Theory of Reactive Systems

Jan Peleska and Wen-ling Huang
{jp,huang}@informatik.uni-bremen.de
Issue 3.3
2022-07-18

Note. These lecture notes are still under development, so it is advisable to check for updates.

Change History

Issue	Date	Change description
2.0	2015-05-18	Change bars introduced. In Chapter 4, new paragraph on model checking techniques. New Chapter 5 on bounded model checking.
2.1	2015-06-01	Typos fixed in Table 5.2. New Section 5.5 on global property checking with BMC.
2.2	2015-06-08	New lattice example in Section B.1. Use terms "lower/upper adjoint" instead of "left/right" for Galois connections. New exercises 601 and 602. Change bars indicate changes since version 2.1
2.3	2015-06-08	New example 7 for implication lattices. Change bars indicate changes since version 2.2
2.4	2015-06-22	Introduction of positive normal form for LTL formulas (Section 3.1). Simplified proofs for Lemma 15 and Theorem 14. Change bars indicate changes since version 2.3
2.5	2015-06-29	Fixed erroneous statement about Galois Connections in Section B.2. Change bars indicate changes since version 2.4
2.6	2015-07-06	Removed Exercise 601. New Lemma 7 with full proof of Galois Connection properties. Added example after Lemma 7 showing that Galois Connections are generally not lattice homomorphisms. Change bars indicate changes since version 2.5
2.7	2015-07-13	New Example illustrating the construction of simulations in Section C.5. Change bars indicate changes since version 2.6
2.8	2017-04-03	Clarify difference between propositional logic and first order logic. Change bars indicate changes since version 2.7

2.9	2017-04-10	Some typos corrected. Change bars indicate changes since version 2.8
2.10	2017-04-24	Small additions and changes in Chapter 1 and Chapter 3. Change bars indicate changes since version 2.9
2.11	2017-05-08	Revised explanation about $K \models \phi$ on first page of Chapter 4. Change bars indicate changes since version 2.10
2.12	2017-05-15	Re-factored CTL model checking algorithm in Chapter 4. These changes have not been marked by change bars.
2.13	2017-06-19	Re-installed the old Chapter 7 as main chapter on data abstraction. Chapters B and C have been moved as advanced material into the Appendix. These changes have not been marked by change bars.
2.14	2018-04-15	Introduced consistent numbering for exercises These changes have not been marked by change bars.
2.17	2020-07-27	Notation for Timed State Machine in Chapters 9 has been changed to "m". These changes have not been marked by change bars.
2.18	2021-02-26	Fix typo in function *checkCTL* Chapter 4. These changes have not been marked by change bars.
2.19	2022-04-21	Use dot notation for sequences. (These changes have not been marked by change bars.) Add information about hybrid systems in Chapter 1.
2.20	2022-04-26	Improve explanations in Exercise 1. Add information on timed computations and traces in Chapter 1.
2.21	2022-05-06	New Section 9.1 about Zeno sequences, time lock, and finite variability.
2.22	2022-05-16	New paragraph about propositional LTL in Section 3.1
2.23	2022-06-13	Add theorem on time complexity of CTL model checking in Chapter 4. New exercise on complexity of CTL model checking algorithms.

3.0	2022-06-27	Add new Lemma 2. Add section on over-approximation of LTL bad-prefix formulae by CTL formulae in Sectin 3.5.
3.1	2022-07-10	Added Biere's Theorem 3.1 about BMC fixpoint evaluation for LTL.
3.2	2022-07-17	New Chapter 6 on nuXmv model checker.
3.3	2022-07-18	Add detailed simulation construction for traffic light example in Chapter 7 on data abstraction.

Contents

List of Figures

Chapter 1

Reactive Systems, Behaviour, Specifications and Models

Reactive Systems. A *reactive computer system* continuously interacts with its operational environment: at any point in time, inputs from the environment to the system may occur, and the system should be ready to react on these inputs in an appropriate way. In general, the interaction takes place over a longer period of time (think of an aircraft engine controller that should certainly be operative during the duration of the flight); in many applications reactive computer systems are not supposed to terminate at all, because the services they deliver do not allow for any downtime (so-called 24/7 systems).

