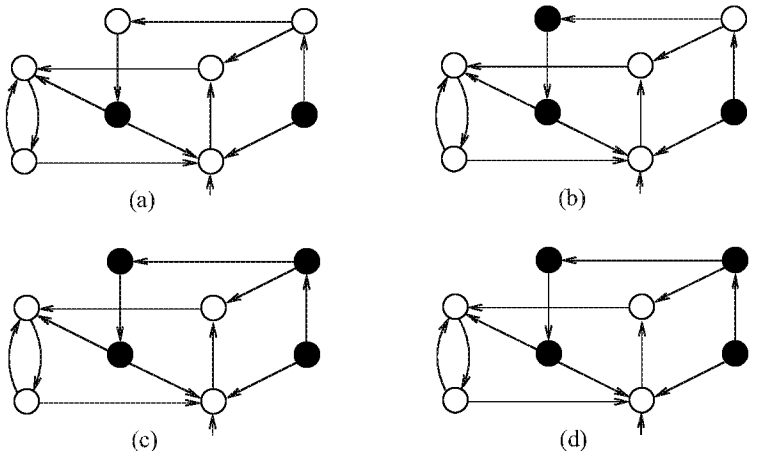


Figure 6.12: Example of backward search for $\exists(\text{true}\,\mathsf{U}\,(a{=}c) \wedge (a{\neq}b))$.

Algorithm 15 Enumerative backward search for computing $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$

Input: finite transition system TS with state set S and CTL formula $\exists(\Phi \,\mathsf{U}\, \Psi)$
Output: $Sat(\exists(\Phi \,\mathsf{U}\, \Psi)) = \{\, s \in S \mid s \models \exists(\Phi \,\mathsf{U}\, \Psi) \,\}$

```
E := Sat(Ψ);                          (* E administers the states s with s ⊨ ∃(Φ U Ψ) *)
T := E;                               (* T contains the already visited states s with s ⊨ ∃(Φ U Ψ) *)
while E ≠ ∅ do
  let s' ∈ E;
  E := E \ { s' };
  for all s ∈ Pre(s') do
    if s ∈ Sat(Φ) \ T then E := E ∪ { s }; T := T ∪ { s } fi
  od
od
return T
```

Let us now consider the computation of $Sat(\exists\Box\,\Phi)$ for the transition system TS. As for the until-operator, the algorithm for $\exists\Box\,\Phi$ is based on the characterization in Theorem 6.23, i.e., $Sat(\exists\Box\,\Phi)$ is the largest set $T \subseteq S$ satisfying

$$T \subseteq Sat(\Phi) \quad \text{and} \quad (s \in T \text{ implies } T \cap Post(s) \neq \varnothing).$$

The basic idea is to compute $Sat(\exists\Box\,\Phi)$ by means of the iteration

$$T_0 \;=\; Sat(\Phi) \quad \text{and} \quad T_{i+1} \;=\; T_i \;\cap\; \{\, s \in Sat(\Phi) \mid Post(s) \cap T_i \neq \varnothing \,\}.$$

Then, for all $j \geqslant 0$, it holds that

$$T_0 \supsetneqq T_1 \supsetneqq T_2 \supsetneqq \ldots \supsetneqq T_j = T_{j+1} = \ldots = T = Sat(\exists\Box\,\Phi).$$

The above iteration can be realized by means of a *backward search* starting with

$$T = Sat(\Phi) \quad \text{and} \quad E = S \setminus Sat(\Phi).$$

Here T equals T_0 and E contains all states that refute $\exists\Box\,\Phi$. During the backward search, states are iteratively removed from T, for which it has been established that they refute $\exists\Box\,\Phi$. This applies to any $s \in T$ satisfying

$$Post(s) \cap T \;=\; \varnothing.$$

Although $s \models \Phi$ (as it is in T), all its successors refute $\exists\Box\,\Phi$ (as they are not in T), and therefore s refutes $\exists\Box\,\Phi$. Once such states are encountered, they are inserted in E to enable the possible removal of other states in T.

Algorithm 16 Enumerative backward search to compute $Sat(\exists\Box\,\Phi)$

Input: finite transition system TS with state set S and CTL formula $\exists\Box\,\Phi$
Output: $Sat(\exists\Box\,\Phi) = \{\, s \in S \mid s \models \exists\Box\,\Phi \,\}$

```
E := S \ Sat(Φ);                          (* E contains any not visited s' with s' ⊭ ∃□Φ *)
T := Sat(Φ);          (* T contains any s for which s ⊨ ∃□Φ has not yet been disproven *)
for all s ∈ Sat(Φ) do count[s] := | Post(s) |; od          (* initialize array count *)
while E ≠ ∅ do
                                   (* loop invariant: count[s] = | Post(s) ∩ (T ∪ E) | *)
  let s' ∈ E;                                                          (* s' ⊭ Φ *)
  E := E \ { s' };                                          (* s' has been considered *)
  for all s ∈ Pre(s') do
                                  (* update counters count[s] for all predecessors s of s' *)
    if s ∈ T then
      count[s] := count[s] − 1;
      if count[s] = 0 then
        T := T \ { s };                          (* s does not have any successor in T *)
        E := E ∪ { s };
      fi
    fi
  od
od
return T
```

The resulting computational procedure is detailed in Algorithm 16 on page 352. In order to support the test whether $Post(s) \cap T = \varnothing$, a counter $count[s]$ is exploited that keeps track of the number of direct successors of s in $T \cup E$:

$$count[s] \;=\; |\; Post(s) \;\cap\; (T \cup E)\;|.$$

Once $count[s] = 0$, we have that $Post(s) \cap (T \cup E) = \varnothing$, and hence $Post(s) \cap T = \varnothing$. Thus, state s is not in $Sat(\exists\Box\,\Phi)$ and therefore can be safely removed from T. On termination, $E = \varnothing$, and thus $count[s] = |\, Post(s) \cap T\,|$. It follows that any state $s \in Sat(\Phi)$ for which $count[s] > 0$ satisfies the CTL formula $\exists\Box\,\Phi$.

It is left to the interested reader to consider how the outlined approach can be modified to compute $Sat(\exists(\Phi\,\mathsf{W}\,\Psi))$.

Example 6.27.

Consider the transition system depicted in Figure 6.11 and the formula $\exists\Box\, b$. Initially,

$$T_0 = \{ s_0, s_1, s_2, s_4 \} \quad \text{and} \quad E = \{ s_3, s_5, s_6, s_7 \} \quad \text{and} \quad \mathit{count} = [1,1,2,1,2,2,1,2].$$

Suppose state $s_3 \in E$ is selected in the first iteration. As $s_1 \in \mathit{Pre}(s_3)$ and $s_1 \in T$, $\mathit{count}[s_1] := 0$. Accordingly, s_1 is deleted from T and added to E. This yields

$$T_1 = \{ s_0, s_2, s_4 \} \quad \text{and} \quad E = \{ s_1, s_5, s_6, s_7 \} \quad \text{and} \quad \mathit{count} = [1,0,2,1,2,2,1,2].$$

We also have $s_7 \in \mathit{Pre}(s_3)$, but as $s_7 \notin T$, this affects neither $\mathit{count}[s_7]$, T nor E.

Suppose s_6 and s_7 are selected in the second and third iteration, respectively. None of these states has a predecessor in T_1, so we obtain

$$T_3 = \{ s_0, s_2, s_4 \} \quad \text{and} \quad E = \{ s_1, s_5 \} \quad \text{and} \quad \mathit{count} = [1,0,2,1,2,2,1,2].$$

Now select $s_1 \in E$ in the next iteration. As $\mathit{Pre}(s_1) \cap T_3 = \{ s_2, s_4 \}$, the counters for s_2 and s_4 are decremented. This yields

$$T_4 = \{ s_0, s_2, s_4 \} \quad \text{and} \quad E = \{ s_5 \} \quad \text{and} \quad \mathit{count} = [1,0,1,1,1,2,1,2].$$

As $\mathit{Pre}(s_5) = \varnothing$, T and *count* are unaffected in the subsequent iteration. As $E = \varnothing$, the algorithm terminates and returns $T = \{ s_0, s_2, s_4 \}$ as the final outcome. ■

This section is concluded by outlining an alternative algorithm for computing $\mathit{Sat}(\exists\Box\,\Phi)$. Since the computation of the satisfaction sets takes place by means of a bottom-up traversal through the parse tree of the formula at hand, it is assumed that $\mathit{Sat}(\Phi)$ is at our disposal. A possibility to compute $\mathit{Sat}(\exists\Box\,\Phi)$ is to only consider the Φ-states of transition system TS and ignore all $\neg\Phi$-states. The justification of this modification to TS is that all removed states will not satisfy $\exists\Box\,\Phi$ (as they violate Φ) and therefore can be safely removed. For $TS = (S, \mathit{Act}, \rightarrow, I, \mathit{AP}, L)$ let $TS[\Phi] = (S', \mathit{Act}, \rightarrow', I', \mathit{AP}, L')$ with $S' = \mathit{Sat}(\Phi)$, $\rightarrow' \;=\; \rightarrow \cap (S' \times \mathit{Act} \times S')$, $I' = I \cap S'$ and $L'(s) = L(s)$ for all $s \in S'$. Then, all nontrivial strongly connected components (SCCs) [4] in the state graph induced by $TS[\Phi]$ are computed. All states in each such SCC C satisfy $\exists\Box\,\Phi$, as any state in C is reachable from any other state in C, and—by construction—all states in C satisfy Φ. Finally, all states in $TS[\Phi]$ are computed that can reach such SCC. If state $s \in S'$ and there exists such a path, then—by construction of $TS[\Phi]$—the property $\exists\Box\,\Phi$ is satisfied by s; otherwise, it is not. This can be done by a backward search. The worst-case time complexity of this

[4] A strongly connected component (SCC) of a digraph G is a maximal, connected subgraph of G. Stated differently, the SCCs of a graph are the equivalence classes of vertices under the "are mutually reachable" relation. A *nontrivial* SCC is an SCC that contains at least one transition.

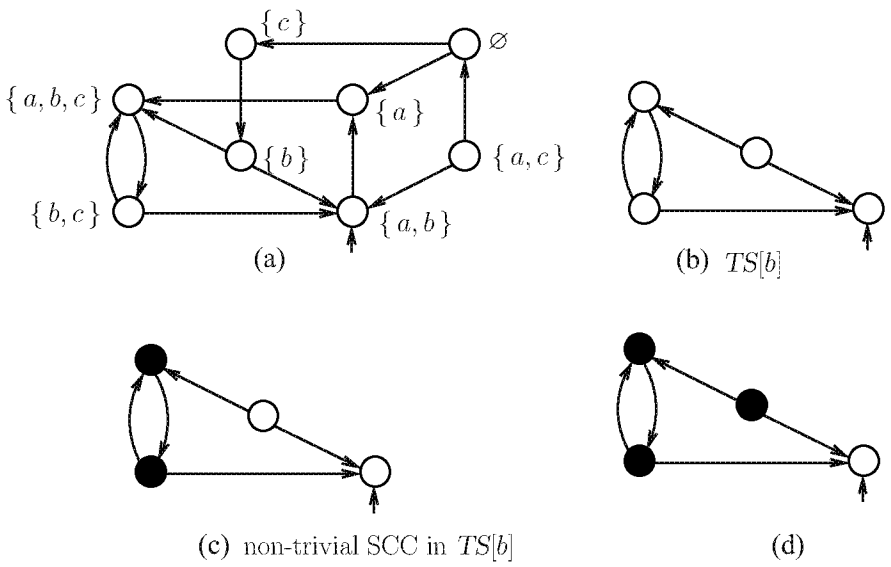


Figure 6.13: Computing $Sat(\exists\Box\, b)$ using strongly connected components in $TS[\Phi]$.

alternative algorithm is the same as for Algorithm 16. This approach is illustrated by the following example.

Example 6.28. Alternative Algorithm for $\exists\Box\,\Phi$

Consider the transition system of Figure 6.11 and CTL formula $\exists\Box\, b$. The modified transition system $TS[b]$ consists of the four states that are labeled with b, see the gray states in Figure 6.13(a). The only nontrivial SCC of this structure is indicated by the black states, see Figure 6.13(b). As there is only a single b-state (that is not in the SCC) that can reach the nontrivial SCC, this state satisfies $\exists\Box\, b$, and the computation finishes, see Figure 6.13(c). ■

Theorem 6.29.

For state s in transition system TS and CTL formula Φ:

$$s \models \exists\Box\,\Phi \text{ iff } s \models \Phi \text{ and there is a nontrivial SCC in } TS[\Phi] \text{ reachable from } s.$$

Proof: $\Rightarrow$: Suppose $s \models \exists\Box\,\Phi$. Clearly, s is a state in $TS[\Phi]$. Let π be a path in TS starting in s such that $\pi \models \Box\,\Phi$. As TS is finite, π has a suffix $\rho = s_1\, s_2 \ldots s_k$ for $k > 1$, representing a cycle that is traversed infinitely often. As π is also a path in $TS[\Phi]$, the

states s_1 through s_k are all in $TS[\Phi]$. Since π is traversed infinitely often, it represents a cycle, and thus any pair of states s_i and s_j is mutually reachable. Stated differently, $\{ s_1, \ldots, s_k \}$ is either an SCC or contained in some SCC in $TS[\Phi]$. As π is a path starting in s, these states are reachable from s.

$\Leftarrow$: Suppose s is a state in $TS[\Phi]$ and there exists an SCC in $TS[\Phi]$ reachable from s. Let s' be a state in such SCC. As the SCC is nontrivial, s' is reachable from itself by a path of length at least one. Repeating this cycle infinitely often yields an infinite path π with $\pi \models \Box\, \Phi$. The path from s to s' followed by $\pi[1..]$ now satisfies $\Box\, \Phi$ and starts in s. Thus, $s \models \exists \Box\, \Phi$. ■

6.4.3 Time and Space Complexity

The time complexity of the CTL model-checking algorithm is determined as follows. Let TS be a finite transition system with N states and K transitions. Under the assumption that the sets of predecessors $Pre(\cdot)$ are represented as linked lists, the time complexity of Algorithms 15 and 16 lies in $\mathcal{O}(N+K)$. Given that the computation of the satisfaction sets $Sat(\Phi)$ is a bottom-up traversal over the parse tree of Φ and thus linear in $|\, \Phi \,|$, the time complexity of Algorithm 14 (see page 348) is in

$$\mathcal{O}((N+K) \cdot |\, \Phi \,|).$$

When the initial states of TS are administrated in, e.g., a linked list, checking whether $I \subseteq Sat(\Phi)$ can be done in $\mathcal{O}(N_0)$ where N_0 is the cardinality of the set I. Recall that the CTL model-checking algorithm requires the CTL formula to be checked to be in existential normal form. As the transformation of any CTL formula into an equivalent CTL formula in ENF may yield an exponential blowup of the formula, it is recommended to treat the modalities such as $\forall \mathsf{U}$, $\forall \Diamond$, $\forall \Box$, $\exists \Diamond$, $\forall \mathsf{W}$, and $\exists \mathsf{W}$, analogous to the introduced approaches for $\exists \mathsf{U}$ and $\exists \Box$, by exploiting the characterizations of $Sat(\forall \Box\, \Phi)$, $Sat(\exists \Diamond\, \Phi)$, and so on. The resulting algorithms are all linear in N and K. Thus, we obtain the following theorem:

Theorem 6.30. Time Complexity of CTL Model Checking

For transition system TS with N states and K transitions, and CTL formula Φ, the CTL model-checking problem $TS \models \Phi$ can be determined in time $\mathcal{O}\,((N+K) \cdot |\, \Phi \,|)$.

Let us compare this complexity bound with that for LTL model checking. Recall that LTL model checking is exponential in the size of the formula. Although the difference in time complexity with respect to the length of the formula seems drastic (exponential for LTL vs.

linear for CTL), this should *not* be interpreted as "CTL model checking is more efficient than LTL model checking". From Theorem 6.18 it follows that whenever $\varphi \equiv \Phi$ for LTL formula φ and CTL formula Φ, then φ is obtained by removing all path quantifiers in Φ. Thus, CTL formulae are at least as long as their equivalent counterparts in LTL (if these exist). In fact, if P $\neq$ NP, then there exist LTL formulae φ_n with $|\,\varphi_n\,| \in \mathcal{O}(poly(n))$ for which CTL-equivalent formulae do exist, but not of polynomial length. LTL formulae may be exponentially shorter than any of their equivalent formulation in CTL. The latter effect is illustrated in the following example. From Lemma 5.45, it follows that the Hamiltonian path problem for digraphs with n nodes can be encoded in LTL formula φ_n whose length is polynomial in n. More precisely, given a digraph G with nodeset $\{\,1,\ldots,n\,\}$ there is an LTL formula φ_n such that (1) G has a Hamiltonian path if and only if $\neg\varphi_n$ does not hold for the transition system associated with G and (2) $\neg\varphi_n$ has an equivalent CTL formula.

To express the existence of a Hamiltonian path in CTL requires a CTL formula of exponential length, unless P = NP: if it is assumed that $\neg\varphi_n \equiv \neg\Phi_n$ for CTL formulae Φ_n of polynomial length, then the Hamiltonian path problem could be solved by CTL model checking, and thus in polynomial time. Since, however, the Hamiltonian path problem is NP-complete, this is only possible for the case of PTIME = NP.

Example 6.31. The Hamiltonian Path Problem

Consider the NP-complete problem of finding a Hamiltonian path in an arbitrary, connected, directed graph $G = (V, E)$ where V denotes the set of vertices and $E \subseteq V \times V$, the set of edges. Let $V = \{\,v_1,\ldots,v_n\,\}$. A Hamiltonian path is a (finite) path through the graph G which visits each state exactly once. Starting from graph G, a transition system $TS(G)$ is derived as well as a CTL formula Φ_n, such that

$$G \text{ contains a Hamiltonian path if and only if } TS \not\models \neg\Phi_n.$$

The transition system TS is defined as follows

$$TS \;=\; (V \cup \{\,b\,\}, \{\,\tau\,\}, \rightarrow, V, V, L)$$

with $L(v_i) = \{\,v_i\,\}$, $L(b) = \varnothing$. The transition relation $\rightarrow$ is defined by

$$\frac{(v_i, v_j) \in E}{v_i \xrightarrow{\tau} v_j} \quad \text{and} \quad \frac{v_i \in V \cup \{\,b\,\}}{v_i \xrightarrow{\tau} b}.$$

All vertices of G are states in the transition system TS. A new state b is introduced that is a direct successor of any state (including b). Figure 6.14 shows (a) an example of a directed graph G and (b) its transition system $TS(G)$. The sole purpose of the new state b is to ensure that the transition system has no terminal states. Note that $TS(G)$ is defined as in the proof for the existence of a polynomial reduction of the Hamiltonian

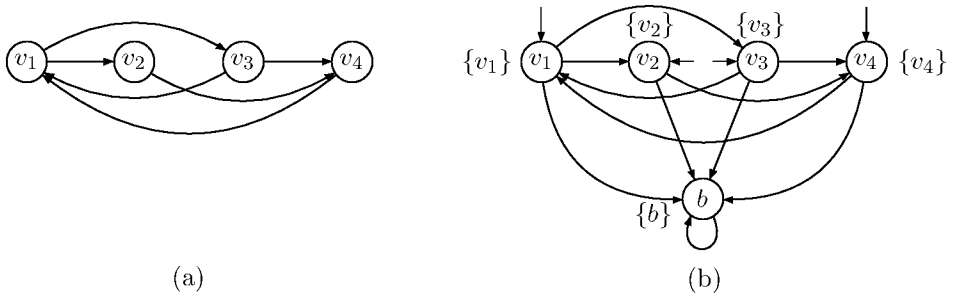


Figure 6.14: Example of encoding the Hamiltonian path problem as a transition system.

path problem onto the complement of the LTL model-checking problem; see Lemma 5.45 on page 288.

It remains to give a recipe for the construction of Φ_n, a CTL formula that captures the existence of a Hamiltonian path. Let Φ_n be defined as follows:

$$\Phi_n \;=\; \bigvee_{\substack{(i_1,\ldots,i_n) \\ \text{permutation of } (1,\ldots,n)}} \Psi(v_{i_1}, v_{i_2}, \ldots, v_{i_n})$$

such that $\Psi(v_{i_1}, \ldots, v_{i_n})$ is a CTL formula that is satisfied if and only if $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ is a Hamiltonian path in G. The formulae $\Psi(v_{i_1}, \ldots, v_{i_n})$ are inductively defined as follows:

$$\begin{aligned} \Psi(v_i) &= v_i \\ \Psi(v_{i_1}, v_{i_2}, \ldots, v_{i_n}) &= v_{i_1} \wedge \exists \bigcirc \Psi(v_{i_2}, \ldots, v_{i_n}) \quad \text{if } n > 1. \end{aligned}$$

Stated in words, $\Psi(v_{i_1}, \ldots, v_{i_n})$ holds if there exists a path on which v_{i_1}, v_{i_2}, v_{i_3} successively hold. Since each state has a transition to the b state, the trace $\{ v_{i_1} \}\{ v_{i_2} \}\ldots\{ v_{i_n} \}$ can be extended with $\{ b \}^\omega$. An example of an instantiation of Ψ_n is

$$\Phi_2 \;=\; (v_1 \wedge \exists \bigcirc v_2) \;\vee\; (v_2 \wedge \exists \bigcirc v_1)$$

and

$$\begin{aligned} \Phi_3 \;=\; & (v_1 \wedge \exists \bigcirc (v_2 \wedge \exists \bigcirc v_3)) \;\vee\; (v_1 \wedge \exists \bigcirc (v_3 \wedge \exists \bigcirc v_2)) \\ \vee\; & (v_2 \wedge \exists \bigcirc (v_1 \wedge \exists \bigcirc v_3)) \;\vee\; (v_2 \wedge \exists \bigcirc (v_3 \wedge \exists \bigcirc v_1)) \\ \vee\; & (v_3 \wedge \exists \bigcirc (v_1 \wedge \exists \bigcirc v_2)) \;\vee\; (v_3 \wedge \exists \bigcirc (v_2 \wedge \exists \bigcirc v_1)). \end{aligned}$$

It is not difficult to infer that

$$\mathit{Sat}(\; \Psi(v_{i_1}, \ldots, v_{i_n}) \;) \;=\; \begin{cases} \{ v_{i_1} \} & \text{if } v_{i_1}, \ldots, v_{i_n} \text{ is a Hamiltonian path in } G \\ \varnothing & \text{otherwise.} \end{cases}$$

Thus:

$TS \not\models \neg\Phi_n$

iff there is an initial state s of TS for which $s \not\models \neg\Phi_n$

iff there is an initial state s of TS for which $s \models \Phi_n$

iff $\exists v$ in G and a permutation $i_1, \ldots, i_n$ of $1, \ldots, n$ with $v \in Sat(\Psi(v_{i_1}, \ldots, v_{i_n}))$,

iff $\exists v$ in G and a permutation $i_1, \ldots, i_n$ of $1, \ldots, n$, such that $v = v_{i_1}$ and $v_{i_1}, \ldots, v_{i_n}$ is a Hamiltonian path in G

iff G has a Hamiltonian path.

Thus, G contains a Hamiltonian path if and only if $TS \not\models \neg\Phi_n$.

By the explicit enumeration of all possible permutations we obtain a formula with a length that is exponential in the number of vertices in the graph. This does not prove that there does not exist an equivalent, but shorter, CTL formula which describes the Hamiltonian path problem. Actually, shorter formalizations in CTL cannot be expected, since the CTL model-checking problem is polynomially solvable whereas the Hamiltonian path problem is NP-complete. ■

6.5 Fairness in CTL

Recall that fairness assumptions (see Section 3.5) are used to rule out certain computations that are considered to be unrealistic for the system under consideration. These unreasonable computations that ignore certain transition alternatives forever are the "unfair" ones; the remaining computations are thus the "fair" ones. As there are different notions of fairness, various distinct forms of fairness can be imposed: unconditional, strong, and weak fairness constraints.

An important distinction between LTL and CTL is that fairness assumptions can be incorporated into LTL without any specific changes while a special treatment of fairness is required for CTL. That is to say, fairness assumptions can be added as premise to the LTL formula to be verified. The LTL model-checking problem where only fair paths are considered (i.e., considering the fair satisfaction relation $\models_{fair}$) can thus be reduced to the traditional LTL model-checking problem, i.e., with respect to the common satisfaction relation $\models$. In this case, the LTL formula φ to be verified is replaced by $fair \rightarrow \varphi$:

$$TS \models_{fair} \varphi \quad \text{if and only if} \quad TS \models (fair \rightarrow \varphi).^5$$

[5] This observation is mainly of theoretical interest since it is more efficient to design special LTL model-

For more details we refer to Section 5.1.6 (see page 257).

An analogous approach is *not* possible for CTL. This stems from the fact that most fairness constraints cannot be encoded in a CTL formula. An indication of this is the fact that persistence properties $\Diamond\Box a$ are inherent in, e.g., strong fairness conditions $\Box\Diamond b \rightarrow \Box\Diamond c \equiv \Diamond\Box\neg b \vee \Box\Diamond c$ and cannot be expressed in CTL. To be a bit more precise: fairness constraints operate on the path level and replace the standard meaning "for all paths" of universal quantification with "for all fair paths" and existential quantification "there exists a path" with "there exists a fair path". Thus, if *fair* expresses the fairness condition on the path level, then CTL formulae that encode the intuitive meaning of $\forall(\mathit{fair} \rightarrow \varphi)$ and $\exists(\mathit{fair} \wedge \varphi)$ would be needed. However, these are not legal CTL formulae since (1) the Boolean connectives $\rightarrow$ and $\wedge$ are not allowed on the level of CTL path formulae and (2) fairness conditions cannot be described by CTL path formulae.

Therefore, an alternative approach is taken to treat fairness in CTL. In order to deal with fairness constraints in CTL, the semantics of CTL is slightly modified such that the state formulae $\forall\varphi$ and $\exists\varphi$ are interpreted over all fair paths rather than over all possible paths. A fair path is a path that satisfies a set of fairness constraints. It is assumed that the given fixed fairness constraint is described by a formula as in LTL.

Definition 6.32. CTL Fairness Assumptions

A *strong CTL fairness constraint* (over AP) is a term of the form

$$\mathit{sfair} = \bigwedge_{1 \leqslant i \leqslant k} (\Box\Diamond\Phi_i \rightarrow \Box\Diamond\Psi_i)$$

where Φ_i and Ψ_i (for $1 \leqslant i \leqslant k$) are CTL formulae over AP. Weak and unconditional CTL fairness constraints are defined analogously by conjunctions of terms of the form $(\Diamond\Box\Phi_i \rightarrow \Box\Diamond\Psi_i)$ and $\Box\Diamond\Psi_i$, respectively. A *CTL fairness assumption* is a conjunction of strong, weak, and unconditional CTL fairness constraints. ■

Note that CTL fairness assumptions are *not* CTL path formulae, but they can be viewed as LTL formulae using CTL state formulae instead of atomic propositions. For instance, imposing the strong fairness constraint $\bigwedge_{1 \leqslant i \leqslant k} (\Box\Diamond\Phi_i \rightarrow \Box\Diamond\Psi_i)$ on paths means that a path must either have only finitely many states satisfying Φ_i or infinitely many states satisfying Ψ_i, for any $1 \leqslant i \leqslant k$ (or both).

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, π be an infinite path fragment in TS, and *fair* a fixed CTL fairness assumption. $\pi \models \mathit{fair}$ denotes

checking algorithms when dealing with fairness assumptions than to apply the reduction to the standard semantics $\models$.

that π satisfies the formula *fair*, where $\models$ should be read as the LTL semantics. Consider, for example, strong fairness. For an infinite path $\pi = s_0\, s_1\, s_2 \ldots$, we have

$$\pi \models \bigwedge_{1 \leqslant i \leqslant k} (\Box \Diamond \Phi_i \rightarrow \Box \Diamond \Psi_i)$$

if and only if for every $i \in \{1, \ldots, k\}$ either $s_j \models \Phi_i$ for finitely many indices j, or $s_j \models \Psi_i$ for infinitely many j (or both). Here, the statement $s_j \models \Phi_i$ should be interpreted according to the CTL semantics, that is, the semantics without taking any fairness into account.

The semantics of CTL under fairness assumption *fair* is identical to the semantics given earlier (see Definition 6.4), except that the path quantifications range over all fair paths rather than over all paths. The fair paths starting in state s are defined as

$$\mathit{FairPaths}(s) \;=\; \{\pi \in \mathit{Paths}(s) \mid \pi \models \mathit{fair}\}.$$

Let $\mathit{FairPaths}(TS)$ denote the set of all fair paths in TS, i.e.:

$$\mathit{FairPaths}(TS) \;=\; \bigcup_{s_0 \in I} \mathit{FairPaths}(s_0).$$

The fair interpretation of CTL is defined in terms of the satisfaction relation $\models_{\mathit{fair}}$. We have $s \models_{\mathit{fair}} \Phi$ if and only if Φ is valid in state s under the fairness assumption *fair*.

Definition 6.33. Satisfaction Relation for CTL with Fairness

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and $s \in S$. The satisfaction relation $\models_{\mathit{fair}}$ for CTL fairness assumption *fair* is defined for state formulae by

$$\begin{array}{lll}
s \models_{\mathit{fair}} a & \text{iff} & a \in L(s) \\
s \models_{\mathit{fair}} \neg \Phi & \text{iff} & \text{not } s \models_{\mathit{fair}} \Phi \\
s \models_{\mathit{fair}} \Phi \wedge \Psi & \text{iff} & (s \models_{\mathit{fair}} \Phi) \text{ and } (s \models_{\mathit{fair}} \Psi) \\
s \models_{\mathit{fair}} \exists \varphi & \text{iff} & \pi \models_{\mathit{fair}} \varphi \text{ for some } \pi \in \mathit{FairPaths}(s) \\
s \models_{\mathit{fair}} \forall \varphi & \text{iff} & \pi \models_{\mathit{fair}} \varphi \text{ for all } \pi \in \mathit{FairPaths}(s)
\end{array}$$

Here, $a \in AP$, Φ, Ψ are CTL state formulae, and φ a CTL path formula. For path π, the satisfaction relation $\models_{\mathit{fair}}$ for path formulae is defined as in Definition 6.4:

$$\begin{array}{lll}
\pi \models_{\mathit{fair}} \bigcirc \Phi & \text{iff} & \pi[1] \models_{\mathit{fair}} \Phi \\
\pi \models_{\mathit{fair}} \Phi \,\mathsf{U}\, \Psi & \text{iff} & \exists\, j \geqslant 0.\ (\pi[j] \models_{\mathit{fair}} \Psi \ \wedge\ (\forall\, 0 \leqslant k < j.\, \pi[k] \models_{\mathit{fair}} \Phi))
\end{array}$$

where for path $\pi = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$, $\pi[i]$ denotes the $(i{+}1)$-th state of π, i.e., $\pi[i] = s_i$. ■

Whereas for LTL, fairness constraints can be specified as part of the formula to be checked, for CTL similar constraints are imposed on the underlying model of the system under consideration, i.e., the transition system.

Definition 6.34. CTL Semantics with Fairness for Transition Systems

For CTL-state formula Φ, and CTL fairness assumption *fair*, the *satisfaction set* $Sat_{fair}(\Phi)$ is defined by

$$Sat_{fair}(\Phi) \;=\; \{\, s \in S \mid s \models_{fair} \Phi \,\}.$$

The transition system TS satisfies CTL formula Φ under fairness assumption *fair* if and only if Φ holds in all initial states of TS:

$$TS \models_{fair} \Phi \quad \text{if and only if} \quad \forall s_0 \in I.\, s_0 \models_{fair} \Phi.$$

This is equivalent to $I \subseteq Sat_{fair}(\Phi)$. ■

Example 6.35. CTL Fairness Assumption

Consider the transition system TS depicted in Figure 6.15 and suppose we are interested in establishing whether or not $TS \models \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$. This formula is invalid since the path $s_0\, s_1\, (s_2\, s_4)^\omega$ never goes through a b-state. The reason that this property is not valid is as follows. At state s_2 there is a nondeterministic choice between moving either to state s_3 or to s_4. By continuously ignoring the possibility of going to s_3 we obtain a computation for which $\forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$ is invalid, hence:

$$TS \not\models \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b).$$

Usually, though, the intuition is that if there is infinitely often a choice of moving to s_3, then s_3 should be visited in some fair way.

Let us impose the unconditional fairness assumption:

$$\mathit{fair} \;\equiv\; \Box\Diamond\, a \;\wedge\; \Box\Diamond\, b.$$

We now check $\forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$ under *fair*, i.e., consider the verification problem $TS \models_{fair} \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b)$. Due to the fairness assumption, paths like $s_0\, s_1\, (s_2\, s_4)^\omega$ are excluded, since s_3 is never visited along this path. It can now easily be established that

$$TS \models_{fair} \forall\Box\,(a \;\Rightarrow\; \forall\Diamond\, b) \quad .$$

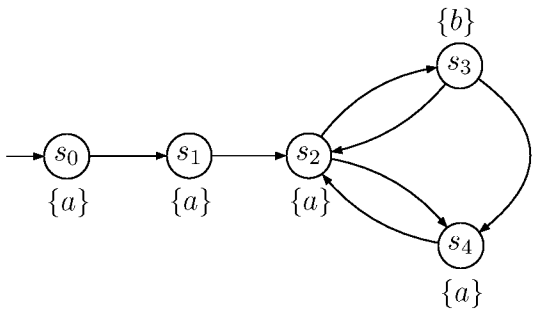


Figure 6.15: An example of a transition system.

■

Example 6.36. Mutual Exclusion

Consider the semaphore-based solution to the two-process mutual exclusion problem. The transition system of this concurrent program is denoted TS_{sem}. The CTL formula

$$\Phi \;=\; (\forall\Box\forall\Diamond\, crit_1) \;\wedge\; (\forall\Box\forall\Diamond\, crit_2)$$

describes the liveness property that both processes infinitely often have access to the critical section. It follows that $TS_{sem} \not\models \Phi$. We first impose the weak fairness assumption

$$wfair \;=\; (\Diamond\Box\, noncrit_1 \rightarrow \Box\Diamond\, wait_1) \wedge (\Diamond\Box\, noncrit_2 \rightarrow \Box\Diamond\, wait_2)$$

and the strong fairness assumption

$$sfair \;=\; (\Box\Diamond\, wait_1 \rightarrow \Box\Diamond\, crit_1) \wedge (\Box\Diamond\, wait_1 \rightarrow \Box\Diamond\, crit_1)\,.$$

It then follows that $TS_{sem} \models_{fair} \Phi$ where $fair = wfair \wedge sfair$.

As a second example, consider the arbiter-based solution to the two-process mutual exclusion problem, see Example 5.27 on page 259. The decision as to which process will acquire access to the critical section is determined by coin flipping by the arbiter. Consider the unconditional fairness assumption

$$ufair \;=\; \Box\Diamond\, head \;\wedge\; \Box\Diamond\, tail.$$

This fairness assumption can be considered to require a fair coin such that the events "heads" and "tails" occur infinitely often with probability 1. Apparently, it follows that

$$TS_1 \parallel Arbiter \parallel TS_2 \;\not\models\; \Phi, \quad \text{and} \quad TS_1 \parallel Arbiter \parallel TS_2 \;\models_{ufair}\; \Phi.$$

■

As explained before, fairness is treated in CTL by considering a fair satisfaction relation, denoted $\models_{fair}$ where *fair* is the fairness assumption considered. In the remainder of this section, algorithms will be provided in order to check whether

$$TS \models_{fair} \Phi$$

for CTL formula Φ and CTL-fairness assumption *fair*. As before, TS is supposed to be finite and have no terminal states and Φ is a CTL formula in ENF. Note that the assumption that Φ is in ENF does not impose any restriction as any CTL formula can be transformed into an equivalent (with respect to $\models_{fair}$) CTL formula in ENF. This can be established in the same way as Theorem 6.14 (page 332).

The basic idea is to exploit the CTL model-checking algorithms (without fairness) to compute $Sat_{fair}(\Phi) = \{ s \in S \mid s \models_{fair} \Phi \}$. Suppose *fair* is the strong CTL fairness constraint:

$$fair \;=\; \bigwedge_{0<i\leqslant k} (\Box \Diamond \Phi_i \rightarrow \Box \Diamond \Psi_i)$$

where Φ_i and Ψ_i are CTL formulae over AP. Recall that Φ_i and Ψ_i are interpreted according to the standard CTL semantics, i.e., without taking any fairness assumptions into account. By applying the CTL model-checking algorithm, first the sets $Sat(\Phi_i)$ and $Sat(\Psi_i)$ are determined. The formulae Φ_i and Ψ_i can thus be replaced by (fresh) atomic propositions a_i and b_i, say. It thus suffices to consider strong fairness assumptions of the form

$$fair \;=\; \bigwedge_{0<i\leqslant k} (\Box \Diamond a_i \rightarrow \Box \Diamond b_i)\,.$$

Once the fairness assumption is simplified, the sets $Sat_{fair}(\Psi)$ are determined for all subformulae Ψ of Φ using the standard CTL model-checking algorithm (i.e., without fairness), together with an algorithm to compute $Sat_{fair}(\exists\Box\, a)$ for $a \in AP$. The outcome of the model-checking procedure is "yes" if $I \subseteq Sat_{fair}(\Phi)$, and "no" otherwise.

The essential ideas are outlined in Algorithm 17 (page 364).

The subformulae of Φ are treated as in the CTL model-checking routine. Based on the syntax tree of Φ, a bottom-up computation is initiated. It is essential that during the computation of $Sat_{fair}(\Psi)$, the maximal genuine subformulae of Ψ have been already processed and replaced by atomic propositions. For the propositional logic fragment,

Algorithm 17 CTL model checking with fairness (basic idea)

Input: finite transition system TS, CTL formula Φ in ENF, and CTL fairness assumption *fair* over k CTL state formulae Φ_i and Ψ_i
Output: $TS \models_{fair} \Phi$

---

for all $0 < i \leqslant k$ **do**
  determine $Sat(\Phi_i)$ and $Sat(\Psi_i)$
  if $s \in Sat(\Phi_i)$ **then** $L(s) := L(s) \cup \{ a_i \}$; **fi**
  if $s \in Sat(\Psi_i)$ **then** $L(s) := L(s) \cup \{ b_i \}$; **fi**
od
compute $Sat_{fair}(\exists\Box\,\mathrm{true}) = \{ s \in S \mid FairPaths(s) \neq \varnothing \}$;
forall $s \in Sat_{fair}(\exists\Box\,\mathrm{true})$ **do** $L(s) := L(s) \cup \{ a_{fair}; \}$ **od**
(* compute $Sat_{fair}(\Phi)$ *)
for all $i \leqslant |\,\Phi\,|$ **do**
  for all $\Psi \in Sub(\Phi)$ with $|\,\Psi\,| = i$ **do**
    switch(Ψ):

true	:	$Sat_{fair}(\Psi) := S$;
a	:	$Sat_{fair}(\Psi) := \{ s \in S \mid a \in L(s) \}$;
$a \wedge a'$	:	$Sat_{fair}(\Psi) := \{ s \in S \mid a, a' \in L(s) \}$;
$\neg a$	:	$Sat_{fair}(\Psi) := \{ s \in S \mid a \notin L(s) \}$;
$\exists\bigcirc a$	:	$Sat_{fair}(\Psi) := Sat(\exists\bigcirc (a \wedge a_{fair}))$;
$\exists(a \,\mathsf{U}\, a')$	:	$Sat_{fair}(\Psi) := Sat(\exists(a \,\mathsf{U}\, (a' \wedge a_{fair})))$;
$\exists\Box a$	:	compute $Sat_{fair}(\exists\Box a)$

    end switch
    replace all occurrences of Ψ in Φ by the atomic proposition a_Ψ;
    forall $s \in Sat_{fair}(\Psi)$ **do** $L(s) := L(s) \cup \{ a_\Psi \}$ **od**
  od
od
return $I \subseteq Sat_{fair}(\Phi)$

---

the approach is straightforward:

$$\begin{array}{lcl} Sat_{fair}(\text{true}) & = & S \\ Sat_{fair}(a) & = & \{\, s \in S \mid a \in L(s) \,\} \\ Sat_{fair}(\neg a) & = & S \setminus Sat_{fair}(a) \\ Sat_{fair}(a \wedge a') & = & Sat_{fair}(a) \cap Sat_{fair}(a'). \end{array}$$

For all nodes of the syntax tree that are labeled with either $\exists\bigcirc$ or $\exists\,\mathsf{U}$ (i.e, nodes that represent a subformula of the form $\Psi = \exists\bigcirc a$ or of the form $\Psi = \exists(a\,\mathsf{U}\,a')$), the following observation is used. For any infinite path fragment π in TS, π is fair if and only if one (or all) suffix(es) of π is (are) fair:

$$\pi \models \mathit{fair} \quad \text{iff} \quad \pi[j..] \models \mathit{fair} \text{ for some } j \geqslant 0 \quad \text{iff} \quad \pi[j..] \models \mathit{fair} \text{ for all } j \geqslant 0.$$

The following two lemmas provide the ingredients for checking subformulae of the "type" $\exists\bigcirc$ and $\exists\,\mathsf{U}$.