Behaviour, States and Events. As a consequence, the *behaviour* of reactive systems cannot simply be described by initial and termination state, as would be possible for sequential terminating software programs. Instead, behaviour is characterised by (possibly infinite) sequences of state changes, called *computations*, *executions* or *runs* of the reactive system:

$$c = s_0.s_1.s_2\ldots$$

denotes a sequence of states s_i which have been observed as "snapshots" of the system state at several points in time during the execution. s_0 was the first observation, s_1 the second, and so on. Observe that computations

represent a *discretised* view on the observable state components: it may be the case that between observations s_i and s_{i+1} additional state changes took place which we could not observe or were not interested in. In theory it would be possible for digital computer systems to observe *every* state change in a computation since the electronic circuits involved process data in discrete steps timed by the digital clock. For physical systems, however, when *time-continuous* observables are involved (e. g. change of temperature over time), computations can never capture the complete evolution of system states.

For reactive real-time systems, the system execution time $t \in \mathbb{R}_{\geq 0}$ will be an explicit state variable, or it can be measured and recorded during system execution. It is often desirable to represent computations as pairs of execution time and other state values. This is usually written in the form

$$c = (t_0, s_0).(t_1, s_1).(t_2, s_2)\ldots$$

We require that such timed computations fulfil the following realistic assumptions

1. *Time is monotonic.* This means that consecutive time stamps in a computation never decrease.

2. *Finite variability.* The computation can perform only finitely many state changes in any finite time interval.

3. *Bounded variability.* There exists global upper bounds β, δ, such that the computation can perform at most β state changes in any time interval of a duration less or equal to δ. This is a stronger alternative to finite variability: it states that the state changes inside the computation cannot be accelerated.

4. *Absence of time convergence.* The sequence of t_i does not converge to a finite time value.

5. *Absence of time lock.* A time lock exists if the computation is time convergent and a computation suffix exists where no state changes

occur anymore. The system is "locked" in this state for infinitely many steps, while the time converges to a finite value.

6. *Non zenoness condition.* Infinitely many state changes in the computation require infinite time. This means that systems cannot be infinitely fast. Zenoness complements time lock: a Zeno computation is time convergent and performs *infinitely many state changes* on a computation suffix.

Systems with mixed discrete and time-continuous observables are called *hybrid systems*; they have been studied in detail by Henzinger [8] and others. The semantics of hybrid systems can be described by an extended notion of computations c_{hybrid} structured as

$$c_{\text{hybrid}} = s_0.\text{flow}_0.s_1.\text{flow}_1.s_2.\text{flow}_2.s_3\ldots$$

Hybrid computations are interpreted in the following way.

1. Sequence $s_0.s_1.s_2.s_3\ldots$ describes the system state at *discrete points in time*, such that the transitions $s_i \longrightarrow s_{i+1}$ capture all *discrete* variable changes (so-called *jumps*) in this computation. The discrete changes may affect both variables with discrete range (like `int`, $\mathbb{Z}$ or subsets thereof) and continuous range (like $\mathbb{R}$). Typically, jumps are interpreted to require zero time, so that time passes *in between* two jumps.

2. Between state changes $s_i \longrightarrow s_{i+1}$ of discrete variables, only the time-continuous variables change over the time available between the entry of state s_i at time t_i and the transition to s_{i+1} at time t_{i+1}. This change of time-continuous variables x is specified by

   $$\text{flow}_i = (f_i : (0, t_{i+1} - t_i) \to \mathbb{R}, \dot{f}_i : (0, t_{i+1} - t_i) \to \mathbb{R})$$

   with a differentiable function f and the constraint

   $$\forall t \in (t_i, t_{i+1}) \,.\, x(t) = x(t_i) + \int_0^t \dot{f}_i(t)\,dt$$

   The variable value $x(t_i)$ is the result of the jump when entering s_i. From there, the value of x changes according to the integrated derivative.

It is possible to abstract from concrete states in the description of reactive system behaviour by recording sequences of *events*. Events denote discrete points in time where certain properties of the state space become true. This abstraction may help to reduce the amount of information in computations to the data which is "relevant" in the application context.

Slightly more formal, any predicate p over state variables gives rise to an event e_p which is triggered according to the following rules.

1. e_p occurs initially (at execution time stamp 0), if and only if p evaluates to `true` initially. (This convention is inspired by the definition of *change events* in SysML.) If several events occur initially, because all of their predicates evaluate to `true`, they are listed in a prefix of the trace of events in arbitrary order, all of them equipped with time stamp 0.

2. e_p occurs at time t_i, $i > 0$, where t_i is a time stamp in the (state-based) computation, if and only if

   - p evaluates to `false` in computation step t_{i-1} and
   - p evaluates to `true` in computation step t_i.

   If several events need to be triggered at the same point in time t_i, because all of their predicates where `false` at t_{i-1} and became `true` at t_i, these events are recorded in the trace with the same time stamp t_i, and in arbitrary order. Having the same time stamp in a trace is always interpreted as the fact that an order of event occurrence cannot be determined.

Example 1. Suppose we observe temperature changes temp in a reactor at discrete points in time, and this results in a run

$$\pi =_{\mathsf{def}} (t_0, \mathrm{temp}_0).(t_1, \mathrm{temp}_1).(t_2, \mathrm{temp}_2) \ldots (t_k, \mathrm{temp}_k) \ldots$$

where the state observations consist of tuples (timestamp t_i, temperature temp_i observed at time t_i). Suppose further that we are interested in

observing whether a temperature threshold max is exceeded at t_k by $temp_k$, and that the computation satisfies

$$\begin{aligned}&\forall i \in \{0, \ldots, k-1\} : temp_i \leq max\\ &\forall i \in \{k, \ldots, k+3\} : temp_i > max\\ &\forall i \in \{k+4, \ldots\} : temp_i \leq max\end{aligned}$$

Introducing two events

- temp_ok, induced by predicate $p_0 \equiv temp \leq max$, and
- temp_too_high, induced by predicate $p_1 \equiv \neg p_0$.

the computation can be abstracted to a *trace of events*

$$\pi_{event} =_{def} (t_0, \text{temp\_ok}).(t_k, \text{temp\_too\_high}).(t_{k+4}, \text{temp\_ok})$$

□

Specifications. A *specification* is a description of the expected or admissible behaviours of a system. In general, first order predicate logic can be used to write specifications by giving logical characterisations of the state sequences or event sequences which are admissible in computations. Since these logical characterisations always deal with sequences of states or events, more elegant logical formalisms (temporal logic, trace logic) have been invented, in order to represent these logical formulas in a more elegant way. Some of these logical formalisms will be presented in the sections below.

Example 2. Suppose we require in Example 1 that the temperature threshold in the reactor should never be exceeded for longer than δ time units. This can be expressed by a formula referring to arbitrary computations

$$\pi =_{def} (t_0, temp_0).(t_1, temp_1).(t_2, temp_2) \ldots$$

in the following way:

$$\forall \pi : \forall i \geq 0 : temp_i > max \Rightarrow (\exists j > 0 : temp_{i+j} \leq max \wedge t_{i+j} - t_i \leq \delta)$$

On the event abstraction level, consider arbitrary computations

$$\pi_{event} =_{def} (t_0, e_0).(t_1, e_1)\ldots$$

Now the requirement can be expressed as

$$\forall \pi_{event} : \forall i \geq 0 : e_i = \text{temp\_too\_high} \Rightarrow (e_{i+1} = \text{temp\_ok} \wedge t_{i+1} - t_i \leq \delta)$$

□

Models. A *model* is a representation of the system from which all possible behaviours can be theoretically derived in a mechanical way by means of *simulations*.

Model Checking. A procedure to investigate whether the possible behaviours of a model satisfy a given specification is called *model checking*, or, more specific, *property checking*.

Another variant of model checking investigates whether two given models produce the same computations (i. e., have the same behaviour). This technique is called *equivalence checking*.

A third variant checks whether the sets of computations associated with two models fulfil a more general relation than equality, as, for example, a subset relation. This variant is usually called *refinement checking*.

Exercise 1. Fig. 1.1 shows a laboratory which is equipped with a laser and a door locking mechanism, both controlled by a controller component. When the laboratory is empty, the door is locked and the laser is switched on. Anyone who wants to enter the room has to push a button whereupon the controller switches the laser off and unlocks the door.

Right after being switched on the laser is in the state *on* which, by itself, changes to *active* after a certain period of time. The same applies to the states *off* and *passive*.