Lemma 6.37. Next Step for Fair Satisfaction Relation

$s \models_{fair} \exists\bigcirc a$ if and only if $\exists s' \in Post(s)$ with $s' \models a$ and $FairPaths(s') \neq \varnothing$.

Proof: $\Rightarrow$: Assume $s \models_{fair} \exists\bigcirc a$. Then there exists a fair path $\pi = s_0\, s_1\, s_2\, s_3 \ldots \in Paths(s)$ with $s_1 \models a$. Since π is fair, the path $\pi[1..] = s_1\, s_2\, s_3 \ldots$ is fair too. Thus, $s' = s_1 \in Post(s)$ satisfies the indicated property.

$\Leftarrow$: Assume $s' \models a$ and $FairPaths(s') \neq \varnothing$ for some $s' \in Post(s)$. Thus there exists a fair path

$$\pi' \;=\; s'\, s'_1\, s'_2\, s'_3 \ldots$$

starting in s'. Therefore, $\pi \;=\; s\, s'\, s'_1\, s'_2\, s'_3 \ldots$ is a fair path starting s such that $\pi \models \bigcirc a$. Thus, $s \models_{fair} \exists\bigcirc a$. ∎

Using analogous arguments we obtain:

Lemma 6.38. Until for Fair Satisfaction Relation

$s \models_{fair} \exists(a_1\,\mathsf{U}\,a_2)$ if and only if there exists a finite path fragment

$$s_0\, s_1\, s_2\, s_3 \ldots s_n \in Paths_{fin}(s) \qquad \text{with } n \geqslant 0$$

such that $s_i \models a_1$ for $0 \leqslant i < n$, $s_n \models a_2$, and $FairPaths(s_n) \neq \varnothing$.

The results of the previous two lemmas lead to the following approach. As a first step, the set

$$Sat_{fair}(\exists\Box\,\text{true}) \;=\; \{\, s \in S \mid FairPaths(s) \neq \varnothing \,\}$$

is computed. (The algorithmic way to do so is explained later in this chapter.) That is, all states are determined for which it is guaranteed that at least one fair path emanates. Once such a state is visited, it is thus guaranteed that a fair continuation is possible. The labels of the thus computed states are extended with the new atomic proposition a_{fair}, i.e.:

$$a_{fair} \in L(s) \quad \text{if and only if} \quad s \in Sat_{fair}(\exists\Box\,\text{true}).$$

The sets $Sat_{fair}(\exists\bigcirc a)$ and $Sat_{fair}(\exists(a \,\mathsf{U}\, a'))$ result from

$$\begin{aligned} Sat_{fair}(\exists\bigcirc a) &= Sat\,(\exists\bigcirc (a \wedge a_{fair}))\,, \\ Sat_{fair}(\exists(a \,\mathsf{U}\, a')) &= Sat\,(\exists(a \,\mathsf{U}\, (a' \wedge a_{fair})))\,. \end{aligned}$$

As a result, these satisfaction sets can be computed using an ordinary CTL model checker. These considerations lead to Algorithm 17 on page 364 and provide the basis for the following result:

Theorem 6.39. Model-Checking CTL with Fairness

The model-checking problem for CTL with fairness can be reduced to

- *the model-checking problem for CTL (without fairness), and*
- *the problem of computing* $Sat_{fair}(\exists\Box a)$ *for the atomic proposition* a.

Note that the set $Sat_{fair}(\exists\Box\,\text{true})$ corresponds to $Sat_{fair}(\exists\Box\, a)$ whenever every state is labeled with a. Thus, an algorithm to compute $Sat_{fair}(\exists\Box\, a)$ can also be used to compute $Sat_{fair}(\exists\Box\,\text{true})$.

In the following, we explain how to compute the satisfaction set $Sat_{fair}(\exists\Box\, a)$ for $a \in AP$ in case *fair* is a *strong* CTL fairness assumption. Weak fairness assumptions can be treated in an analogous way. As we will see, unconditional fairness assumptions are a special case. Arbitrary fairness assumptions with unconditional, strong, and weak fairness constraints can be treated by using algorithms in which the corresponding techniques are appropriately combined.

Consider the strong CTL fairness assumption over the atomic propositions a_i and b_i ($0 < i \leqslant k$):

$$sfair \;=\; \bigwedge_{0<i\leqslant k} (\Box\Diamond a_i \rightarrow \Box\Diamond b_i).$$

The following lemma provides a graph-theoretical characterization of the fair satisfaction set $Sat_{sfair}(\exists\Box\, a)$ where a is an atomic proposition.

Lemma 6.40. Characterization of $Sat_{sfair}(\exists\Box a)$

$s \models_{sfair} \exists\Box\, a$ if and only if there exists a finite path fragment $s_0\, s_1 \ldots s_n$ and a cycle $s'_0\, s'_1 \ldots s'_r$ such that

(1) $s_0 = s \quad$ *and* $\quad s_n = s'_0 = s'_r$,

(2) $s_i \models a$, *for any* $0 \leqslant i \leqslant n$, *and* $s'_j \models a$, *for any* $0 < j \leqslant r$, *and*

(3) $Sat(a_i) \cap \{\, s'_1, \ldots, s'_r \,\} = \varnothing \quad$ *or* $\quad Sat(b_i) \cap \{\, s'_1, \ldots, s'_r \,\} \neq \varnothing$ *for all* $1 \leqslant i \leqslant k$.

Proof: $\Leftarrow$: Assume there exists a finite path fragment $s_0\, s_1 \ldots s_n$ and a cycle $s'_0\, s'_1 \ldots s'_r$ such that the conditions (1) through (3) hold for state s. Consider the infinite path fragment $\pi = s_0\, s_1 \ldots s_n\, s'_1 \ldots s'_r\, s'_1 \ldots s'_r \ldots \in Paths(s)$ which is obtained by traversing the cycle $s'_0\, s'_1 \ldots s'_r$ infinitely often. From constraints (1) and (3), it follows that π is fair, i.e., $\pi \models sfair$. From (2), it follows that $\pi \models \Box a$. Thus, $s \models_{sfair} \exists\Box\, a$.

$\Rightarrow$: Assume $s \models_{sfair} \exists\Box\, a$. Since $s \models_{sfair} \exists\Box\, a$, there exists an infinite path fragment $\pi = s_0\, s_1\, s_2 \ldots$ with $\pi \models \Box a$ and $\pi \models sfair$. Let $i \in \{\, 1, \ldots, k \,\}$. Distinguish between two cases:

1. $\pi \models \Box\Diamond a_i$. Since $\pi \models sfair$, there is a state $s' \in Sat(b_i)$ which is infinitely often visited in π . Let $n \in \mathbb{N}$ such that $s_n = s'$ and $s' \notin \{\, s_0, s_1, \ldots, s_{n-1} \,\}$. That is, s_n is the first occurrence of s' in π. Further, let $r > n$ such that $s_n = s_r$. Then $s_n\, s_{n+1} \ldots s_{r-1}\, s_r = s_n$ is a cycle with

$$s_n \in Sat(b_i) \cap \{\, s_n, s_{n+1}, \ldots, s_{r-1}, s_r \,\}.$$

2. $\pi \not\models \Box\Diamond a_i$. Then there exists an index n such that $s_n, s_{n+1}, s_{n+2} \ldots \notin Sat(a_i)$. Assume without loss of generality that s_n occurs infinitely often in π. Since there are finitely many states, there exists an $r > n$ such that $s_n = s_r$. Thus, $s_n\, s_{n+1} \ldots s_{r-1}\, s_r = s_n$ is a cycle with $Sat(a_i) \cap \{\, s_n, s_{n+1}, \ldots, s_{r-1}, s_r \,\} = \varnothing$.

From both cases it follows that there exists a finite path fragment $s_0\, s_1 \ldots s_n$ and a cycle $s'_0 \ldots s'_r$ satisfying the constraints (1) through (3). ■

This characterization is used to compute $Sat_{sfair}(\exists\Box\, a)$ in the following way. Consider the directed graph $G[a] = (S, E_a)$ whose set of edges E_a is defined as

$$(s, s') \in E_a \quad \text{if and only if} \quad s' \in Post(s) \ \wedge\ s \models a \ \wedge\ s' \models a.$$

Stated in words, $G[a]$ is obtained from the state graph G_{TS} by eliminating all edges (s, s') for which either $s \not\models a$ or $s' \not\models a$. (Thus, $G[a]$ is the state graph of the transition system $TS[a]$.) Apparently, each infinite path in $G[a]$ (that starts in $s \in I$) corresponds to an infinite path in the transition system TS satisfying $\Box a$. Conversely, each infinite path fragment π of TS with $\pi \models \Box a$ is a path in $G[a]$. In particular,

$$s \models_{sfair} \exists \Box a$$

if and only if there exists a nontrivial, strongly connected set of nodes D in $G[a]$ that is reachable from s, such that for all $1 \leqslant i \leqslant k$:

$$D \cap Sat(a_i) = \varnothing \qquad \text{or} \qquad D \cap Sat(b_i) \neq \varnothing.$$

Let T be the union of all nontrivial strongly connected components C in the graph $G[a]$ such that *sfair* is realizable in C, i.e., there exists a nontrivial strongly connected subset D of C satisfying $D \cap Sat(a_i) = \varnothing$ or $D \cap Sat(b_i) \neq \varnothing$, for all $1 \leqslant i \leqslant k$. It then follows that

$$Sat_{sfair}(\exists \Box\, a) \;=\; \{\, s \in S \mid Reach_{G[a]}(s) \cap T \neq \varnothing \,\}.$$

Here, $Reach_{G[a]}(s)$ is the set of states that is reachable from s in $G[a]$. Note that as SCC D is contained in SCC C, D can be reached from any state in C; reachability of C is thus of relevance.

The key part of the model-checking algorithm is now the computation of the set T. This amounts to investigating the SCCs C of $G[a]$ and checking for which SCC *sfair* is realizable. As the computation of T for the general case is slightly involved, we first consider two special (and simpler) cases: a_i = true for all i, i.e., unconditional fairness, and the case for which k=1, i.e., a single strong fairness constraint.

In the sequel, each set C of nodes in $G[a]$ is identified with the subgraph (C, E_C) where E_C results from the edge relation in $G[a]$ by removing all edges with starting or target vertex not in C.

Unconditional Fairness Let a_i = true for all $1 \leqslant i \leqslant k$. In this case

$$sfair \;\equiv \bigwedge_{1 \leqslant i \leqslant k} \Box \Diamond b_i.$$

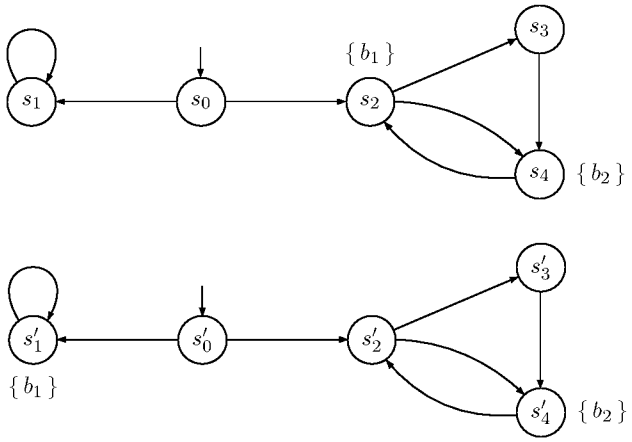
Obviously, $s \models_{sfair} \exists \Box\, a$ if and only if there exists a nontrivial SCC C in $G[a]$ that is reachable from s, such that C contains at least one b_i-state for any i. In that case, $\Box \Diamond b_i$ is realizable in SCC C, for any i. Let T be the set union of all nontrivial SCCs C of $G[a]$ satisfying $C \cap Sat(b_i) \neq \varnothing$ for all $1 \leqslant i \leqslant k$. It now follows that

$$s \models_{sfair} \exists \Box\, a \quad \text{if and only if} \quad Reach_{G[a]}(s) \cap T \neq \varnothing.$$

The following example illustrates this.

Example 6.41.

Consider the transition systems TS (upper) and TS' (lower) in which it is implicitly assumed that all states are labeled with the atomic proposition a:



(Due to this assumption, $TS[a] = TS$ and $TS'[a] = TS'$.) Consider the unconditional fairness assumption:

$$sfair \;=\; \Box\Diamond b_1 \;\wedge\; \Box\Diamond b_2.$$

The transition system TS contains the reachable nontrivial SCC $C = \{ s_2, s_3, s_4 \}$ such that $C \cap Sat(b_1) \neq \varnothing$ and $C \cap Sat(b_2) \neq \varnothing$. We thus have $s_0 \models_{sfair} \exists\Box\, a$, and hence $TS \models_{sfair} \exists\Box\, a$. On the other hand, TS' contains two non-trivial SCCs, but none that contains a b_1-state *and* a b_2-state. Therefore $s'_0 \not\models_{sfair} \exists\Box\, a$ and hence $TS' \not\models_{sfair} \exists\Box\, a$. ■

Strong Fairness with a Single Fairness Constraint Assume k=1. i.e., *sfair* is assumed to be of the form

$$sfair \;=\; \Box\Diamond a_1 \rightarrow \Box\Diamond b_1.$$

We have that $s \models_{sfair} \exists\Box a$ if and only if there exists a nontrivial strongly connected component C of $G[a]$ with $C \subseteq Reach_{G[a]}(s)$ such that at least one of the following two conditions (1) or (2) holds:

(1) $C \cap Sat(b_1) \neq \varnothing$, or

(2) $D \cap Sat(a_1) = \varnothing$ for some nontrivial SCC D of C.

Algorithm 18 Computation of $Sat_{sfair}(\exists\Box a)$

Input: A finite TS without terminal states, $a \in AP$ and $fair = \bigwedge_{0<i\leqslant k} sfair_i$ with $sfair_i = \Box\Diamond a_i \rightarrow \Box\Diamond b_i$
Output: $\{ s \in S \mid s \models_{fair} \exists\Box a \}$

```
compute the SCCs of the state graph G[a] of TS[a];
T := ∅;
for all nontrivial SCCs C in G[a] do
                        (* check whether the fairness assumption sfair can be realized in C *)
  if CheckFair(C, k, sfair_1, ..., sfair_k) then
    T := T ∪ C;
  fi
od
return { s ∈ S | Reach_{G[a]}(s) ∩ T ≠ ∅ }          (* e.g., backwards reachability *)
```

Intuitively, in case (1) C stands for a cyclic set of states that realizes $\Box\Diamond b_1$, while D in (2) represents a cyclic set of states for which $\neg a_1$ continuously holds. The SCCs D of C that do not contain any a_1-state can easily be computed by determining the nontrivial SCCs in the graph that is obtained from C by eliminating all a_1-states. (Stated differently, C realizes the unconditional fairness constraint $\Box\Diamond b_1$, while D realizes the unconditional fairness constraint $\Box\Diamond \mathrm{true}$ in the transition system induced by C after eliminating all a_1-states. This characterization is helpful to understand the general algorithm below.) Every path that has a suffix consisting of the infinite repetition of a cycle that runs through all states of C (case (1)) or that eventually reaches D and stays there forever satisfies the fairness assumption *sfair*. Accordingly, we may define T as the union of all nontrivial SCCs C of $G[a]$ satisfying the above constraints (1) and (2). Then, $s \models_{sfair} \exists\Box a$ if and only if $Reach_{G[a]}(s) \cap T \neq \varnothing$.

Strong Fairness with $k > 1$ Fairness Constraints In this case, *sfair* is defined for $k > 1$ by

$$sfair \;=\; \bigwedge_{1\leqslant i\leqslant k} sfair_i \quad \text{with} \quad sfair_i \;= \Box\Diamond a_i \rightarrow \Box\Diamond b_i.$$

As for the other cases, the initial step is to determine $G[a]$ and its set of SCCs. We have to compute all nontrivial SCCs C such that for some cyclic subset D of C all fairness constraints $\Box\Diamond a_i \rightarrow \Box\Diamond b_i$ are realizable in D, i.e., for $1 \leqslant i \leqslant k$:

$$D \cap Sat(a_i) = \varnothing \quad \text{or} \quad D \cap Sat(b_i) \neq \varnothing.$$

Algorithm 18 on page 370 provides the schema for the computation of $Sat_{sfair}(\exists\Box\, a)$. It uses the recursive procedure *CheckFair* (see Algorithm 19, page 372) to check the real-

izability of *sfair* in SCC C of $G[a]$. The inputs of *CheckFair* are a cyclic strongly connected subgraph D of $G[a]$, index j, and j strong fairness assumptions $sfair_{i_1}, \ldots, sfair_{i_j}$. $CheckFair(D, j, sfair_{i_1}, \ldots, sfair_{i_j})$ returns true if $\bigwedge_{1 \leqslant \ell \leqslant j} sfair_{i_\ell}$ is realizable in D. Thus, the invocation $CheckFair(C, k, sfair_1, \ldots, sfair_k)$ yields an affirmative answer whether $C \subseteq T$ by returning true if and only if the fairness assumption $sfair = \bigwedge_{1 \leqslant i \leqslant k} sfair_i$ is realizable in C.

The idea of Algorithm 19, $CheckFair(C, k, sfair_1, \ldots, sfair_k)$, is as follows. First, it is checked whether $C \cap Sat(b_i) \neq \varnothing$ for each fairness constraint $sfair_i = \Box\Diamond a_i \rightarrow \Box\Diamond b_i$.

- If yes, then $sfair = \bigwedge_{0<i\leqslant k} sfair_i$ is realizable in C.
- Otherwise, there exists an index $j \in \{1, \ldots, k\}$ such that $C \cap Sat(b_j) = \varnothing$. The goal is then to realize the condition

$$\bigwedge_{\substack{0<i\leqslant k \\ i\neq j}} sfair_i \ \wedge \ \Box\neg a_j$$

  in C. For this, we investigate the subgraph $C[\neg a_j]$ of C that arises by removing all states where a_j holds and their incoming and outgoing edges. The goal is to check the existence of a cyclic subgraph of $C[\neg a_j]$ such that the remaining $k-1$ fairness constraints $sfair_1, \ldots, sfair_{j-1}, sfair_{j+1}, \ldots, sfair_k$ are realizable in this subgraph. This is done by analyzing the nontrivial SCCs D of $C[\neg a_j]$.

  - If there is no nontrivial SCC D of $C[\neg a_j]$, then *sfair* is not realizable in C.
  - Otherwise, invoke $CheckFair(D, k{-}1, \ldots)$ for each of these nontrivial SCCs D of $C[\neg a_j]$ to check whether the remaining $k{-}1$ fairness constraints are realizable in D.

The main steps of this procedure are summarized in Algorithm 19 on page 372. Note that in each recursive call one of the fairness constraints is removed, and thus the recursion depth is at most k, in which case $k = 0$ and the condition "$\forall i \in \{1, \ldots, k\}.C \cap Sat(b_j) \neq \varnothing$" of the first ifstatement is obviously fulfilled.

The cost function for $CheckFair(C, k, sfair_1, \ldots, sfair_k)$, is given by the recurrence equation:

$$T(n,k) = \mathcal{O}(n) + \max\{ \sum_{1\leqslant\ell\leqslant r} T(n_\ell, k{-}1) \mid n_1, \ldots, n_r \geqslant 1, n_1 + \ldots + n_r \leqslant n\}$$

where n is the size (number of vertices and edges) of the subgraph C of $G[a]$. The values $n_1, \ldots, n_r$ denote the sizes of the nontrivial strongly connected components $D_1, \ldots, D_r$ of $C[\neg a_j]$. It is easy to see that the solution of this recurrence is $\mathcal{O}(n{\cdot}k)$. This yields:

Algorithm 19 Recursive algorithm $CheckFair(C, k, sfair_1, \ldots, sfair_k)$

Input: nontrivial SCC C in $G[a]$, and strong fairness constraints $sfair_i = \Box\Diamond a_i \rightarrow \Box\Diamond b_i$, $i = 1, \ldots, k$.
Output: true if $\bigwedge_{1 \leqslant i \leqslant k} sfair_i$ is realizable in C. Otherwise false.

---

if $\forall i \in \{1, \ldots, k\}. C \cap Sat(b_j) \neq \varnothing$ **then**
  return true (* $\bigwedge_{1 \leqslant i \leqslant k} \Box\Diamond b_i$ is realizable in C *)
else
  choose an index $j \in \{1, \ldots, k\}$ with $C \cap Sat(b_j) = \varnothing$;
  if $C[\neg a_j]$ is acyclic (or empty) **then**
    return false
  else
    compute the nontrivial SCCs of $C[\neg a_j]$;
    for all nontrivial SCC D of $C[\neg a_j]$ **do**
      if $CheckFair(D, k-1, sfair_1, \ldots, sfair_{j-1}, sfair_{j+1}, \ldots, sfair_k)$ **then**
        return true
      fi
    od
  fi

fi
return false

Theorem 6.42. Time Complexity of Verifying $\exists\Box a$ under Fairness

For transition system TS with N states and K transitions, and CTL strong fairness assumption fair with k conjuncts, the set $Sat_{fair}(\exists\Box\, a)$ can be computed in $\mathcal{O}((N+K)\cdot k)$.

The time complexity is thus linear in the size of the transition system and the number of imposed fairness constraints. The set T resulting from all SCCs C of $G[a]$, for which $CheckFair(C,1)$ returns the value true, can be computed (with appropriate implementation) in time $\mathcal{O}(size(G[a])\cdot k)$. A reachability analysis (by, e.g., depth-first search) results in the set

$$Sat_{fair}(\exists\Box a) \quad = \quad \{\, s \in S \mid Reach_{G[a]}(s) \cap T \neq \varnothing \,\}.$$

Gathering these results yields that CTL model checking under strong fairness constraints and for CTL formulae in ENF can be done in time linear in the size of the transition system, the length of the formula, and the number of (strong) fairness constraints. This, in fact, also holds for arbitrary CTL formulae (with universal quantification which can be treated by techniques similar as the ones we presented for existential quantification) and weak fairness constraints, or any mixture of unconditional, strong, and weak fairness constraints. We thus obtain:

Theorem 6.43. Time Complexity of CTL Model Checking with Fairness

For transition system TS with N states and K transitions, CTL formula Φ, and CTL fairness assumption fair with k conjuncts, the CTL model-checking problem $TS \models_{fair} \Phi$ can be determined in time $\mathcal{O}((N+K)\cdot|\Phi|\cdot k)$.

6.6 Counterexamples and Witnesses

A major strength of model checking is the possibility of generating a counterexample in case a formula is refuted. Let us first explain what is meant by a counterexample. In the case of LTL, a counterexample for $TS \not\models \varphi$ is a sufficiently long prefix of a path π that indicates why π refutes φ. For instance, a counterexample for the LTL formula $\Diamond\, a$ is a finite prefix of just $\neg a$-states that ends with a single cycle traversal. Such counterexample suggests that there is a $\Box\,\neg a$-path. Similarly, a counterexample for $\bigcirc\, a$ consists of a path π for which $\pi[1]$ violates a.

For CTL the situation is somewhat more involved due to the existential path quantification. For CTL formulae of the form $\forall\varphi$ a sufficiently long prefix of π with $\pi \not\models \varphi$ provides—as in LTL—sufficient information about the source of the refutation. In the

case of a path formula of the form $\exists\varphi$, it is unclear what a counterexample will be: if $TS \not\models \exists\varphi$, then all paths violate φ and no path satisfies φ. However, if one checks the formula $\exists\varphi$ for a transition system TS, then it is quite natural that for the answer "yes, $TS \models \exists\varphi$" one aims at an initial path where φ holds, while the answer "no" might be sufficient.[6] Therefore, CTL model checking supports the system diagnosis by providing *counterxamples* or *witnesses*. Intuitively, counterexamples indicate the refutation of universally quantified path formulae, while witnesses indicate the satisfaction of existentially quantified path formulae. From a path-based view, the concepts are counterexamples and witnesses can be explained as follows:

- a sufficiently long prefix of a path π with $\pi \not\models \varphi$ is a *counterexample* for the CTL path formula $\forall\varphi$, and
- a sufficiently long prefix of a path π with $\pi \models \varphi$ is a *witness* of the CTL path formula $\exists\varphi$.

To exemplify the idea of generating a witness, consider the following well-known combinatorial problem.

Example 6.44. The Wolf-Goat-Cabbage Problem

Consider the problem of bringing a ferryman (f), a goat (g), a cabbage (c) and a wolf (w) from one side of a river to the other side. The only available means to go from one riverside to another is a small boat that is capable of carrying at most two occupants. In order for the boat to be steered, one of the occupants needs to be the ferryman. Of course, the boat does not need to be fully occupied, and the ferryman can cross the river alone. For obvious reasons, neither the goat and the cabbage nor the goat and the wolf should be left unobserved by the ferryman. The question is whether there exists a series of boat movements such that all three items can be carried by the ferryman to the other side of the river.

This problem can be represented as a CTL model-checking problem in a rather natural way. The behavior of the goat, wolf, and cabbage is provided by a simple two-state transition system depicted in Figure 6.16. The state identifiers indicate the position of each of the items: 0 stands for the current (i.e., starting) riverside, and 1 for the other side

[6] Since the standard CTL model-checking procedure calculates the set of states where the given (state) formula Φ holds, also some information extracted from $Sat(\Phi)$ could be returned to the user. For instance, if $TS \not\models \exists\varphi$, then the model checker might return the set of initial states s_0 where $s_0 \not\models \exists\varphi$. This information could be understood as a counterexample and used for debugging purposes. This issue will not be addressed here. Instead we will discuss the concept of path-based counterexamples (also often called "error traces") and their duals, i.e., computations that provide a proof for the existence of computations with certain properties.

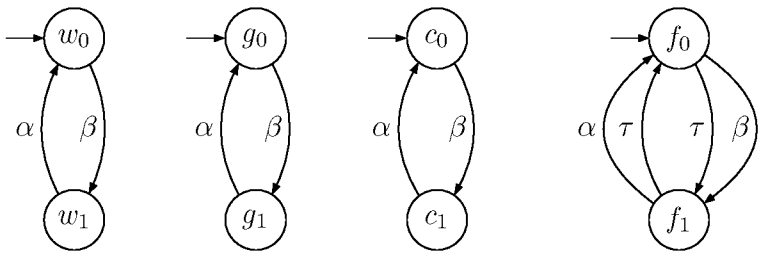


Figure 6.16: Transition systems for the wolf, goat, cabbage, and ferryman.

(i.e., the goal). The synchronization actions α and β are used to describe the boat trips that are taking place together with the ferryman. The ferryman behaves very similarly, but has in addition the possibility to cross the river alone. This corresponds to the two τ-labeled transitions. The resulting transition system representing the entire "system"

$$TS \;=\; (\mathit{wolf} \,|||\, \mathit{goat} \,|||\, \mathit{cabbage}) \,\|\, \mathit{ferryman},$$

has $2^4 = 16$ states. The resulting transition system is depicted in Figure 6.17. Note that the transitions are all bidirectional as each boat movement can be reversed. The existence of a sequence of boat movements to bring the two animals and the cabbage to the other riverbank can be expressed as the CTL state formula $\exists\varphi$ where

$$\varphi \;=\; \Big(\bigwedge_{i=0,1} (w_i \wedge g_i \rightarrow f_i) \;\wedge\; (c_i \wedge g_i \rightarrow f_i) \Big) \;\mathsf{U}\; (c_1 \wedge f_1 \wedge g_1 \wedge w_1).$$

Here, the left part of the until formula forbids the scenarios in which the wolf and the goat, or the cabbage and the goat are left unobserved. A witness of the CTL path formula φ is an initial finite path fragment which leads from

$$\text{initial state } \langle c_0, f_0, g_0, w_0 \rangle \text{ to target state } \langle c_1, f_1, g_1, w_1 \rangle$$

and which does not pass through any of the (six) states in which the wolf and the goat or the goat and the cabbage are left alone on one riverbank. That is, e.g., the states $\langle c_0, f_0, g_1, w_1 \rangle$ and $\langle c_1, f_0, g_1, w_0 \rangle$ should be avoided. An example witness for φ is:

$\langle c_0, f_0, g_0, w_0 \rangle$	goat to riverbank 1
$\langle c_0, f_1, g_1, w_0 \rangle$	ferryman comes back to riverbank 0
$\langle c_0, f_0, g_1, w_0 \rangle$	cabbage to riverbank 1
$\langle c_1, f_1, g_1, w_0 \rangle$	goat back to riverbank 0
$\langle c_1, f_0, g_0, w_0 \rangle$	wolf to riverbank 1
$\langle c_1, f_1, g_0, w_1 \rangle$	ferryman comes back to riverbank 0
$\langle c_1, f_0, g_0, w_1 \rangle$	goat to riverbank 1
$\langle c_1, f_1, g_1, w_1 \rangle$	

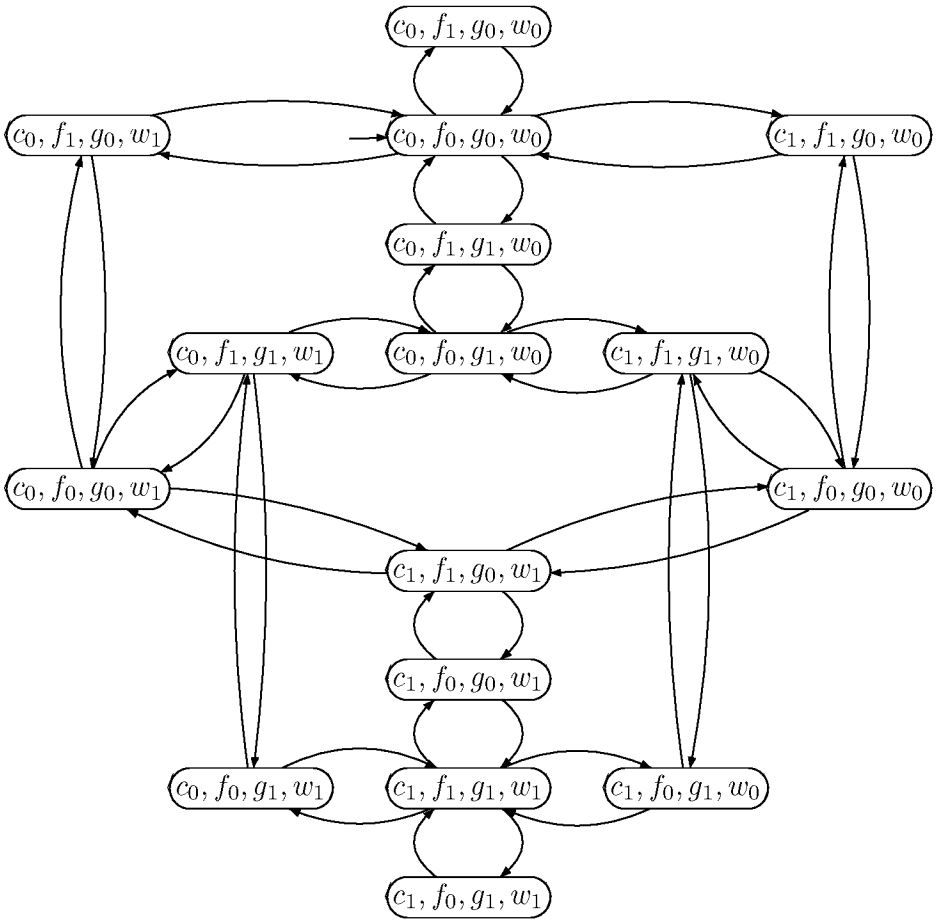


Figure 6.17: Transition system of the wolf-goat-cabbage problem.

■

6.6.1 Counterexamples in CTL

Now we explain how to generate counterexamples or witnesses for CTL (path) formulae. We consider here path formulae of the form $\bigcirc \Phi$, $\Phi \,\mathsf{U}\, \Psi$, and $\Box\Phi$. (Techniques for other operators, such as W or R, can be derived. Alternatively, corresponding considerations can be made for them.)

In the sequel, let $TS = (S, Act, \rightarrow, I, AP, L)$ be a finite transition system without terminal states.

The Next Operator A counterexample of $\varphi = \bigcirc \Phi$ is a pair of states (s, s') with $s \in I$ and $s' \in \mathit{Post}(s)$ such that $s' \not\models \Phi$. A witness of $\varphi = \bigcirc \Phi$ is a pair of states (s, s') with $s \in I$ and $s' \in \mathit{Post}(s)$ with $s' \models \Phi$. Thus, counterexamples and witnesses for the next-step operator result from inspecting the immediate successors of the initial states of TS.

The Until Operator A witness of $\varphi = \Phi \,\mathsf{U}\, \Psi$ is an initial path fragment $s_0\, s_1 \ldots s_n$ for which

$$s_n \models \Psi \quad \text{and} \quad s_i \models \Phi \text{ for } 0 \leqslant i < n.$$

Witnesses can be determined by a backward search starting in the set of Ψ-states.

A counterexample is an initial path fragment that indicates a path π for which either:

$$\pi \models \Box(\Phi \wedge \neg\Psi) \quad \text{or} \quad \pi \models (\Phi \wedge \neg\Psi) \,\mathsf{U}\, (\neg\Phi \wedge \neg\Psi).$$

For the first case, a counterexample is an initial path fragment of the form

$$\underbrace{s_0\, s_1 \ldots s_{n-1}\, \underbrace{s_n\, s'_1 \ldots s'_r}_{\text{cycle}}}_{\text{satisfy } \Phi \wedge \neg\Psi} \quad \text{with } s_n = s'_r.$$

For the second case, an initial path fragment of the form

$$\underbrace{s_0\, s_1 \ldots s_{n-1}}_{\text{satisfy } \Phi \wedge \neg\Psi}\; s_n \quad \text{with } s_n \models \neg\Phi \wedge \neg\Psi$$

does suffice as counterexample. Counterexamples can be determined by an analysis of the digraph $G = (S, E)$ where

$$E \;=\; \{\, (s, s') \in S \times S \mid s' \in \mathit{Post}(s) \;\wedge\; s \models \Phi \wedge \neg\Psi \,\}.$$

Then the SCCs of G are determined. Each path in G that starts in an initial state $s_0 \in S$ and leads to a nontrivial SCC C in G provides a counterexample of the form

$$s_0\, s_1 \ldots s_n \underbrace{s'_1 \ldots s'_r}_{\in C} \quad \text{with} \quad s_n = s'_r.$$

Each path in G that leads from an initial state s_0 to a trivial terminal SCC

$$C = \{\, s' \,\} \quad \text{with} \quad s' \not\models \Psi$$

provides a counterexample of the form $s_0\, s_1 \ldots s_n$ with $s_n \models \neg\Phi \wedge \neg\Psi$.

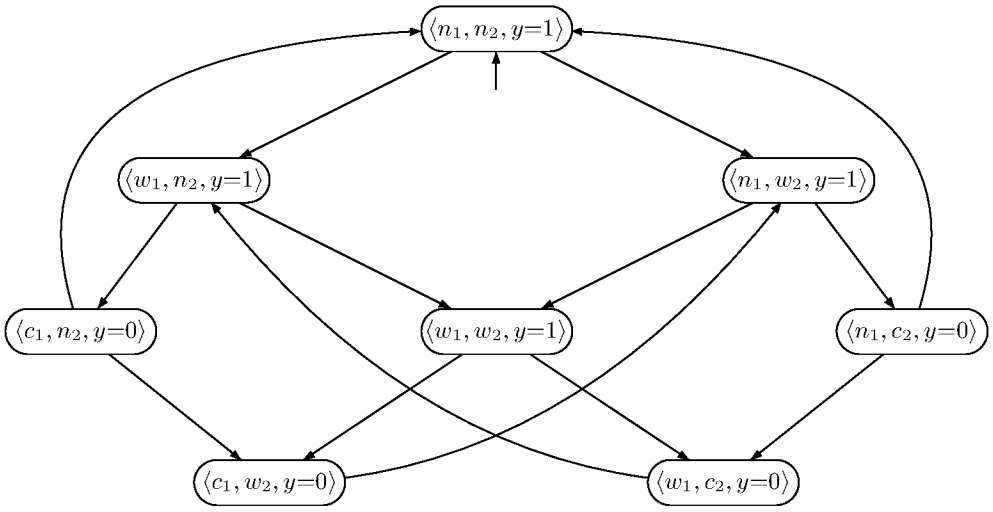


Figure 6.18: Transition system of semaphore-based mutual exclusion algorithm.

The Always Operator A counterexample for the formula $\varphi = \Box\Phi$ is an initial path fragment $s_0\, s_1 \ldots s_n$ such that $s_i \models \Phi$ for $0 \leqslant i < n$ and $s_n \not\models \Phi$. Counterexamples may be determined by a backward search starting in $\neg\Phi$-states.

A witness of $\varphi = \Box\,\Phi$ consists of an initial path fragment of the form

$$\underbrace{s_0\, s_1 \ldots s_n\, s'_1 \ldots s'_r}_{\text{satisfy } \Phi} \quad \text{with} \quad s_n = s'_r.$$

Witnesses can be determined by a simple cycle search in the digraph $G = (S, E)$ where the set of edges E is obtained from the transitions emanating from Φ-states, i.e., $E = \{\, (s, s') \mid s' \in \mathit{Post}(s) \ \wedge \ s \models \Phi \,\}$.

Example 6.45. Counterexamples and Semaphore-Based Mutual Exclusion

Recall the two-process mutual exclusion algorithm that exploits a binary semaphore y to resolve contention (see Eaxmple 3.9 on page 98). For convenience, the transition system TS_{Sem} of this algorithm is depicted in Figure 6.18. Consider the CTL formula over $AP = \{\, c_1, c_2, n_1, n_2, w_1, w_2 \,\}$:

$$\forall(\underbrace{((n_1 \wedge n_2) \ \vee \ w_2)}_{\Phi} \ \mathsf{U} \ \underbrace{c_2}_{\Psi}).$$

It expresses that process P_2 acquires access to the critical section once it starts waiting to enter it.

Note that the state labeling of TS_{Sem} can be directly obtained from the information in

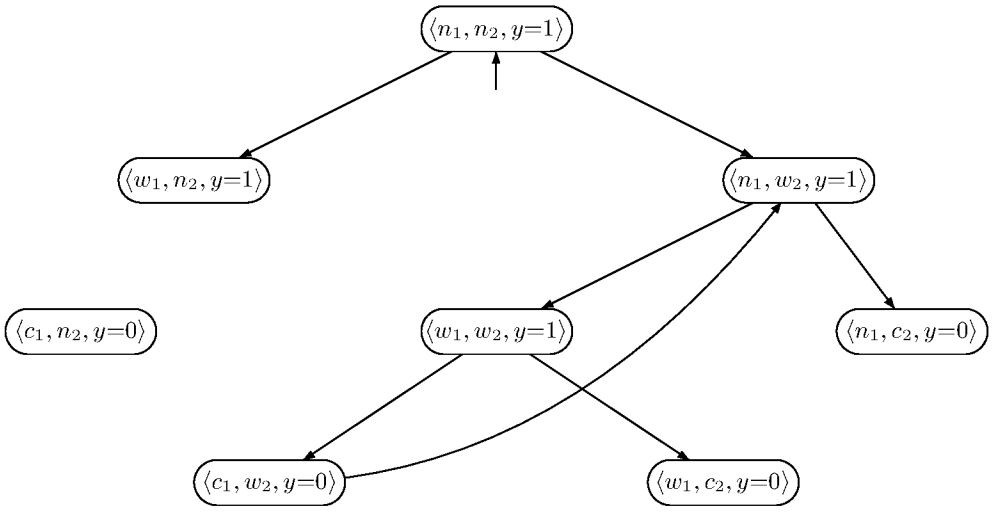


Figure 6.19: Graph to detect counterexamples for $\forall(((n_1 \wedge n_2) \vee w_2) \,\mathsf{U}\, c_2)$.

the states. Evidently, the CTL formula is refuted by TS_{Sem}. Counterexamples can be established as follows. First, the graph G is determined by only allowing edges that emanate from states satisfying $\Phi \wedge \neg\Psi$. This entails that all edges emanating from s with $s \models \neg((n_1 \wedge n_2) \vee w_2) \vee c_2$ are eliminated. This yields the graph depicted in Figure 6.19. The graph G contains a single nontrivial SCC C that is reachable from the initial state in TS_{Sem}. The initial path fragment:

$$\langle n_1, n_2, y{=}1\rangle \underbrace{\langle n_1, w_2, y{=}1\rangle \langle w_1, w_2, y{=}1\rangle \langle c_1, w_2, y{=}0\rangle}_{\in C}$$

is a counterexample as it shows that there is a path π in TS_{Sem} satisfying $\Box(((n_1 \wedge n_2) \vee w_2) \wedge \neg c_2)$, i.e., a path for which c_2 is never established.

Alternatively, any path from the initial state to the trivial terminal SCC satisfying $\neg c_2$ is a counterexample as well. This only holds for the terminal SCC $\{\langle w_1, n_2, y{=}1\rangle\}$. ■

Theorem 6.46. Time Complexity of Counterexample Generation

Let TS be a transition system TS with N states and K transitions and φ a CTL- path formula. If $TS \not\models \forall\varphi$, then a counterexample for φ in TS can be determined in time $\mathcal{O}(N{+}K)$. The same holds for a witness for φ, provided that $TS \models \exists\varphi$.

6.6.2 Counterexamples and Witnesses in CTL with Fairness

In the case CTL fairness assumptions are imposed, witnesses and counterexamples can be provided in a similar way as for CTL. Suppose *fair* is the fairness assumption of interest.

The Next Operator A witness for $\pi \models_{fair} \bigcirc a$ originates from a witness for $\pi \models \bigcirc(a \wedge a_{fair})$. Note that the latter is obtained as for CTL, as no fairness is involved. A counterexample to $\bigcirc a$ is a prefix of a fair path $\pi = s_0\, s_1\, s_2 \ldots$ in TS with $\pi \not\models \bigcirc a$. As π is fair and $\pi \not\models \bigcirc a$, it follows that $s_1 \models a_{fair}$ and $s_1 \not\models a$. So, $s_1 \not\models a_{fair} \rightarrow a$. A counterexample to $\bigcirc a$ with respect to $\models_{fair}$ thus results from a counterexample to the formula $\bigcirc(a_{fair} \rightarrow a)$ for CTL without fairness.

The Until Operator A witness of $a \,\mathsf{U}\, a'$ with respect to the fair semantics is a witness of $a \,\mathsf{U}\, (a' \wedge a_{fair})$ with respect to the standard CTL semantics. A counterexample for $a \,\mathsf{U}\, a'$ with respect to the fair semantics is either a witness of $(a \wedge a') \,\mathsf{U}\, (\neg a \wedge \neg a' \wedge a_{fair})$ under the common semantics $\models$ or a witness of $\Box(a \wedge \neg a')$ with respect to the fair semantics (explained below).

The Always Operator For a formula of the form $\Box a$, a counterexample with respect to the fair semantics is an initial path fragment $s_0\, s_1 \ldots s_n$ such that:

$$s_n \models \neg a \wedge a_{fair} \quad \text{and} \quad s_i \models a \quad \text{for } 0 \leqslant i < n.$$

Consider the strong fairness assumption of the form:

$$\mathit{sfair} \;=\; \bigwedge_{0<i\leqslant k} (\Box \Diamond a_i \rightarrow \Box \Diamond b_i).$$

A witness of $\Box a$ under *sfair* is an initial path fragment

$$\underbrace{s_0\, s_1 \ldots s_n\, s'_1\, s'_2 \ldots s'_r}_{\models a} \qquad \text{with } s_n = s'_r$$

such that for all $0 < i \leqslant k$ it holds that

$$\mathit{Sat}(a_i) \cap \{\, s'_1, \ldots, s'_r \,\} = \varnothing \text{ or } \mathit{Sat}(b_i) \cap \{\, s'_1, \ldots, s'_r \,\} \neq \varnothing.$$

A witness can be computed by means of an analysis of the SCCs of a digraph that originates from the state graph of TS after some slight modifications. The costs are (multiplicatively) linear in the number of fairness conditions and in the size of the state graph.

Theorem 6.47. Time Complexity of Fair Counterexample Generation

For transition system TS with N states and K transitions, CTL path formula φ and CTL fairness assumption fair with k conjuncts such that $TS \not\models_{fair} \forall\varphi$, a counterexample for φ in TS can be determined in time $\mathcal{O}((N+K)\cdot k)$. The same holds for a witness for φ if $TS \models \exists\varphi$.

Example 6.48.

Consider the transition system depicted in Figure 6.11 (page 350) and assume the formula of interest is:

$$\exists\Box\,(a \vee (b = c)) \quad \text{under fairness constraint} \quad \mathit{sfair} = \Box\Diamond\,(q \wedge r) \rightarrow \Box\Diamond\,\neg(q \vee r).$$

The strong CTL fairness constraint asserts that in case either state s_0 or s_2 is visited infinitely often, then also state s_3 or s_7 needs to be visited infinitely often. The path $s_1\, s_3\, s_0\, s_2\, s_0$ is a witness of $\exists\Box\,(a \vee (b = c))$ in the absence of any fairness constraint. It is, however, no witness under *sfair* as it visits s_0 (and s_2) infinitely often, but not s_3 or s_7. On the other hand, the path $s_1\, s_3\, s_0\, s_2\, s_1$ is a witness under *sfair*. ■

6.7 Symbolic CTL Model Checking

The CTL model-checking procedure described so far relies on the assumption that the transition system has an explicit representation by the predecessor and successor lists per state. Such an *enumerative* representation is not adequate for very large transition systems. To attack the state explosion problem, the CTL model-checking procedure can be reformulated in a *symbolic* way where sets of states and sets of transitions are represented rather than single states and transitions. This set-based approach is very natural for CTL, since its semantics and model-checking algorithm rely on satisfaction sets for subformulae. There are several possibilities to realize the CTL model checking algorithm in a purely set-based stetting. The most prominent ones rely on a binary encoding of the states, which permits identifying subsets of the state space and the transition relation with switching functions. To obtain compact representations of switching functions, special data structures have been developed, such as ordered *binary decision diagrams*. Other forms for the representation of switching functions, such as conjunctive normal forms, could be used as well. They are widely used in the context of so-called *SAT-based model checking* where the model-checking problem is reduced to the satisfiability problem for propositional formulae (SAT). The SAT-based techniques will not be described in this book. Instead we will explain the main ideas of the approach with binary decision diagrams.

We first explain the general ideas behind symbolic approaches which operate on *sets of states* rather than single states and rely on a representation of transition systems by switching functions. In the sequel, let $TS = (S, \rightarrow, I, AP, L)$ be a "large", but finite transition system. The set of actions is irrelevant here and has been skipped. That is, the transition relation $\rightarrow$ is a subset of $S \times S$. Let $n \geqslant \lceil \log |S| \rceil$. (Since we suppose a large transition system, we can safely assume that $|S| \geqslant 2$.) We choose an arbitrary (injective) encoding $enc : S \rightarrow \{0,1\}^n$ of the states by bit vectors of length n. Although enc might not be surjective, it is no restriction to suppose that $enc(S) = \{0,1\}^n$, since all elements $(b_1, \ldots, b_n) \in \{0,1\}^n \setminus enc(S)$ can be treated as the encoding of pseudo states that cannot be reached from any proper state $s \in S$. The transitions of these pseudo states are arbitrary. The idea is now to identify the states $s \in S = enc^{-1}(\{0,1\}^n)$ with their encoding $enc(s) \in \{0,1\}^n$ and to represent any subset T of S by its characteristic function $\chi_T : \{0,1\}^n \rightarrow \{0,1\}$, which evaluates to true exactly for the (encodings of the) states $s \in T$. Similarly, the transition relation $\rightarrow \subseteq S \times S$ can be represented by a Boolean function $\Delta : \{0,1\}^{2n} \rightarrow \{0,1\}$ that assigns 1 to exactly those pairs (s, s') of bit vectors of length n each where $s \rightarrow s'$.

On the basis of this encoding, the CTL model-checking procedure can be reformulated to an algorithm that operates on the representation of TS by binary decision diagrams for Δ and the characteristic functions $\chi_{Sat(a)}$ for the satisfaction sets of the atomic propositions $a \in AP$. The remainder of this section is concerned with explanations on this approach. Section 6.7.1 summarizes our notations for switching functions and operations on them. The encoding of transition systems by switching functions and a corresponding reformulation of the CTL model-checking algorithm will be presented in Section 6.7.2. The main concepts of (ordered) binary decision diagrams are summarized in Section 6.7.3.

6.7.1 Switching Functions

For technical reasons, it is more appropriate to consider switching functions as mappings from evaluations for certain Boolean variables to the values 0 or 1 rather than functions $\{0,1\}^n \rightarrow \{0,1\}$. This permits simpler definitions of composition operators as we just have to identify the common variables, rather than refer to common arguments via their positions in bit tuples. Furthermore, the reference of the arguments of a switching function by means of variable names is also central for binary decision diagrams.

Let $z_1 \ldots, z_m$ be Boolean variables and $Var = \{z_1, \ldots, z_m\}$. Let $Eval(z_1, \ldots, z_m)$ denote the set of evaluations for $z_1, \ldots, z_m$, i.e., functions $\eta : Var \rightarrow \{0,1\}$. Evaluations are written as $[z_1 = b_1, \ldots, z_m = b_m]$. We often use tuple-notations such as $\overline{z}$ for the variable tuple $(z_1, \ldots, z_m)$, $\overline{b}$ for a bit tuple $(b_1, \ldots, b_m) \in \{0,1\}^m$, and $[\overline{z} = \overline{b}]$ as a shorthand for the evaluation $[z_1 = b_1, \ldots, z_m = b_m]$.

Notation 6.49. Switching Function

A *switching function* for $Var = \{z_1, \ldots, z_m\}$ is a function $f : Eval(Var) \to \{0,1\}$. The special case $m = 0$ (i.e., $Var = \varnothing$) is allowed. The switching functions for the empty variable set are just constants 0 or 1. ■

To indicate the underlying set of variables of a switching function we often write $f(\overline{z})$ or $f(z_1, \ldots, z_m)$ rather than f. When an enumeration of the variables in Var is clear from the context, say $z_1, \ldots, z_m$, then we often simply write $f(b_1, \ldots, b_m)$ or $f(\overline{b})$ instead of $f([z_1 = b_1, \ldots, z_m = b_m])$ (or $f([\overline{z} = \overline{b}])$).

Disjunction, conjunction, negation and other Boolean connectives are defined for switching functions in the obvious way. For example, if f_1 is a switching function for $\{z_1, \ldots, z_n, \ldots z_m\}$ and f_2 a switching function for $\{z_n, \ldots z_m, \ldots, z_k\}$, where the z_i's are supposed to be pairwise distinct and $0 \leqslant n \leqslant m \leqslant k$, then $f_1 \vee f_2$ is a switching function for $\{z_1, \ldots, z_k\}$ and the values of $f_1 \vee f_2$ are given by

$$\begin{array}{l}(f_1 \vee f_2)([z_1 = b_1, \ldots, z_k = b_k]) \\ = \ \max\{\, f_1([z_1 = b_1, \ldots, z_m = b_m]),\ f_2([z_n = b_n, \ldots, z_k = b_k])\,\}.\end{array}$$

We often simply write z_i for the *projection function* $pr_{z_i} : Eval(\overline{z}) \to \{0,1\}$, $pr_{z_i}([\overline{z} = \overline{b}]) = b_i$ and 0 or 1 for the constant switching functions. With these notations, switching functions can be represented by Boolean connections of the variables z_i (viewed as projection functions) and constants. For instance, $z_1 \vee (z_2 \wedge \neg z_3)$ stands for a switching function.

Notation 6.50. Cofactor and Essential Variable

Let $f : Eval(z, y_1, \ldots, y_m) \to \{0,1\}$ be a switching function. The *positive cofactor* of f for variable z is the switching function $f|_{z=1} : Eval(z, y_1, \ldots, y_m) \to \{0,1\}$ given by

$$f|_{z=1}(\mathfrak{c}, b_1, \ldots, b_m) \ = \ f(1, b_1, \ldots, b_m)$$

where the bit tuple $(\mathfrak{c}, b_1, \ldots, b_m) \in \{0,1\}^{m+1}$ is short for the evaluation $[z = \mathfrak{c}, y_1 = b_1, \ldots, y_m = b_m]$. Similarly, the *negative cofactor* of f for variable z is the switching function $f|_{z=0} : Eval(z, y_1, \ldots, y_m) \to \{0,1\}$ given by $f|_{z=0}(\mathfrak{c}, b_1, \ldots, b_m) = f(0, b_1, \ldots, b_m)$. If f is a switching function for $\{z_1, \ldots, z_k,\ y_1, \ldots, y_m\}$, then we write $f|_{z_1=b_1,\ldots,z_k=b_k}$ for the *iterated cofactor*, also simply called *cofactor* of f, given by

$$f|_{z_1=b_1,\ldots,z_k=b_k} \ = \ (\ldots(f|_{z_1=b_1})|_{z_2=b_2}\ldots)|_{z_k=b_k}.$$

Variable z is called *essential* for f if $f|_{z=0} \neq f|_{z=1}$. ■

The values of $f|_{z_1=b_1,\ldots,z_k=b_k}$ for $f = f(z_1, \ldots, z_k, y_1, \ldots, y_m)$ are given by

$$f|_{z_1=b_1,\ldots,z_k=b_k}(\mathfrak{c}_1, \ldots, \mathfrak{c}_k, a_1, \ldots, a_m) \ = \ f(b_1, \ldots, b_k, a_1, \ldots, a_m)$$

where $(\mathfrak{c}_1, \ldots, \mathfrak{c}_k, a_1, \ldots, a_m)$ is identified with the evaluation $[z_1 = \mathfrak{c}_1, \ldots, z_k = \mathfrak{c}_k, y_1 = a_1, \ldots, y_m = a_m]$. As a consequence, the definition of (iterated) cofactors does not depend on the order in which the cofactors for single variables are considered, i.e.:

$$f|_{z_1=b_1,\ldots,z_k=b_k} \;=\; (\ldots(f|_{z_{i_1}=b_{i_1}})|_{z_{i_2}=b_{i_2}}\ldots)|_{z_{i_k}=b_{i_k}}$$

for each permutation $(i_1, \ldots, i_k)$ of $(1, \ldots, k)$.

Obviously, variable z is not essential for the cofactors $f|_{z=0}$ and $f|_{z=1}$. Thus, at most the variables in $Var \setminus \{z_1, \ldots, z_k\}$ are essential for $f|_{z_1=b_1,\ldots,z_k=b_k}$, provided that f is a switching function for Var.

Example 6.51. Cofactors and Essential Variables

Consider the switching function $f(z_1, z_2, z_3)$ given by $(z_1 \vee \neg z_2) \wedge z_3$. Then, $f|_{z_1=1} = z_3$ and $f|_{z_1=0} = \neg z_2 \wedge z_3$. In particular, variable z_1 is essential for f.

When we fix the variable set $\{z_1, z_2, z_3\}$, then variables z_2 and z_3 are not essential for the projection function $pr_{z_1} = z_1$. In fact, we have $z_1|_{z_2=0} = z_1|_{z_2=1} = z_1$. On the other hand, z_1 is essential for the projection function z_1 as we have $z_1|_{z_1=1} = 1 \neq 0 = z_1|_{z_1=0}$.

For another example, variables z_1 and z_2 are essential for $f(z_1, z_2, z_3) = z_1 \vee \neg z_2 \vee (z_1 \wedge z_2 \wedge \neg z_3)$, while variable z_3 is not, as $f|_{z_3=1} = z_1 \vee \neg z_2$ agrees with $f|_{z_3=0} = z_1 \vee \neg z_2 \vee (z_1 \wedge z_2)$. ■

The following lemma yields a decomposition of f into its cofactors. This simple observation relies on the fact that for any evaluation where z is assigned to 0 the value of $f(z, \overline{y})$ agrees with the value of $f|_{z=0}$ under the same evaluation for $\overline{y}$. And similarly, $f([z = 1, \overline{y} = \overline{b}]) = f|_{z=1}([\overline{y} = \overline{b}])$.

Lemma 6.52. Shannon Expansion

If f is a switching function for Var , then for each variable $z \in Var$:

$$f \;=\; (\neg z \wedge f|_{z=0}) \;\vee\; (z \wedge f|_{z=1}).$$

A simple consequence of the Shannon expansion is that z is not essential for f if and only if $f = f|_{z=0} = f|_{z=1}$.

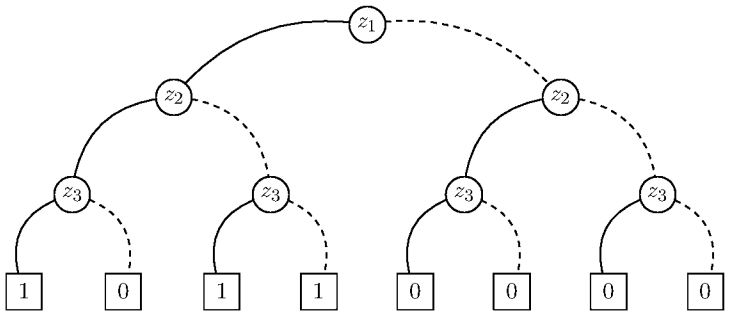


Figure 6.20: Binary decision tree for $z_1 \wedge (\neg z_2 \vee z_3)$.

Remark 6.53. Binary Decision Trees

The Shannon expansion is inherent in the representation of switching functions by *binary decision trees*. Given a switching function f for some variable set *Var*, one first fixes an arbitrary enumeration $z_1, \ldots, z_m$ for the variables in *Var* and then represents f by a binary tree of height m such that the two outgoing edges of the inner nodes at level i stand for the cases $z_i = 0$ (depicted by a dashed line) and $z_i = 1$ (depicted by a solid line). Thus, the paths from the root to a leaf in that tree represent the evaluations and the corresponding value. The leaves stand for the function values 0 or 1 of f. That is, given the evaluation $s = [z_1 = b_1, \ldots, z_m = b_m]$, then $f(s)$ is the value of the terminal node that is reached by traversing the tree from the root using the branch $z_i = b_i$ for the node at level i. The subtree of node v of the binary decision tree for f and the variable ordering $z_1, \ldots, z_m$ yields a representation of the iterated cofactor $f|_{z_1=b_1,\ldots,z_{i-1}=b_{i-1}}$ (viewed as a switching function for $\{z_i, \ldots, z_m\}$) where $z_1 = b_1$, ..., $z_{i-1} = b_{i-1}$ is the sequence of decisions made along the path from the root to node v.

An example of a binary decision tree for $f(z_1, z_2, z_3) = z_1 \wedge (\neg z_2 \vee z_3)$ is given in Figure 6.20. We use dashed lines for the edges from an inner node for variable z representing the case $z = 0$ and solid edges for the case $z = 1$. ■

Further operators on switching functions that will be needed later are existential quantification over variables and renaming of variables.

Notation 6.54. Existential and Universal Quantification

Let f be a switching function for *Var* and $z \in$ *Var*. Then, $\exists z.f$ is the switching function given by:

$$\exists z.f \;=\; f|_{z=0} \vee f|_{z=1}.$$

If $\overline{z} = (z_1, \ldots, z_k)$ and $z_i \in Var$ for $1 \leqslant i \leqslant k$, then $\exists \overline{z}.f$ is a short-form notation for $\exists z_1.\exists z_2. \ldots \exists z_k.f$. Similarly, universal quantification is defined by

$$\forall z.f \;=\; f|_{z=0} \wedge f|_{z=1}$$

and $\forall \overline{z}.f \;=\; \forall z_1.\forall z_2. \ldots \forall z_k.f$. ■

For example, if $f(z, y_1, y_2)$ is given by $(z \vee y_1) \wedge (\neg z \vee y_2)$, then

$$\exists z.f \;=\; f|_{z=0} \vee f|_{z=1} \;=\; y_1 \vee y_2$$

and $\forall z.f \;=\; f|_{z=0} \wedge f|_{z=1} \;=\; y_1 \wedge y_2$.

The rename operator simply replaces certain variables with other ones. E.g., renaming variable z into y in $f(z, x) = \neg z \vee x$ yields the switching function $\neg y \vee x$. Formally:

Notation 6.55. Rename Operator

Let $\overline{z} = (z_1, \ldots, z_m)$, $\overline{y} = (y_1, \ldots, y_m)$ be variable tuples of the same length and let $\overline{x} = (x_1, \ldots, x_k)$ be a variable tuple such that none of the variables y_i or z_i is contained in $\overline{x}$. For the evaluation $s = [\overline{y} = \overline{b}] \in Eval(\overline{y}, \overline{x})$, $s\{\overline{y} \leftarrow \overline{z}\}$ denotes the evaluation in $Eval(\overline{z}, \overline{x})$ that is obtained by composing the variable renaming function $y_i \mapsto z_i$, for $1 \leqslant i \leqslant m$ with the evaluation s. That is, $s\{\overline{y} \leftarrow \overline{z}\}$ agrees with s for the variables in $\overline{x}$ and $s\{\overline{y} \leftarrow \overline{z}\}$ assigns the same value $b \in \{0, 1\}$ to variable z_i as s to variable y_i. Given a switching function $f : Eval(\overline{z}, \overline{x}) \to \{0, 1\}$, then the switching function $f\{\overline{z} \leftarrow \overline{y}\} : Eval(\overline{y}, \overline{x}) \to \{0, 1\}$ is given by

$$f\{\overline{z} \leftarrow \overline{y}\}(s) \;=\; f(\, s\{\overline{y} \leftarrow \overline{z}\}\,),$$

i.e., $f\{\overline{z} \leftarrow \overline{y}\}([\overline{y} = \overline{b}, \overline{x} = \overline{c}]) \;=\; f([\overline{z} = \overline{b}, \overline{x} = \overline{c}])$. If it is clear from the context which variables have to be renamed, then we simply write $f(\overline{y}, \overline{x})$ rather than $f\{\overline{z} \leftarrow \overline{y}\}$. ■

6.7.2 Encoding Transition Systems by Switching Functions

After this excursus on switching functions, we return to the question of a symbolic representation of a transition system $TS = (S, \to, I, AP, L)$. As mentioned above (see page 382), the action set is irrelevant for our purposes and therefore omitted. For the encoding of the states $s \in S$ we use n Boolean variables $x_1, \ldots, x_n$ and identify any evaluation $[x_1 = b_1, \ldots, x_n = b_n] \in Eval(\overline{x})$ with the unique state $s \in S$ such that $enc(s) = (b_1, \ldots, b_n)$. In the sequel, we suppose $S = Eval(\overline{x})$. Given a subset B of S, then the *characteristic function* $\chi_B : S \to \{0, 1\}$ of B assigns 1 to all states $s \in B$

and 0 to all states $s \notin B$. As we assume $S = \mathit{Eval}(\overline{x})$, the characteristic function is the switching function given by

$$\chi_B : \mathit{Eval}(\overline{x}) \to \{0,1\}, \quad \chi_B(s) \;=\; \begin{cases} 1 & \text{if } s \in B \\ 0 & \text{otherwise.} \end{cases}$$

In particular, for any atomic proposition $a \in AP$, the satisfaction set $\mathit{Sat}(a) = \{s \in S \mid s \models a\}$ can be represented by the switching function $f_a = \chi_{\mathit{Sat}(a)}$ for $\overline{x}$. This yields a symbolic representation of the labeling function by means of a family $(f_a)_{a \in AP}$ of switching functions for $\overline{x}$.

The symbolic representation of the transition relation $\to \subseteq S \times S$ relies on the same idea: we identify $\to$ with its characteristic function $S \times S \to \{0,1\}$ where truth-value 1 is assigned to the state pair (s,t) if and only if $s \to t$. Formally, we deal with the variable tuple $\overline{x} = (x_1, \ldots, x_n)$ to encode the starting state s of a transition and a copy $\overline{x}' = (x_1', \ldots, x_n')$ of $\overline{x}$ to encode the target state. That is, for each of the variables x_i we introduce a new variable x_i'. The original (unprimed) variables x_i serve to encode the current state, while the primed variables x_i' are used to encode the next state. Then, we identify the transition relation $\to$ of TS with the switching function

$$\Delta : \mathit{Eval}(\overline{x}, \overline{x}') \to \{0,1\}, \quad \Delta(s, t\{\overline{x}' \leftarrow \overline{x}\}) \;=\; \begin{cases} 1 & \text{if } s \to t \\ 0 & \text{otherwise} \end{cases}$$

where s and t are elements of the state space $S = \mathit{Eval}(\overline{x})$ and the second argument $t\{\overline{x}' \leftarrow \overline{x}\}$ in $\Delta(s, t\{\overline{x}/\overline{x}'\})$ is the evaluation for $\overline{x}'$ that assigns the same value (1 or 0) to variable x_i' as t assigns to variable x_i (cf. Notation 6.55).

Example 6.56. Symbolic Representation of Transition Relation

Suppose that TS has two states s_0 and s_1 and the transitions $s_0 \to s_0$, $s_0 \to s_1$ and $s_1 \to s_0$, then we need a single Boolean variable $x_1 = x$ for the encoding. Say we identify state s_0 by 0 and state s_1 by 1. The transition relation $\to$ is represented by the switching function $\Delta : \mathit{Eval}(x, x') \to \{0,1\}$,

$$\Delta \;=\; \neg x \;\vee\; \neg x'.$$

Let us check why. The satisfying assignments for Δ are $[x = 0, x' = 0]$, $[x = 0, x' = 1]$ and $[x = 1, x' = 0]$. The first two evaluations (where $x = 0 = 0$) represent the two outgoing transitions from state $s_0 = 0$, while $[x = 1, x' = 0]$ stands for the transition $s_1 \to s_0$. ■

Example 6.57. Symbolic Representation of a Ring

Consider a transition system TS with states $\{s_0, \ldots, s_{k-1}\}$ where $k = 2^n$ that are organized in a ring, i.e., TS has the transitions

$$s_i \;\to\; s_{(i+1) \bmod k}$$

for $0 \leqslant i < k$. We use the encoding that identifies any state s_i with the binary encoding of its index i. E.g., if $k = 16$, then $n = 4$ and state s_1 is identified with 0001, state s_{10} with 1010, and state s_{15} with 1111. We use the Boolean variables $x_1, \ldots, x_n$ where x_n represents the most significant bit (i.e., the evaluation $[x_n = b_n, \ldots, x_1 = b_1]$ stands for state $\sum_{1 \leqslant i \leqslant n} b_i 2^{i-1}$). Then, Δ is a function with $2n$ variables, namely $x_1, x_2, \ldots, x_n$ and their copies $x'_1, x'_2, \ldots x'_n$. The values of the switching function $\Delta(\overline{x}, \overline{x}')$ are given by

$$\bigwedge_{1 \leqslant i < n} \Big(x_1 \wedge \ldots \wedge x_i \wedge \neg x_{i+1} \;\rightarrow\; x' \wedge \ldots \wedge x'_i \wedge x'_{i+1} \wedge \bigwedge_{j < i \leqslant n} (x_j \leftrightarrow x'_j) \Big)$$
$$\wedge\, (x_1 \wedge \ldots \wedge x_n \;\rightarrow\; \neg x'_1 \wedge \ldots \wedge \neg x'_n).$$

The set $B = \{\, s_{2i} \mid 0 \leqslant i < 2^{n-1} \,\}$ is given by the switching function $\chi_B(\overline{x}) = x_1$. ■

Given the switching function Δ and a state $s \in S = \mathit{Eval}(\overline{x})$, then the successor set $\mathit{Post}(s) = \{s' \in S \mid s \rightarrow s'\}$ arises from Δ by fixing evaluation s for $\overline{x}$. More precisely, if $s = [x_1 = b_1, \ldots, x_n = b_n]$, then a switching function $\chi_{\mathit{Post}(s)}$ for $\mathit{Post}(s)$ is obtained from Δ by building the cofactor for the variables $x_1, \ldots, x_n$ and the values $b_1, \ldots, b_n$:

$$\chi_{\mathit{Post}(s)} \;=\; \Delta|_s\{\overline{x} \leftarrow \overline{x}'\}$$

where $\Delta|_s$ stands for the iterated cofactor $\Delta|_{x_1=b_1,\ldots,x_n=b_n}$. As $\Delta|_s$ is a switching function for $\{x'_1, \ldots, x'_n\}$, the renaming operator $\{\overline{x} \leftarrow \overline{x}'\}$ yields a representation of $\mathit{Post}(s)$ by the variables $x_1, \ldots, x_n$.

Example 6.58.

The successor set $\mathit{Post}(s_0) = \{\, s_0, s_1 \,\}$ for the simple system in Example 6.56 is obtained symbolically by

$$\Delta|_{x=0}\{x \leftarrow x'\} \;=\; \underbrace{(\neg x \vee \neg x')|_{x=0}}_{=1}\{x \leftarrow x'\} \;=\; 1,$$

which reflects the fact that after identifying state s_0 with the evaluation $[x = 0]$ and state s_1 with $[x = 1]$ the successor set of s_0 is $\mathit{Eval}(x) = \{s_0, s_1\}$, and its characteristic function is the constant 1. For state $s_1 = [x = 1]$, a symbolic representation of the successor set $\mathit{Post}(s_1) = \{s_0\} = \{[x = 0]\}$ is obtained by

$$\Delta|_{x=1}\{x \leftarrow x'\} \;=\; \underbrace{(\neg x \vee \neg x')|_{x=1}}_{=\neg x'}\{x \leftarrow x'\} \;=\; \neg x$$

■

Remark 6.59. Symbolic Composition Operators

As we explained in Chapter 2, a crucial aspect for the feasibility of any algorithmic verification technique is the automatic construction of "large" transition systems to be analyzed by means of operators that compose several "small" transition systems $TS_1, \ldots, TS_m$ representing the processes to be executed in parallel. Let us suppose that we have appropriate representations for the switching functions $\Delta_1, \ldots, \Delta_m$ for transition systems $TS_1, \ldots, TS_m$ at our disposal. If TS arises by $TS_1, \ldots, TS_m$ through the synchronous product operator of TS, then the transition relation of TS is given by

$$\Delta(\overline{x}_1, \ldots, \overline{x}_m, \overline{x}'_1, \ldots, \overline{x}'_m) \;=\; \bigwedge_{1 \leqslant i \leqslant n} \Delta_i(\overline{x}_i, \overline{x}'_i)$$

where $\overline{x}_i$ denotes the variable tuple that is used to encode the states in TS_i. The justification for the above equation is that each transition $\langle s_1, \ldots, s_m \rangle \to \langle s'_1, \ldots, s'_m \rangle$ of $TS = TS_1 \otimes \ldots \otimes TS_m$ is composed of individual transitions $s_i \to s'_i$ in TS_i for each of the transition systems TS_i. For the other extreme, suppose that $TS = TS_1 \,|||\, \ldots \,|||\, TS_m$ arises by the interleaving operator of the TS_i's (without any synchronization or communication). Then,

$$\Delta(\overline{x}_1, \ldots, \overline{x}_m, \overline{x}'_1, \ldots, \overline{x}'_m) \;=\; \bigvee_{1 \leqslant i \leqslant n} \Big(\Delta_i(\overline{x}_i, \overline{x}'_i) \;\wedge \bigwedge_{\substack{1 \leqslant j \leqslant m \\ i \neq j}} \overline{x}_j = \overline{x}'_j \Big)$$

where for $\overline{x}_j = (x_{1,j}, \ldots, x_{n_j,j})$ and $\overline{x}'_j = (x'_{1,j}, \ldots, x'_{n_j,j})$ notation $\overline{x}_j = \overline{x}'_j$ abbreviates $\bigwedge_{1 \leqslant k \leqslant n_j} (x_{k,j} \leftrightarrow x'_{k,j})$. The justification for the above equation for Δ is that each transition in TS has the form

$$\langle s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_m \rangle \to \langle s_1, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, s_m \rangle$$

where exactly one transition system makes a move $s_i \to s'_i$, while the others do not change their local state. For the parallel operator $\|_H$ which relies on an interleaving semantics for actions outside H and synchronization over the actions in H, we need a combination of the above formulae for Δ. Here, in fact, we need a refined representation of the transition relation in TS_i with actions for the transitions by means of a switching function $\Delta_i(\overline{x}_i, \overline{z}, \overline{x}'_i)$ where the variable tuple $\overline{z}$ serves to encode the action names. In case $m = 2$, the switching function $\Delta(\overline{x}_1, \overline{x}_2, \overline{x}'_1, \overline{x}'_2)$ for the transition relation in $TS = TS_1 \|_H TS_2$ is given by

$$\begin{array}{lll}
\exists \overline{z}. \Big(& (\quad \chi_H(\overline{z}) \;\wedge\; \Delta_1(\overline{x}_1, \overline{z}, \overline{x}'_1) \;\wedge\; \Delta_2(\overline{x}_1, \overline{z}, \overline{x}'_2) \quad) & \vee \\
& (\neg \chi_H(\overline{z}) \;\wedge\; \Delta_1(\overline{x}_1, \overline{z}, \overline{x}'_1) \;\wedge\; \overline{x}_2 = \overline{x}'_2 \quad) & \vee \\
& (\neg \chi_H(\overline{z}) \;\wedge\; \Delta_2(\overline{x}_2, \overline{z}, \overline{x}'_2) \;\wedge\; \overline{x}_1 = \overline{x}'_1 \quad) & \Big).
\end{array}$$

In case TS is the transition system of a program graph or channel system then the Boolean variables x_i serve to encode the locations, the potential values of the variables, and the

channel contents. The effect of the actions has then to be rewritten in terms of these variables. ■

Given the switching function $\Delta(\overline{x},\overline{x}')$ and the characteristic function $\chi_B(\overline{x})$ for some set B of states, we can now describe the backward BFS-based reachability analysis to compute all states in $Pre^*(B) = \{s \in S \mid s \models \exists\Diamond B\}$ on the basis of switching functions. Initially, we start with the switching function $f_0 = \chi_B$ that characterizes the set $T_0 = B$. Then, we successively compute the characteristic functions $f_{j+1} = \chi_{T_{j+1}}$ of

$$T_{j+1} \;=\; T_j \cup \{s \in S \mid \exists s' \in S \text{ s.t. } s' \in \mathit{Post}(s) \wedge s' \in T_j \;\}$$

The set of states s where the condition $\exists s' \in S$ s.t. $s' \in \mathit{Post}(s)$ and $s' \in T_j$ holds is given by the switching function:

$$\exists\overline{x}'.\,(\;\underbrace{\Delta(\overline{x},\overline{x}')}_{s' \,\in\, \mathit{Post}(s)} \wedge \underbrace{f_j(\overline{x}')}_{s' \in T_j}\;).$$

Recall that $f_j(\overline{x}')$ is just a short notation for $f_j\{\overline{x}' \leftarrow \overline{x}\}$, i.e., arises from f_j by renaming the unprimed variables x_i into their primed copies x_i' for $1 \leqslant i \leqslant n$. This BFS-based technique can easily be adapted to treat constrained reachability properties $\exists(C \,\mathsf{U}\, B)$ for subsets B, C of S, as shown in Algorithm 20 on page 390.

Algorithm 20 Symbolic computation of $Sat(\exists(C \,\mathsf{U}\, B))$

$f_0(\overline{x}) := \chi_B(\overline{x})$;
$j := 0$;
repeat
  $f_{j+1}(\overline{x}) := f_{j+1}(\overline{x}) \vee (\,\chi_C(\overline{x}) \wedge \exists\overline{x}'.\,(\,\Delta(\overline{x},\overline{x}') \wedge f_j(\overline{x}')\,)\,)$;
  $j := j + 1$
until $f_j(\overline{x}) = f_{j-1}(\overline{x})$;
return $f_j(\overline{x})$.

If we are just interested in the one-step predecessors, i.e., properties of the form $\exists \bigcirc B$, then no iteration is needed and the characteristic function of the set of states $s \in S$ with $\mathit{Post}(s) \cap B \neq \varnothing$ is obtained by

$$\exists\overline{x}'.\,(\;\underbrace{\Delta(\overline{x},\overline{x}')}_{s' \,\in\, \mathit{Post}(s)} \wedge \underbrace{\chi_B(\overline{x}')}_{s' \in B}\;).$$

In a similar way, the set $Sat(\exists\Box B)$ of all states s that have an infinite path consisting of states in a given set B can be computed symbolically. Here, we mimic the computation of the largest set $T \subseteq B$ with $\mathit{Post}(t) \cap T \neq \varnothing$ for all $t \in T$ by the iteration $T_0 = B$ and

$$T_{j+1} \;=\; T_j \cap \{s \in S \mid \exists s' \in S \text{ s.t. } s' \in \mathit{Post}(s) \wedge s' \in T_j \;\}$$

Algorithm 21 Symbolic computation of $Sat(\exists\Box B)$

$f_0(\overline{x}) := \chi_B(\overline{x})$;
$j := 0$;
repeat
  $f_{j+1}(\overline{x}) := f_{j+1}(\overline{x}) \wedge \exists\overline{x}'.\,(\,\Delta(\overline{x},\overline{x}') \wedge f_j(\overline{x}')\,)$;
  $j := j + 1$
until $f_j(\overline{x}) = f_{j-1}(\overline{x})$;
return $f_j(\overline{x})$.

in a symbolic way, as shown in Algorithm 21 on page 391.

These considerations show that the CTL model checking problem "Does CTL formula Φ hold for TS?" can be solved symbolically by means of switching functions f_Ψ that represent the satisfaction sets $Sat(\Psi)$ of the subformulae Ψ of Φ. The satisfaction sets f_a for the atomic propositions $a \in AP$ are supposed to be given. Union, intersection, and complementation of sets of states correspond to disjunction, conjunction, and negation on the level of switching functions. E.g., the satisfaction set for $\Psi_1 \wedge \neg\Psi_2$, is obtained by $f_{\Psi_1} \wedge \neg f_{\Psi_2}$. The treatment of formulae $\exists(\Psi_1 \,\mathsf{U}\, \Psi_2)$ and $\exists\Box\Psi$ relies on the techniques sketched in Algorithms 20 and 21. To treat full CTL, we can either transform any CTL formula into an equivalent CTL formula in existential normal form or use analogous symbolic algorithms for universally quantified formulae such as $\forall(\Phi \,\mathsf{U}\, \Psi)$.

The major challenge is to find an appropriate data structure for the switching functions. Besides yielding compact representations of the satisfaction sets and the transition relation, this data structure has to support the Boolean connectives (disjunction, conjunction, complementation) and the comparison of two switching functions. The latter is needed in the termination criteria for the repeat loops in Algorithms 20 and 21.

Before presenting the definition of binary decision diagrams that have proven to be very efficient in many applications, let us first discuss other potential data structures. Truth tables can be ruled as they always have the size 2^n for switching functions with n variables. The same holds for binary decision trees since they always have $2^{n+1} - 1$ nodes. Conjunctive or disjunctive normal forms for the representation of switching functions by propositional logic formulae yield the problem that checking equivalence (i.e., equality for the represented switching functions) is expensive, namely coNP-complete. Furthermore, there are switching functions f_m with m essential variables where the length of any representation by a conjunctive normal form formula grows exponentially in m. The same holds for disjunctive normal forms. However, the latter is no proper argument against conjunctive or disjunctive normal forms, because there is *no* data structure that ensures representations of polynomial size for all switching functions. The reason for this is that the

number of switching functions for m variables $z_1, \ldots, z_m$ is double exponential in m. Note that $|Eval(z_1, \ldots, z_m)| = 2^m$, and hence, the number of functions $Eval(z_1, \ldots, z_m) \to \{0,1\}$ is 2^{2^m}. Suppose we are given a universal data structure for switching functions (i.e., a data structure that can represent any switching function) such that K_m is the number of switching functions for $z_1, \ldots, z_m$ that can be represented by at most 2^{m-1} bits. Then:

$$K_m \;\leqslant\; \sum_{i=0}^{2^{m-1}} 2^i \;=\; 2^{2^{m-1}+1} - 1 \;<\; 2^{2^{m-1}+1}.$$

But then there are at least

$$2^{2^m} \;-\; 2^{2^{m-1}+1} \;=\; 2^{2^{m-1}+1} \cdot \left(2^{2^m - 2^{m-1} - 1} \;-\; 1\right) \;=\; 2^{2^{m-1}+1} \cdot \left(2^{2^{m-1}-1} - 1\right)$$

switching functions for $z_1, \ldots, z_m$ where the representation requires more than 2^{m-1} bits. This calculation shows that we cannot expect a data structure which is efficient for all switching functions. Nevertheless there are data structures which yield compact representations for many switching functions that appear in practical applications. A data structure that has been proven to be very successful for model checking purposes, in particular in the area of hardware verification, is *ordered binary decision diagrams (OBDDs)*. Besides yielding compact representation for many "realistic" transition systems, they enjoy the property that the Boolean connectives can be realized in time linear in the size of the input OBDDs and that (with appropriate implementation techniques) equivalence checking can even be performed in constant time.

In the remainder of this section, we explain those aspects of ordered binary decision diagrams that are relevant to understanding the main concepts of symbolic model checking with these data structures. Further theoretical aspects on binary decision diagrams, their variants and applications, can be found, e.g., in the textbooks [134, 180, 292, 300, 418]. For details on OBDD-based symbolic model checking we refer to [74, 92, 288, 374].

6.7.3 Ordered Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs for short), originally proposed by Bryant [70], yield a data structure for switching functions that relies on a compactification of binary decision trees. The rough idea is to skip redundant fragments of a binary decision tree. This means collapsing constant subtrees (i.e., subtrees where all terminal nodes have the same value) into a single node and identifying nodes with isomorphic subtrees. In this way, we obtain a directed acyclic graph of outdegree 2 where – as in binary decision trees – the inner nodes are labeled by variables and their outgoing edges stand for the possible evaluations of the corresponding variable. The terminal nodes are labeled by the function value.