At any time, the door is either *open* or *closed*. After the door has been opened, it closes automatically. A counter counts how often the door has

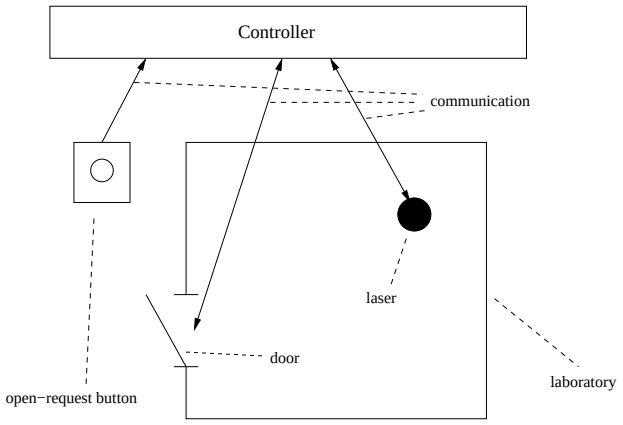


Figure 1.1: Laboratory setup from Exercise 1

been opened or closed. It can be assumed that at any time at most one person has access to the open-request button and may enter the lab.

Assuming t, *door*, *dcnt* and *laser* being variables reflecting the point in time, the door state, the door counter and the laser state respectively, computations c are of the form $(t_0, door_0, dcnt_0, laser_0).(t_1, door_1, dcnt_1, laser_1)\ldots$ with domains

$$D(t) = \mathbb{R},\ D(door) = \{\ \mathit{open}, \mathit{closed}\ \},\ D(dcnt) = \mathbb{N},$$
$$D(laser) = \{\mathit{on}, \mathit{active}, \mathit{off}, \mathit{passive}\}$$

1.1 Find logical formulae to express the following textual requirements:

a) In the initial state the door is *closed*, the counter is 0 and the laser is in the *active* state.

b) Whenever the laser is in the state *on*, it will transit to state *active* after a finite number of steps.

c) The change of laser state from *off* to *passive* takes at most X time units.

d) If the laser is not in the state *passive*, the room has to be empty and the door has to be closed.

e) The laser has to be in the state *passive*, if the room is not empty or the door is open.

1.2 Define events $e_0, \ldots, e_n$ abstracting concrete computations c to abstract computations c_E of the form $(t_0, e_0).(t_1, e_1)\ldots$. Adapt the logical formulae from part 1.1 to abstract computations over these events. □

Chapter 2

Transition Systems and Kripke Structures

The operational semantics of specification formalisms for reactive systems, as well as of computer programs, can be described by means of state transition systems. For the verification of properties of specifications or programs it is useful to extend the notion of transition systems by adding information about the basic properties which are true in each state. This leads to the definition of Kripke structures. The definitions below follow closely [5, pp. 14].

Definition 1 *A* State Transition System *is a triple* $\mathrm{TS} = (S, S_0, R)$*, where*

- S *is the set of* states*,*
- $S_0 \subseteq S$ *is the set of* initial states*,*
- $R \subseteq S \times S$ *is the* transition relation*.*

□

Given a state transition system, its computations can be determined as follows. Let S^ω denote the set of infinite sequences of elements from S, that is, infinite sequences of states: from now on we only consider non-terminating systems, so that computations are never finite[1]. Every com-

[1] Observe that even for terminating systems we can assume that their computations are infinite by repeating all termination states *ad infinitum*.

putation of TS has to start in one of the initial states from S_0, and each pair of consecutive states in the sequence has to be compatible with the transition relation. This leads to

$$\mathrm{Comp}(TS) = \{\pi \in S^\omega \mid \pi(0) \in S_0 \wedge \forall i \geq 0 : (\pi(i), \pi(i+1)) \in R\}$$

It is interesting to note that S^ω is actually "quite big" in the following sense.

Lemma 1 *If S contains at least two states then S^ω is uncountable.*

Proof. The proof applies *Cantor's Diagonal Argument* which has originally been used to prove that the set of real numbers is uncountable: suppose that S has just two states s_0, s_1. Suppose further that S^ω were countable. Then an enumeration of S^ω would exist that could be presented in tabular form as follows.