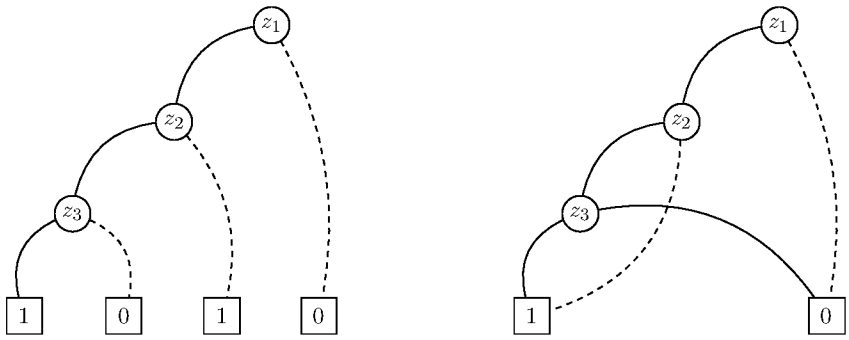


Figure 6.21: Binary decision diagram for $z_1 \wedge (\neg z_2 \vee z_3)$.

Example 6.60. From Binary Decision Tree to OBDD

Before presenting the formal definition of binary decision diagrams, we explain the main ideas by means of the function $f(z_1, z_2, z_3) = z_1 \wedge (\neg z_2 \vee z_3)$. A binary decision tree for f has been shown in Figure 6.20 on page 385. Since all terminal nodes in the right subtree of the root have value 0 (which reflects the fact that the cofactor $f|_{z_1=0}$ agrees with the constant function 0), the inner tests for variables z_2 and z_3 in that subtree are redundant and the whole subtree can be replaced by a terminal node with value 0. Similarly, the subtree of the z_3-node representing the cofactor $f|_{z_1=1,z_2=0} = 1$ can be replaced with a terminal node for the value 1. This leads to the graph shown on the left of Figure 6.21. Finally, we may identify all terminal nodes with the same value, which yields the graph shown on the right of Figure 6.21. ■

Example 6.61. From Binary Decision Tree to OBDD

As another example, consider the switching function $f = (z_1 \wedge z_3) \vee (z_2 \wedge z_3)$. The upper part of Figure 6.22 on page 394 shows a binary decision tree for f. The subtree of the z_3 node on the right is constant and can be replaced by a terminal node. The three subtrees of the z_3-nodes for the cofactors $f|_{z_1=0,z_2=1}$, $f|_{z_1=1,z_2=0}$, and $f|_{z_1=1,z_2=1}$ are isomorphic and can thus be collapsed. This yields the decision graph shown in the middle part of Figure 6.22. But now the z_2-node for the cofactor $f|_{z_1=1}$ becomes redundant, as regardless whether $z_2 = 0$ or $z_2 = 1$, the same node will be reached. This permits removing this z_2-node and redirecting its incoming edge. This yields the binary decision diagram depicted in the lower part of Figure 6.22. ■

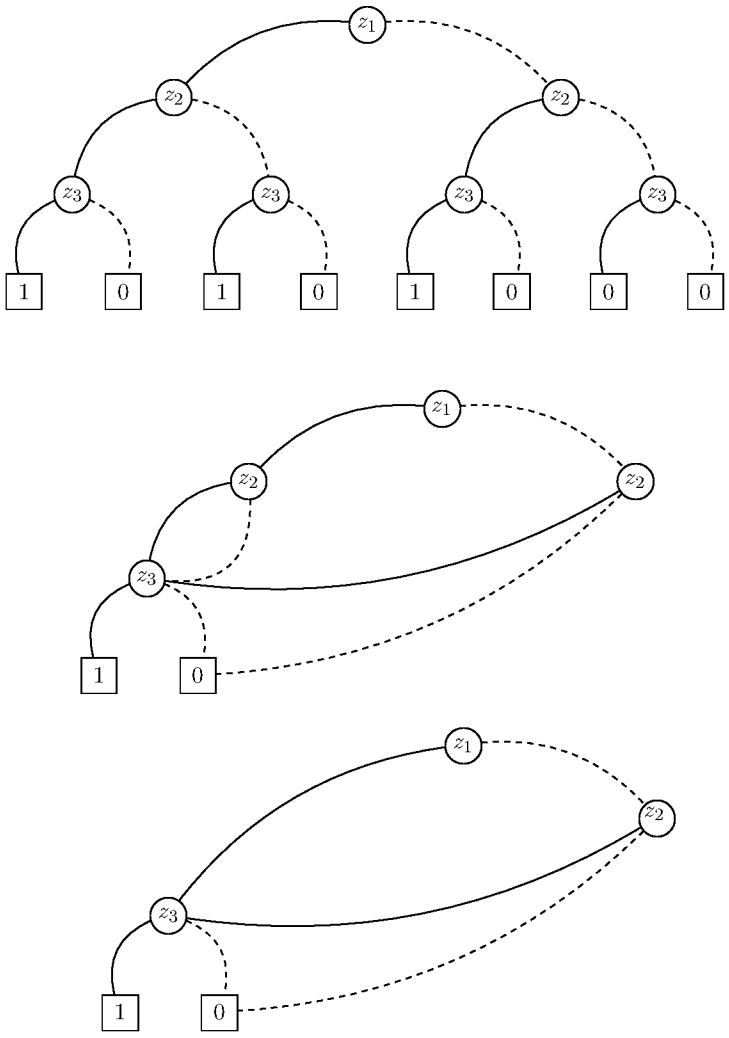


Figure 6.22: Binary decision diagrams for $f = (z_1 \wedge z_3) \vee (z_2 \wedge z_3)$.

Notation 6.62. Variable Ordering

Let *Var* be a finite set of variables. A *variable ordering* for *Var* denotes any tuple $\wp = (z_1, \ldots, z_m)$ such that $Var = \{z_1, \ldots, z_m\}$ and $z_i \neq z_j$ for $1 \leqslant i < j \leqslant m$. We write $<_\wp$ for the induced total order on *Var*. I.e., for $\wp = (z_1, \ldots, z_m)$ the binary relation $<_\wp$ on *Var* is given by $z_i <_\wp z_j$ if and only if $i < j$. We write $z_i \leqslant_\wp z_j$ iff either $z_i <_\wp z_j$ or $i = j$. ■

Definition 6.63. Ordered Binary Decision Diagram (OBDD)

Let $\wp$ be a variable ordering for *Var*. An *$\wp$-ordered binary decision diagram* ($\wp$-OBDD for short) is a tuple

$$\mathfrak{B} \;=\; (V, V_I, V_T, succ_0, succ_1, var, val, v_0)$$

consisting of

- a finite set V of nodes, disjointly partitioned into V_I and V_T where the nodes in V_I are called *inner nodes*, while the nodes in V_T are called *terminal nodes* or *drains*;
- successor functions $succ_0$, $succ_1 : V_I \to V$ that assign to each inner node v a 0-successor $succ_0(v) \in V$ and a 1-successor $succ_1(v) \in V$;
- a variable labeling function $var : V_I \to Var$ that assigns to each inner node v a variable $var(v) \in Var$;
- a value function $val : V_T \to \{0, 1\}$ that assigns to each drain a function value 0 or 1;
- a *root (node)* $v_0 \in V$.

Consistency of the variable labeling function with the variable ordering $\wp$ is required in the following sense. If $\wp = (z_1, \ldots, z_m)$, then for each inner node v: if $var(v) = z_i$ and $w \in \{succ_0(v), succ_1(v)\} \cap V_I$, then $var(w) = z_j$ for some $j > i$. Furthermore, it is required that each node $v \in V \setminus \{v_0\}$ has at least one predecessor, i.e., $v = succ_b(w)$ for some $w \in V$ and $b \in \{0, 1\}$. ■

The underlying digraph of a $\wp$-OBDD is obtained by using V as node set and establishing an edge from node v to w if and only if w is an successor of v, i.e., if $w \in \{succ_0(v), succ_1(v)\}$. The last condition in Definition 6.63, which states that each node of an OBDD, except for the root, has a predecessor, is equivalent to the condition that all nodes of an OBDD are reachable from the root. The *size* of an OBDD $\mathfrak{B}$, denoted $size(\mathfrak{B})$, is the number of nodes in $\mathfrak{B}$.

An equivalent formulation for the order consistency of $\mathfrak{B}$ (i.e., the condition stating that the variable of an inner node appears in $\wp$ before the variables of its successors, provided they are inner nodes) is the requirement that for each path $v_0\, v_1 \ldots v_n$ in (the underlying graph of) $\mathfrak{B}$ we have $v_i \in V_I$ for $1 \leqslant i < n$ and

$$\mathit{var}(v_0) <_\wp \mathit{var}(v_1) <_\wp \ldots <_\wp \mathit{var}(v_n),$$

where for the drains we put $\mathit{var}(v) = \bot$ (undefined) and extend $<_\wp$ by $z <_\wp \bot$ for all variables $z \in \mathit{Var}$. That is, we regard $<_\wp$ as a total order on $\mathit{Var} \cup \{\bot\}$ and consider the variable labeling function as a function $\mathit{var} : V \to \mathit{Var} \cup \{\bot\}$. This yields that the underlying graph of an OBDD is acyclic. In particular, $v \neq \mathit{succ}_b(v)$ for all inner nodes v and $b \in \{0,1\}$.

Examples of OBDDs are the binary decision trees and graphs shown in Figures 6.20, 6.21, and 6.22. All these OBDDs rely on the variable ordering $\wp = (z_1, z_2, z_3)$.

In the sequel, we often refer to an inner node v with $\mathit{var}(v) = z$ as a z-node. As the pictures for OBDDs suggest, we may think of the node set V of an OBDD to be partitioned into *levels*. Dealing with the ordering $\wp = (z_1, \ldots, z_m)$, then the z_i-nodes constitute the nodes at level i. The drains yields the bottom level, while the top level consists of the root node.

Definition 6.64. Semantics of OBDDs

Let $\mathfrak{B}$ be an $\wp$-OBDD as in Definition 6.63. The semantics $\mathfrak{B}$ is the switching function $f_\mathfrak{B}$ for *Var* where $f_\mathfrak{B}([z_1 = b_1, \ldots, z_m = b_m])$ is the value of the drain that will be reached when traversing the graph starting in the root v_0 and branching according to the evaluation $[z_1 = b_1, \ldots, z_m = b_m]$. That is, if the current node v is a z_i-node, then the traversal continues with the b_i-successor of v. ■

Definition 6.65. Sub-OBDD, Switching Function for the Nodes

Let $\mathfrak{B}$ be a $\wp$-OBDD. If v is a node in $\mathfrak{B}$, then the *sub-OBDD* induced by v, denoted $\mathfrak{B}_v$ arises from $\mathfrak{B}$ by declaring v as the root node and removing all nodes that are not reachable from v. The switching function for node, denoted f_v, is the switching function for *Var* that is given by the sub-OBDD $\mathfrak{B}_v$. ■

Clearly, at most the variables x with $\mathit{var}(v) \leqslant_\wp x$ can be essential for f_v, since none of the variables z with $z <_\wp \mathit{var}(v)$ appear in the sub-OBDD. Hence, f_v could also be viewed as a switching function for the whole variable set *Var* or as a switching function for the variables x with $\mathit{var}(v) \leqslant_\wp x$, but this is irrelevant here. In particular, if v is a z-node, then the order condition $z = \mathit{var}(v) <_\wp \mathit{var}(\mathit{succ}_b(v))$ yields that $f_{\mathit{succ}_b(v)}|_{z=\mathfrak{c}} = f_{\mathit{succ}_b(v)}$,

since variable z is not essential for $f_{succ_b(v)}$. But then:

$$f_v|_{z=0} = (\neg z \wedge f_{succ_0(v)})|_{z=0} \vee \underbrace{(z \wedge f_{succ_1(v)})|_{z=0}}_{=0}$$

$$= f_{succ_0(v)}|_{z=0} = f_{succ_0(v)}$$

and, similarly, $f_v|_{z=1} = f_{succ_1(v)}$. Thus, the *Shannon expansion* yields the following bottom-up characterization of the functions f_v:

Lemma 6.66. Bottom-up Characterization of the Functions f_v

Let $\mathfrak{B}$ be a $\wp$-OBDD. The switching functions f_v for the nodes $v \in V$ are given as follows:

- *If v is a drain, then f_v is the constant switching function with value $val(v)$.*
- *If v is a z-node, then $f_v = (\neg z \wedge f_{succ_0(v)}) \wedge (z \wedge f_{succ_1(v)})$.*

Furthermore, $f_{\mathfrak{B}} = f_{v_0}$ for the root v_0 of $\mathfrak{B}$.

This yields that $f_v = f_{\mathfrak{B}}|_{z_1=b_1,\ldots,z_i=b_i}$ where $[z_1 = b_1, \ldots, z_i = b_i]$ is an evaluation which leads from the root v_0 of $\mathfrak{B}$ to node v. In fact, all concepts of OBDD-based approaches rely on the decomposition of switching functions into their cofactors. However, only the cofactors are relevant that arise from f by assigning values to the first i variables of $\wp$ for some i.

Notation 6.67. $\wp$-Consistent Cofactor

Let f be a switching function for *Var* and $\wp = (z_1, \ldots, z_m)$ a variable ordering for *Var*. A switching function f' for *Var* is called a *$\wp$-consistent cofactor* of f if there exists some $i \in \{0, 1, \ldots, m\}$ such that $f' = f|_{z_1=b_1,\ldots,z_i=b_i}$. (Including the case $i = 0$ means that we regard f as a cofactor of itself.) ■

For instance, if $f = z_1 \wedge (z_2 \vee \neg z_3)$ and $\wp = (z_1, z_2, z_3)$, then the $\wp$-consistent cofactors of f are the switching functions f, $f|_{z_1=1} = z_2 \vee \neg z_3$, $f|_{z_1=1,z_2=0} = \neg z_3$ and the constants 0 and 1. The cofactors $f|_{z_3=0} = z_1$ and $f|_{z_2=0} = z_1 \wedge \neg z_3$ are not $\wp$-consistent. Since it is possible that some cofactors of f arise by several evaluations, it is possible that cofactors $f|_{z_{i_1}=b_1,\ldots,z_{i_k}=b_k}$ are $\wp$-consistent for the variable ordering $\wp = (z_1, \ldots, z_m)$, even if $(z_{i_1}, \ldots, z_{i_k})$ is *not* a permutation of $(z_1, \ldots, z_k)$. For example, for $f = z_1 \wedge (z_2 \vee \neg z_3)$ and $\wp = (z_1, z_2, z_3)$ the cofactor $f|_{z_2=0,z_3=1}$ is $\wp$-consistent as it agrees with the cofactors $f_{z_1=0}$ or $f|_{z_1=1,z_2=0,z_3=1}$. (They all agree with the constant function 0.)

The observation made above can now be rephrased as follows:

Lemma 6.68. Nodes in OBDDs and $\wp$-Consistent Cofactors

For each node v of an $\wp$-OBDD $\mathfrak{B}$, the switching function f_v is a $\wp$-consistent cofactor of $f_{\mathfrak{B}}$. Vice versa, for each $\wp$-consistent cofactor f' of $f_{\mathfrak{B}}$ there is at least one node v in $\mathfrak{B}$ such that $f_v = f'$.

However, given a $\wp$-OBDD $\mathfrak{B}$ and a $\wp$-consistent cofactor f' of $f_{\mathfrak{B}}$ there could be more than one node in $\mathfrak{B}$ representing f'. This, for instance, applies to the binary decision tree shown in Figure 6.20 on page 385, viewed as $\wp$-OBDD for $\wp = (z_1, z_2, z_3)$, where we have $f_{\mathfrak{B}} = z_1 \wedge (\neg z_2 \vee z_3)$ and the $\wp$-consistent cofactors represented by the nodes in the left subtree of the root agree as we have

$$f|_{z_1=0} \;=\; f|_{z_1=0,z_2=b} \;=\; f|_{z_1=0,z_2=b,z_3=\mathfrak{c}} \;=\; 0$$

for all b, $\mathfrak{c} \in \{0,1\}$. For the $\wp$-OBDD shown on the left of Figure 6.21 on page 393, all inner nodes represent different switching functions. However, the two drains with value 0 represent the same cofactors of $f_{\mathfrak{B}}$. The same holds for the two drains with value 1. However, in the $\wp$-OBDD shown on the right of Figure 6.21, every $\wp$-conistent cofactor is represented by a single node. In this sense, this $\wp$-OBDD is free of redundancies, and therefore called reduced:

Definition 6.69. Reduced OBDD

Let $\mathfrak{B}$ be a $\wp$-OBDD. $\mathfrak{B}$ is called *reduced* if for every pair (v, w) of nodes in $\mathfrak{B}$: $v \neq w$ implies $f_v \neq f_w$. Let $\wp$-ROBDD denote a reduced $\wp$-OBDD. ■

Thus, in reduced $\wp$-OBDDs any $\wp$-consistent cofactor is represented by *exactly one* node. This is the key property to prove that reduced OBDDs yield a universal and canonical data structure for switching functions. Universality means that any switching function can be represented by an OBDD. Canonicity means that any two $\wp$-OBDDs for the same function agree up to isomorphism, i.e., renaming of the nodes.

Theorem 6.70. Universality and Canonicity of ROBDDs

Let Var be a finite set of Boolean variables and $\wp$ a variable ordering for Var. Then:

(a) For each switching function f for Var there exists a $\wp$-ROBDD $\mathfrak{B}$ with $f_{\mathfrak{B}} = f$.

(b) Given two $\wp$-ROBDDs $\mathfrak{B}$ and $\mathfrak{C}$ with $f_{\mathfrak{B}} = f_{\mathfrak{C}}$, then $\mathfrak{B}$ and $\mathfrak{C}$ are isomorphic, i.e., agree up to renaming of the nodes.

Proof: ad (a). Clearly, the constant functions 0 and 1 can be represented by a $\wp$-ROBDD consisting of a single drain. Given a nonconstant switching function f for *Var* and a variable ordering $\wp$ for *Var*, we construct a reduced $\wp$-OBDD $\mathfrak{B}$ for f as follows. Let V be the set of $\wp$-consistent cofactors of f. The root of $\mathfrak{B}$ is f. The constant cofactors are the drains with the obvious values. For $f' \in V$, $f' \notin \{0,1\}$, let

$$\mathit{var}(f') \;=\; \min\{\, z \in \mathit{Var} \mid z \text{ is essential for } f' \,\}$$

be the first essential variable where the minimum is taken according to the total order $<_\wp$ induced by $\wp$. (We use here the trivial fact that any nonconstant switching function has at least one essential variable.) The successor functions are given by

$$\mathit{succ}_0(f') \;=\; f'|_{z=0}, \quad \mathit{succ}_1(f') = f'|_{z=1}$$

where $z = \mathit{var}(f')$. The definition of $\mathit{var}(\cdot)$ yields that $\mathfrak{B}$ is a $\wp$-OBDD. By the Shannon expansion we get that the semantics of $f' \in V$ (i.e., the switching function of f' as a node of $\mathfrak{B}$) is f'. In particular, this yields that $f_{\mathfrak{B}} = f$ (the function for the root f) and the reducedness of $\mathfrak{B}$ (as any two nodes represent different cofactors of f).

To prove the statement in (b), it suffices to show that any reduced $\wp$-OBDD $\mathfrak{C}$ with $f_{\mathfrak{C}} = f$ is isomorphic to the $\wp$-ROBDD $\mathfrak{B}$ constructed above. Let $V^{\mathfrak{C}}$ be the node set of $\mathfrak{C}$, $v_0^{\mathfrak{C}}$ the root of $\mathfrak{C}$, $\mathit{var}^{\mathfrak{C}}$ the variable labeling function, and $\mathit{succ}_0^{\mathfrak{C}}$, $\mathit{succ}_1^{\mathfrak{C}}$ the successor functions in $\mathfrak{C}$. Let function $\imath : V^{\mathfrak{C}} \to V$ be given by $\imath(v) = f_v$. (Recall that the functions f_v are $\wp$-consistent cofactors of $f_{\mathfrak{C}} = f$. This ensures that $f_v \in V$.) Since $\mathfrak{C}$ is reduced, $\imath$ is a bijection. It remains to show that $\imath$ preserves the variable labeling of inner nodes and maps the successors of an inner node v of $\mathfrak{C}$ to the successors of $f_v = \imath(v)$ in $\mathfrak{B}$.

Let v be an inner node of $\mathfrak{C}$, say a z-node, and let w_0 and w_1 be the 0- and 1-successors of v in $\mathfrak{C}$. Then, the cofactor $f_v|_{z=0}$ agrees with f_{w_0}, and similarly, $f_{w_1} = f_v|_{z=1}$. (This holds in any OBDD.) As $\mathfrak{C}$ is reduced, f_v is nonconstant (since otherwise $f_v = f_{w_0} = f_{w_1}$). Variable z must be the first essential variable of f_v according to $<_\wp$, i.e., $z = \mathit{var}(f_v)$. Let us see why. Let $y = \mathit{var}(f_v)$. The assumption $z <_\wp y$ yields that z is not essential for f_v, and therefore $f_{w_0} = f_v|_{z=0} = f = f_v|_{z=1} = f_{w_1}$. But then w_0, w_1 and v represent the same function. Since $w_0 \neq v$ and $w_1 \neq v$, this contradicts the assumption that $\mathfrak{C}$ is reduced. The assumption $y <_\wp z$ is also impossible since then no y-node would appear in the sub-OBDD $\mathfrak{C}_v$, which is impossible as $y = \mathit{var}(f_v)$ is essential for f_v by definition.

But then $\mathit{var}(\imath(v)) = z = \mathit{var}^{\mathfrak{C}}(v)$ and, for $b \in \{0,1\}$:

$$\mathit{succ}_b(\imath(v)) \;=\; f_v|_{z=b} \;=\; f_{\mathit{succ}_b^{\mathfrak{C}}(v)} \;=\; \imath(\mathit{succ}_b^{\mathfrak{C}}(v))$$

Hence, $\imath$ is an isomorphism. ∎

Theorem 6.70 permits speaking about "the $\wp$-ROBDD" for a given switching function f for *Var*. The $\wp$-ROBDD-size of f denotes the size (i.e., number of nodes) in the $\wp$-ROBDD for f.

Corollary 6.71. Minimality of Reduced OBDDs

Let $\mathfrak{B}$ be a $\wp$-OBDD for f. Then, $\mathfrak{B}$ is reduced if and only if size($\mathfrak{B}$) $\leqslant$ size($\mathfrak{C}$) for each $\wp$-OBDD $\mathfrak{C}$ for f.

Proof: This follows from the facts that (i) each $\wp$-consistent confactor of f is represented in any $\wp$-OBDD $\mathfrak{C}$ for f by at least one node, and (ii) a $\wp$-OBDD $\mathfrak{B}$ for f is reduced if and only if the nodes of $\mathfrak{B}$ stand in one-to-one-correspondence to the $\wp$-consistent cofactors of f. ■

Reduction rules. When the variable ordering $\wp$ for *Var* is fixed, then reduced $\wp$-OBDDs provide unique representations of switching functions for *Var*. (Of course, uniqueness is up to isomorphism.) Although reducedness is a global condition for an OBDD, there are two simple local *reduction rules* (see Figure 6.23), which can successively be applied to transform a given nonreduced $\wp$-OBDD into an equivalent $\wp$-ROBDD.

Elimination rule: If v is an inner node of $\mathfrak{B}$ with $succ_0(v) = succ_1(v) = w$, then remove v and redirect all incoming edges $u \to v$ to w.

Isomorphism rule: If v, w are nodes in $\mathfrak{B}$ with $v \neq w$ and either v, w are drains with $val(v) = val(w)$ or v, w are inner nodes with

$$\langle var(v), succ_1(v), succ_0(v)\rangle \;=\; \langle var(w), succ_1(w), succ_0(w)\rangle,$$

then remove node v and redirect all incoming edges $u \to v$ to node w.

Both rules delete node v. The redirection of the incoming edges $u \to v$ to node w means the replacement of the edges $u \to v$ which $u \to w$. Formally, this means that we deal with the modified successor functions given by

$$succ'_b(u) \;=\; \begin{cases} succ_b(u) & \text{if } succ_b(u) \neq v \\ w & \text{if } succ_b(u) = v \end{cases}$$

for $b = 0, 1$. The transformations described in Examples 6.60 and 6.61 rely on the application of the isomorphism and elimination rule.

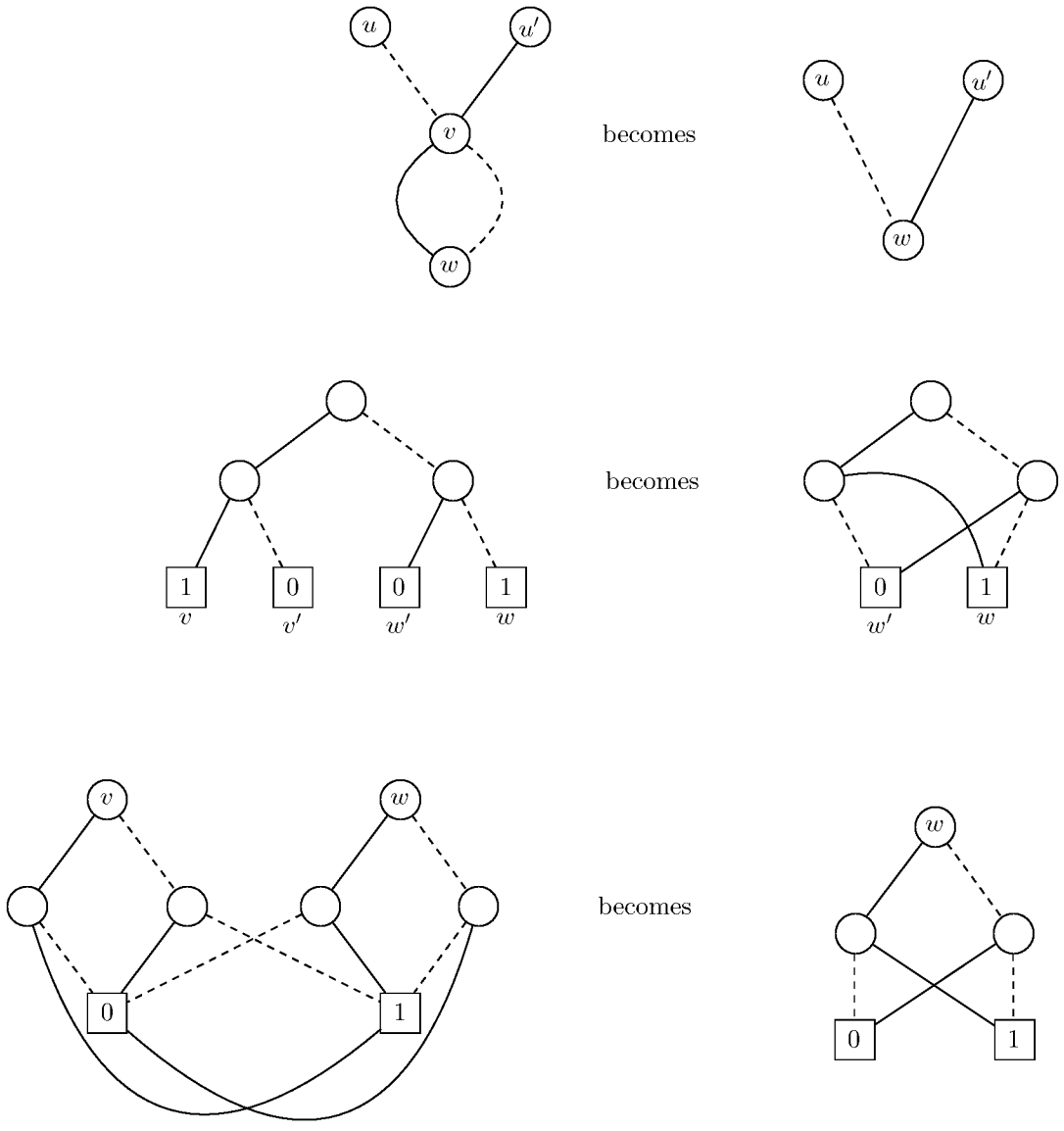


Figure 6.23: Reduction rules for OBDDs.

Both reduction rules are sound in the sense that they do not affect the semantics, i.e., if $\mathfrak{C}$ arises from a $\wp$-OBDD $\mathfrak{B}$ by applying the elimination or isomorphism rule, then $\mathfrak{C}$ is again a $\wp$-OBDD and $f_{\mathfrak{B}} = f_{\mathfrak{C}}$. This is due to the fact that both rules simply collapse two nodes v and w with $f_v = f_w$. For the elimination rule applied to a z-node v with $w = succ_0(v) = succ_1(v)$, we have

$$f_v \;=\; (\neg z \wedge f_{succ_0(v)}) \wedge (z \wedge f_{succ_1(v)}) \;=\; (\neg z \wedge f_w) \wedge (z \wedge f_w) \;=\; f_w.$$

Similarly, if the isomorphism rule is applied to z-nodes v and w then

$$f_v \;=\; (\neg z \wedge f_{succ_0(v)}) \wedge (z \wedge f_{succ_1(v)}) \;=\; (\neg z \wedge f_{succ_0(w)}) \wedge (z \wedge f_{succ_1(w)}) \;=\; f_w.$$

Since the application of the reduction rules decreases the number of nodes, the process to generate an equivalent $\wp$-OBDD by applying the reduction rules as long as possible always terminates. In fact, the resulting OBDD is reduced:

Theorem 6.72. Completeness of Reduction Rules

$\wp$-OBDD $\mathfrak{B}$ is reduced if and only if no reduction rule is applicable to $\mathfrak{B}$.

Proof: $\Rightarrow$: The applicability of a reduction rule implies the existence of at least two nodes representing the same switching function. Thus, if $\mathfrak{B}$ is reduced, then no reduction rule is applicable.

$\Leftarrow$: We prove the other direction by induction on the number of variables. More precisely, suppose that we are given a $\wp$-OBDD $\mathfrak{B}$ for the variable ordering $\wp = (z_1, \ldots, z_m)$ such that neither the elimination nor the isomorphism rule is applicable and show by induction on i that

$$f_v \neq f_w \text{ for all nodes } v,\, w \in V_i \text{ where } v \neq w.$$

Here, V_i denotes the set of all nodes $v \in V$ on level i or lower. Formally, V_i is the set of all nodes v in $\mathfrak{B}$ such that $z_i \leqslant_\wp var(v)$. Recall that $var(v) = \perp$ (undefined) for each drain v and that $z <_\wp \perp$ for all variables z.

We start the induction with the bottom level $i = m + 1$. The statement $f_v \neq f_w$ for all drains v, w where $v \neq w$ is trivial since the nonapplicability of the isomorphism rule yields that there is at most one drain with value 0 and at most one drain with value 1. In the induction step $i + 1 \Longrightarrow i$ $(m \geqslant i \geqslant 0)$ we suppose that $f_v \neq f_w$ for all nodes $v, w \in V_{i+1}$ with $v \neq w$ (induction hypothesis). Suppose there are two nodes v, $w \in V_i$ with $v \neq w$ and $f_v = f_w$. At least one of the nodes v or w must be on level i. Say $v \in V_i \setminus V_{i+1}$. Then, $var(v) = z_i$.

Let us first suppose that $w \in V_{i+1}$. Then, either w is a drain or a z_j-node for some $j > i$. In either case, variable z_i is not essential for $f_v = f_w$. As v is a z_i-node this yields that f_v agrees with the switching functions f_{v_0} and f_{v_1} of its successors $v_0 = succ_0(v)$ and $v_1 = succ_1(v)$. But then $v_0, v_1 \in V_{i+1}$ and $f_{v_0} = f_{v_1}$. The induction hypothesis yields that $v_0 = v_1$. But then the elimination rule would be applicable. Contradiction.

Suppose now that w is a z_i-node too. Let $v_0 = succ_0(v)$, $v_1 = succ_1(v)$ and $w_0 = succ_0(w)$, $w_1 = succ_1(w)$. The assumption $f_v = f_w$ yields that

$$f_{v_0} \;=\; f_v|_{z_i=0} \;=\; f_w|_{z_i=0} \;=\; f_{w_0}$$

and $f_{v_1} = f_{w_1}$. As $v_0, v_1, w_0, w_1 \in V_{i+1}$ the induction hypothesis yields that $v_0 = w_0$ and $v_1 = w_1$. But then the ismorphism rule is applicable. Contradiction. ■

Theorem 6.70 suggests a *reduction algorithm* which takes as input a nonreduced $\wp$-OBDD $\mathfrak{B}$ and constructs an equivalent $\wp$-OBDD by applying the reduction rules as long as possible. According to the inductive proof of the completeness of the reduction rules in Theorem 6.72, this technique is complete if the candidate nodes for the reduction rules are considered in a bottom-up manner. That is, initially we identify all drains with the same value. Then, for the levels $m, m-1, \ldots, 1$ (in this order) we apply the elimination and isomorphism rule. At level i, we first remove all nodes with identical successors (elimination rule) and then check the pairs of z_i-nodes where the isomorphism rule is applicable. To support the isomorphism rule a bucket technique can be used that groups together (1) all z_i-nodes with the same 0-successor and (2) splits all buckets consisting of z_i-nodes with the same 0-successor into buckets consisting of z_i-nodes with exactly the same successors. Then, application of the isomorphism rule simply means that the nodes in the buckets resulting from step (2) have to be collapsed into a single node. The time complexity of this algorithm is in $\mathcal{O}(size(\mathfrak{B}))$. In particular, given two $\wp$-OBDDs $\mathfrak{B}$ and $\mathfrak{C}$, the equivalence problem "Does $f_{\mathfrak{B}} = f_{\mathfrak{C}}$ hold?" can be solved by applying the reduction algorithm to both and then checking isomorphism for the reduced $\wp$-ROBDDs (see Exercise 6.12 on page 436). We will see later that with tricky implementation techniques, which integrate the steps of the reduction algorithm in the synthesis algorithms for ROBDDs and thus ensure that at any time the decision graph is reduced, the equivalence problem for ROBDDs can even be solved in constant time.

The Variable Ordering Problem The result stating the canonicity of reduced OBDDs crucially depends on the fact that the variable ordering $\wp$ is assumed to be fixed. Varying the variable ordering can lead to totally different ROBDDs, possibly ranging from ROBDDs of linear size to ROBDDs of exponential size. The results established before yield that the size (i.e., number of nodes) in the $\wp$-ROBDD for a switching function f agrees

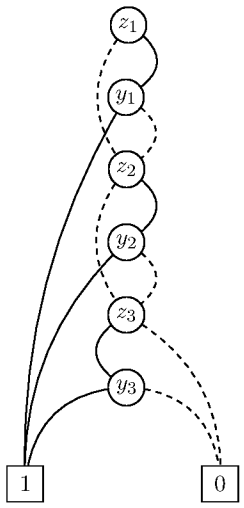


Figure 6.24: ROBDD for the function $f_3 = (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee (z_3 \wedge y_3)$ for the variable ordering $\wp = (z_1, y_1, z_2, y_2, z_3, y_3)$.

with the number of $\wp$-consistent cofactors of f. Thus, reasoning about the memory requirements of ROBDD-based approaches relies on counting the number of order-consistent cofactors.

Example 6.73. A Function with Small and Exponential-Size ROBDDs

To illustrate how the size of ROBDDs can be determined by analyzing the cofactors we consider a simple switching function which has both ROBDDs of linear size and ROBDDs with exponentially many nodes. Let $m \geqslant 1$ and

$$f_m = (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee \ldots \vee (z_m \wedge y_m).$$

For the variable ordering $\wp = (z_m, y_m, z_{m-1}, y_{m-1}, \ldots, z_1, y_1)$, the $\wp$-ROBDD for f_m has $2m + 2$ nodes, while $\Omega(2^m)$ nodes are required for the ordering $\wp' = (z_1, z_2, \ldots, z_m, y_1, \ldots, y_m)$. Figures 6.25 and 6.24 show the ROBDDs for the linear-size and exponential-size variable orderings for m=3. We first consider the ordering $\wp$ which groups the variables z_i and y_i that appear in the same clause. In fact, the variable ordering $\wp$ is optimal for f_m as the $\wp$-ROBDD for f_m contains one node for each variable. (This is the best we can hope to have as all $2n$ variables are essential for f_m and must appear in any ROBDD for

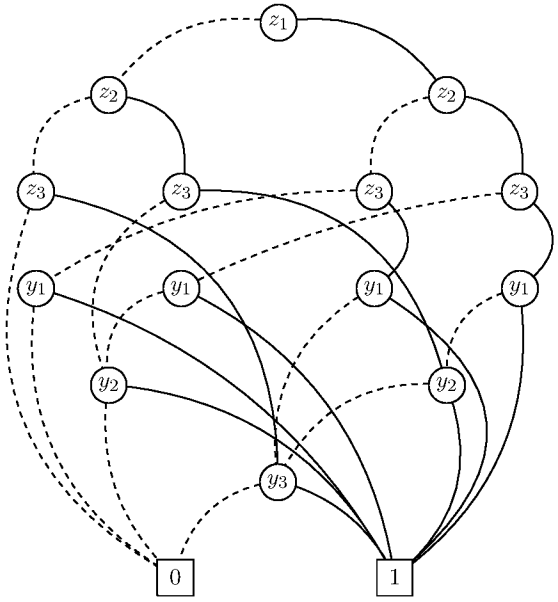


Figure 6.25: ROBDD for the function $f_3 = (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee (z_3 \wedge y_3)$ for the variable ordering $\wp = (z_1, z_2, z_3, y_1, y_2, y_3)$.

f_m.) Note that for $1 \leqslant i \leqslant m$:

$$f_m|_{z_m=a_m, z_m=b_m, \ldots, z_i=a_i, z_i=b_i} = \begin{cases} 1 & \text{if } a_j = b_j = 1 \\ & \text{for some } j \in \{i, \ldots, m\} \\ f_{i-1} & \text{otherwise} \end{cases}$$

$$f_m|_{z_m=a_m, z_m=b_m, \ldots, z_{i+1}=a_{i+1}, z_{i+1}=b_{i+1}, z_i=a_i} \in \{1,\ y_i \vee f_{i-1}\}$$

where $f_0 = 0$. Hence, the $\wp$-ROBDD for f_m has exactly one z_i-node representing the function f_i, exactly one y_i-node for the function $y_i \vee f_{i-1}$ (for $1 \leqslant i \leqslant m$), and two drains.

To see why the variable ordering $\wp'$ leads to a $\wp'$-ROBDD of exponential size, we consider the $\wp'$-consistent cofactors

$$f_{\overline{b}} = f_m|_{z_1=b_1, \ldots, z_m=b_m} = \bigvee_{i \in I_{\overline{b}}} y_i$$

where $\overline{b} = (b_1, \ldots, b_n)$ and $I_{\overline{b}} = \{i \in \{1, \ldots, m\} \mid b_i = 1$. The set of essential variables of $f_{\overline{b}}$ is $\{y_i \mid i \in I_{\overline{b}}\}$. As f $\overline{b}, \overline{c} \in \{0,1\}^m$, $\overline{b} \neq \overline{c}$ the index sets $I_{\overline{b}}$ and $I_{\overline{c}}$ are different, the $f_{\overline{b}}$ and $f_{\overline{c}}$ have different essential variables. Therefore, $f_{\overline{b}} \neq f_{\overline{c}}$ if $\overline{b} \neq \overline{c}$. But then the number of $\wp'$-consistent cofactors is at least 2^m. This yields that the $\wp'$-ROBDD for f_m has at most 2^m nodes. ■

Since for many switching functions the ROBDD sizes for different variable ordering can vary enormously, the efficiency of BDD-based computations crucially relies on the use of techniques that improve a given variable ordering. Although the problem to find an optimal variable ordering is known to be computationally hard (already the problem to decide whether a given variable ordering is optimal is NP-hard [56, 386]), there are several efficient heuristics for improving the current ordering. Most promiment is the so-called *sifting algorithm* [358] which relies on a local search for the best position for each variable, when the ordering of the other variables is supposed to be fixed. Explanations on such variable reordering algorithms and further discussions on the variable ordering problem are beyond the scope of this monograph. We refer the interested reader to the textbooks [292, 418] and conclude the comments on the influence of the variable ordering by the remark that there are also types of switching functions with ROBDDs of polynomial size under all variable orderings and types of switching functions where any variable ordering leads to a ROBDD of exponential size. An example of the latter is the middle bit of the *multiplication function* [71]. Examples of switching functions where each variable ordering leads to a ROBDD of at most quadratic size are *symmetric functions*. These are switching functions where the function values just depend on the number of variables that are assigned to 1. Stated differently, $f \in Eval(z_1, \ldots, z_m)$ is symmetric if and only if

$$f([z_1 = b_1, \ldots, z_m = b_m]) \;=\; f([z_1 = b_{i_1}, \ldots, z_m = b_{i_m}])$$

for each permutation $(i_1, \ldots, i_m)$ of $(1, \ldots, m)$. Examples of symmetric functions for $Var = \{z_1, \ldots, z_m\}$ are $z_1 \vee z_2 \vee \ldots \vee z_m$, $z_1 \wedge z_2 \wedge \ldots \wedge z_m$, the parity function $z_1 \oplus z_2 \oplus \ldots \oplus z_m$ (which returns 1 iff the number of variables that are assigned to 1 is odd), and the majority function (which returns 1 iff the number of variables that are assigned to 1 is greater than the number of variables that are assigned to 0). The ROBDDs for symmetric functions have the same topological structure for all variable orderings. This follows from the fact that the $\wp$-ROBDD for a symmetric function can be transformed into the $\wp'$-ROBDD by just modifying the variable labeling function.

Lemma 6.74. ROBDD-Size for Symmetric Functions

If f is a symmetric function with m essential variables, then for each variable ordering $\wp$ the $\wp$-ROBDD has size $\mathcal{O}(m^2)$.

Proof: Given a symmetric function f for m variables and a variable ordering $\wp$, say $\wp = (z_1, \ldots, z_m)$, then the $\wp$-consistent cofactors $f|_{z_1=b_1,\ldots,z_i=b_i}$ and $f|_{z_1=\mathfrak{c}_1,\ldots,z_i=\mathfrak{c}_i}$ agree for all bit tuples $(b_1, \ldots, b_i)$ and $(\mathfrak{c}_1, \ldots, \mathfrak{c}_i)$ that contain the same number of 1's. Thus, there are at most $i+1$ different $\wp$-consistent cofactors of f that arise by assigning values to the first i variables. Hence, the total number of $\wp$-consistent cofactors is bounded above by $\sum_{i=0}^{m}(i+1) = \mathcal{O}(m^2)$. ■

ROBDDs vs. CNF/DNF Both the parity and the majority function provide examples for switching functions with small ROBDD representations, while any representation of them by conjunctive or disjunctive normal forms (CNF, DNF) requires formulae of exponential length. Vice versa, there are also switching functions with short conjunctive or disjunctive normal forms, while the ROBDD under any variable ordering has exponential length (see, e.g., [418]). In fact, ROBDDs yield a totally different picture for complexity theoretic considerations than CNF or DNF. For example, given a CNF representation for a switching function f, the task to generate a CNF for $\neg f$ is expensive, as f might be expressible by a CNF of polynomial length, while any CNF for $\neg f$ has at least exponentially many clauses. For ROBDDs, however, negation is trivial as we may simply swap the values of the drains. In particular, for any variable ordering $\wp$, the $\wp$-ROBDDs for f and $\neg f$ have the same size. For another example, regard the satisfiability problem, which is known to be NP-complete for CNF, but again trivial for ROBDDs, since $f \neq 0$ if and only if the $\wp$-ROBDD for f does not contain a 0-drain. Similarly, the question whether two CNFs are equivalent is computationally hard (coNP-complete), but can be solved for $\wp$-ROBDDs $\mathfrak{B}$ and $\mathfrak{C}$ by checking isomorphism. The latter can be performed by a simultaneous traversal of the $\wp$-OBDDs in time linear in the sizes of $\mathfrak{B}$ and $\mathfrak{C}$. See Exercise 6.12, page 436. Note that these results do not contradict the complexity theoretic lower bounds, since "linear time" for a ROBDD-based algorithm means linear in the size of the input ROBDDs, which could be exponentially larger than equivalent input formulae (e.g., CNF).

6.7.4 Implementation of ROBDD-Based Algorithms

The efficiency of ROBDD-based algorithms to manipulate switching functions crucially relies on appropriate implementation techniques. In fact, with tricky implementation techniques, equivalence checking for ROBDDs can even be realized in *constant time*. In the sequel, we will explain the main ideas of such techniques, which provide the basis for most BDD packages and serve as a platform for an efficient realization of synthesis algorithms on ROBDDs. The purpose of *synthesis algorithms* is to construct a $\wp$-ROBDD for a function $f_1 \; op \; f_2$ (where op is a Boolean connective such as disjunction, conjunction, implication, etc.), when $\wp$-ROBDDs for f_1 and f_2 are given. Recall that the symbolic realization of the CTL model-checking procedure relies on such synthesis operations.

The idea, originally proposed in [301], is to use a single reduced decision graph with one global variable ordering $\wp$ to represent several switching functions, rather than using separate $\wp$-ROBDDs for each of the switching functions. All computations on these decision graphs are interleaved with the reduction rules to guarantee redundance-freedom at any time. Thus, the comparison of two represented functions simply requires checking equality of the nodes for them, rather than analyzing their sub-OBDDs.

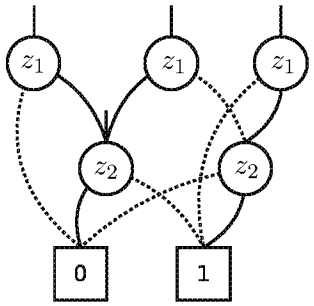


Figure 6.26: Example of a shared OBDD.

We start with the formal definition of a shared $\wp$-OBDD which is the same as a $\wp$-ROBDD, the only difference being that we can have more than more root node.

Definition 6.75. Shared OBDD

Let *Var* be a finite set of Boolean variables and $\wp$ a variable ordering for *Var*. A shared $\wp$-OBDD ($\wp$-SOBDD for short) is a tuple $\overline{\mathfrak{B}} = (V, V_I, V_T, succ_0, succ_1, var, val, \overline{v}_0)$ where V, V_I, V_T, $succ_0$, $succ_1$, *var*, and *val* are as in $\wp$-OBDDs (see Definition 6.63 on page 395). The last component is a tuple $\overline{v}_0 = (v_0^1, \ldots, v_0^k)$ of roots. The requirements are as in $\wp$-ROBDDs, i.e., for all nodes v, $w \in V$, (1) $var(v) <_\wp var(succ_b(v))$ if $v \in V_I$ and $b \in \{0,1\}$ and (2) $v \neq w$ implies $f_v \neq f_w$, where the switching function f_v for the nodes $v \in V$ is defined as for OBDDs. ■

Figure 6.26 shows a shared OBDD with four root nodes that represent the functions $z_1 \wedge \neg z_2$, $\neg z_2$, $z_1 \oplus z_2$ and $\neg z_1 \vee z_2$.

If v is a node in a $\wp$-SOBDD $\overline{\mathfrak{B}}$, then the sub-OBDD $\overline{\mathfrak{B}}_v$ is the $\wp$-ROBDD that results from $\overline{\mathfrak{B}}$ by removing all nodes that are not reachable from v and declaring v to be the root node. In fact, $\overline{\mathfrak{B}}_v$ is the $\wp$-ROBDD for f_v, and thus, the size N_v of $\overline{\mathfrak{B}}_v$ is with the $\wp$-ROBDD size of f_v. Thus, an alternative view of an SOBDD is the combination of several ROBDDs for the same variable ordering $\wp$ by sharing nodes for common $\wp$-consistent cofactors. In particular, an SOBDD has exactly two drains for the constant functions 0 and 1 (where we ignore the pathological case of an SOBDD with a single root node representing a constant function). Thus, if $f_1, \ldots, f_k$ are the functions for the root nodes $v_0^1, \ldots, v_0^k$ of $\overline{\mathfrak{B}}$, then the size (i.e., total number of nodes) of $\overline{\mathfrak{B}}$ is often smaller than but at most $N_{f_1} + \ldots + N_{f_k}$ where N_f denotes the $\wp$-ROBDD size of f.

For the symbolic representation of a transition system by means of switching functions $\Delta(\overline{x}, \overline{x}')$ for the transition relation and switching functions $f_a(\overline{x})$, $a \in AP$, for the satisfaction sets of the atomic propositions (see page 386), one might use a shared OBDD with

root nodes for Δ and the f_a's. As we mentioned before, the chosen variable ordering $\wp$ can be crucial for the size of the SOBDD representing a transition system. Experimental studies have shown that typically good variable orderings are obtained when grouping together the unprimed variables x_i and their copies x_i'. Later we will give some formal arguments why such interleaved variable orderings, like $\wp = (x_1, x_1', \ldots, x_n, x_n')$, are advantageous.

To perform the CTL model-checking procedure in a symbolic way, the shared OBDD $\overline{\mathfrak{B}}$ with root nodes for Δ and the f_a's has to be extended by new root nodes representing the characteristic functions of the satisfaction sets $Sat(\Psi)$ for the state subformulae Ψ of the CTL formula Φ to be checked. For instance, if $\Phi = a \wedge \neg b$ with atomic propositions a, b, then we first have to insert a root node for the characteristic function $f_{\neg b} = \neg f_b$ for $Sat(\neg b)$ and then a root node for the switching function $f_a \wedge f_{\neg b}$. The treatment of formulae of the form, e.g., $\exists\Diamond\Psi$ or $\exists\Box\Psi$ by means of Algorithms 20 or 21, requires creating additional root nodes for the functions f_i representing the current approximations of satisfaction sets. Of course, adding a new root node for some switching function f also means that we have to add nodes for all order-consistent cofactors of f that are not yet represented by a node in $\overline{\mathfrak{B}}$.

To support such dynamic changes of the set of switching functions to be represented, the realization of shared OBDDs typically relies on the use of two tables: the *unique table*, which contains the relevant information about the nodes and serves to keep the diagram reduced during the execution of synthesis algorithms, and a *computed table*, which is needed for efficiency reasons. Let us first explain the ideas behind the use of the unique table. The implementation of synthesis algorithms and the use of the computed table will be explained later.

The Unique Table The entries of the unique table are triples of the form

$$\mathit{info}(v) \;=\; \langle \mathit{var}(v), \mathit{succ}_0(v), \mathit{succ}_0(v) \rangle$$

for each inner node v. Note that these info-triples contain the relevant information which is necessary for the applicability of the isomorphism rule. Accessing the unique table is supported by a *find_or_add*-operation which takes as argument a triple $\langle z, v_1, v_0 \rangle$ consisting of a variable z and two nodes v_1 and v_0 with $v_1 \neq v_0$. The task of the *find_or_add*-operation is to check whether there exists a node v in the shared OBDD $\overline{\mathfrak{B}}$ such that $\mathit{info}(v) = \langle z, v_1, v_0 \rangle$. If so, then it returns node v, otherwise it creates a new z-node v with 1-successor v_1 and 0-successor v_0, and makes a corresponding entry in the unique table. Thus, the *find_or_add* operation can be viewed as the SOBDD realization of the *isomorphism rule*. In most BDD packages, the unique table is organized using appropriate hashing techniques. We skip such details here and assume constant expected time to access the into-triple for any node, and to perform the *find_or_add*-operation.

Boolean Operators Let us now consider how *synthesis algorithms* can be realized on SOBDDs, using the unique table. An elegant, but also very efficient way is to support a ternary operator, called ITE for "if-then-else", that covers all Boolean connectives. The *ITE operator* takes as arguments three switching functions g, f_1, f_2 and composes them according to "if g then f_1 else f_2". Formally:

$$ITE(g, f_1, f_2) \quad = \quad (g \wedge f_1) \ \vee \ (\neg\, g \wedge f_2)$$

For the special case where g is constant we have $ITE(0, f_1, f_2) = f_2$ and $ITE(1, f_1, f_2) = f_1$. The ITE operator fits very well with the representation of the SOBDD nodes in the unique table by their info-triples as we have:

$$f_v \ = \ ITE(\, z, \, f_{succ_1(v)}, \, f_{succ_0(v)} \,).$$

The negation operator is obtained by $\neg f = ITE(f, 0, 1)$. Also all other Boolean connectives can be expressed by the ITE-operator. For example:

$$\begin{array}{lcll} f_1 \vee f_2 & = & ITE(f_1, 1, f_2) & \\ f_1 \wedge f_2 & = & ITE(f_1, f_2, 0) & \\ f_1 \oplus f_2 & = & ITE(f_1, \neg f_2, f_2) & = \quad ITE(f_1, ITE(f_2, 0, 1), f_2) \\ f_1 \rightarrow f_2 & = & ITE(f_1, f_2, 1) & \end{array}$$

The realization of the ITE operator on an SOBDD $\overline{\mathfrak{B}}$ requires a procedure that takes as input three nodes u, v_1, v_2 of $\overline{\mathfrak{B}}$ and returns a possibly new node w such that $f_w = ITE(f_u, f_{v_1}, f_{v_2})$, by reusing existing nodes whenever possible and adding new nodes to $\overline{\mathfrak{B}}$ if necessary. For this, the sub-OBDDs for the input nodes u, v_1, and v_2 are traversed simultaneously in a top-down fashion, while the synthesis of the sub-ROBDD for w (and generation of new nodes) is performed in a bottom-up manner. This method relies on the following observation.

Lemma 6.76. Cofactors of $ITE(\cdot)$

If g, f_1, f_2 are switching functions for Var, $z \in$ Var and $b \in \{0, 1\}$, then

$$ITE(g, f_1, f_2)|_{z=b} \ = \ ITE(g|_{z=b}, f_1|_{z=b}, f_2|_{z=b}).$$

Proof: For simplicity, let us assume that g, f_1, f_2 are switching functions for the same variable set $Var = \{z, y_1, \ldots, y_m\}$. This, in fact, is no proper restriction as we may simply take the union of the variable sets of g, f_1 and f_2 and regard all three functions as switching functions for the resulting variable sets. Let $(a, \overline{\mathfrak{r}})$ be a short-form notation for

the evaluation $[z = a, \overline{y} = \overline{\mathfrak{c}}] \in \mathit{Eval}(\mathit{Var})$. Then, we have

$$\begin{aligned}
& \mathit{ITE}(g, f_1, f_2)|_{z=b}(a, \overline{\mathfrak{c}}) \\
= \;& \mathit{ITE}(g, f_1, f_2)(b, \overline{\mathfrak{c}}) \\
= \;& (\, g(b, \overline{\mathfrak{c}}) \wedge f_1(b, \overline{\mathfrak{c}}) \,) \vee (\, \neg g(b, \overline{\mathfrak{c}}) \wedge f_2(b, \overline{\mathfrak{c}}) \,) \\
= \;& (\, g|_{z=b}(a, \overline{\mathfrak{c}}) \wedge f_1|_{z=b}(a, \overline{\mathfrak{c}}) \,) \vee (\, \neg g|_{z=b}(a, \overline{\mathfrak{c}}) \wedge f_2|_{z=b}(a, \overline{\mathfrak{c}}) \,) \\
= \;& \mathit{ITE}(g|_{z=b}, f_1|_{z=b}, f_2|_{z=b})(a, \overline{\mathfrak{c}}).
\end{aligned}$$

■

Thus, a node in a $\wp$-SOBDD for representing $\mathit{ITE}(g, f_1, f_2)$ is a node w such that $\mathit{info}(w) = \langle z, w_1, w_0 \rangle$ where

- z is the minimal essential variable of $\mathit{ITE}(g, f_1, f_2)$ according to $<_\wp$,
- w_1, w_0 are SOBDD nodes with:

$$f_{w_1} = \mathit{ITE}(g|_{z=1}, f_1|_{z=1}, f_2|_{z=1}) \quad \text{and} \quad f_{w_0} = \mathit{ITE}(g|_{z=0}, f_1|_{z=0}, f_2|_{z=0}).$$

This observation suggests a recursive algorithm which determines z and then recursively computes the nodes for ITE applied to the cofactors of $g = f_u$, $f_1 = f_{v_1}$, $f_2 = f_{v_2}$ for variable z. Since the explicit computation of z can be hard, we use the decomposition into cofactors for the minimal variable z that is essential for f_u, f_{v_1} or f_{v_2}:

$$z = \min\{\mathit{var}(u), \mathit{var}(v_1), \mathit{var}(v_2)\}$$

where the minimum is taken according to the total order $<_\wp$ on $\mathit{Var} \cup \{\bot\}$. (Recall that we put $\mathit{var}(v) = \bot$ for any drain v and that $x <_\wp \bot$ for all $x \in \mathit{Var}$.) If z' is the first essential variable of $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})$, then $z \leqslant_\wp z'$, since no variable $y <_\wp z$ appears in the sub-OBDDs of nodes u, v_1, v_2, and hence, no such variable y can be essential for $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})$. The case $z <_\wp z'$ is possible, if accidentally the cofactors $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=0}$ and $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=1}$ agree. In this case, however, we are in the situation of the elimination rule and the ITE algorithm returns the node representing $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=0}$. Otherwise, i.e., if the nodes w_0 and w_1 that have been recursively determined for $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=0}$ and $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=1}$, respectively, are different, then $z' = z$ and a node for representing $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})$ is obtained by the *find_or_add*-operation applied to the info-triple $\langle z, w_1, w_0 \rangle$.

The question remains how to obtain the cofactors $f_u|_{z=b}$, $f_{v_1}|_{z=b}$, and $f_{v_2}|_{z=b}$. Nodes that represent these functions are obtained easily since (by choice of variable z) nodes u, v_1, and v_2 are on the z-level or below, i.e., $z \leqslant_\wp \mathit{var}(v)$ for $v \in \{u, v_1, v_2\}$. If $\mathit{var}(v) = z$, then $f_v|_{z=b}$ is represented by the b-successor of v. If $z <_\wp \mathit{var}(v)$, then z is not essential for f_v and we have $f_v|_{z=b} = f_v$. Thus, if we define

$$v|_{z=b} \;=\; \begin{cases} \mathit{succ}_b(v) & \text{if } \mathit{var}(v) = z \\ u & \text{if } z <_\wp \mathit{var}(v), \end{cases}$$

then $v|_{z=b}$ is the node in $\overline{\mathfrak{B}}$ representing $f_v|_{z=b}$. Hence, a node representing the function $\mathit{ITE}(f_u, f_{v_1}, f_{v_2})|_{z=b}$ is obtained by a recursive call of the ITE algorithm with the arguments $u|_{z=b}$, $v_1|_{z=b}$ and $v_2|_{z=b}$ (Lemma 6.76). Note that these are already existing nodes in the SOBDD $\overline{\mathfrak{B}}$.

The steps to realize the ITE-operator on a shared OBDD by means of a DFS-based traversal of the sub-OBDDs of u, v_1, and v_2 to determine the relevant cofactors (where recursive calls of the ITE-algorithm are required) have been summarized in Algorithm 22 on page 412.

Algorithm 22 $\mathit{ITE}(u, v_1, v_2)$ (first version)

```
if u is terminal then
  if val(u) = 1 then
    w := v_1                                   (* ITE(1, f_{v_1}, f_{v_2}) = f_{v_1} *)
  else
    w := v_2                                   (* ITE(0, f_{v_1}, f_{v_2}) = f_{v_2} *)
  fi
else
  z := min{var(u), var(v_1), var(v_2)};
  w_1 := ITE(u|_{z=1}, v_1|_{z=1}, v_2|_{z=1});
  w_0 := ITE(u|_{z=0}, v_1|_{z=0}, v_2|_{z=0});
  if w_0 = w_1 then
    w := w_1;                                  (* elimination rule *)
  else
    w := find_or_add(z, w_1, w_0);             (* isomorphism rule (?) *)
  fi
fi
return w
```

Before discussing the complexity of the ITE algorithms, we will first study how the size of the SOBDD can change through performing the ITE algorithm. The size of the sub-OBDD for the (possibly new) node w representing $\mathit{ITE}(u, v_1, v_2)$ is bounded by $N_u \cdot N_{v_1} \cdot N_{v_2}$ where N_v denotes the number of the nodes in the sub-OBDD $\overline{\mathfrak{B}}_v$ for node v. This follows from

the fact that each node w' in the generated sub-OBDD for $ITE(u, v_1, v_2)$ corresponds to one or more triples (u', v_1', v_2'), where u' is a node in $\overline{\mathfrak{B}}_u$ and v_i' a node in $\overline{\mathfrak{B}}_{v_i}$ Formally:

Lemma 6.77. ROBDD Size of $ITE(g, f_1, f_2)$

The size of the $\wp$-ROBDD for $ITE(g, f_1, f_2)$ is bounded by $N_g \cdot N_{f_1} \cdot N_{f_2}$ where N_f denotes the size of the $\wp$-ROBDD for f.

Proof: Let $\wp = (z_1, \ldots, z_m)$ and let $\mathfrak{B}_f$ denote the $\wp$-ROBDD for f where the nodes are the $\wp$-consistent cofactors of f (see the proof of part (a) of Theorem 6.70). We write V_f for the node set of $\mathfrak{B}_f$, i.e.,

$$V_f \;=\; \{\, f|_{z_1=b_1,\ldots,z_i=b_i} \mid 0 \leqslant i \leqslant m,\, b_1, \ldots, b_i \in \{0,1\} \,\}.$$

Note that several of the cofactors $f|_{z_1=b_1,\ldots,z_i=b_i}$ might agree, and hence, they stand for the same element (node) of V_f. By Lemma 6.76, the node set $V_{ITE(g,f_1,f_2)}$ of the $\wp$-ROBDD $\mathfrak{B}_{ITE(g,f_1,f_2)}$ for $ITE(g, f_1, f_2)$ agrees with the set of switching functions

$$ITE(g|_{z_1=b_1,\ldots,z_i=b_i}, f_1|_{z_1=b_1,\ldots,z_i=b_i}, f_2 z_1 = b_1, \ldots, z_i = b_i)$$

where $0 \leqslant i \leqslant m$, $b_1, \ldots, b_i \in \{0,1\}$. Thus, the function

$$\imath : V_g \times V_{f_1} \times V_{f_2} \to V_{ITE(g,f_1,f_2)}, \quad \imath(g', f_1', f_2') \;=\; ITE(g', f_1', f_2')$$

that maps any triple (g', f_1', f_2') where g' is a node in $\mathfrak{B}_g$ (i.e., a $\wp$-consistent cofactor of g) and f_i' a node in $\mathfrak{B}_{f_i}$ (i.e., a $\wp$-consistent cofactor of f_i) to the node $ITE(g', f_1', f_2')$ of $\mathfrak{B}_{ITE(g,f_1,f_2)}$ yields a surjective mapping from $V_g \times V_{f_1} \times V_{f_2}$ to some superset of $V_{ITE(g,f_1,f_2)}$. Hence:

$$N_{ITE(g,f_1,f_2)} \;=\; |V_{ITE(g,f_1,f_2)}| \;\leqslant\; |V_g \times V_{f_1} \times V_{f_2}| \;=\; N_g \cdot N_{f_1} \cdot N_{f_2}$$

Observe that only the triples $(g', f_1', f_2') \in V_g \times V_{f_1} \times V_{f_2}$ where g', f_1', f_2' arise from g, f_1, f_2 by the same evaluation $[z_1 = b_1, \ldots, z_i = b_i]$ are mapped via $\imath$ to nodes of $\mathfrak{B}_{ITE(g,f_1,f_2)}$. Furthermore, $ITE(g', f_1', f_2') = ITE(g'', f_1'', f_2'')$ is possible for $(g', f_1', f_2') \neq (g'', f_1'', f_2'')$. Therefore, $N_{ITE(g,f_1,f_2)}$ can be much smaller than $N_g \cdot N_{f_1} \cdot N_{f_2}$. ■

As a consequence of Lemma 6.77, the size of the $\wp$-ROBDD for $f_1 \vee f_2$ is bounded above by the product of the sizes of the $\wp$-ROBDDs for f_1 and f_2. Recall that $f_1 \vee f_2 = ITE(f_1, 1, f_2)$ and hence

$$N_{f_1 \vee f_2} \;\leqslant\; N_{f_1} \cdot N_1 \cdot N_{f_2} \;=\; N_{f_1} \cdot N_{f_2}.$$

The same holds for conjunction and even for any other binary Boolean connective. This also applies to operators like $\oplus$ (xor, parity) where $f \oplus g = ITE(f, \neg g, g)$, i.e., negation

is needed to express $f \oplus g$ by ITE, f and g. Lemma 6.77 yields the bound $N_f \cdot N_g^2$ for the $\wp$-ROBDD size for $f \oplus g$. However, since the $\wp$-ROBDDs for g and $\neg g$ are isomorphic up to exchanging the values of the drains, the recursive calls in the ITE-algorithms have the form $ITE(u, v, w)$ where $f_v = \neg f_w$. Hence, the number of nodes in the $\wp$-ROBDD for $f \oplus g$ is bounded by the number of triples $(f', \neg g', g')$ where f' is a $\wp$-consistent cofactor of f and g' a $\wp$-consistent cofactor of g. This yields the upper bound $N_f \cdot N_g$ for the size of the $\wp$-ROBDD of $f \oplus g$.

Computed Table The problem with Algorithm 22 is that its worst-case running time is exponential. The reason for this is that, if there are several paths that lead from (u, v_1, v_2) to (u', v_1', v_2'), then $ITE(u', v_1', v_2')$ is invoked several times and the whole sub-OBDDs of these nodes u', v_1', v_2' are traversed in each of these recursive invocations. To avoid such redundant recursive invocations one uses a *computed table* that stores the tuples (u, v_1, v_2), where $ITE(u, v_1, v_2)$ has already been executed, together with the result, i.e., the SOBDDD-node w with $f_w = ITE(f_u, f_{v_1}, f_{v_2})$. Thus, we may refine the ITE-algorithm as shown in Algorithm 23 on page 414.

Algorithm 23 $ITE(u, v_1, v_2)$

if there is an entry for (u, v_1, v_2, w) in the computed table **then**
  return node w
else
  (* no entry for $ITE(u, v_1, v_2)$ in the computed table *)
  if u is terminal **then**
    if $val(u) = 1$ **then** $w := v_1$ **else** $w := v_2$ **fi**
  else
    $z := \min\{var(u), var(v_1), var(v_2)\}$;
    $w_1 := ITE(u|_{z=1}, v_1|_{z=1}, v_2|_{z=1})$;
    $w_0 := ITE(u|_{z=0}, v_1|_{z=0}, v_2|_{z=0})$;
    if $w_0 = w_1$ **then** $w := w_1$ **else** $w := find\_or\_add(z, w_1, w_0)$ **fi**;
    insert (u, v_1, v_2, w) in the computed table;
    return node w
  fi
fi

The number of recursive calls in Algorithm 23 for the input-nodes u, v_1, v_2 agrees with the $\wp$-ROBDD size of $ITE(f_u, f_{v_1}, f_{v_2})$, which is bounded by $N_u \cdot N_{v_1} \cdot N_{v_2}$, where $N_v = N_{f_v}$ denotes the number of nodes in the sub-OBDD of node v. The cost per recursive call is constant when the time to access the computed table and to perform the *find_or_add*-operation is assumed to be constant. This is an adequate assumption if suitable hashing techniques are used to organize both tables. However, in practice the running time of the ITE algorithm is often much better than this upper bound. First, only in extreme cases

is the size of the $\wp$-ROBDD for $ITE(f_u, f_{v_1}, f_{v_2})$ roughly $N_u \cdot N_{v_1} \cdot N_{v_2}$. Second, the use of the ITE operator yields the advantage that all synthesis algorithms that are expressible via ITE rely on the same computed table. This increases the hit rate and makes it possible that the computation aborts due to an entry in the computed table that has been made during the synthesis of another function. Moreover, there are several tricks to intensify this phenomenon. One simple trick is to make use of equivalence rules, such as

$$f_1 \vee f_2 = ITE(f_1, 1, f_2) = ITE(f_2, 1, f_1) = ITE(f_1, f_1, f_2) = \ldots,$$

and to transform the arguments of ITE into so-called *standard triples*. Furthermore, the ITE-algorithm can be refined by terminating the traversal in certain special cases. E.g., we have $ITE(g, f, f) = f$ and $ITE(g, 1, 0) = g$, which allows aborting the computation if either the last two arguments agree or they equal the pair consisting of the 1- and 0-drains.

Remark 6.78. The Negation Operator

We noticed above that the negation operator can be realized as an instance of the ITE operator as we have $\neg f = ITE(f, 0, 1)$. However, applying the ITE algorithm seems to be unnecessarily complicated since the $\wp$-ROBDDs for f and $\neg f$ just differ in the values of the drains. In fact, swapping the values of the drains is an adequate technique to realize negation on ROBDDs, but it is not for shared OBDDs (since changing the values of the drains also affects the functions of all other root nodes). However, there is a simple trick to perform negation in SOBDDs in constant time. It relies on the use of *complement bits* for the edges. This permits the representation of f and $\neg f$ by a single node. Negation then just means swapping the value of the complement bit of the incoming edge. Besides leading to smaller SOBDD sizes, the use of complement bits also tightens the effect of standard triples, since now more equivalence rules can be used to identify the input triples for ITE or for early termination. E.g., we have

$$ITE(f_1, 1, f_2) = f_1 \vee f_2 = \neg(\neg f_1 \wedge \neg f_2) \;=\; \neg ITE(\neg f_1, \neg f_2, 0)$$

and $ITE(g, 0, 1) = \neg g$. However, to ensure canonicity some extra requirements are needed. For instance, the constant function 0 could be represented by the 0-drain (with unnegated complement bit) or the 1-drain with negated complement bit. To guarantee the uniqueness, one could require that only the 1-drain be used and that complement bits be used only for the 0-edges (i.e., edges from the inner nodes to their 0-successors) and the pointers to the root nodes. For more information on such advanced implementation techniques, further theoretical considerations on OBDDs and variants thereof, we refer to textbooks on BDDs, such as [134, 292, 300, 418]. ■

Other Operators on OBDDs Although all Boolean connectives can be expressed by the ITE operator, some more operators are required to perform, e.g., the CTL model

checking procedure with an SOBDD representation of the transition system. Recall that in the symbolic computation of $Sat(\exists\Diamond B)$ and $Sat(\exists\Box B)$ (see Algorithms 20 and 21 on page 391) we use iterations of the form

$$f_{j+1}(\overline{x}) \;:=\; f_j(\overline{x}) \;\; op \;\; \exists\overline{x}'.(\,\Delta(\overline{x},\overline{x}') \;\wedge\; f_j(\overline{x}')\,)$$

where $op \in \{\vee,\wedge\}$ and $f_j = \chi_T$ is the characteristic function of some set T. Thus, f_{j+1} is the characteristic function of $T \cap Pre(T)$ (if $op = \wedge$) and $T \cup Pre(T)$ (if $op = \vee$). (For the treatment of constrained reachability properties like $\exists(C \,\mathsf{U}\, B)$, we have an additional conjunction with χ_C, but this is just a technical detail.) Besides disjunction and conjunction, these iterations use existential quantification and renaming. The major difficulty is the *preimage* computation, i.e., the computation of the symbolic representation of $Pre(T)$ by means of the expression $\exists\overline{x}'.(\Delta(\overline{x},\overline{x}') \wedge f_j(\overline{x}'))$, which is often called a *relational product*.

Let us start with the *rename* operator which is inherent in $f_j(\overline{x}')$ since f_j is a switching function for the variables in $\overline{x}$ and $f_j(\overline{x}') = f_j\{\overline{x}' \leftarrow \overline{x}\}$ means the function that results from f_j when renaming the unprimed variables x_i into their primed copies x_i'. At first glance, the renaming operator appears to be trivial as we simply may modify the variable labeling function by replacing x_i with x_i'. This operation certainly transforms a given ROBDD for $f(\overline{x})$ into a ROBDD for $f(\overline{x}')$. However, if we are given an arbitrary variable ordering $\wp$ where the relative order of the unprimed variables can be different from the relative order of the primed variables (i.e., $x_i <_\wp x_j$ while $x_j' <_\wp x_i'$) then this renaming operator is no longer adequate, since the resulting ROBDD for $f(\overline{x}')$ would rely on another ordering than $\wp$. Furthermore, for an implementation with shared OBDDs, modifying existing nodes is not appropriate since then the functions for all root nodes might be affected. In fact, it is not possible to design a general rename operator which runs in time polynomial in the size of the input ROBDD. To see why, consider the function

$$f \;=\; (z_1 \wedge y_1) \vee (z_2 \wedge y_2) \vee \ldots \vee (z_m \wedge y_m)$$

of Example 6.73 (page 404). Suppose $Var = \{z_i, y_i, z_i', y_i' \mid 1 \leqslant i \leqslant m\}$ and $\wp = (z_m, y_m, \ldots, z_1, y_1, z_1', \ldots, z_m', y_1', \ldots, y_m')$ and that we are given the $\wp$-ROBDD $\mathfrak{B}_f$ for f in the form of a root node v of a $\wp$-SOBDD. The goal is now to rename z_i into z_i' and y_i into y_i' in f, i.e., to compute the $\wp$-ROBDD representation of

$$f\{z_i' \leftarrow z_i, y_i' \leftarrow y_i \mid 1 \leqslant i \leqslant m\} \;=\; (z_1' \wedge y_1') \vee \ldots \vee (z_m' \wedge y_m').$$

By the results of Example 6.73: while the $\wp$-ROBDD size of f is $2m+2$, the $\wp$-ROBDD size of $f\{\ldots\}$ is $\Omega(2^m)$. This observation shows that there is no linear-time algorithm that realizes the rename operator for arbitrary variable orderings. However, if we suppose that x_i and x_i' are neighbors in the ordering $\wp$, e.g., $x_i <_\wp x_i'$ and there is no variable z with $x_s <_\wp z <_\wp x_i'$, then renaming x_i into x_i' for a function $f(\overline{x})$ is simple. As for the ITE operator, we can work with a DFS-based traversal of the sub-OBDD for the node representing $f(\overline{x})$ and

generate the ROBDD for $f(\overline{x}')$ in a bottom-up manner; see Algorithm 24. The algorithm takes as input a node v of a π-SOBDD and tuples $\overline{x} = (x_1, \ldots, x_n)$, $\overline{x}' = (x_1', \ldots, x_n')$, of pairwise distinct variables such that $x_1', \ldots, x_n'$ are not essential for f_v and x_i und x_i' are neighbors in π. The output is a node w such that $f_w = f_v\{\overline{x} \leftarrow \overline{x}'\}$. To avoid multiple invocations of the algorithms with the same input node v, we use a computed table that stores all nodes v where $\mathit{Rename}(v, \overline{x} \leftarrow \overline{x}')$ has already been executed together with the output node w, i.e., the node w with $f_w = f_v\{\overline{x} \leftarrow \overline{x}'\}$.

Algorithm 24 $\mathit{Rename}(v, \overline{x} \leftarrow \overline{x}')$

```
if there is an entry (v, w) in the computed table then
  return w
else
  if v ist terminal then
    w := v
  else
    w_0 := Rename(succ_0(v), x̄ ← x̄');
    w_1 := Rename(succ_1(v), x̄ ← x̄');
    if var(v) = z_j for some j ∈ {1, ..., n} then
      z := z_j'                                  (* replace z_j with z_j' *)
    else
      z := var(v)
    fi
    w := find_or_add(z, w_1, w_0);
  fi
  insert (u, w) in the computed table;
  return w
fi
```

Remark 6.79. Interleaved Variable Orderings for Transition Systems

We noticed before (page 409) that interleaved variable orderings, such as $(x_1, x_1', \ldots, x_n, x_n')$, are favorable for the representation of transition systems. The rename operator yields one reason, as interleaved variable orderings permit use of the renaming that is inherent in the OBDD algorithms for the preimage computation by the above algorithm. Another formal argument for interleaved variable orderings is that they are beneficial for the construction of the ROBDD representation for the transition relation of a composite transition system. In Remark 6.59 (page 389) we saw that if TS arises from the synchronous product of transition systems $TS_1, \ldots, TS_m$, then the switching function $\Delta(\overline{x}_1, \ldots, \overline{x}_n, \overline{x}_1', \ldots, \overline{x}_n')$ for TS's transition relation is obtained by the conjunction of the switching functions $\Delta_i(\overline{x}_i, \overline{x}_i')$ for the transition relations in TS_i, $i = 1, \ldots, m$. Since the Δ_i's do not have common variables the $\wp$-ROBDD size of Δ is bounded by

$$N_\Delta \leqslant N_{\Delta_1} + \ldots + N_{\Delta_m}$$

whenever $\wp$ is an interleaved variable ordering where all variables in $\overline{x}_i$ and $\overline{x}'_i$ are grouped together. Thus, there is no exponential blowup for the ROBDD sizes! Although Lemma 6.77 yields the upper bound $N_{\Delta_1} \cdot \ldots \cdot N_{\Delta_m}$ for any variable ordering, for such interleaved variable orderings $\wp$ at most linear growth of the $\wp$-ROBDD sizes is guaranteed. This is due to the fact that the $\wp$-ROBDD for Δ arises by linking the $\wp$-ROBDDs for $\Delta_1, \ldots, \Delta_m$. E.g., if we suppose that all variables in $\overline{x}_i, \overline{x}'_i$ appear after the variables in $\overline{x}_1, \overline{x}'_1, \ldots, \overline{x}_{i-1}, \overline{x}'_{i-1}$ in $\wp$, then we may simply redirect any edge to the 1-drain in the $\wp$-ROBDD for Δ_{i-1} to the root of the $\wp$-ROBDD for Δ_i (for $1 \leqslant i < m$). This yields the $\wp$-ROBDD for Δ.

A slightly more involved argument applies to the interleaving $TS = TS_1 \,|||\, \ldots \,|||\, TS_m$ where Δ arises by the disjunction of the Δ_i's together with the side conditions $\overline{x}_j = \overline{x}'_j$ for $i \neq j$. With an interleaved variable ordering where the variables for TS_i appear before the variables of $TS_{i+1}, \ldots, TS_m$, we can guarantee that the $\wp$-ROBDD size N_Δ is bounded by $\mathcal{O}(\,(N_{\Delta_1} + \ldots + N_{\Delta_m}) \cdot n^2\,)$ where the n is the total number of variables in TS. The additional factor $\mathcal{O}(n^2)$ stands for the representation of the conditions $\overline{x}_i = \overline{x}'_i$. ■

Existential quantification is reducible to ITE and the cofactor operator, as we have $\exists x.f = f|_{x=0} \vee f|_{x=1}$. As we mentioned in the explanations for the ITE operator, the cofactor operator $f \mapsto f|_{x=b}$ is trivial if $x \leqslant_\wp z$ for the first essential variable z of f in the given variable ordering $\wp$, since then $f|_{z=b}$ is represented by the b-successor of the node representing f, if $x = z$, and $f|_{x=b} = f$, if $x <_\wp z$ or f is constant. If a representation for $f|_{x=b}$ is required where $z <_\wp x$, then we may use the fact that $(f|_{x=b})|_{z=\mathfrak{c}} = (f|_{z=\mathfrak{c}})|_{x=b}$ and apply the cofactor-operator recursively to the successors of the node representing f. This leads to Algorithm 25 on 419. Here, again, we use a computed table that organizes all pairs (v, w) where the cofactor $f_v|_{x=b}$ is known to be represented by node w.

The time complexity of the renaming algorithm, as well as the algorithm for obtaining cofactors is bounded by $\mathcal{O}(\mathit{size}(\overline{\mathfrak{B}}_v))$ as both rely on a DFS-based traversal of the sub-OBDD $\overline{\mathfrak{B}}_v$, assuming constant time for accessing the entries in the unique and computed table. The $\wp$-ROBDD size of f agrees with the $\wp$-ROBDD size of $f\{\overline{x} \leftarrow \overline{x}'\}$ under the assumptions we made for $\overline{x}$, $\overline{x}'$ and $\wp$. The $\wp$-ROBDD size of $f|_{x=b}$ is at most the $\wp$-ROBDD size of f. This follows from the fact that given the $\wp$-ROBDD $\mathfrak{B}_f$ for f, an $\wp$-ROBDD for $f|_{x=\overline{b}}$ is obtained by redirecting any edge $w \to u$ that leads to an x-node u to the b-successor of u and then removing all x-nodes.[7] In summary, the preimage computation via the *relational product*

$$\exists \overline{x}.(\Delta \wedge f\{\overline{x}' \leftarrow \overline{x}\})$$

– which is required for, e.g., the symbolic computation of $\mathit{Sat}(\exists\Box B)$ – could be performed by first applying the rename operator to f, then the conjunction operator (as an instance of

[7] Although this yields a correct operator for constructing the $\wp$-ROBDD for $f|_{x=b}$ from the $\wp$-ROBDD for f, the redirection of edges is not adequate for an implementation with shared OBDDs.

Algorithm 25 *Cof*(v, x, b)

```
if there is an entry (v, w) in the computed table then
  return w
else
  if v ist terminal then
    w := v
  else
    if x ⩽_℘ var(v) then
      w := v|_{x=b}
    else
      z := var(v);
      w_1 := Cof(succ_1(v), x, b);   w_0 := Cof(succ_0(v), x, b);
      if w_0 = w_1 then w := w_1 else w := find_or_add(z, w_1, w_0) fi
    fi
  fi
  insert (u, w) in the computed table;
  return node w
fi
```

ITE operator) to the nodes representing Δ and $f(\overline{x}')$, and finally computing the existential quantifications by cofactors and disjunctions. This naive approach is very time-consuming as it relies on several top-down traversals in the shared OBDD. It also yields the problem that the ROBDD representation for $\Delta \wedge f\{\overline{x}' \leftarrow \overline{x}\}$ could be very large.