No.	Element of S^ω
0	$a_{00}, a_{01}, a_{02}, a_{03}, \ldots$
1	$a_{10}, a_{11}, a_{12}, a_{13}, \ldots$
2	$a_{20}, a_{21}, a_{22}, a_{23}, \ldots$
3	$\ldots$
$\ldots$	$\ldots$

with $a_{ij} \in \{s_0, s_1\}$. Now define the following infinite sequence of states from $\{s_0, s_1\}$:

$$\pi = b_0.b_1.b_2\ldots$$

such that

$$b_i = \begin{cases} s_0 & \text{if} \quad a_{ii} = s_1 \\ s_1 & \text{if} \quad a_{ii} = s_0 \end{cases}$$

Obviously π is not contained in the table above, because for all $i \geq 0$ its i^{th} element differs from table entry number i at place a_{ii}. This contradicts the assumption that the table enumerates all elements from S^ω, and hence S^ω must be uncountable. □

Definition 2 *A* Labelled Transition System *is a tuple* $\mathrm{LTS} = (S, S_0, \Sigma, R)$, *where*

- S *is the set of* states,
- $S_0 \subseteq S$ *is the set of* initial states,
- Σ *is a set of* labels, *also called* events,
- $R \subseteq S \times \Sigma \times S$ *is the* transition relation.

□

If we abstract from states and observe events only, the computations of a labelled transition system are given by

$$\text{Comp}_1(\text{LTS}) = \{e \in \Sigma^\omega \mid \exists \pi \in S^\omega : \pi(0) \in S_0 \wedge \forall i \geq 0 : (\pi(i), e(i), \pi(i+1)) \in R\}$$

This type of computations is typically used in the world of *process algebras*, such as CSP [9]. In other scenarios it is desirable to investigate both events and states, so that computations of the kind

$$\begin{aligned} \text{Comp}_2(\text{LTS}) \quad = \quad & \{\pi_e \in (S \cup \Sigma)^\omega \mid \forall i \geq 0 : \pi_e(2i) \in S \wedge \pi_e(2i+1) \in \Sigma \wedge \\ & \pi_e(0) \in S_0 \wedge (\pi_e(2i), \pi_e(2i+1), \pi_e(2i+2)) \in R\} \end{aligned}$$

State transition systems are the preferred mathematical models to reason about state-based reactive systems, where communication takes place according to the shared variable paradigm. Labelled transition systems are the preferred model for reasoning on the event abstraction level. In the sections to follow we focus on state-based systems represented by state transition systems.

A *proposition* is a logical expression which consists of Boolean operands composed by the logical operators $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$. The variables occurring in propositions are called *free* variables, because they can be associated with concrete values, making the formula true or false. In *propositional logic*, the operands in propositions are elementary statements p *without* free variables, so p corresponds to a Boolean variable. In *first order logic*, propositions may contain arbitrary Boolean expressions over free variables with different types as operands. Moreover, first order expressions may contain universal quantification $\forall$ and/or existential quantification $\exists$ operators and associated bound variables. For example, $\forall x \in \mathbb{Z} : x^2 + y \geq 0$ is

a first order formula. It has free variable y and bound variable x. For any $y \geq 0$, the formula evaluates to `true`, for any $y < 0$, the formula evaluates to `false`.

An *atomic proposition* is a logical proposition which cannot be divided further. Examples are `true`, `false`, a, $x < y$, but $x < y \wedge a$ is not considered as atomic because it represents the conjunction of a and $x < y$.

Definition 3 *A* Kripke Structure $K = (S, S_0, R, L, AP)$ *is a state transition system* (S, S_0, R) *augmented by a set* AP *of* atomic propositions *and a* labelling function

$$L : S \to 2^{AP}$$

mapping each state s *of* K *to the set of atomic propositions valid in* s*. Furthermore it is required that the transition relation* R *is* total *in the sense that* $\forall s \in S : \exists s' \in S : (s, s') \in R$. □

If a state transition system contains *terminal states*, that is, states $s \in S$ satisfying $\forall s' \in S : (s, s') \notin R$, we can always extend R to a total transition relation $\overline{R}$ suitable for Kripke structures by adding *self loops* to the terminal states in R:

$$\overline{R} = R \cup \{(s, s) \mid s \in S \wedge (\forall s' \in S : (s, s') \notin R)\}$$