A more elegant approach for a BDD-based preimage computation by means of the relational product $\exists \overline{x}'.\,(\,\Delta \wedge f\{\overline{x}' \leftarrow \overline{x}\}\,)$ is to realize the existential quantifications, renaming and conjunction simultaneously by a single DFS traversal of the sub-OBDDs of the nodes representing Δ and f, with intermediate calls of the ITE operator to realize the disjunctions that are inherent in the existential quantification. Algorithm 26 on page 420 summarizes the main steps of this approach.

The input of Algorithm 26 are two nodes u and v of a $\wp$-SOBDD such that $f_u = \Delta(\overline{x}, \overline{x}')$ and $f_v = f(\overline{x}')$. The assumptions about the variable tuples $\overline{x}$, $\overline{x}'$ are as above, i.e., $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{x}' = (x_1', \ldots, x_n')$ are tuples consisting of pairwise distinct variables such that the unprimed variables x_i and their primed copies x_i' are neighbours in $\wp$. For simplicity, let us suppose that $\wp$ interleaves the unprimed and primed variables, say

$$x_1 <_\wp x_1' <_\wp x_2 <_\wp x_2' <_\wp \ldots <_\wp x_n <_\wp x_n'.$$

The output of Algorithm 26 is a (possibly new) node w with $f_w = \exists \overline{x}'.\,(\,\Delta(\overline{x}, \overline{x}') \wedge f(\overline{x}')\,) = \exists \overline{x}'.(f_u \wedge f_v)$. The termination condition of Algorithm 26 is given by the cases: (i) there exists an entry in the computed table, or (ii) $\Delta = 0$ or $f = 0$ in which case $\exists \overline{x}'.(\Delta \wedge f) = 0$,

Algorithm 26 Relational product $RelProd(u, v)$

if there exists an entry (u, v, w) in the computed table **then return** w **fi**;
if u or v is the 0-drain **then return** the 0-drain **fi**;
if u and v are the 1-drain **then return** the 1-drain **fi**;

$y := \min\{var(u), var(v)\}$, say $y \in \{x_i, x_i'\}$
if $y = x_i$ **then**
  $w_{1,0} := RelProd(u|_{x_i=1,x_i'=0}, v|_{x_i=0})$;
  $w_{1,1} := RelProd(u|_{x_i=1,x_i'=1}, v|_{x_i=1})$;
  $w_1 := ITE(w_{1,0}, 1, w_{1,1})$;

  $w_{0,0} := RelProd(u|_{x_i=0,x_i'=0}, v|_{x_i=0}, \overline{x}, \overline{x}')$;
  $w_{0,1} := RelProd(u|_{x_i=0,x_i'=1}, v|_{x_i=1}, \overline{x}, \overline{x}')$;
  $w_0 := ITE(w_{0,0}, 1, w_{0,1})$;

  if $w_1 = w_0$ **then**
    $w := w_1$ (* elimination rule *)
  else
    $w := find\_or\_add(x_i, w_1, w_0)$
  fi
else
  $w_0 := RelProd(u|_{x_i'=0}, v)$; $w_1 := RelProd(u|_{x_i'=1}, v)$;
  $w := ITE(w_0, 1, w_1)$
fi
insert (u, v, w) in the computed table;
return w

or (iii) $\Delta = f = 1$ in which case $\exists \overline{x'}.(\Delta \wedge f) = 1$. In the remaining cases, the traversal of the sub-OBDDs of u and v relies on the expansion rule:

$$\begin{aligned}
&\exists x_1' \exists x_2' \ldots \exists x_n'.\,(\,\Delta \;\wedge\; f\{x_1' \leftarrow x_1, x_2' \leftarrow x_2, \ldots, x_n' \leftarrow x_n\}\,)\,)|_{x_1=b} \\
&= \quad \exists x_2' \ldots \exists x_n'.\,(\,\Delta|_{x_1=b,x_1'=0} \;\wedge\; f|_{x_1=0}\{x_2' \leftarrow x_2, \ldots, x_n' \leftarrow x_n\}\,) \quad \vee \\
&\qquad \exists x_2' \ldots \exists x_n'.\,(\,\Delta|_{x_1=b,x_1'=1} \;\wedge\; f|_{x_1=1}\{x_2' \leftarrow x_2, \ldots, x_n' \leftarrow x_n\}\,)
\end{aligned}$$

In the literature, several techniques have been proposed that serve to improve the image or preimage computation. These range from techniques that rely on partitionings of the variables that attempt to perform the existential quantification as soon as possible (as they decrease the number of essential variables and often lead to smaller ROBDD sizes). Other techniques rely on so-called input- or output splitting (which use alternative expansion rules), and special ROBDD operators that attempt to replace, e.g., Δ with other switching functions $\tilde{\Delta}$ such that $\Delta \wedge f = \tilde{\Delta} \wedge f$ and such that the $\wp$-ROBDD for $\overline{\Delta}$ is smaller than the $\wp$-ROBDD for Δ. In the case of an iterated preimage computation by $T_0 = B$ and $T_{j+1} = T_j \cup \mathit{Pre}(T_j)$ for the symbolic computation of $\mathit{Pre}^*(B)$, one might also switch from T_j to any set $\tilde{T}$ such that $T_j \setminus T_{j-1} \subseteq \tilde{T} \subseteq T_j$ and compute $T_j \cup \mathit{Pre}(\tilde{T})$. For such advanced techniques, we refer the interested reader to [92, 292, 374] and the literature mentioned there.

Summary We now have all ingredients for a symbolic realization of the standard CTL model-checking approach which recursively computes the satisfaction sets of the subformulae by means of shared OBDDs. Initially, one has to construct the ROBDD representation of the transition system to be analyzed. This can be done in a compositional way by means of synthesis operators (disjunction, conjunction, etc.) as mentioned in Remark 6.59. Furthermore, we assume that ROBDD representations of the satisfaction sets for the atomic propositions are given. This assumption is justified since often the atomic propositions can serve as variables for the encoding of the states. (And in this case their satisfaction set is just given by a projection function.) The CTL model-checking procedure can then be performed by means of the ITE algorithm (to treat the propositional logic fragment of CTL) and the symbolic BFS-based algorithms sketched in Algorithms 20 and 21. Both rely on an iterative preimage computation. Techniques to do this efficiently have been discussed above. The termination condition requires checking the equality of two switching functions. This, in fact, is trivial for shared OBDDs since it simply amounts to the comparison of the corresponding nodes and can be performed in constant time.

6.8 CTL*

We saw in Theorem 6.21 on page 337 that CTL and LTL have incomparable expressiveness. An extension of CTL, proposed by Emerson and Halpern, called CTL*, combines the features of both logics, and thus is more expressive than either of them.

6.8.1 Logic, Expressiveness, and Equivalence

CTL* is an extension of CTL as it allows path quantifiers $\exists$ and $\forall$ to be arbitrarily nested with linear temporal operators such as $\bigcirc$ and U. In contrast, in CTL each linear temporal operator must be immediately preceded by a path quantifier. As in CTL, the syntax of CTL* distinguishes between state and path formulae. The syntax of CTL* state formulae is roughly as in CTL, while the CTL* path formulae are defined as LTL formulae, the only difference being that arbitrary CTL* state formulae can be used as atoms. For example, $\forall \bigcirc \bigcirc a$ is a legal CTL* formula, but does not belong to CTL. The same applies to the CTL* formulae $\exists \Box \Diamond a$ and $\forall \Box \Diamond a$. (However, $\forall \Box \Diamond a$ is equivalent to the CTL formula $\forall \Box \forall \Diamond a$.)

Definition 6.80. Syntax of CTL*

CTL* *state formulae* over the set AP of atomic propositions, briefly called CTL* formulae, are formed according to the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi$$

where $a \in AP$ and φ is a path formula. The syntax of CTL* *path formulae* is given by the following grammar:

$$\varphi ::= \Phi \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

where Φ is a state formula, and φ, φ_1, and φ_2 are path formulae. ■

As for LTL or CTL, we use derived propositional logic operators like $\vee, \rightarrow, \ldots$ and let

$$\Diamond\varphi \;=\; \text{true}\,\mathsf{U}\,\varphi \quad \text{and} \quad \Box\varphi \;=\; \neg\Diamond\neg\varphi.$$

The universal path quantifier $\forall$ can be defined in CTL* by existential quantification and negation:

$$\forall\varphi \;=\; \neg\exists\neg\varphi.$$

(Note that this is not the case for CTL.)

For example, the following formulae are syntactically correct CTL* formulae:

$$\forall\Box\,(\,\bigcirc\Diamond a \;\wedge\; \neg(b\,\mathsf{U}\,\Box c)\,)$$

and

$$\forall\bigcirc\Box\neg a \quad\wedge\quad \exists\Diamond\Box\,(a \;\vee\; \forall(b\,\mathsf{U}\,a)).$$

Note that these formulae are not CTL formulae.

Definition 6.81. Satisfaction Relation for CTL*

Let $a \in AP$ be an atomic proposition, $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, state $s \in S$, Φ, Ψ be CTL* state formulae, and φ, φ_1 and φ_2 be CTL* path formulae. The satisfaction relation $\models$ is defined for state formulae by

$$\begin{array}{lll}
s \models a & \text{iff} & a \in L(s),\\
s \models \neg\,\Phi & \text{iff} & \text{not } s \models \Phi,\\
s \models \Phi \wedge \Psi & \text{iff} & (s \models \Phi) \text{ and } (s \models \Psi),\\
s \models \exists\varphi & \text{iff} & \pi \models \varphi \text{ for some } \pi \in \mathit{Paths}(s).
\end{array}$$

For path π, the satisfaction relation $\models$ for path formulae is defined by:

$$\begin{array}{lll}
\pi \models \Phi & \text{iff} & s_0 \models \Phi,\\
\pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2,\\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi,\\
\pi \models \bigcirc\,\varphi & \text{iff} & \pi[1..] \models \varphi,\\
\pi \models \varphi_1\,\mathsf{U}\,\varphi_2 & \text{iff} & \exists\, j \geqslant 0.\ (\pi[j..] \models \varphi_2 \;\wedge\; (\forall\, 0 \leqslant k < j.\ \pi[k..] \models \varphi_1))
\end{array}$$

where for path $\pi = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$, $\pi[i..]$ denotes the suffix of π from index i on. ■

Definition 6.82. CTL* Semantics for Transition Systems

For CTL*-state formula Φ, the *satisfaction set* $\mathit{Sat}(\Phi)$ is defined by

$$\mathit{Sat}(\Phi) \;=\; \{\, s \in S \mid s \models \Phi \,\}.$$

The transition system TS satisfies CTL* formula Φ if and only if Φ holds in all initial states of TS:

$$TS \models \Phi \quad \text{if and only if} \quad \forall s_0 \in I.\ s_0 \models \Phi.$$

■

Thus, $TS \models \Phi$ if and only if all initial states of TS satisfy the formula Φ.

LTL formulae are CTL* path formulae in which the elementary state formulae Φ are restricted to atomic propositions. Apparently, the interpretation of LTL over the paths of a transition system (see Definition 5.7, page 237) corresponds to the semantics of LTL obtained as a sublogic of CTL*. The following theorem demonstrates that the corresponding claim also holds for states. Thereby, every LTL formula φ is identified with the CTL* formula $\forall\varphi$, and the semantics of LTL over states is taken as a reference. Recall that according to the LTL semantics in paths, $s \models \varphi$ if and only if $\pi \models \varphi$ for all $\pi \in Paths(s)$.

Theorem 6.83. Embedding of LTL in CTL*

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states. For each LTL formula φ over AP and for each $s \in S$:

$$\underbrace{s \models \varphi}_{\text{LTL semantics}} \quad \text{if and only if} \quad \underbrace{s \models \forall\varphi}_{\text{CTL}^* \text{ semantics}} \quad .$$

In particular, $TS \models \varphi$ (with respect to the LTL semantics) if and only if $TS \models \forall\varphi$ (with respect to the CTL* semantics)

As a result, it is justified to understand LTL (with interpretation over the states of a transition system) as a *sublogic* of CTL*. Theorem 6.21 (page 337) stated that the expressivenesses of LTL and CTL are incomparable. Since LTL is a sublogic of CTL*, it now follows that CTL* subsumes LTL and CTL, i.e., there exist CTL* formulae which can be expressed neither in LTL nor in CTL.

Theorem 6.84. CTL* is More Expressive Than LTL and CTL

For the CTL* formula over $AP = \{a, b\}$,

$$\Phi \;=\; (\forall\Diamond\Box a) \;\vee\; (\forall\Box\exists\Diamond b),$$

there does not exist any equivalent LTL or CTL formula.

Proof: This follows directly from the fact that $\forall\Box\exists\Diamond b$ is a CTL formula that cannot be expressed in LTL, whereas $\Diamond\Box a$ is an LTL formula that cannot be expressed in CTL. Both these facts follow from Theorem 6.21 (page 337). ■

The relationship between LTL, CTL, and CTL* is depicted in Figure 6.27.

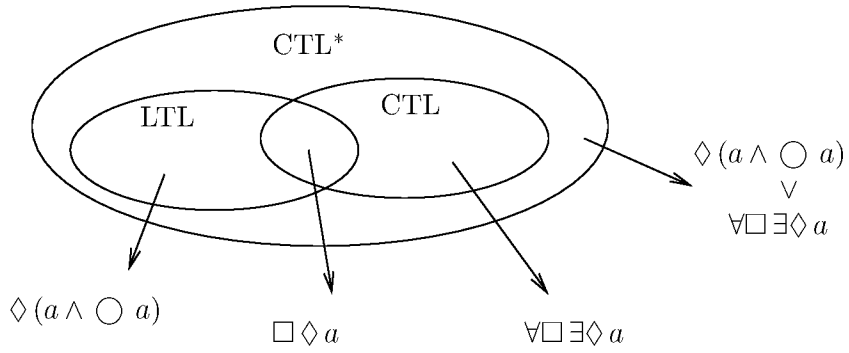


Figure 6.27: Relationship between LTL CTL and CTL*.

One of the consequences of this fact, is that—as in LTL—fairness assumptions can be expressed syntactically in CTL*. For instance, for fairness assumption *fair*, formulae of the form

$$\forall(\mathit{fair} \rightarrow \varphi) \quad \text{or} \quad \exists(\mathit{fair} \ \wedge \ \varphi)$$

are legal CTL* formulae. As for CTL or LTL, semantic equivalence $\equiv$ can be defined for CTL* formulae. Apart from the equivalence laws for CTL and the laws resulting from the equivalence laws for LTL, there exists a series of other important laws that are specific to CTL*. This includes, among others, the laws listed in Figure 6.28.

Example instances of the duality laws for the path quantifiers are

$$\neg\forall\Box\Diamond a \ \equiv \ \exists\Diamond\Box\neg a \quad \text{and} \quad \neg\exists\Box\Diamond a \ \equiv \ \forall\Diamond\Box\neg a.$$

As usual, universal quantification does not distribute over disjunction, and the same applies to existential quantification and conjunction:

$$\forall(\varphi \ \vee \ \psi) \ \not\equiv \ \forall\varphi \ \vee \ \forall\psi \quad \text{and} \quad \exists(\varphi \wedge \psi) \ \not\equiv \ \exists\varphi \wedge \exists\psi.$$

Path quantifier elimination should also be considered with care. For instance,

$$\forall\Diamond\Box\varphi \ \not\equiv \ \forall\Diamond\forall\Box\varphi \quad \text{and} \quad \exists\Box\Diamond\Phi \ \not\equiv \ \exists\Box\exists\Diamond\varphi.$$

Finally, we remark that for CTL*-state formula Φ we have that

$$\exists\Phi \ \equiv \Phi \quad \text{and} \quad \forall\Phi \ \equiv \ \Phi.$$

This is illustrated by means of an example. Consider the CTL* formula $\exists\forall\Diamond a$. This formula holds in state s whenever there exists an infinite path fragment $\pi = s_0\, s_1\, s_2 \ldots \in \mathit{Paths}(s)$, such that $\pi \models \forall\Diamond a$. Since $\pi \models \forall\Diamond a$ holds if and only if $s = s_0 \models \forall\Diamond a$, the formula $\exists\forall\Diamond a$ is equivalent to $\forall\Diamond a$.

duality laws for path quantifiers

$$\begin{aligned} \neg\forall\varphi &\equiv \exists\neg\varphi \\ \neg\exists\varphi &\equiv \forall\neg\varphi \end{aligned}$$

distributive laws

$$\begin{aligned} \forall(\varphi_1 \wedge \varphi_2) &\equiv \forall\varphi_1 \wedge \forall\varphi_2 \\ \exists(\varphi_1 \vee \varphi_2) &\equiv \exists\varphi_1 \vee \exists\varphi_2 \end{aligned}$$

quantifier absorption laws

$$\begin{aligned} \forall\Box\Diamond\varphi &\equiv \forall\Box\,\forall\Diamond\varphi \\ \exists\Diamond\Box\varphi &\equiv \exists\Diamond\,\exists\Box\varphi \end{aligned}$$

Figure 6.28: Some equivalence laws for CTL*.

Remark 6.85. Extending CTL with Boolean Connectors for Path Formulae (CTL$^+$)
Consider the following fragment, called CTL$^+$, of CTL*, which extends CTL by allowing Boolean operators in path formulae. CTL$^+$ state formulae over the set AP of atomic proposition are formed according to the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula. CTL$^+$ path formulae are formed according to the following grammar:

$$\varphi ::= \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\,\Phi \mid \Phi_1\,\mathsf{U}\,\Phi_2$$

where Φ, Φ_1, and Φ_2 are state formulae, and φ_1, φ_2 are path formulae. Surprisingly, CTL$^+$ is as expressive as CTL, i.e., for any CTL$^+$ state formula Φ^+ there exists an equivalent CTL formula Φ. For example:

$$\underbrace{\exists(a\,\mathsf{W}\,b)}_{\text{CTL formula}} \equiv \underbrace{\exists(\ (a\,\mathsf{U}\,b)\ \vee\ \Box a\ )}_{\text{CTL}^+\text{ formula}}$$

or

$$\underbrace{\exists(\Diamond a \wedge \Diamond b)}_{\text{CTL}^+\text{ formula}} \equiv \underbrace{\exists\Diamond(a \wedge \exists\Diamond b)\ \wedge\ \exists\Diamond(b \wedge \exists\Diamond a)}_{\text{CTL formula}}.$$

We do not provide here a full proof for transforming CTL$^+$ formulae into equivalent CTL formulae. The transformation relies on equivalence laws such as

$$\begin{aligned}
\exists(\neg \bigcirc\ \Phi) &\equiv \exists \bigcirc\ \neg\Phi \\
\exists(\neg(\Phi_1 \,\mathsf{U}\, \Phi_2)) &\equiv \exists(\ (\Phi_1 \wedge \neg\Phi_2) \,\mathsf{U}\, (\neg\Phi_1 \wedge \neg\Phi_2)\ ) \ \vee\ \exists\Box\neg\Phi_2 \\
\exists(\bigcirc\, \Phi_1 \wedge \bigcirc\, \Phi_2) &\equiv \exists \bigcirc\ (\Phi_1 \wedge \Phi_2) \\
\exists(\bigcirc\, \Phi \wedge (\Phi_1 \,\mathsf{U}\, \Phi_2)) &\equiv (\Phi_2 \wedge \exists \bigcirc\ \Phi) \ \vee\ (\Phi_1 \wedge \exists \bigcirc\ (\Phi \wedge \exists(\Phi_1 \,\mathsf{U}\, \Phi_2))) \\
\exists((\Phi_1 \,\mathsf{U}\, \Phi_2)\ \wedge\ (\Psi_1 \,\mathsf{U}\, \Psi_2)) &\equiv \exists((\Phi_1 \wedge \Psi_1) \,\mathsf{U}\, (\Phi_2 \wedge \exists(\Psi_1 \,\mathsf{U}\, \Psi_2))) \ \vee \\
&\qquad \exists((\Phi_1 \wedge \Psi_1) \,\mathsf{U}\, (\Psi_2 \wedge \exists(\Phi_1 \,\mathsf{U}\, \Phi_2))) \\
&\ \ \vdots
\end{aligned}$$

Thus, CTL can be expanded by means of a Boolean operator for path formulae without changing the expressiveness. However, CTL$^+$ formulae can be much shorter than the shortest equivalent CTL formulae. ■

6.8.2 CTL* Model Checking

This section treats a model-checking algorithm for CTL*. The CTL* model-checking problem is to establish whether $TS \models \Phi$ holds for a given finite transition system TS (without terminal states) and the CTL* state formula Φ. As we will see, an appropriate combination of the model-checking algorithms for LTL and CTL suffices.

As for CTL, the model-checking algorithm for CTL* is based on a bottom-up traversal of the syntax tree of the formula Φ to be checked. Due to the bottom-up nature of the algorithm, the satisfaction set $Sat(\Psi)$ for any state sub formulae Ψ of Φ has been computed before, and can be used to determine $Sat(\Phi)$. This holds in particular for the maximal proper state subformulae of Φ.

Definition 6.86. Maximal Proper State Subformula

State formula Ψ is a *maximal proper state subformula* of Φ whenever Ψ is a subformula of Φ that differs from Φ and that is not contained in any other proper state subformula of Φ. ■

The basic concept is to replace all maximal proper state subformulae of Φ by fresh atomic propositions $a_1, \ldots, a_k$, say. These propositions do not occur in Φ and are such that $a_i \in L(s)$ if and only if $s \in Sat(\Psi_i)$, the ith maximal state subformula of Φ. For state

subformulae whose "top-level" operator is a Boolean operator (such as negation or conjunction), the treatment is obvious. Let us consider the more interesting case of $\Psi = \exists\varphi$. By replacing all maximal state subformulae in φ, an LTL formula results! Since

$$s \models \exists\varphi \quad \text{iff} \quad \underbrace{s \not\models \forall\neg\varphi}_{\text{CTL}^* \text{ semantics}} \quad \text{iff} \quad \underbrace{s \not\models \neg\varphi}_{\text{LTL semantics}},$$

it suffices to compute the satisfaction set

$$Sat_{LTL}(\neg\varphi) \;=\; \{\, s \in S \mid s \models_{LTL} \neg\varphi \,\}$$

by means of an LTL model checker. (Here, the notations $Sat_{LTL}(\cdot)$ and $\models_{LTL}$ are used to emphasize that the basis is the LTL satisfaction relation.) The satisfaction set for $\Phi = \exists\varphi$ is now obtained by complementation:

$$Sat_{CTL^*}(\exists\varphi) \;=\; S \setminus Sat_{LTL}(\neg\varphi).$$

For CTL* formulae where the outermost operator is an universal quantification we simply may deal with

$$Sat_{CTL^*}(\forall\varphi) \;=\; Sat_{LTL}(\varphi)$$

where, as before, it is assumed that φ is an LTL formula resulting from the replacement of the maximal state subformula with fresh atomic propositions.

The main steps of the CTL* model-checking procedure are presented in Algorithm 27 on page 429.

Example 6.87. Abstract Example of CTL Model Checking*

The CTL* model-checking approach is illustrated by considering the CTL* formula:

$$\exists\varphi \quad \text{where} \quad \varphi \;=\; \bigcirc(\forall\Box\,\exists\Diamond\, a) \;\wedge\; \Diamond\Box\,\exists(\bigcirc a \wedge \Box\, b).$$

The maximal proper state subformulae of φ are

$$\Phi_1 \;=\; \forall\Box\,\exists\Diamond\, a \quad \text{and} \quad \Phi_2 \;=\; \exists(\bigcirc a \wedge \Box\, b).$$

Thus:

$$\varphi \;=\; \bigcirc \underbrace{(\forall\Box\,\exists\Diamond\, a)}_{\Phi_1} \;\wedge\; \Diamond\Box\, \underbrace{\exists(\bigcirc a \wedge \Box\, b)}_{\Phi_2} \;=\; \bigcirc\Phi_1 \;\wedge\; \Diamond\Box\,\Phi_2.$$

According to the model-checking algorithm for CTL*, the satisfaction sets $Sat(\Phi_i)$ are computed recursively. Subsequently, Φ_1 and Φ_2 are replaced with the atomic propositions a_1 and a_2, say. This yields the following LTL formula over the set of propositions $AP' = \{\, a_1, a_2 \,\}$:

$$\varphi' \;=\; \bigcirc a_1 \;\wedge\; \Diamond\Box\, a_2.$$

Algorithm 27 CTL* model checking algorithm (basic idea)

Input: finite transition system TS with initial states I, and CTL* formula Φ
Output: $I \subseteq Sat(\Phi)$

```
for all i ⩽ |Φ| do
  for all Ψ ∈ Sub(Φ) with |Ψ| = i do
    switch(Ψ):
        true    :  Sat(Ψ) := S;
        a       :  Sat(Ψ) := { s ∈ S | a ∈ L(s) };
        a1 ∧ a2 :  Sat(Ψ) := Sat(a1) ∩ Sat(a2);
        ¬a      :  Sat(Ψ) := S \ Sat(a);
        ∃φ      :  determine Sat_LTL(¬φ) by means of an LTL model-checker;
                :  Sat(Ψ) := S \ Sat_LTL(¬φ)
    end switch
    AP := AP ∪ { a_Ψ };                          (* introduce fresh atomic proposition *)
    replace Ψ with a_Ψ
    forall s ∈ Sat(Ψ) do L(s) := L(s) ∪ { a_Ψ }; od
  od
od
return I ⊆ Sat(Φ)
```

The labeling function $L' : S \to 2^{AP'}$ is given by:

$$a_i \in L'(s) \quad \text{if and only if} \quad s \in Sat(\Phi_i) \quad \text{for } i \in \{1, 2\}.$$

Applying the LTL model-checking algorithm to the formula $\neg\varphi'$ yields the set of states satisfying (with respect to the LTL semantics) $\neg\varphi'$, i.e., $Sat_{LTL}(\neg\varphi')$. The complement of $Sat_{LTL}(\neg\varphi')$ provides the set $Sat_{CTL^*}(\varphi)$. ■

Evidently, the time complexity of the CTL* model-checking algorithm is dominated by the LTL model-checking phases. The additional effort that is necessary for CTL* model checking is polynomial in the size of the transition system and the length of the formula. Hence, a time complexity is obtained which is exponential in the length of the formula and linear in the size of the transition system.

Theorem 6.88. Time Complexity of CTL* Model Checking

For transition system TS with N states and K transitions, and CTL* formula Φ, the CTL* model-checking problem $TS \models \Phi$ can be determined in time $\mathcal{O}((N+K)\cdot 2^{|\Phi|})$.

Note that CTL* model checking can be solved by *any* LTL model-checking algorithm. These considerations show that there is a polynomial reduction of the CTL* model-

	CTL	LTL	CTL*
model checking	PTIME	PSPACE-complete	PSPACE-complete
without fairness	$size(TS) \cdot \lvert\Phi\rvert$	$size(TS) \cdot \exp(\lvert\Phi\rvert)$	$size(TS) \cdot \exp(\lvert\Phi\rvert)$
with fairness	$size(TS) \cdot \lvert\Phi\rvert \cdot \lvert fair\rvert$	$size(TS) \cdot \exp(\lvert\Phi\rvert) \cdot \lvert fair\rvert$	$size(TS) \cdot \exp(\lvert\Phi\rvert) \cdot \lvert fair\rvert$
for fixed specifications (model complexity)	$size(TS)$	$size(TS)$	$size(TS)$
satisfiability check	EXPTIME	PSPACE-complete	2EXPTIME
best known technique	$\exp(\lvert\Phi\rvert)$	$\exp(\lvert\Phi\rvert)$	$\exp(\exp(\lvert\Phi\rvert))$

Figure 6.29: Complexity of the model-checking algorithms and satisfiability checking.

checking problem to the LTL model-checking problem. As a result, the theoretical complexity results for LTL also apply to CTL*. Table 6.29 summarizes the complexity results for model checking CTL, CTL*, and LTL.

Theorem 6.89. Theoretical Complexity of CTL* Model Checking

The CTL* model-checking problem is PSPACE*-complete.*

6.9 Summary

- Computation Tree Logic (CTL) is a logic for formalizing properties over computation trees, i.e., the branching structure of the states.
- The expressivenesses of LTL and CTL are incomparable.
- Although fairness constraints cannot be encoded in CTL formulae, fairness assumptions can be incorporated in CTL by adapting the CTL semantics such that quantification is over fair paths, rather than over all paths.
- The CTL model-checking problem can be solved by a recursive descent procedure over the parse tree of the state formula to be checked. The set of states satisfying $\exists(\Phi \,\mathsf{U}\, \Psi)$ can be determined using a smallest fixed-point procedure; for $\exists\Box\,\Phi$ this is a largest fixed-point procedure.

- The time complexity of the CTL model-checking algorithm is linear in the size of the transition system and the length of the formula. In case fairness constraints are considered, an additional multiplicative factor that is proportional to the number of fairness constraints needs to be taken into account.
- Counterexamples and witnesses for CTL path formulae can be determined using a standard graph analysis.
- The CTL model-checking procedure can be realized symbolically by means of ordered binary decision diagrams. These provide a universal and canonical data structure for switching functions.
- Extended Computation Tree Logic (CTL*) is more expressive than either CTL and LTL.
- The CTL* model-checking problem can be solved by an appropriate combination of the recursive descent procedure (as for CTL) and the LTL model-checking algorithm.
- The CTL* model-checking problem is PSPACE-complete.

6.10 Bibliographic Notes

Branching temporal logics. Various types of branching temporal logic have been proposed in the literature. We mention a few important ones in increasing expressive power: Hennessy-Milner logic (HML [197]), Unified System of Branching-Time Logic [42], Computation Tree Logic (CTL [86]), Extended Computation Tree Logic (CTL* [86]), and modal μ-Calculus [243]. The modal μ-calculus is the most expressive among these languages, and HML is the least expressive. CTL has been extended with fairness by Emerson and Halpern [140] and by Emerson and Lei [143].

Not treated in this textbook is the task of proving satisfiability of formulae by means of algorithms or deductive techniques. The satisfiability problems for CTL, CTL*, and other temporal logics have been addressed by many researchers and used, e.g., in the context of synthesis problems where the goal is to design a system model from a given temporal specification. Emerson [138] has shown that checking CTL satisfiability is in the complexity class EXPTIME. This means that the time complexity of checking CTL satisfiability is exponential in the length of the formula. For CTL* this problem is double exponential [141] in the length of the formula. A complete axiomatization of CTL has been given by Ben-Ari, Manna and Pnueli [42] and Emerson and Halpern [139].

Branching vs. linear temporal logics. The discussion of the relative merits of linear- vs. branching-time logics goes back to the early eighties. Pnueli [338] established that linear

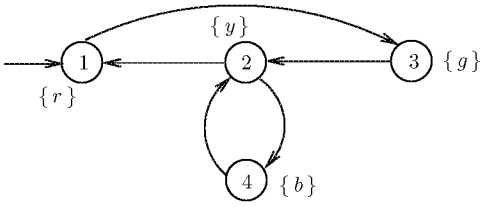
and branching temporal logics are based on two distinct notions of time. Various papers [85, 140, 259] show that the expressivenesses of LTL and CTL are incomparable. A somewhat more practical view on comparing the usefulness of LTL vs. CTL was recently given by Vardi [410]. The logic CTL* that encompasses LTL and CTL was defined by Clarke and Emerson [86].

CTL model checking. The first algorithms for CTL model checking were presented by Clarke and Emerson [86] in 1981 and (for a logic similar to CTL) by Queille and Sifakis [347] in 1982. The algorithm by Clarke and Emerson was polynomial in both the size of the transition system and the length of the formula, and could handle fairness. Clarke, Emerson, and Sistla [87] presented an efficiency improvement using the detection of strongly connected components and backward breadth-first search, yielding an algorithm that is linear in both the size of the system and the length of the formula. CTL model checking based on a forward search has been proposed by Iwashita, Nakata, and Hirose [224]. Emerson and Lei [143] showed that CTL* can be checked with essentially the same complexity as LTL, using a combination of the algorithms for LTL and CTL. The same authors consider in [142] CTL model checking under a broad class of fairness assumptions. Practical aspects of CTL* model checking have been reported by Bhat, Cleaveland, and Grumberg [50], and more recently by Visser and Barringer [414]. Algorithms for generating counterexamples and witnesses originate from the works by Clarke et al. [91] and Hojati, Brayton, and Kurshan [204]. More recent developments are the use of satisfiability solvers for propositional logic or quantified Boolean formulae to find counterexamples up to certain length, as proposed by Clarke et al. [84], and the use of tree-like counterexamples as opposed to linear ones by Clarke [93].

CTL model checkers. Clarke and Emerson [86] reported the first (fair) CTL model checker, called EMC. About the same time, Queille and Sifakis [347] announced CESAR, a model checker for a branching logic very similar to CTL. EMC was improved in [87] and constituted the basis for SMV (Symbolic Model Verifier), an efficient CTL model checker by McMillan based on a symbolic OBDD-based representation of the state space [288]. The concept of ordered binary decision diagrams has been proposed by Bryant [70]. The milestone for symbolic model checking with OBDDs is the paper by Burch et al.[74]. Further references for OBDD-based approaches have been given in Section 6.7. Recent variants of SMV are NuSMV [83] developed by the teams of Cimatti *et al.*, and SMV by McMillan and colleagues at Cadence Berkeley Laboratories that is focused on compositionality. Both tools are freely available. Another symbolic CTL model checker is VIS [62].

6.11 Exercises

EXERCISE 6.1. Consider the following transition system over $AP = \{ b, g, r, y \}$:

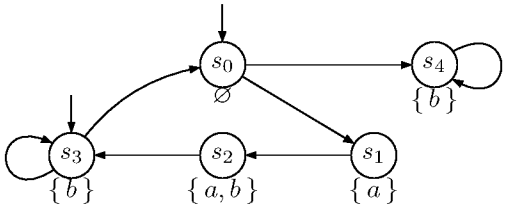


The following atomic propositions are used: r (red), y (yellow), g (green), and b (black). The model is intended to describe a traffic light that is able to blink yellow. You are requested to indicate for each of the following CTL formulae the set of states for which these formulae hold:

(a) $\forall \Diamond\, y$
(b) $\forall \Box\, y$
(c) $\forall \Box\, \forall \Diamond\, y$
(d) $\forall \Diamond\, g$
(e) $\exists \Diamond\, g$
(f) $\exists \Box\, g$
(g) $\exists \Box\, \neg\, g$
(h) $\forall (b \,\mathsf{U}\, \neg\, b)$
(i) $\exists (b \,\mathsf{U}\, \neg\, b)$
(j) $\forall (\neg\, b \,\mathsf{U}\, \exists \Diamond\, b)$
(k) $\forall (g \,\mathsf{U}\, \forall (y \,\mathsf{U}\, r))$
(l) $\forall (\neg\, b \,\mathsf{U}\, b)$

EXERCISE 6.2. Consider the following CTL formulae and the transition system TS outlined on the right:

$$\Phi_1 = \forall (a \,\mathsf{U}\, b) \vee \exists \bigcirc (\forall \Box\, b)$$
$$\Phi_2 = \forall \Box\, \forall (a \,\mathsf{U}\, b)$$
$$\Phi_3 = (a \wedge b) \rightarrow \exists \Box\, \exists \bigcirc\, \forall (b \,\mathsf{W}\, a)$$
$$\Phi_4 = (\forall \Box\, \exists \Diamond\, \Phi_3)$$



Determine the satisfaction sets $Sat(\Phi_i)$ and decide whether $TS \models \Phi_i$ $(1 \leqslant i \leqslant 4)$.

EXERCISE 6.3. Which of the following assertions are correct? Provide a proof or a counterexample.

(a) If $s \models \exists \Box\, a$, then $s \models \forall \Box\, a$.
(b) If $s \models \forall \Box\, a$, then $s \models \exists \Box\, a$.
(c) If $s \models \forall \Diamond\, a \;\vee\; \forall \Diamond\, b$, then $s \models \forall \Diamond\, (a \;\vee\; b)$.
(d) If $s \models \forall \Diamond\, (a \;\vee\; b)$, then $s \models \forall \Diamond\, a \;\vee\; \forall \Diamond\, b$.

(e) If $s \models \forall(a \,\mathsf{U}\, b)$, then $s \models \neg\,(\exists(\neg b \,\mathsf{U}\, (\neg a \;\wedge\; \neg b)) \;\vee\; \exists\Box\, \neg b)$.

EXERCISE 6.4. Let Φ and Ψ be arbitrary CTL formulae. Which of the following equivalences for CTL formulae are correct?

(a) $\forall\bigcirc\, \forall\Diamond\, \Phi \;\equiv\; \forall\Diamond\, \forall\bigcirc\, \Phi$

(b) $\exists\bigcirc\, \exists\Diamond\, \Phi \;\equiv\; \exists\Diamond\, \exists\bigcirc\, \Phi$

(c) $\forall\bigcirc\, \forall\Box\, \Phi \;\equiv\; \forall\Box\, \forall\bigcirc\, \Phi$

(d) $\exists\bigcirc\, \exists\Box\, \Phi \;\equiv\; \exists\Box\, \exists\bigcirc\, \Phi$

(e) $\exists\Diamond\, \exists\Box\, \Phi \;\equiv\; \exists\Box\, \exists\Diamond\, \Phi$

(f) $\forall\Box\, (\Phi \;\Rightarrow\; (\neg\Psi \;\wedge\; \exists\bigcirc\, \Phi)) \;\equiv\; (\Phi \;\Rightarrow\; \neg\forall\Diamond\, \Psi)$

(g) $\forall\Box\, (\Phi \;\Rightarrow\; \Psi) \;\equiv\; (\exists\bigcirc\, \Phi \;\Rightarrow\; \exists\bigcirc\, \Psi)$

(h) $\neg\forall(\Phi \;\mathsf{U}\; \Psi) \;\equiv\; \exists(\Phi \;\mathsf{U}\; \neg\Psi)$

(i) $\exists((\Phi \wedge \Psi) \;\mathsf{U}\; (\neg\Phi \wedge \Psi)) \;\equiv\; \exists(\Phi \;\mathsf{U}\; (\neg\Phi \wedge \Psi))$

(j) $\forall(\Phi \;\mathsf{W}\; \Psi) \equiv \neg\exists(\neg\Phi \;\mathsf{W}\; \neg\Psi)$

(k) $\exists(\Phi \;\mathsf{U}\; \Psi) \equiv \exists(\Phi \;\mathsf{U}\; \Psi) \;\wedge\; \exists\Diamond\Psi$

(l) $\exists(\Psi \;\mathsf{W}\; \neg\Psi) \vee\; \forall(\Psi \;\mathsf{U}\; \text{false}) \equiv \exists\bigcirc\; \Phi \;\vee\; \forall\bigcirc\; \neg\Phi$

(m) $\forall\Box\Phi \;\wedge\; (\neg\Phi \;\vee\; \exists\bigcirc\; \exists\Diamond\neg\Phi) \equiv \exists X\neg\Phi \;\wedge\; \forall\bigcirc\; \Phi$

(n) $\forall\Box\forall\Diamond\Phi \equiv \Phi \;\wedge\; (\forall\bigcirc\; \forall\Box\forall\Diamond\Phi) \;\vee\; \forall\bigcirc\; (\forall\Diamond\Phi \;\wedge\; \forall\Box\forall\Diamond\Phi)$

(o) $\forall\Box\Phi \equiv \Phi \;\vee\; \forall\bigcirc\; \forall\Box\Phi$

EXERCISE 6.5. Consider an elevator system that services $N > 0$ floors numbered 0 through $N-1$. There is an elevator door at each floor with a call button and an indicator light that signals whether or not the elevator has been called. In the elevator cabin there are N send buttons (one per floor) and N indicator lights that inform to which floor(s) is going to be sent. For simplicity consider $N = 4$. Present a set of atomic propositions—try to minimize the number of propositions—that are needed to describe the following properties of the elevator system as CTL formulae and give the corresponding CTL formulae:

(a) The doors are "safe", i.e., a floor door is never open if the cabin is not present at the given floor.

(b) The indicator lights correctly reflect the current requests. That is, each time a button is pressed, there is a corresponding request that needs to be memorized until fulfillment (if ever).

(c) The elevator only services the requested floors and does not move when there is no request.

(d) All requests are eventually satisfied.

EXERCISE 6.6. Consider the single pulser circuit, a hardware circuit that is part of a set of benchmark circuits for hardware verification. The single pulser has the following informal specification: "For every pulse at the input *inp* there appears exactly one pulse of length 1 at output *outp*, independent of the length of the input pulse". Thus, the single pulser circuit is required to generate an output pulse between two rising edges of the input signal. The following questions require the formulation of the circuit in terms of CTL. Suppose we have the proposition *rise_edge* at our disposal which is true if the input was low (0) at time instant $n-1$ and high (1) at time instant n (for natural $n > 0$). It is assumed that input sequences of the circuit are well behaved, i.e., more that the rising edge appears in the input sequence.
Questions: specify the following requirements of the circuit in CTL:

(a) A rising edge at the inputs leads to an output pulse.

(b) There is at most one output pulse for each rising edge.

(c) There is at most one rising edge for each output pulse.

(This exercise is taken from [246].)

EXERCISE 6.7. Transform the following CTL formulae into ENF and PNF. Show all intermediate steps.

$$\Phi_1 \;=\; \forall\big(\,(\neg a)\,\mathsf{W}\,(b \rightarrow \forall\bigcirc\, c)\,\big)$$

$$\Phi_2 \;=\; \forall\bigcirc\,\big(\;\exists((\neg a)\,\mathsf{U}\,(b \wedge \neg c)) \;\;\vee\;\; \exists\Box\,\forall\bigcirc\, a\,\big)$$

EXERCISE 6.8. Provide two finite transition systems TS_1 and TS_2 (without terminal states, and over the same set of atomic propositions) and a CTL formula Φ such that $\mathit{Traces}(TS_1) = \mathit{Traces}(TS_2)$ and $TS_1 \models \Phi$, but $TS_2 \not\models \Phi$.

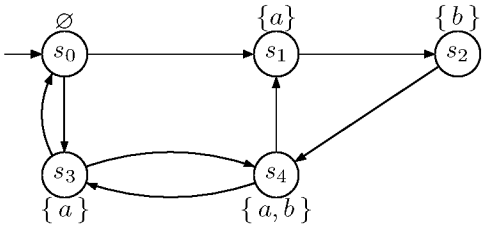
EXERCISE 6.9. Consider the CTL formula

$$\Phi = \forall\Box\,\big(a \rightarrow \forall\Diamond\,(b \wedge \neg a)\big)$$

and the following CTL fairness assumption:

$$\mathit{fair} = \forall\Diamond\,\forall\bigcirc\,(a \wedge \neg b) \rightarrow \forall\Diamond\,\forall\bigcirc\,(b \wedge \neg a) \wedge \Diamond\,\Box\,\exists\Diamond\, b \rightarrow \Box\,\Diamond\, b.$$

Prove that $TS \models_{fair} \Phi$ where transition system TS is depicted below.



EXERCISE 6.10. Let $\wp = (z_1, z_2, z_3, z_4, z_5, z_6)$. Depict the $\wp$-ROBDD for the majority function

$$MAJ([z_1 = b_1, z_2 = b_2, \ldots, z_6 = b_6]) \;=\; \begin{cases} 1 & \text{if } b_1 + b_2 + \ldots + b_6 \geqslant 4 \\ 0 & \text{otherwise.} \end{cases}$$

EXERCISE 6.11. Consider the function

$$F(x_0, \ldots, x_{n-1}, a_0, \ldots, a_{k-1}) = x_m$$

where $n = 2^k$, and $m = \sum_{j=0}^{k-1} a_j 2^j$. Let k=3. Questions:

(a) Depict the $\wp$-ROBDD for $\wp = (a_0, \ldots, a_{k-1}, x_0, \ldots, x_{n-1})$.

(b) Depict the $\wp$-ROBDD for $\wp = (a_0, x_0, \ldots, a_{k-1}, x_{k-1}, x_k, \ldots, x_{n-1})$.

EXERCISE 6.12. Let $\mathfrak{B}$ and $\mathfrak{C}$ be two $\wp$-ROBDDs. Design an algorithm that checks whether $f_{\mathfrak{B}} = f_{\mathfrak{C}}$ and runs in time linear in the sizes of $\mathfrak{B}$ and $\mathfrak{C}$.

(*Hint: It is assumed that* $\mathfrak{B}$ *and* $\mathfrak{C}$ *are given as separate graphs (and not by nodes of a shared OBDD).*)

EXERCISE 6.13. Let TS be a finite transition system (over AP) without terminal states, and Φ and Ψ be CTL state formulae (over AP). Prove or disprove

$$TS \models \exists(\Phi \,\mathsf{U}\, \Psi) \quad \text{if and only if} \quad TS' \models \exists\Diamond\, \Psi$$

where TS' is obtained from TS by eliminating all outgoing transitions from states s such that $s \models \Psi \;\vee\; \neg\Phi$.

EXERCISE 6.14. Check for each of the following formula pairs (Φ_i, φ_i) whether the CTL formula Φ_i is equivalent to the LTL formula φ_i. Prove the equivalence or provide a counterexample that illustrates why $\Phi_i \not\equiv \varphi_i$.

(a) $\Phi_1 = \forall\Box\forall\bigcirc a.$ and $\varphi_1 = \Box\bigcirc a$

(b) $\Phi_2 = \forall\Diamond\forall\bigcirc a$ and $\varphi_2 = \Diamond\bigcirc a.$

(c) $\Phi_3 = \forall\Diamond(a \wedge \exists\bigcirc a)$ and $\varphi_3 = \Diamond(a \wedge \bigcirc a).$

(d) $\Phi_4 = \forall\Diamond a \vee \forall\Diamond b$ and $\varphi_4 = \Diamond(a \vee b).$

(e) $\Phi_5 = \forall\Box(a \rightarrow \forall\Diamond b)$ and $\varphi_5 = \Box(a \rightarrow \Diamond b).$

(f) $\Phi_6 = \forall(b \,\mathsf{U}\, (a \wedge \forall\Box b))$ and $\varphi_6 = \Diamond a \wedge \Box b.$

EXERCISE 6.15.

(a) Prove, using Theorem 6.18, that there does not exist an equivalent LTL formula for the CTL formula $\Phi_1 = \forall\Diamond\,(a \wedge \exists\bigcirc a)$.

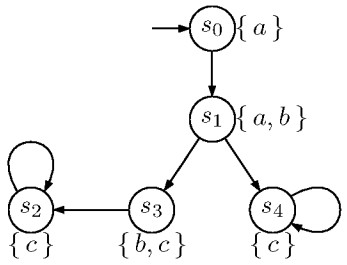
(b) Now prove directly (i.e. without Theorem 6.18), that there does not exist an equivalent LTL formula for the CTL formula $\Phi_2 = \forall\Diamond\,\exists\bigcirc\forall\Diamond\,\neg a$. (*Hint: Argument by contraposition.*)

EXERCISE 6.16.

Consider the following CTL formulae

$$\Phi_1 = \exists\Diamond\forall\Box\, c \quad \text{and} \quad \Phi_2 = \forall(a \,\mathsf{U}\, \forall\Diamond\, c)$$

and the transition system TS outlined on the right. Decide whether $TS \models \Phi_i$ for $i = 1, 2$ using the CTL model-checking algorithm. Sketch its main steps.



EXERCISE 6.17. Provide an algorithm in pseudo code to compute $Sat(\forall(\Phi \,\mathsf{U}\, \Psi))$ in a direct manner, i.e., without transforming the formula into ENF.

EXERCISE 6.18.

(a) Prove that $Sat(\exists(\Phi \,\mathsf{W}\, \Psi))$ is the largest set T such that

$$T \subseteq Sat(\Psi) \cup \big\{s \in Sat(\Phi) \mid Post(s) \cap T \neq \varnothing\big\}.$$

(b) Prove that $Sat(\forall(\Phi \,\mathsf{W}\, \Psi))$ is the largest set T such that

$$T \subseteq Sat(\Psi) \cup \big\{s \in Sat(\Phi) \mid Post(s) \subseteq T\big\}.$$

Use the above characterizations to provide efficient algorithms for computing the sets $Sat(\exists(\Phi \,\mathsf{W}\, \Psi))$ and $Sat(\forall(\Phi \,\mathsf{W}\, \Psi))$ in a direct manner.

EXERCISE 6.19. Consider the fragment ECTL of CTL which consists of formulae built according to the following grammar:

$$\begin{aligned} \Phi &::= a \mid \neg a \mid \Phi \wedge \Phi \mid \exists \varphi \\ \varphi &::= \bigcirc \Phi \mid \Box \Phi \mid \Phi \,\mathsf{U}\, \Phi \end{aligned}$$

For two transition systems $TS_1 = (S_1, Act, \rightarrow_1, I_1, AP, L_1)$ and $TS_2 = (S_2, Act, \rightarrow_2, I_2, AP, L_2)$, let $TS_1 \subseteq TS_2$ iff $S_1 \subseteq S_2$, $\rightarrow_1 \subseteq \rightarrow_2$, $I_1 = I_2$ and $L_1(s) = L_2(s)$ for all $s \in S_1$.

(a) Prove that for all ECTL formulae Φ and all transition systems TS_1, TS_2 with $TS_1 \subseteq TS_2$, it holds:

$$TS_1 \models \Phi \quad \Longrightarrow \quad TS_2 \models \Phi.$$

(b) Give a CTL formula which is not equivalent to any other ECTL formula. Justify your answer.

EXERCISE 6.20. In CTL, the release operator is defined by

$$\exists(\Phi \,\mathsf{R}\, \Psi) = \neg\forall((\neg\Phi) \,\mathsf{U}\, (\neg\Psi)) \quad \text{and} \quad \forall(\Phi \,\mathsf{R}\, \Psi) = \neg\exists((\neg\Phi) \,\mathsf{U}\, (\neg\Psi)).$$

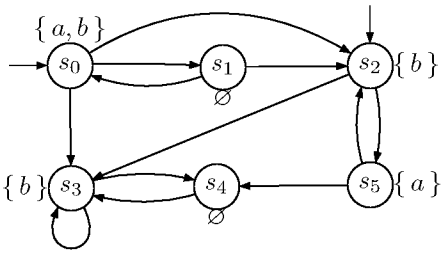
(a) Provide expansion laws for $\exists(\Phi \,\mathsf{R}\, \Psi)$ and $\forall(\Phi \,\mathsf{R}\, \Psi)$.

(b) Give a pseudo code algorithm for computing $Sat(\exists(\Phi \,\mathsf{R}\, \Psi))$ and do the same for computing $Sat(\forall(\Phi \,\mathsf{R}\, \Psi))$.

EXERCISE 6.21. Consider the CTL formula Φ and the strong fairness assumption *sfair*:

$$\begin{aligned} \Phi &= \forall\Box\forall\Diamond\, a \\ \mathit{sfair} &= \Box\Diamond \underbrace{(b \wedge \neg a)}_{\Phi_1} \rightarrow \Box\Diamond \underbrace{\exists\,(b \,\mathsf{U}\, (a \wedge \neg b))}_{\Psi_1} \end{aligned}$$

and transition system TS over $AP = \{\, a, b \,\}$ which is given by



Questions:

(a) Determine $Sat(\Phi_1)$ and $Sat(\Psi_1)$ (without fairness).

(b) Determine $Sat_{sfair}(\exists\Box \text{ true})$.

(c) Determine $Sat_{sfair}(\Phi)$.

EXERCISE 6.22. Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, $a, b \in AP$ and $s \in S$. Furthermore, let *fair* be a CTL fairness assumption and $a_{fair} \in AP$ an atomic proposition with $a_{fair} \in L(s)$ iff $s \models_{fair} \exists\Box\text{true}$.

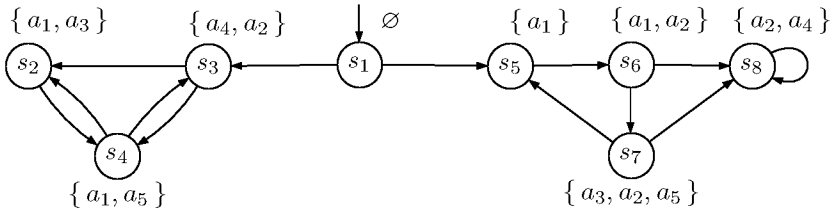
Which of the following assertions are correct? Give a proof or counterexample.

(a) $s \models_{fair} \forall(a \,\mathsf{U}\, b) \quad \text{iff} \quad s \models \forall(a \,\mathsf{U}\, (b \wedge a_{fair}))$.

(b) $s \models_{fair} \exists(a \,\mathsf{W}\, b) \quad \text{iff} \quad s \models \exists(a \,\mathsf{W}\, (b \wedge a_{fair}))$.

(c) $s \models_{fair} \forall(a \,\mathsf{W}\, b) \quad \text{iff} \quad s \models \forall(a \,\mathsf{W}\, (a_{fair} \rightarrow b))$.

EXERCISE 6.23. Consider the following transition system TS over $AP = \{ a_1, \ldots, a_6 \}$.



Let $\Phi = \exists \bigcirc \big(a_1 \rightarrow \exists(a_1 \,\mathsf{U}\, a_2)\big)$ and $\mathit{sfair} = \mathit{sfair}_1 \wedge \mathit{sfair}_2 \wedge \mathit{sfair}_3$ a strong CTL fairness assumption where

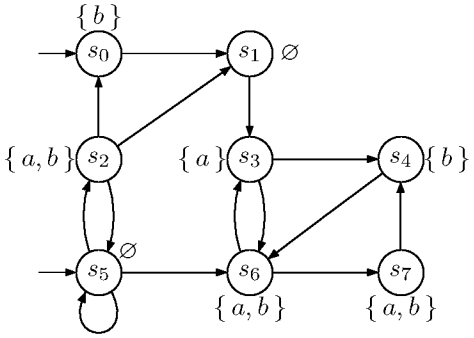
$$\begin{aligned} \mathit{sfair}_1 &= \Box\Diamond\, \forall\Diamond(a_1 \vee a_3) \longrightarrow \Box\Diamond a_4 \\ \mathit{sfair}_2 &= \Box\Diamond(a_3 \wedge \neg a_4) \longrightarrow \Box\Diamond a_5 \\ \mathit{sfair}_3 &= \Box\Diamond(a_2 \wedge a_5) \longrightarrow \Box\Diamond a_6 \end{aligned}$$

Sketch the main steps for computing the satisfaction sets $\mathit{Sat}_{\mathit{sfair}}(\exists\Box\mathrm{true})$ and $\mathit{Sat}_{\mathit{sfair}}(\Phi)$.

Exercise 6.24. Consider the CTL* formula over $AP = \{a, b\}$:

$$\Phi = \forall\Diamond\Box\exists\bigcirc\,(a \,\mathsf{U}\, \exists\Box b)$$

and the transition system TS outlined below:



Apply the CTL* model-checking algorithm to compute $\mathit{Sat}(\Phi)$ and decide whether $TS \models \Phi$. (*Hint: You may infer the satisfaction sets for LTL formulae directly.*)

Exercise 6.25. The model-checking algorithm presented in Section 6.5 treats CTL with strong fairness assumptions. Explain which modifications are necessary to deal with weak CTL fairness assumptions:

$$\mathit{wfair} = \bigwedge_{1 \leqslant i \leqslant k} \big(\, \Diamond\Box a_i \longrightarrow \Box\Diamond b_i \,\big).$$

You may assume that $a_i, b_i \in \{\mathrm{true}\} \cup AP$.

Exercise 6.26. Which of the following equivalences for CTL* are correct? Provide a proof or a counterexample.

(a) $\forall \bigcirc \forall \Box \Phi \equiv \forall \bigcirc \Box \Phi$

(b) $\exists \bigcirc \exists \Box \Phi \equiv \exists \bigcirc \Box \Phi$

(c) $\forall(\varphi \wedge \psi) \equiv \forall\varphi \wedge \forall\psi$

(d) $\exists(\varphi \wedge \psi) \equiv \exists\varphi \wedge \exists\psi$

(e) $\neg\forall(\varphi \rightarrow \psi) \equiv \exists(\varphi \wedge \neg\psi)$

(f) $\exists\Box\exists \bigcirc \Phi \wedge \neg\forall \bigcirc \neg\Phi \equiv \exists\Box(\neg \bigcirc \neg\Phi)$

(g) $\forall(\Diamond\Psi \wedge \Box\Phi) \equiv \forall\Diamond(\Psi \wedge \forall\Box\Phi) \wedge \forall\Box(\Phi \wedge \forall\Diamond\Psi)$

(h) $\exists(\Diamond\Psi \wedge \Box\Phi) \equiv \exists\Diamond(\Psi \wedge \exists\Box\Phi)$

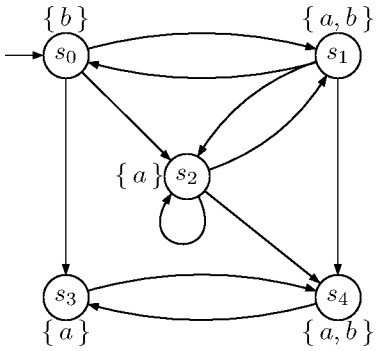
Here, Φ, Ψ are arbitrary CTL* state formulae and ψ, φ are CTL* path formulae.

Exercise 6.27.

Consider the transition system TS and the CTL* formula

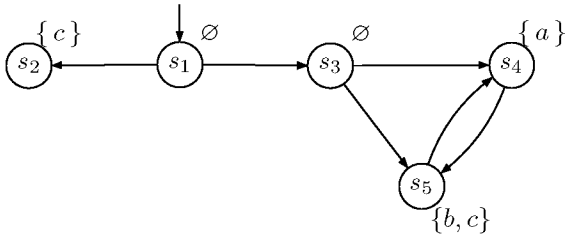
$$\Phi = \exists\big(\bigcirc (a \wedge \neg b) \wedge \bigcirc \forall (b \,\mathsf{U}\, \Box\, a)\big).$$

Apply the CTL* model-checking algorithm to check whether $TS \models \Phi$ and sketch its main steps as well as its output. (*Hint: You may compute the LTL satisfaction sets directly.*)



Exercise 6.28. Consider the transition system TS depicted below and the CTL* formula

$$\Phi \;=\; \exists\,(\Box\Diamond b \;\wedge\; \Box\exists\Diamond \bigcirc\; a) \;\wedge\; \forall\Box\Diamond \bigcirc\; c.$$



Sketch the main steps that the CTL* model checking algorithm performs for checking whether $TS \models \Phi$.

EXERCISE 6.29. Provide an example of a CTL* formula which is not a CTL$^+$ formula, but there exists an equivalent CTL formula.

EXERCISE 6.30. Provide equivalent CTL formulae for the CTL$^+$ formulae $\forall(\Diamond a \wedge \Box b)$ and $\forall(\bigcirc a \wedge \neg(a \,\mathsf{U}\, (\Box b)))$.

Practical Exercises

The remaining exercises are modeling and veriifcation exercises using the CTL model checker NuSMV [83].

EXERCISE 6.31. The following program is a mutual exclusion protocol for two processes due to Pnueli (taken from [118]). There is a single shared variable s which is either 0 or 1, and initially equals 1. Besides, each process has a local Boolean variable y that initially equals 0. The program text for process P_i $(i = 0, 1)$ is as follows:

l0: **loop forever do**
  begin
  l1: Noncritical section
  l2: $(y_i, s) := (1, i)$;
  l3: **wait until** $((y_{1-i} = 0) \vee (s \neq i))$;
  l4: Critical section
  l5: $y_i := 0$
end.

Here, the statement $(y_i, s) := (1, i)$ is a *multiple assignment* in which variable $y_i := 1$ and $s := i$ is a single, atomic step.

The intuition behind this protocol is as follows. The variables y_0 and y_1 are used by each process to signal the other process of active interest in entering the critical section. On leaving the noncritical section, process P_i sets its own local variable y_i to 1. In a similar way this variable is reset to 0 once the critical section is left. The global variable s is used to resolve a tie situation between the processes. It serves as a logbook in which each process that sets its y variable to 1 signs at the same time. The test at line l3 says that P_0 may enter its critical section if either y_1 equals 0 – implying that its competitor is not interested in entering its critical section – or if s differs from 0 – implying that the competitor process P_1 performed its assignment to y_1 after p_0 assigned 1 to y_0.

Questions concerning this mutual exclusion protocol:

(a) Model this protocol in NuSMV; formulate the property of mutual exclusion in CTL and check this property.

(b) Check whether Pnueli's protocol ensures absence of unbounded overtaking, i.e., when a process wants to enter its critical section, it eventually will be able to do so. Provide a counterexample (and an explanation thereof) in case this property is violated.

(c) Add the fairness constraint `FAIRNESS running` to the process specification in your NuSMV model of the mutual exclusion program, and check again the property of absence of unbounded overtaking. Compare the obtained results with the results obtained in the previous question without using the fairness constraint.

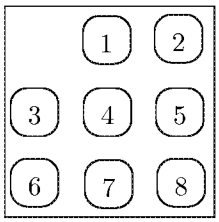
(d) Express in CTL that each process will occupy its critical section infinitely often. Check the property (use again the `FAIRNESS running`).

(e) A practical problem with this mutual exclusion protocol is that it is too demanding in the sense that it enforces performing the assignments to y_i and s (in line l2) in a single step. Most existing hardware systems cannot perform such assignments in one step. Therefore, it is requested to investigate whether any of the four possible realizations of this protocol – in which the aforementioned assignments are not atomic anymore – is a correct mutual exclusion protocol.

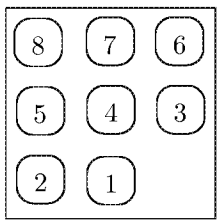
(I) Report for each possible implementation your results, including possible counterexamples and their explanation.

(II) Compare your results with the results of your PROMELA experiments with this exercise in the previous exercise series.

EXERCISE 6.32. In this exercise you are confronted with a nonstandard example for model checking. The purpose of this exercise is to present the model checker as a solver for combinatorial problems rather than a tool for correctness analysis. These problems involve a search (involving backtracking) of optimal or cost-minimizing strategies such as schedulers or puzzle solutions. The exercise is concerned with Loyd's puzzle that consists of an $N \cdot K$ grid in which there are $N \cdot K-1$ numbered tiles and a single blank space. The goal of the puzzle is to achieve a predetermined order on the tiles. The initial and final configuration of the tiles for $N = 3$ and $K = 3$ is as follows:



initial configuration



final configuration

Note that there are approximately $4 \cdot (N \cdot K)!$ possible moves in this puzzle. For $N = 3$ and $K = 3$ this already amounts to about $1.45\ 10^6$ possible configurations.

Questions concerning Loyd's puzzle:

(a) Complete the (partial) model of Loyd's puzzle in NuSMV that is given below. In this model, there is an array `h` that keeps track of the horizontal positions of the tiles and an array `v` that records the vertical positions of the tiles such that the position of tile `i` is given by the pair `h[i], v[i]`. Tile `0` represents the blank tile. Position `h[i] = 1` and `v[i] = 1` is the lowest left corner of the puzzle.

```
MODULE main

DEFINE N := 3; K := 3;

VAR move: {u, d, l, r};     -- the possible tile-moves
    h: array 0..8 of 1..3;  -- the horizontal positions of all tiles
    v: array 0..8 of 1..3;  -- .... and their vertical positions

ASSIGN -- the initial horizontal and vertical positions of all tiles

init(h[0]) := 1;  init(v[0]) := 3;
init(h[1]) := 2;  init(v[1]) := 3;
init(h[2]) := 3;  init(v[2]) := 3;
init(h[3]) := 1;  init(v[3]) := 2;
init(h[4]) := 2;  init(v[4]) := 2;
init(h[5]) := 3;  init(v[5]) := 2;
init(h[6]) := 1;  init(v[6]) := 1;
init(h[7]) := 2;  init(v[7]) := 1;
init(h[8]) := 3;  init(v[8]) := 1;

ASSIGN

-- determine the next positions of the blank tile

next(h[0]) :=      -- horizontal position of the blank tile
       case
                              -- one position right
                              -- one position left
              1 : h[0];       -- keep the same horizontal position

       esac;

next(v[0]) :=      -- vertical position of the blank tile
       case
                              -- one position down
                              -- one position up
              1 : v[0];       -- keep the same vertical position
       esac;

-- determine the next positions of all non-blank tiles

next(h[1]) :=      -- horizontal position of tile 1
       case


       esac;
```

```
next(v[1]) :=      -- vertical position of tile 1
       case


       esac;

-- and similar for all remaining tiles
```

A possible way to proceed is as follows:

(i) First, consider the possible moves of the blank tile (i.e., the blank space). Notice that the blank space cannot be moved to the left in all positions. The same applies to moves upward, downward and to the right.

(ii) Then try to find the possible moves of tile `[1]`. The code for tiles `[2]` through `[8]` are obtained by simply copying the code for tile `[1]` while replacing all references to `[1]` with references of the appropriate tile number.

(iii) Test the possible moves by running a simulation.

(b) Define an atomic proposition `goal` that describes the desired goal configuration of the puzzle. Add this definition to your NuSMV specification by incorporating the following line(s) in your NuSMV model:

```
DEFINE goal :=   ..............  ;
```

where the dotted lines contain your description of the goal configuration.

(c) Find a solution to the puzzle by imposing the appropriate CTL formula to the NuSMV specification, and running the model checker on this formula.

This exercise has been taken from [95].

EXERCISE 6.33. Consider the mutual exclusion algorithm by the Dutch mathematician Dekker. There are two processes P_1 and P_2, two Boolean-valued variables b_1 and b_2 whose initial values are false, and a variable k which may take the values 1 and 2 and whose initial value is arbitrary. The ith process (i=1, 2) may be described as follows, where j is the index of the other process:

```
while true do
begin b_i := true;
      while b_j do
            if k = j then begin
                          b_i := false;
                          while k = j do skip;
                          b_i := true
                          end;
      ⟨ critical section ⟩;
      k := j;
      b_i := false
end
```

Questions:

(a) Model Dekker's algorithm in NuSMV.

(b) Verify whether this algorithm satisfies the following properties:

(c) Mutual exclusion: two processes cannot be in their critical section at the same time.

(d) Absence of individual starvation: if a process wants to enter its critical section, it is eventually able to do so.

(*Hint: use the* `FAIRNESS running` *statement in your NuSMV specification for proving the latter property in order to prohibit unfair executions that might trivially violate these requirements.*)

EXERCISE 6.34. In the original mutual exclusion protocol by Dijkstra in 1965 (another Dutch mathematician), it is assumed that there are $n \geqslant 2$ processes, and global variables b, c : **array** $[1 \ldots n]$ of **Boolean** and an integer k. Initially all elements of b and of c have the value true and the value of k belongs to $1, 2, \ldots, n$. The ith process may be represented as follows:

```
var j : integer;
while true do
begin b[i] := false;
      L_i : if k ≠ i then begin c[i] := true;
                                if b[k] then k := i;
                                goto L_i
                          end;
            else begin c[i] := false;
                       for j := 1 to n do
                       if (j ≠ i ∧ ¬(c[j])) then goto L_i
                 end
      ⟨ critical section ⟩;
      c[i] := true;
      b[i] := true
end
```

Questions:

(a) Model this algorithm in NuSMV.

(b) Check the mutual exclusion property (at most one process can be in its critical section at any point in time) in two different ways: by means of a CTL formula using `SPEC` and by using invariants. Try to check this property for n=2 through n=5 by increasing the number of processes gradually and compare the sizes of the state spaces and the runtime needed for the two ways of verifying the mutual exclusion property.

(c) Check the absence of individual starvation property: if a process wants to enter its critical section, it is eventually able to do so.

EXERCISE 6.35. In order to find a fair solution for N processes, Peterson proposed in 1981 the following protocol. Let $Q[1 \ldots N]$ (Q for queue) and $T[1 \ldots N]$ (T for turn), be two shared arrays which are initially 0 and 1, respectively. The variables i and j are local to the process with i containing the process number. The code of process i is as follows:

while true **do**
for $j := 1$ **to** $N - 1$ **do**
begin
    $Q[i] := j$;
    $T[j] := i$;
    wait until $(T[j] \neq i \ \vee \ (\textbf{forall}\ k \neq i . Q[k] < j))$
end;
$\langle$ critical section $\rangle$;
$Q[i] := 0$
end

Questions:

(a) Model Peterson's algorithm in NuSMV.

(b) Verify whether this algorithm satisfies the following properties:

  (i) Mutual exclusion.

  (ii) Absence of individual starvation.

Chapter 7

Equivalences and Abstraction

Transition systems can model a piece of software or hardware at various abstraction levels. The lower the abstraction level, the more implementation details are present. At high abstraction levels, such details are deliberately left unspecified. Binary relations between states (henceforth implementation relations) are useful to relate or to compare transition systems, possibly at different abstraction levels. When two models are related, one model is said to be refined by the other, or, reversely, the second is said to be an *abstraction* of the first. If the implementation relation is an equivalence, then it identifies all transition systems that cannot be distinguished; such models fulfill the same observable properties at the relevant abstraction level.

Implementation relations are predominantly used for comparing two models of the same system. Given a transition system TS that acts as an abstract system specification, and a more detailed system model TS', implementation relations allow checking whether TS' is a correct implementation (or: refinement) of TS. Alternatively, for system analysis purposes, implementation relations provide ample means to *abstract* from certain system details, preferably details that are irrelevant for the analysis of the property, φ say, at hand. In this way, a transition system TS' comprising very many, maybe even infinitely many, states can be abstracted by a smaller model TS. Provided the abstraction preserves the property to be checked, the analysis of the (hopefully small) abstract model TS suffices to decide the satisfaction of the properties in TS'. Formally, from $TS \models \varphi$ we may safely conclude $TS' \models \varphi$.

This chapter will introduce several implementation relations, ranging from very strict ones ("strong" relations) that require transition systems to mutually mimic each transition, to more liberal ones ("weak" relations) in which this is only required for certain transitions.

In fact, in this monograph we have already encountered implementation relations that were aimed at comparing transition systems by considering their linear-time behavior, i.e., (finite or infinite) traces. Examples of such relations are trace inclusion and trace equivalence. Linear-time properties or LTL formulae are preserved by trace relations based on infinite traces; for trace-equivalent TS and TS' and linear-time property P, we have $TS' \models P$ whenever $TS \models P$. A similar result is obtained when replacing P by an LTL formula.

Besides the introduction of a weak variant of trace equivalence, the goal of this chapter is primarily to study relations that respect the branching-time behavior. Classical representatives of such relations are *bisimulation* equivalences and *simulation* preorder relations. Whereas bisimulation relates states that mutually mimic all individual transitions, simulation requires that one state can mimic all stepwise behavior of the other, but not the reverse. Weak variants of these relations only require this for certain ("observable") transitions, and not for other ("silent") transitions. This chapter will formally define strong and weak variants of (bi)simulation, and will treat their relationship to trace-based relations. The preservation of CTL and CTL* formulae is shown; for bisimulation the truth value of all such formulae is preserved, while for simulation this applies to a (large) fragment thereof. These results provide us with the tools to simplify establishing $TS' \models \varphi$, for CTL (or CTL*) formula φ by checking whether $TS \models \varphi$.

This provides the theoretical underpinning of exploiting bisimulation and simulation relations for the purpose of abstraction. The remaining issue is how to obtain the more abstract TS from the (larger, and more concrete) transition system TS'. This chapter will treat polynomial-time algorithms for several notions of (bi)simulation. These algorithms allow checking whether two given transition systems are (bi)similar, and can be used to generate an abstract transition system from a more concrete one in an automated manner.

As in the previous chapters, we will follow the state-based approach. This means that we consider branching-time relations that refer to the state labels, i.e., the atomic propositions that hold in the states. Action labels of the transitions are not considered. All concepts (definitions, theorems, algorithms) treated in this chapter can, however, easily be reformulated for the approach that is focused on action labels rather than state labels. This connection is discussed in Section 7.1.2.

Throughout this chapter, the transition systems under consideration may have terminal states, and hence, may exhibit both finite and infinite paths and traces. When considering temporal logics, though, transition systems without terminal states (thus only having infinite behaviors) are assumed.

7.1 Bisimulation

Bisimulation equivalence aims to identify transition systems with the same branching structure, and which thus can simulate each other in a stepwise manner. Roughly speaking, a transition system TS' can simulate transition system TS if every step of TS can be matched by one (or more) steps in TS'. *Bi*simulation equivalence denotes the possibility of mutual, stepwise simulation. We first introduce bisimulation equivalence as a binary relation between transition systems (over the same set of atomic propositions); later on, bisimulation is also treated as a relation between states of a single transition system. Bisimulation is defined coinductively, i.e., as the largest relation satisfying certain properties.

Definition 7.1. Bisimulation Equivalence

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$, be transition systems over AP. A *bisimulation* for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

(A) $\forall s_1 \in I_1 \ (\exists s_2 \in I_2.\ (s_1, s_2) \in \mathcal{R})$ and $\forall s_2 \in I_2 \ (\exists s_1 \in I_1.\ (s_1, s_2) \in \mathcal{R})$

(B) for all $(s_1, s_2) \in \mathcal{R}$ it holds:

(1) $L_1(s_1) = L_2(s_2)$

(2) if $s_1' \in Post(s_1)$ then there exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$

(3) if $s_2' \in Post(s_2)$ then there exsist $s_1' \in Post(s_1)$ with $(s_1', s_2') \in \mathcal{R}$.

TS_1 and TS_2 are *bisimulation-equivalent* (bisimilar, for short), denoted $TS_1 \sim TS_2$, if there exists a bisimulation $\mathcal{R}$ for (TS_1, TS_2). ■

Condition (A) asserts that every initial state of TS_1 is related to an initial state of TS_2, and vice versa. According to condition (B.1), the states s_1 and s_2 are equally labeled. This can be considered as ensuring the "local" equivalence of s_1 and s_2. Condition (B.2) states that every outgoing transition of s_1 must be matched by an outgoing transition of s_2; the reverse is stated by (B.3). Figure 7.1 summarises the latter two conditions.

Example 7.2. Two Beverage Vending Machines

Let $AP = \{ pay, beer, soda \}$. Consider the transition systems depicted in Figure 7.2. These model a beverage vending machine, but differ in the number of possibilities for supplying beer. The fact that the right-hand transition system (TS_2) has an additional option to deliver beer is not observable by a user. This suggests an equivalence between

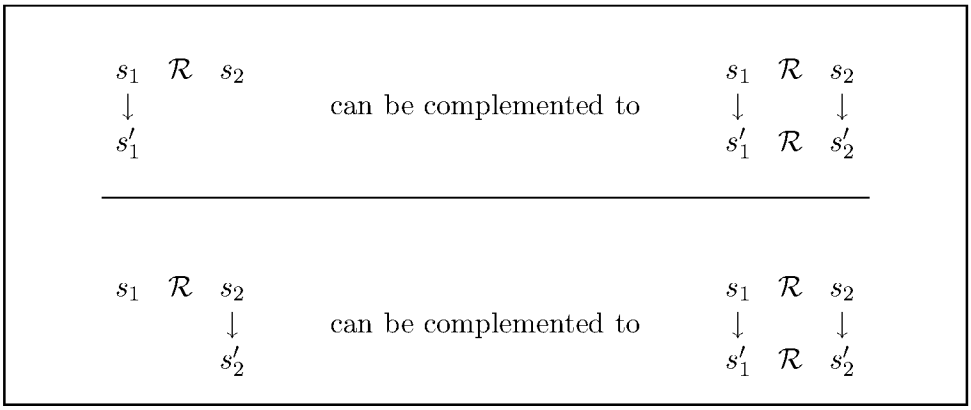


Figure 7.1: Conditions (B.2) and (B.3) of bisimulation equivalence.

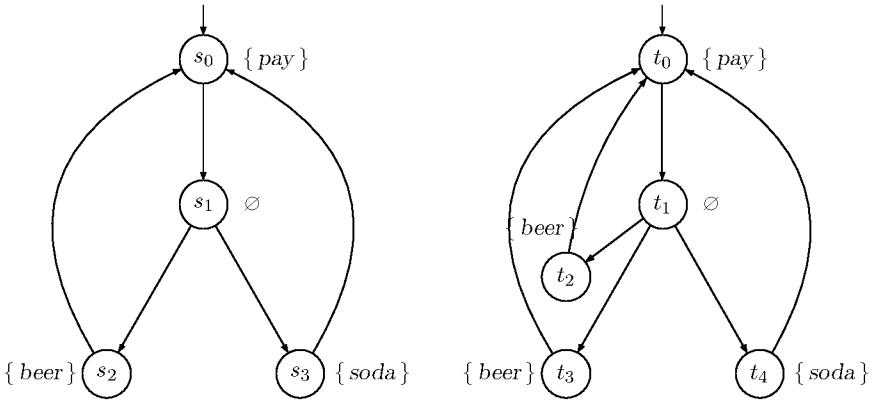


Figure 7.2: Two bisimilar beverage vending machines

the transition systems. Indeed, the equivalence of TS_1 and TS_2 follows from the fact that the relation

$$\mathcal{R} \;=\; \{(s_0,t_0),(s_1,t_1),(s_2,t_2),(s_2,t_3),(s_3,t_4)\}$$

is a bisimulation for (TS_1, TS_2). It can easily be verified that $\mathcal{R}$ indeed satisfies all requirements of Definition 7.1.

Now consider an alternative model (TS_3) of the vending machine where the user selects the drink on inserting a coin, see Figure 7.3, depicting TS_1 (left) and TS_3 (right). AP is as before. It follows that TS_1 and TS_3 are not bisimilar, since the state s_1 in TS_1 cannot be mimicked by a state in TS_3. This can be seen as follows. Due to the labeling condition (B.1), the only candidates for mimicking state s_1 are the states u_1 and u_2 in TS_3. However, neither of these states can mimic all transitions of s_1 in TS_1: either the possibility for *soda* or for *beer* is missing. Thus, $TS_1 \not\sim TS_3$ for $AP = \{\, pay, beer, soda \,\}$.

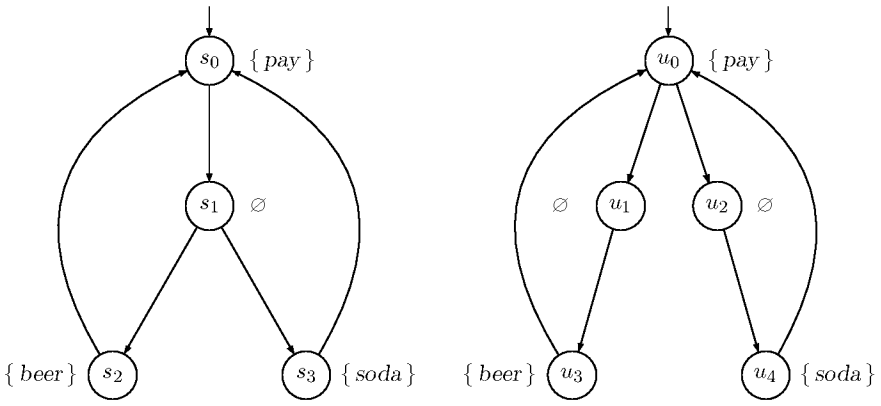


Figure 7.3: Nonbisimulation-equivalent beverage vending machines.

As a final example, reconsider TS_1 and TS_3 for $AP = \{ pay, drink \}$. The labelings of the transition systems are obvious: $L(s_0) = L(u_0) = \{ pay \}$, $L(s_1) = L(u_1) = L(u_2) = \varnothing$, and all remaining states are labeled with $\{ drink \}$. It can now be established that the relation

$$\{ (s_0, u_0), (s_1, u_1), (s_1, u_2), (s_2, u_3), (s_2, u_4), (s_3, u_3), (s_3, u_4) \}$$

is a bisimulation for (TS_1, TS_3). Henceforth, $TS_1 \sim TS_3$ for $AP = \{ pay, drink \}$. ■

Remark 7.3. The Relevant Set of Atomic Propositions

The fixed set AP plays a crucial role in comparing transition systems using bisimulation. Intuitively, AP stands for the set of all "relevant" atomic propositions. All other atomic propositions are understood as negligible and are ignored in the comparison. In case TS is a refinement of TS', e.g., TS is obtained from TS' by incorporating some implementation details, then the set AP of atomic propositions of TS generally is a proper superset of the set AP' of propositions of TS'. To compare TS and TS', the set of common atomic propositions, AP', is a reasonable choice. In this way, it is possible to check whether the branching structure of TS agrees with that of TS' when considering all observable information in AP'. If we are only interested in checking the equivalence of TS and TS' with respect to the satisfaction of a temporal logic formula Φ, it suffices to consider AP as the atomic propositions that occur in Φ. ■

Lemma 7.4. Reflexivity, Transitivity, and Symmetry of $\sim$

For a fixed set AP of atomic propositions, the relation $\sim$ is an equivalence relation.

Proof: Let AP be a set of atomic propositions. We show reflexivity, symmetry, and transitivity of $\sim$:

- Reflexivity: For transition system TS with state space S, the identity relation $\mathcal{R} = \{ (s, s) \mid s \in S \}$ is a bisimulation for (TS, TS).

- Symmetry: Sssume that $\mathcal{R}$ is a bisimulation for (TS_1, TS_2). Consider

$$\mathcal{R}^{-1} \;=\; \{(s_2, s_1) \mid (s_1, s_2) \in \mathcal{R}\}$$

  that is obtained by swapping the states in any pair in $\mathcal{R}$. Clearly, relation $\mathcal{R}^{-1}$ satisfies conditions (A) and (B.1). By symmetry of (B.2) and (B.3), we immediately conclude that $\mathcal{R}^{-1}$ is a bisimulation for (TS_2, TS_1).