State Space of Valuation Functions. Next, we specialise on specification formalisms where the state space can always be defined by a tuple of variables, together with their current values. In this context, a state is a mapping from symbols to current values. The mapping is partial, since the visibility of symbols may depend on scope rules. Let $V = \{x_0, x_1, \ldots\}$ be the set of all variable symbols associated with a specification, a model or a program. For each variable $x \in V$, let D_x denote its type (also called *domain*) comprising all possible values x can assume. We require a special element $\top$ to be contained in each D_x, denoting an undefined variable state, such as an arbitrary input value or a stack variable which is still in an undefined state since no assignments to the variable have been performed so far. Let $D = \bigcup_{x \in V} D_x$ be the union over all domains of variables from V.

A *valuation (function)* is a partial mapping

$$s : V \nrightarrow D$$

which is compatible with the symbol types D_x, in the sense that

$$\forall x \in \text{dom } s : s(x) \in D_x$$

Expression Valuation. Given a valuation function $s : V \nrightarrow D$ and a well-typed expression $e(x_1, \ldots, x_n)$ with free variables $x_i \in V$ we can *evaluate e in state s* by inserting the valuation of each x_i in state s into the expression. This extends the valuation function on variable symbols to well-typed expressions in a natural way:

$$s(e(x_1, \ldots, x_n)) =_{\text{def}} e(s(x_1), \ldots, s(x_n))$$

If $e(x_1, \ldots, x_n)$ is a Boolean expression and $s(e(x_1, \ldots, x_n)) = \texttt{true}$ then we say that $e(x_1, \ldots, x_n)$ *holds in state* s and write

$$s \models e(x_1, \ldots, x_n).$$

Instead of "e holds in state s" we also say that s *is a model for* e.

Kripke Structures With State Spaces of Valuation Functions. In the transition systems and Kripke structures to consider from now on, the state space will always be represented by a set of valuation functions. This has a consequence on the atomic propositions to consider: All information that can be obtained from the fact that a system is in state $s : V \nrightarrow D$ is a consequence from the atomic propositions specifying exactly the valuation of each variable in the current state s, that is,

$$x_0 = s(x_0), x_1 = s(x_1), \ldots \qquad (*)$$

Every other atomic proposition, say, $x_0 < x_1$ can be derived from the propositions (*): For example, $x_0 < x_1$ holds in state s if and only if $s(x_0) < s(x_1)$. For the moment, our sets of atomic propositions will therefore fulfil the subset relation

$$AP \subseteq \{x = d \mid x \in V \wedge d \in D_x\} \qquad (**)$$

Observe, however, that we will also consider other atomic propositions later on in order to avoid the state explosion that would occur if we enumerated AP from (**) for variables x with large data types, such as 32 and 64 bit integers and floats.

The special nature of the atomic propositions from AP in (**) implies that the mapping L can be easily determined for a Kripke structure as soon as their state space, initial state and transition relation is known: Considering (*) and (**), the atomic propositions valid in some state s are obviously

$$L(s) = \{x = d \in AP \mid s(x) = d\} \qquad (***)$$

Note that this choice of atomic propositions and labelling function in a Kripke Structure $\mathcal{K} = (S, S_0, R, L, AP)$ does not introduce any abstraction information about the state transition system $\mathcal{S} = (S, S_0, R)$: the labelling function $(***)$ just asserts that the variables have values as specified by the state valuation functions.

In the more general – and practically relevant – case, the atomic propositions in AP are used to abstract the concrete variable values (like $x = 5$) to more general information items (such as $x < \texttt{threshold}$). The general rule applies, however, that $L(s)$ shall always be consistent to the valuation function s in the sense that

$$\forall s \in S : L(s) = \{p \in AP \mid s \models p\}.$$

This means that $L(s)$ always contains exactly those atomic propositions from AP that evaluate to `true` in s.

First Order Representations. Let ϕ be a first order logical formula, x a free variable in ϕ and ε an expression. Then $\phi[\varepsilon/x]$ denotes the formula which results from replacement of every free occurrence of x by ε. This term replacement can be applied more than once, which is written $\phi[\varepsilon_0/x_0, \varepsilon_1/x_1, \ldots]$; in which case the replacements are applied from left to right.