- Transitivity: Let $\mathcal{R}_{1,2}$ and $\mathcal{R}_{2,3}$ be bisimulations for (TS_1, TS_2) and $(TS_2,\ TS_3)$, respectively. The relation $\mathcal{R} = \mathcal{R}_{1,2} \circ \mathcal{R}_{2,3}$, given by

$$\mathcal{R} \;=\; \{(s_1, s_3) \mid \exists s_2 \in S_2.\ (s_1, s_2) \in \mathcal{R}_{1,2} \ \wedge\ (s_2, s_3) \in \mathcal{R}_{2,3} \ \},$$

  is a bisimulation for (TS_1, TS_3) where S_2 denotes the set of states in TS_2. This can be seen by checking all conditions for a bisimulation.

  (A) Consider the initial state s_1 of TS_1. Since $\mathcal{R}_{1,2}$ is a bisimulation, there is an initial state s_2 of TS_2 with $(s_1, s_2) \in \mathcal{R}_{1,2}$. As $\mathcal{R}_{2,3}$ is a bisimulation, there is an initial state s_3 of TS_3 with $(s_2, s_3) \in \mathcal{R}_{2,3}$. Thus, $(s_1, s_3) \in \mathcal{R}$. In the same way we can check that for any initial state s_3 of TS_3, there is an initial state s_1 of TS_1 with $(s_1, s_3) \in \mathcal{R}$.

  (B.1) By definition of $\mathcal{R}$, there is a state s_2 in TS_2 with $(s_1, s_2) \in \mathcal{R}_{1,2}$ and $(s_2, s_3) \in \mathcal{R}_{2,3}$. Then, $L_1(s_1) = L_2(s_2) = L_3(s_3)$.

  (B.2) Assume $(s_1, s_3) \in \mathcal{R}$. As $(s_1, s_2) \in \mathcal{R}_{1,2}$, it follows that if $s_1' \in \mathit{Post}(s_1)$ then $(s_1', s_2') \in \mathcal{R}_{1,2}$ for some $s_2' \in \mathit{Post}(s_2)$. Since $(s_2, s_3) \in \mathcal{R}_{2,3}$, we have $(s_2', s_3') \in \mathcal{R}_{2,3}$ for some $s_3' \in \mathit{Post}(s_3)$. Hence, $(s_1', s_3') \in \mathcal{R}$.

  (B.3) Similar to the proof for (B.2).

■

Bisimulation is defined in terms of the direct successors of states. By using an inductive argument over states, we obtain a relation between (finite or infinite) paths.

Lemma 7.5. Bisimulation on Paths

Let TS_1 and TS_2 be transition systems over AP, $\mathcal{R}$ a bisimulation for (TS_1, TS_2), and $(s_1, s_2) \in \mathcal{R}$. Then for each (finite or infinite) path $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots \in \mathit{Paths}(s_1)$ there exists a path $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots \in \mathit{Paths}(s_2)$ of the same length such that $(s_{j,1}, s_{j,2}) \in \mathcal{R}$ for all j.

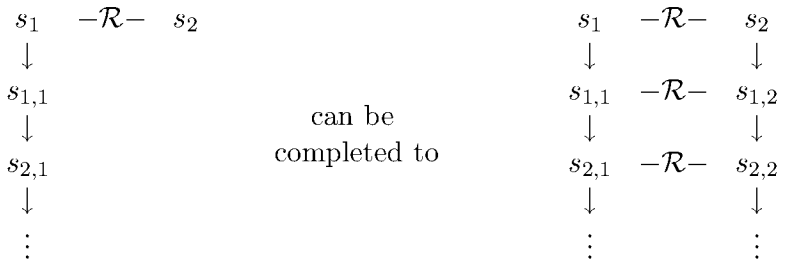


Figure 7.4: Construction of statewise bisimilar paths.

Proof: Let $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots \in \mathit{Paths}(s_1)$ be a maximal path fragment in TS_1 starting in $s_1 = s_{0,1}$ and assume $(s_1, s_2) \in \mathcal{R}$. We successively define a "corresponding" maximal path fragment in TS_2 starting in $s_2 = s_{0,2}$, where the transitions $s_{i,1} \to_1 s_{i+1,1}$ are matched by transitions $s_{i,2} \to_2 s_{i+1,2}$ such that $(s_{i+1,1}, s_{i+1,2}) \in \mathcal{R}$. This is done by induction on i, see Figure 7.4 on page 455. For each case we distinguish between s_i being a terminal state or not.

- Base case: $i = 0$. In case s_1 is a terminal state, it follows directly from $(s_1, s_2) \in \mathcal{R}$ (by condition (B.3)) that s_2 is a terminal state too. Thus $s_2 = s_{0,2}$ is a maximal path fragment in TS_2. If s_1 is not a terminal state, it follows from $(s_{0,1}, s_{0,2}) = (s_1, s_2) \in \mathcal{R}$ that the transition $s_1 = s_{0,1} \to_1 s_{1,1}$ can be matched by a transition $s_2 \to_2 s_{1,2}$ such that $(s_{1,1}, s_{1,2}) \in \mathcal{R}$. This yields the path fragment $s_2\, s_{1,2}$ in TS_2.

- Induction step: Assume $i \geqslant 0$ and that the path fragment $s_2\, s_{1,2}\, s_{2,2} \ldots s_{i,2}$ is already constructed with $(s_{j,1}, s_{j,2}) \in \mathcal{R}$ for $j = 1, \ldots, i$.

  If π_1 has length i, then π_1 is maximal and $s_{i,1}$ is a terminal state. By condition (B.3), $s_{i,2}$ is terminal too. Thus, $\pi_2 = s_2\, s_{1,2}\, s_{2,2} \ldots s_{i,2}$ is a maximal path fragment in TS_2 which is statewise related to $\pi_1 = s_1\, s_{1,1}\, s_{2,1} \ldots s_{i,1}$.

  Now assume that $s_{i,1}$ is not terminal. We consider the step $s_{i,1} \to_1 s_{i+1,1}$ in π_1. Since $(s_{i,1}, s_{i,2}) \in \mathcal{R}$, there exists a transition $s_{i,2} \to_2 s_{i+1,2}$ with $(s_{i+1,1}, s_{i+1,2}) \in \mathcal{R}$. This yields a path fragment $s_2\, s_{1,2} \ldots s_{i,2}\, s_{i+1,2}$ which is statewise related to the prefix $s_1\, s_{1,1} \ldots s_{i,1}\, s_{i+1,1}$ of π_1.

■

By symmetry, for each path $\pi_2 \in \mathit{Paths}(s_2)$ there exists a path $\pi_1 \in \mathit{Paths}(s_1)$ of the same length which is statewise related to π_2. As one can construct statewise bisimilar paths, bisimilar transition systems are trace-equivalent. It is mostly easier to prove that two transition systems are bisimilar rather than prove their trace equivalence. The intuitive

reason for this discrepancy is that proving bisimulation equivalence just requires "local" reasoning about state behavior instead of considering entire paths. The following result is thus of importance in checking trace equivalence as well.

Theorem 7.6. Bisimulation and Trace Equivalence

$TS_1 \sim TS_2$ *implies* $Traces(TS_1) = Traces(TS_2)$.

Proof: Let $\mathcal{R}$ be a bisimulation for (TS_1, TS_2). By Lemma 7.5, any path $\pi_1 = s_{0,1}\, s_{1,1}\, s_{2,1} \ldots$ in TS_1 can be lifted to a path $\pi_2 = s_{0,2}\, s_{1,2}\, s_{2,2} \ldots$ in TS_2 such that $(s_{i,1}, s_{i,2}) \in \mathcal{R}$ for all indices i. According to condition (B.1), $L_1(s_{i,1}) = L_2(s_{i,2})$ for all i. Thus, $trace(\pi_1) = trace(\pi_2)$. This shows $Traces(TS_1) \subseteq Traces(TS_2)$. By symmetry, one obtains that TS_1 and TS_2 are trace equivalent. ■

As trace-equivalent transition systems fulfill the same linear-time properties, it thus now follows that bisimilar transition systems fulfill the same linear-time properties.

7.1.1 Bisimulation Quotient

So far, bisimulation has been defined as a relation between transition systems. This enables comparing different transition systems. An alternative perspective is to consider bisimulation as a relation between states within a single transition system. By considering the quotient transition system under such a relation, smaller models are obtained. This minimization recipe can be exploited for efficient model checking. In the following, we define bisimulation as a relation on states, relate it to the notion of bisimulation between transition systems, and define the quotient transition system under such relation.

Definition 7.7. Bisimulation Equivalence as Relation on States

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *bisimulation* for TS is a binary relation $\mathcal{R}$ on S such that for all $(s_1, s_2) \in \mathcal{R}$:

1. $L(s_1) = L(s_2)$.
2. If $s_1' \in Post(s_1)$, then there exists an $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$.
3. If $s_2' \in Post(s_2)$, then there exists an $s_1' \in Post(s_1)$ with $(s_1', s_2') \in \mathcal{R}$.

States s_1 and s_2 are *bisimulation-equivalent* (or bisimilar), denoted $s_1 \sim_{TS} s_2$, if there exists a bisimulation $\mathcal{R}$ for TS with $(s_1, s_2) \in \mathcal{R}$. ■

Thus, a bisimulation (on states) for TS is a bisimulation (on transition systems) for the pair (TS, TS), except that condition (A) is not required. This condition could be ensured by adding the pairs (s, s) to $\mathcal{R}$ for any state s. Moreover, for all states s_1 and s_2 in TS it holds that

$$\underbrace{s_1 \sim_{TS} s_2}_{\text{as states of } TS \text{ (Def. 7.7)}} \quad \text{iff} \quad \underbrace{TS_{s_1} \sim TS_{s_2},}_{\text{in the sense of Def. 7.1}}$$

where TS_{s_i} denotes the transition system obtained from TS by declaring s_i as the unique initial state. Vice versa, the definition of bisimulation between transition systems (Definition 7.1) arises from Definition 7.7 as follows. Take transition systems TS_1 and TS_2 over AP, and combine them in a single transition system $TS_1 \oplus TS_2$, which basically results from the disjoint union of state spaces (see below). We then subsequently "compare" the initial states of TS_1 and TS_2 as states of the composite transition system $TS_1 \oplus TS_2$ to ensure condition (A).

The formal definition of $TS_1 \oplus TS_2$ is as follows. For $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$, $i = 1, 2$:

$$TS_1 \oplus TS_2 \quad = \quad (S_1 \uplus S_2, Act_1 \cup Act_2, \rightarrow_1 \cup \rightarrow_2, I_1 \cup I_2, AP, L)$$

where $\uplus$ stands for disjoint union and where $L(s) = L_i(s)$ if $s \in S_i$. Then $TS_1 \sim TS_2$ if and only if, for every initial state s_1 of TS_1, there exists a bisimilar initial state s_2 of TS_2, and vice versa. That is, $s_1 \sim_{TS_1 \oplus TS_2} s_2$. Stated in terms of equivalence classes, $TS_1 \sim TS_2$ if and only if

$$\forall C \in (S_1 \uplus S_2)/\sim_{TS_1 \oplus TS_2} . \; I_1 \cap C \neq \varnothing \quad \text{iff} \quad I_2 \cap C \neq \varnothing \quad .$$

Here, $(S_1 \uplus S_2)/\sim_{TS_1 \oplus TS_2}$denotes the quotient space with respect to $\sim_{TS_1 \oplus TS_2}$, i.e., the set of all bisimulation equivalence classes in $S_1 \uplus S_2$. The latter observation is based on the fact that $\sim_{TS_1 \oplus TS_2}$ is an equivalence relation, see the first part of the following lemma.

Lemma 7.8. Coarsest Bisimulation

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ it holds that:

1. *$\sim_{TS}$ is an equivalence relation on S.*
2. *$\sim_{TS}$ is a bisimulation for TS.*
3. *$\sim_{TS}$ is the coarsest bisimulation for TS.*

Proof: The first claim follows directly from the characterization of $\sim_{TS}$ in terms of $\sim$, and Lemma 7.4 on page 453. The last claim states that each bisimulation $\mathcal{R}$ for TS is

finer than $\sim_{TS}$; this immediately follows from the definition of $\sim_{TS}$. It remains to prove that $\sim_{TS}$ is a bisimulation for TS. We show that $\sim_{TS}$ satisfies conditions (1) and (2) of Definition 7.7. Condition (3) follows by symmetry. Let $s_1 \sim_{TS} s_2$. Then, there exists a bisimulation $\mathcal{R}$ that contains (s_1, s_2). From condition (1), it follows that $L(s_1) = L(s_2)$. Condition (2) yields that for any transition $s_1 \to s_1'$ there is a transition $s_2 \to s_2'$ with $(s_1', s_2') \in \mathcal{R}$. Hence, $s_1' \sim_{TS} s_2'$. ■

Stated differently, the relation $\sim_{TS}$ is the coarsest equivalence on the state space of TS such that equivalent states are equally labeled and can simulate each other as shown in Figure 7.5.

$$\begin{array}{ccc} s_1 & \sim_{TS} & s_2 \\ \downarrow & & \\ s_1' & & \end{array} \qquad \text{can be complemented to} \qquad \begin{array}{ccc} s_1 & \sim_{TS} & s_2 \\ \downarrow & & \downarrow \\ s_1' & \sim_{TS} & s_2' \end{array}$$

Figure 7.5: Condition (2) for bisimulation equivalence $\sim_{TS}$ on states.

Remark 7.9. Union of Bisimulations

For finite index set I and $(\mathcal{R}_i)_{i \in I}$ a family of bisimulation relations for TS, $\bigcup_{i \in I} \mathcal{R}_i$ is a bisimulation for TS too (see Exercise 7.2). Since $\sim_{TS}$ is the coarsest bisimulation for TS, $\sim_{TS}$ coincides with the union of all bisimulation relations for TS. ■

As stated before, bisimilar transition systems satisfy the same linear-time properties. Such properties—and, as we will see later, all temporal formulae that can be expressed in CTL*—can thus be checked on the quotient system instead of on the original (and possibly much larger) transition system. Before providing the definition of quotient transition systems with respect to $\sim_{TS}$, let us first fix some notations.

Notation 7.10. Equivalence Classes, Quotient Space

Let S be a set and $\mathcal{R}$ an equivalence on S. For $s \in S$, $[s]_{\mathcal{R}}$ denotes the equivalence class of state s under $\mathcal{R}$, i.e., $[s]_{\mathcal{R}} = \{ s' \in S \mid (s, s') \in \mathcal{R} \}$. Note that for $s' \in [s]_{\mathcal{R}}$ we have $[s']_{\mathcal{R}} = [s]_{\mathcal{R}}$. The set $[s]_{\mathcal{R}}$ is often referred to as the $\mathcal{R}$-equivalence class of s. The quotient space of S under $\mathcal{R}$, denoted by $S/\mathcal{R} = \{ [s]_{\mathcal{R}} \mid s \in S \}$, is the set consisting of all $\mathcal{R}$-equivalence classes. ■

Definition 7.11. Bisimulation Quotient

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ and bisimulation $\sim_{TS}$, the *quotient transition system* $TS/\sim_{TS}$ is defined by

$$TS/\sim_{TS} = (S/\sim_{TS}, \{\tau\}, \rightarrow', I', AP, L')$$

where:

- $I' = \{[s]_\sim \mid s \in I\}$,
- $\rightarrow'$ is defined by
$$\frac{s \xrightarrow{\alpha} s'}{[s]_\sim \xrightarrow{\tau} [s']_\sim},$$
- $L'([s]_\sim) = L(s)$.

■

In the sequel, $TS/\sim_{TS}$ is referred to as the *bisimulation quotient* of TS. For the sake of simplicity, we write $TS/\sim$ rather than $TS/\sim_{TS}$.

The state space of $TS/\sim$ is the quotient of S under $\sim$. The initial states in $TS/\sim$ are the $\sim$-equivalence classes of the initial states in TS. Each transition $s \rightarrow s'$ in TS induces a transition $[s]_\sim \rightarrow' [s']_\sim$. As the action label is irrelevant, these labels are omitted from now on; this is reflected in the definition by replacing any action $\alpha \in Act$ by an arbitrary action, τ say. The state-labeling function L' is well-defined, since all states in $[s]_\sim$ are equally labeled (see the definition of bisimulation). According to the definition of $\rightarrow'$ it follows for any $B, C \in S/\sim$:

$$B \rightarrow' C \quad \text{if and only if} \quad \exists s \in B.\, \exists s' \in C.\, s \rightarrow s'$$

By condition (2) of Definition 7.7, this is equivalent to

$$B \rightarrow' C \quad \text{if and only if} \quad \forall s \in B.\, \exists s' \in C.\, s \rightarrow s'.$$

The following two examples show that enormous state-space reductions may be obtained by considering the bisimulation quotient. In some cases, it even allows obtaining a finite quotient transition system for infinite transition systems.

Example 7.12. Many Printers

Consider a system consisting of n printers, each represented as extremely simplified by two states, *ready* and *print*. The initial state is *ready*, and once started, each printer alternates between being *ready* and *print*ing. The entire system is given by

$$TS_n = \underbrace{Printer \,|||\, \ldots \,|||\, Printer}_{n \text{ times}}.$$

Assume the states of TS_n are labeled with atomic propositions from the set $AP = \{0, 1, \ldots, n\}$. Intuitively, $L(s)$ denotes the number of printers available in state s, i.e., which are in the local state *ready*. The number of states of TS_n is exponential in n (it is 2^n); for n=3, TS_n is depicted in Figure 7.6, where r denotes ready, and p denotes print. The quotient transition system $TS_n/\sim$, however, only contains n+1 states. For n=3, $TS_n/\sim$ is depicted in Figure 7.7.

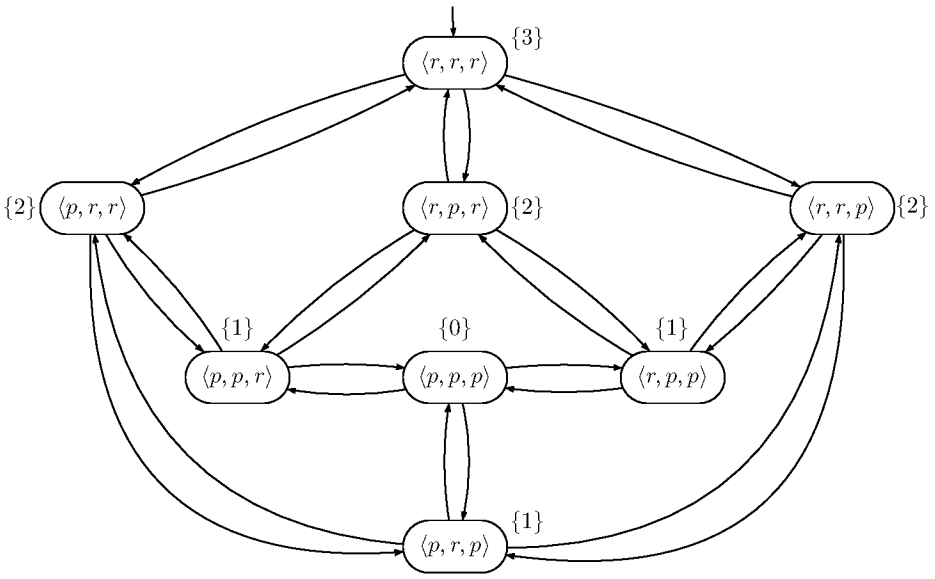


Figure 7.6: Transition system TS_3 for three independent printers

Figure 7.7: Bisimulation quotient $TS_3/\!\sim$.

■

Example 7.13. The Bakery Algorithm

We consider a mutual exclusion algorithm, originally proposed by Lamport, which is known as the *Bakery* algorithm. Although the principle of the Bakery algorithm allows guaranteeing mutual exclusion for an arbitrary number of processes, we consider the simpler setting with two processes. Let P_1 and P_2 be two processes, and x_1 and x_2 be two shared variables that both initially equal zero, see the following program text:

Process 1:		Process 2:	
	.....		.....
	while true {		**while** true {
	.....		.....
n_1 :	$x_1 := x_2 + 1$;	n_2 :	$x_2 := x_1 + 1$;
w_1 :	**wait until** $(x_2 = 0 \,\|\|\, x_1 < x_2)$ {	w_2 :	**wait until** $(x_1 = 0 \,\|\|\, x_2 < x_1)$ {
c_1 :	... critical section ...}	c_2 :	... critical section ...}
	$x_1 := 0$;		$x_2 := 0$;
	.....		.....
	}		}

These variables are used to resolve a conflict if both processes want to enter the critical section. (One might consider the value of a variable as a ticket, i.e., a number that one typically gets upon entering a bakery. The holder of the lowest number is the next customer to be served.) If process P_i is waiting, and $x_i < x_j$ or x_j=0, then it may enter the critical section. We have $x_i > 0$ whenever process P_i is either waiting to acquire access to the critical section, or is in the critical section. On requesting access to the critical section, process P_i sets x_i to x_j+1, where $i \neq j$. Intuitively, process P_i gives priority to its opponent, process P_j.

As the value of x_1 and x_2 may grow unboundedly, the underlying transition system of the parallel composition of P_1 and P_2 is infinite; a fragment of the transition system is depicted in Figure 7.8. An example of a path fragment that visits infinitely many different states is:

process P_1	process P_2	x_1	x_2	effect
$noncrit_1$	$noncrit_2$	0	0	P_1 requests access to critical section
$wait_1$	$noncrit_2$	1	0	P_2 requests access to critical section
$wait_1$	$wait_2$	1	2	P_1 enters the critical section
$crit_1$	$wait_2$	1	2	P_1 leaves the critical section
$noncrit_1$	$wait_2$	0	2	P_1 requests access to critical section
$wait_1$	$wait_2$	3	2	P_2 enters the critical section
$wait_1$	$crit_2$	3	2	P_2 leaves the critical section
$wait_1$	$noncrit_2$	3	0	P_2 requests access to critical section
$wait_1$	$wait_2$	3	4	P_2 enters the critical section
...	...	..	..	...

Although algorithmic analysis, such as LTL model checking, is impossible due to the infinity of the transition system, it is not difficult to check that the Bakery algorithm suffers neither from deadlock nor starvation:

- A deadlock only occurs whenever none of the processes can enter the critical section, i.e., if $x_1{=}x_2 > 0$. It is easy to see, however, that apart from the initial situation, we always have $x_1 \neq x_2$.
- Starvation only occurs if a process that wants to enter the critical section is never able to do so. Such situation, however, can never occur: in case both processes want to enter the critical section, it is impossible that a process acquires access to the critical section twice in a row.

Alternatively, the correctness of the Bakery algorithm can also be established algorithmically. This is done by considering an abstraction of the infinite transition system such that, instead of the concrete values of x_1 and x_2, it is only recorded whether

$$x_1 > x_2 > 0 \quad \text{or} \quad x_2 > x_1 > 0 \quad \text{or} \quad x_1 = 0 \quad \text{or} \quad x_2 = 0$$

Note that this information is sufficient to determine which process may acquire access to the critical section. By means of this *data abstraction*, we obtain a finite transition system $TS_{Bak}^{abstract}$, e.g., the infinite set of states for which x_1 and x_2 exceed 0 is mapped onto the single abstract state $\langle wait_1, wait_2, x_1 > x_2 > 0 \rangle$. When considering the atomic propositions

$$\{\, noncrit_i, wait_i, crit_i \mid i = 1,2 \,\} \;\cup\; \{\, x_1 > x_2 > 0, x_2 > x_1 > 0, x_1 = 0, x_2 = 0 \,\}$$

the finite transition system $TS_{Bak}^{abstract}$ (with the obvious state labeling) is trace-equivalent to the original infinite transition system TS_{Bak}. Due to the fact that trace-equivalent transition systems satisfy the same LT properties, it follows that each LT property that is

shown to hold for the finite (abstract) transition system also holds for the original infinite transition system! The following LT properties, expressed as LTL formulae, indeed hold for $TS_{Bak}^{abstract}$:

$$\Box(\neg crit_1 \ \vee \ \neg crit_2) \quad \text{and} \quad (\Box \Diamond \, wait_1 \ \Rightarrow \ \Box \Diamond \, crit_1) \ \wedge \ (\Box \Diamond \, wait_2 \ \Rightarrow \ \Box \Diamond \, crit_2) \, .$$

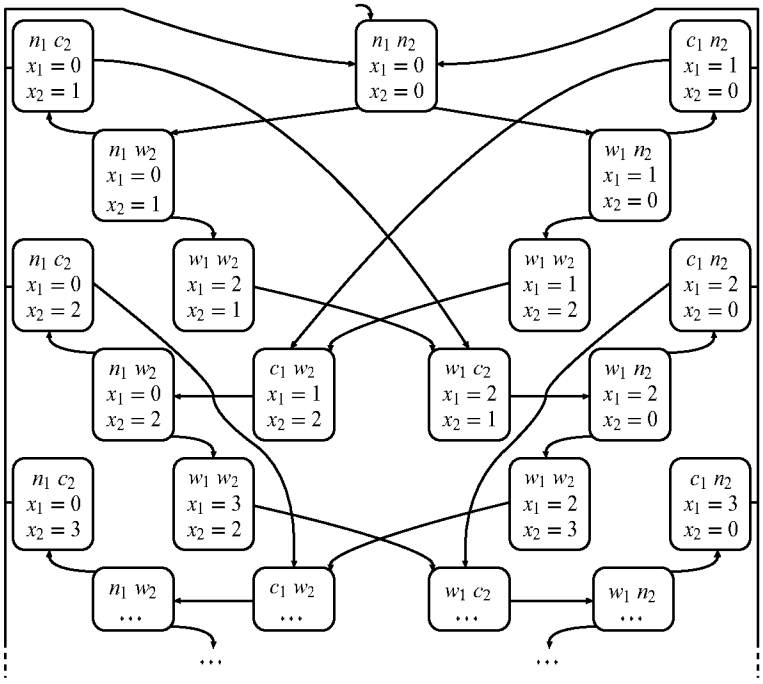


Figure 7.8: Fragment of the infinite transition system of the Bakery algorithm.

Transition systems TS_{Bak} and $TS_{Bak}^{abstract}$ are bisimilar. This can be easily proven by indicating a bisimulation relation. The data abstraction described above is formalized by means of the function $f : S \to S'$ where S and S' denote the set of reachable states of TS_{Bak} and $TS_{Bak}^{abstract}$, respectively. The function f associates to any reachable state s of TS_{Bak}, an (abstract) state $f(s)$ of $TS_{Bak}^{abstract}$. Let $s = \langle \ell_1, \ell_2, x_1 = b_1, x_2 = b_2 \rangle$ be a state of TS_{Bak} with $\ell_i \in \{ \, noncrit_i, wait_i, crit_i \, \}$ and $b_i \in \mathbb{N}$ for $i = 1, 2$. Then, we define

$$f(s) \ = \ \begin{cases} \langle \ell_1, \ell_2, x_1 = 0, x_2 = 0 \rangle & \text{if } b_1 = b_2 = 0 \\ \langle \ell_1, \ell_2, x_1 = 0, x_2 > 0 \rangle & \text{if } b_1 = 0 \text{ and } b_2 > 0 \\ \langle \ell_1, \ell_2, x_1 > 0, x_2 = 0 \rangle & \text{if } b_1 > 0 \text{ and } b_2 = 0 \\ \langle \ell_1, \ell_2, x_1 > x_2 > 0 \rangle & \text{if } b_1 > b_2 > 0 \\ \langle \ell_1, \ell_2, x_2 > x_1 > 0 \rangle & \text{if } b_2 > b_1 > 0 \quad . \end{cases}$$

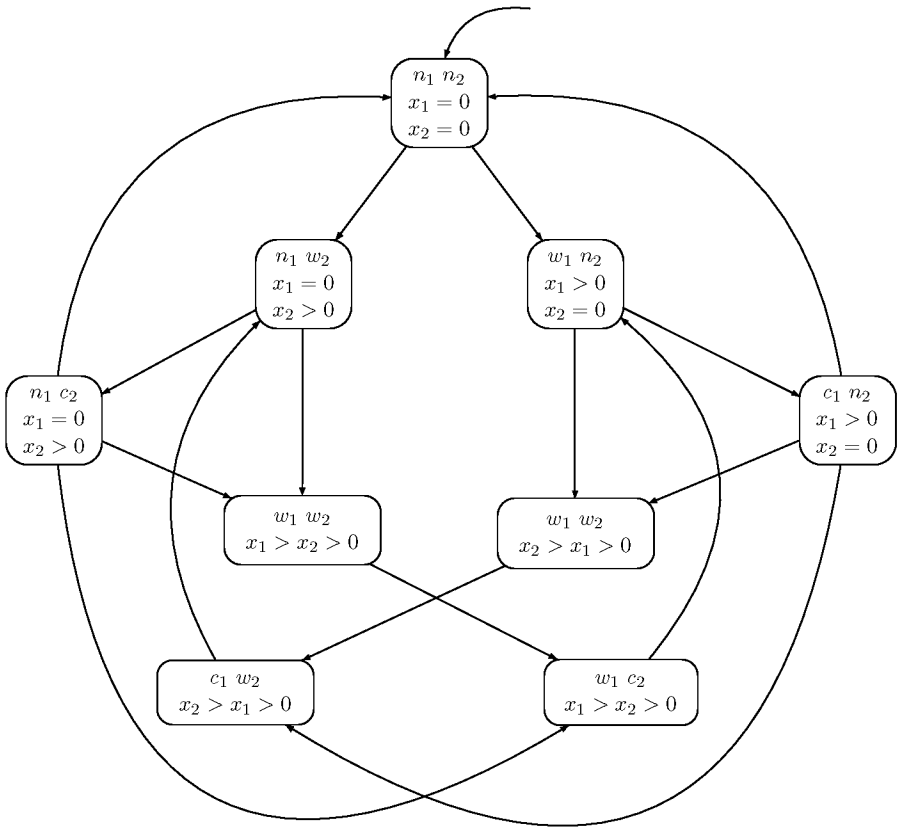


Figure 7.9: Bisimulation quotient transition system of the Bakery algorithm

It follows by straightforward reasoning that $\mathcal{R} = \{ (s, f(s)) \mid s \in S \}$ is a bisimulation for $(TS_{Bak}, TS_{Bak}^{abstract})$ for any subset of $AP = \{ noncrit_i, wait_i, crit_i \mid i = 1, 2 \}$. The transition system $TS_{Bak}^{abstract}$ in Figure 7.9 above is the bisimulation quotient system

$$TS_{Bak}^{abstract} \quad = \quad TS_{Bak}/\!\sim \quad \text{for} \quad AP = \{ crit_1, crit_2 \}.$$

Whereas the original transition system is *infinite*, its bisimulation quotient is *finite*. ■

Theorem 7.14. Bisimulation Equivalence of TS ***and*** $TS/\sim$

For any transition system TS *it holds that* $TS \sim TS/\sim$.

Proof: Follows immediately from the fact that $\{ (s, s') \mid s' \in [s]_\sim, s \in S \}$ is a bisimulation for $(TS, TS/\mathcal{R})$ in the sense of Definition 7.1 (page 451). ■

In general, the quotient transition system $TS/\mathcal{R}$ with respect to a bisimulation $\mathcal{R}$ contains more states than $TS/\sim$ since $\sim$ is the coarsest bisimulation relation. It is often simple to manually indicate a (meaningful) bisimulation, while the computation of the quotient space $S/\sim$ requires an algorithmic analysis of the complete transition system (see Section 7.3 on page 476 and further). Therefore, it may be advantageous to generate the quotient system $TS/\mathcal{R}$ instead of $TS/\sim$.

7.1.2 Action-Based Bisimulation

As the prime interest of this monograph is model checking, the notions of bisimulation are all focused on the state labelings; labels of transitions are simply ignored. In other contexts, most notably process algebras, analogous notions of bisimulation are considered that do the reverse—these notions ignore state labelings and are focused on transitions labels, i.e. actions. The aim of this subsection is to relate these notions.

Definition 7.15. Action-Based Bisimulation Equivalence

Let $TS_i = (S_i, Act, \rightarrow_i, I_i, AP_i, L_i)$, i=1,2, be transition systems over the set Act of actions. An *action-based bisimulation* for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that

(A) $\forall s_1 \in I_1 \, \exists s_2 \in I_2 .\ (s_1, s_2) \in \mathcal{R}$ and $\forall s_2 \in I_2 \, \exists s_1 \in I_1 .\ (s_1, s_2) \in \mathcal{R}$

(B) for any $(s_1, s_2) \in \mathcal{R}$ it holds that

(2') if $s_1 \xrightarrow{\alpha}_1 s_1'$, then $s_2 \xrightarrow{\alpha}_2 s_2'$ with $(s_1', s_2') \in \mathcal{R}$ for some $s_2' \in S_2$

(3') if $s_2 \xrightarrow{\alpha}_2 s_2'$, then $s_1 \xrightarrow{\alpha}_1 s_1'$ with $(s_1', s_2') \in \mathcal{R}$, for some $s_1' \in S_1'$.

TS_1 and TS_2 are *action-based bisimulation equivalent* (or action-based bisimilar), denoted $TS_1 \sim^{Act} TS_2$, if there exists an action-based bisimulation $\mathcal{R}$ for (TS_1, TS_2). ■

Action-based bisimulation differs from the variant for state labels (see Definition 7.1) in the following way: the state-labeling condition (B.1) is reformulated by means of the transition labels, and thus is encoded by conditions (B.2') and (B.3'). All results and concepts presented for $\sim$ can be adapted for $\sim^{Act}$ in a straightforward manner. For instance, $\sim^{Act}$ is an equivalence and can be adapted to an equivalence $\sim_{TS}^{Act}$ for the states of a single transition system TS in a similar way as before. The action-based bisimulation quotient system $TS/\sim^{Act}$ is defined as $TS/\sim$ except that we deal with an empty set of atomic propositions and lift any transition $s \xrightarrow{\alpha} s'$ to an equally labeled transition $B \xrightarrow{\alpha} B'$

where B and B' denote the action-based bisimulation equivalence classes of states s and s', respectively.

In the context of process calculi, an important aspect of bisimulation is whether it enjoys a substitutivity property with respect to syntactic operators in the process calculus, such as parallel composition. The following result states that action-based bisimulation is a congruence for the parallel composition $\|_H$ with synchronization over handshaking actions , see Definition 2.26 on page 48.

Lemma 7.16. Congruence w.r.t. Handshaking

For transition systems TS_1, TS_1' over Act_1, TS_2, TS_2' over Act_2, and $H \subseteq Act_1 \cap Act_2$ it holds that

$$TS_1 \sim^{Act} TS_1' \quad \textit{and} \quad TS_2 \sim^{Act} TS_2' \quad \textit{implies } TS_1 \|_H TS_2 \sim^{Act} TS_1' \|_H TS_2'.$$

Proof: Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP, L_i)$ and $TS_i' = (S_i', Act_i, \rightarrow_i', I_i', AP, L_i')$ and let $\mathcal{R}_i \subseteq S_i \times S_i'$ be an action-based bisimulation for (TS_i, TS_i'), i=1, 2. Then, the relation:

$$\mathcal{R} = \{ (\langle s_1, s_2 \rangle, \langle s_1', s_2' \rangle) \mid (s_1, s_1') \in \mathcal{R}_1 \ \wedge \ (s_2, s_2') \in \mathcal{R}_2 \}$$

is an action-based bisimulation for $(TS_1 \|_H TS_2, TS_1' \|_H TS_2')$. This can be seen as follows. The fulfillment of condition (A) is obvious. To check condition (B.2'), assume that (1) there is a transition $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle t_1, t_2 \rangle$ in $TS_1 \|_H TS_2$, and (2) $(\langle s_1, s_2 \rangle, \langle t_1, t_2 \rangle)$ in $\mathcal{R}$. Distinguish two cases.

1. $\alpha \in Act \setminus H$. Then $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle t_1, t_2 \rangle$ arises by an individual move of either TS_1 or TS_2. By symmetry, we may assume that $s_1 \xrightarrow{\alpha}_1 t_1$ and $s_2 = t_2$. Since (s_1, s_1') belongs to bisimulation $\mathcal{R}_1$, there exists a transition $s_1' \xrightarrow{\alpha}_1' t_1'$ in TS_1' with $(t_1, t_1') \in \mathcal{R}_1$. Thus, $\langle s_1', s_2' \rangle \xrightarrow{\alpha} \langle t_1', s_2' \rangle$ is a transition in $TS_1' \|_H TS_2'$ and $(\langle t_1, s_2 \rangle, \langle t_1', s_2' \rangle) \in \mathcal{R}$.
2. $\alpha \in H$. Then $\langle s_1, s_2 \rangle \xrightarrow{\alpha} \langle t_1, t_2 \rangle$ arises by synchronizing transitions in TS_1 and TS_2. That is, $s_1 \xrightarrow{\alpha}_1 t_1$ is a transition in TS_1 and $s_2 \xrightarrow{\alpha}_2 t_2$ a transition in TS_2. Since $(s_i, s_i') \in \mathcal{R}_i$, there exists a transition $s_i' \xrightarrow{\alpha}_i' t_i'$ in TS_i' (for $i = 1, 2$) with $(t_i, t_i') \in \mathcal{R}_i$. Thus, $\langle s_1', s_2' \rangle \xrightarrow{\alpha} \langle t_1', t_2' \rangle$ is a transition in $TS_1' \|_H TS_2'$ and $(\langle t_1, t_2 \rangle, \langle t_1', t_2' \rangle) \in \mathcal{R}$.

The fulfillment of condition (B.3') follows by a symmetric argument. ■

Let us now consider the relation between state-based and action-based bisimulation in more detail. We first discuss how an action-based bisimulation can be obtained from

a state-based bisimulation, and the reverse. This is done for transition system $TS = (S, Act, \rightarrow, I, AP, L)$.

Consider the bisimulation $\sim_{TS}$ on S. The intention is to define a transition system

$$TS_{act} = (S_{act}, Act, \rightarrow_{act}, I_{act}, AP_{act}, L_{act})$$

such that $\sim_{TS}$ and the action-based bisimulation $\sim_{TS}^{Act}$ coincide. As our interest is in action-based bisimulation on TS_{act}, AP_{act} and L_{act} are irrelevant, and can be taken as AP and L, respectively. Let $S_{act} = S \cup \{ t \}$ where t is a new state (i.e., $t \notin S$). TS_{act} has the same initial states as TS, i.e., $I_{act} = I$, and is equipped with the action set $Act = 2^{AP} \cup \{ \tau \}$. The transition relation $\rightarrow_{act}$ is given by the rules:

$$\frac{s \longrightarrow s'}{s \xrightarrow{L(s)}_{act} s'} \qquad \text{and} \qquad \frac{s \text{ is a terminal state in } TS}{s \xrightarrow{\tau}_{act} t}$$

Thus, the new state t serves to characterize the terminal states in TS. For bisimulation $\mathcal{R}$ for TS, $\mathcal{R}_{act} = \mathcal{R} \cup \{ (t,t) \}$ is an action-based bisimulation for TS_{act}. Vice versa, for action-based bisimulation $\mathcal{R}_{act}$ for TS_{act}, $\mathcal{R}_{act} \cap (S \times S)$ is a bisimulation for TS. Thus, for all states $s_1, s_2 \in S$:

$$s_1 \sim_{TS} s_2 \quad \text{if and only if} \quad s_1 \sim_{TS_{act}}^{Act} s_2.$$

Let us now consider the reverse direction. Consider the action-based bisimulation $\sim_{TS}^{Act}$ on S. The intention is to define a transition system

$$TS_{state} = (S_{state}, Act_{state}, \rightarrow_{state}, I_{state}, AP_{state}, L_{state})$$

such that $\sim_{TS}^{Act}$ and the bisimulation $\sim_{TS_{state}}$ coincide. As our interest is in a state-based bisimulation, the action-set Act_{state} is not of importance. Let $S_{state} = S \cup (S \times Act)$ where it is assumed that $S \cap (S \times Act) = \varnothing$. (Such construction is also used to compare action-based vs. state-based fairness on page 263) Take $I_{state} = I$. The actions of TS serve as atomic propositions in TS_{state}, i.e., $AP_{state} = Act$. The labeling function of L_{state} is defined by $L(s) = \varnothing$ and $L(\langle s, \alpha \rangle) = \{ \alpha \}$. The transition relation $\rightarrow_{state}$ is defined by the rules:

$$\frac{s \xrightarrow{\alpha} s'}{s \longrightarrow_{state} \langle s', \alpha \rangle} \qquad \text{and} \qquad \frac{s \xrightarrow{\alpha} s', \quad \beta \in Act}{\langle s, \beta \rangle \longrightarrow_{state} \langle s', \alpha \rangle}$$

That is, state $\langle s, \alpha \rangle$ in TS_{state} serves to simulate state s in TS. In fact, the second component α indicates the action via which s is entered. It now follows that for all states $s_1, s_2 \in S$ (see Exercise 7.5):

$$s_1 \sim_{TS}^{Act} s_2 \quad \text{if and only if} \quad s_1 \sim_{TS_{state}} s_2.$$