Let $s \in S$ be a valuation and ϕ a (first order) logical formula with free variables from $V = \{x_0, x_1, \ldots\}$. We say that ϕ *holds in state* s and write

$s \models \phi$, if the formula evaluates to true when replacing every free variable x occurring in ϕ by its valuation $s(x)$; that is, $\phi[s(x_0)/x_0, s(x_1)/x_1, \ldots]$ is a tautology.

Based on this replacement concept, the initial states S_0 of a transition system based on variables and valuations can be specified by means of a first order logical formula I, if S_0 coincides with the set of all valuations where I holds, that is,

$$S_0 = \{s : V \nrightarrow D \mid s \models I\}$$

Conversely, given S_0 and assuming that S_0 is finite, we can always construct such an I by setting

$$I \equiv \bigvee_{s \in S_0} (\bigwedge_{x \in V} x = s(x))$$

If the finiteness assumptions do not hold we can write

$$I \equiv \exists s \in S_0 : \forall x \in V : x = s(x)$$

In analogy, we can specify transition relations by means of first order formulas. In contrast to the initial state formula, however, we now have to consider pre- and post states. Therefore we consider formulas with free variables in V and $V' = \{x' \mid x \in V\}$ and associate unprimed variable symbols x with the prestate and primed variables with the poststate. Let s, s' two valuations and ψ a first-order formula with free variables in V, V'. We say that ψ holds in (s, s') and write $(s, s') \models \psi$ if

$$\psi[s(x_0)/x_0, s(x_1)/x_1, \ldots, s'(x_0)/x'_0, s'(x_1)/x'_1, \ldots]$$

evaluates to true. With this notation a formula T with free variables in V, V' specifies a transition relation $R \subseteq S \times S$ by setting

$$R = \{(s, s') \in S \times S \mid (s, s') \models T\}$$

Conversely, given transition relation R we can construct a suitable formula T by

$$T \equiv \exists (s, s') \in R : \forall x \in V, x' \in V' : x = s(x) \wedge x' = s'(x)$$

Example 3. Consider two parallel processes `P0`, `P1` acting on global variables `s`, `c0`, `c1`. Suppose the processes are executed on a single-core CPU such that each assignment is atomic, but the processes may have to release the CPU between two arbitrary statements.

```
int s  = 0;
int c0 = 0;
int c1 = 0;
```

```
1      P0 {
2       do { s = 0;
3        while ( s == 0 );
4        c0 = 1; // process data
5        c0 = 0;
6       } while (1);
7      }
8
```

```
1      P1 {
2       do { s = 1;
3        while ( s == 1 );
4        c1 = 1; // process data
5        c1 = 0;
6       } while (1);
7      }
8
```

To capture the complete state space, we add two program counters p_0, p_1 in range $\{1, 2, \ldots, 7\}$ indicating the next statement to be executed by `P0`, `P1`, respectively. The semantics of this little parallel program is specified as follows: The symbol set of the parallel system is $V = \{p_0, p_1, s, c_0, c_1\}$ with $p_0, p_1 \in \{1, 2, \ldots, 7\}$, $c_0, c_1, s \in \mathbb{B}$. The initial state is captured by the formula

$$I \equiv p_0 = 1 \wedge p_1 = 1 \wedge s = 0 \wedge c_0 = 0 \wedge c_1 = 0$$

The transition relation is specified by the formula

$$
\begin{aligned}
T \equiv\ & (p_0 = 1 \wedge p_0' = 2 \wedge p_1' = p_1 \wedge s' = s \wedge c_0' = c_0 \wedge c_1' = c_1) \vee \\
& (p_0 = 2 \wedge p_0' = 3 \wedge p_1' = p_1 \wedge s' = 0 \wedge c_0' = c_0 \wedge c_1' = c_1) \vee \\
& (p_0 = 3 \wedge s = 0 \wedge p_0' = 3 \wedge p_1' = p_1 \wedge s' = s \wedge c_0' = c_0 \wedge c_1' = c_1) \vee \\
& (p_0 = 3 \wedge s \neq 0 \wedge p_0' = 4 \wedge p_1' = p_1 \wedge s' = s \wedge c_0' = c_0 \wedge c_1' = c_1) \vee \\
& (p_0 = 4 \wedge p_0' = 5 \wedge p_1' = p_1 \wedge s' = s \wedge c_0' = 1 \wedge c_1' = c_1) \vee \\
& (p_0 = 5 \wedge p_0' = 6 \wedge p_1' = p_1 \wedge s' = s \wedge c_0' = 0 \wedge c_1' = c_1) \vee \\
& (p_0 = 6 \wedge p_0' = 2 \wedge p_1' = p_1 \wedge s' = s \wedge c_0' = c_0 \wedge c_1' = c_1) \vee \\
& (p_1 = 1 \wedge p_1' = 2 \wedge p_0' = p_0 \wedge s' = s \wedge c_1' = c_1 \wedge c_0' = c_0) \vee \\
& (p_1 = 2 \wedge p_1' = 3 \wedge p_0' = p_0 \wedge s' = 1 \wedge c_1' = c_1 \wedge c_0' = c_0) \vee \\
& (p_1 = 3 \wedge s = 1 \wedge p_1' = 3 \wedge p_0' = p_0 \wedge s' = s \wedge c_1' = c_1 \wedge c_0' = c_0) \vee \\
& (p_1 = 3 \wedge s \neq 1 \wedge p_1' = 4 \wedge p_0' = p_0 \wedge s' = s \wedge c_1' = c_1 \wedge c_0' = c_0) \vee \\
& (p_1 = 4 \wedge p_1' = 5 \wedge p_0' = p_0 \wedge s' = s \wedge c_1' = 1 \wedge c_0' = c_0) \vee \\
& (p_1 = 5 \wedge p_1' = 6 \wedge p_0' = p_0 \wedge s' = s \wedge c_1' = 0 \wedge c_0' = c_0) \vee \\
& (p_1 = 6 \wedge p_1' = 2 \wedge p_0' = p_0 \wedge s' = s \wedge c_1' = c_1 \wedge c_0' = c_0)
\end{aligned}
$$

Studying T induces the following intuitive interpretation of the parallel process behaviour.

- Following the single-core CPU paradigm, T expresses an *interleaving semantics*: in each program state, either `P0` or `P1` performs a transition, but never both of them. This is reflected in T by the fact that no disjunct allows p_0 and p_1 to change their value in the same transition.

- The undetermined scheduling strategy is reflected by the *non-determinism* in T: the pre-conditions in several disjuncts may be enabled in the same program state. Only one of the enabled disjuncts will lead to a transition, and the selection is non-deterministic. Practically, this means that we do not know when the – possibly unfair! – scheduler will assign the single CPU core available to `P0` and when to `P1`.

- As a consequence of the non-deterministic scheduling strategy, the program may lead to *starvation* of `P0` or `P1`: if, for example, $p_0 =$

$3 \wedge s = 0$, P0 may perform an active wait transition, where the program counter remains unchanged. Then P1 cannot progress, and T allows the active-wait transition with pre-condition $p_0 = 3 \wedge s = 0$ to be taken infinitely often. As a consequence, P1 cannot progress though it has an enabled transition where it *might* progress.

For representing the associated Kripke structure we use the encoding $\boxed{\pi_0, \pi_1, \sigma, \zeta_0, \zeta_1}$ for a Kripke state s where $L(s) = \{p_0 = \pi_0, p_1 = \pi_1, s = \sigma, c_0 = \zeta_0, c_1 = \zeta_1\}$. For unfolding the Kripke structure from the specification of the transition system we proceed as follows:

1. **Construct the initial states:** This is done by finding all solutions $s : V \not\to D$ of the formula I describing the initial state. In our example this is trivial, since I specifies exactly one admissible initial value for each variable, so S_0 consists just of the one valuation $s_0 = \{p_0 \mapsto 1, p_1 \mapsto 1, s \mapsto 0, c_0 \mapsto 0, c_1 \mapsto 0\}$. In the general case, the set of all valuations s with $s \models I$ has to be constructed. Each initial state s is labelled as described above by $L(s) = \{x_0 = s(x_0), x_1 = s(x_1), \ldots\}$. If the number of variables involved and their data ranges are small this can be done using truth tables for I. For more complex applications more sophisticated methods will be introduced later on.

2. **Expand from the initial states:** Starting with each initial state, expand the Kripke structure by applying the transition relation. This process stops as soon as the expansions of all states generated so far have already been generated before, that is, as soon as the expansion process reaches a *fixed point*. More formally, given a state s which has already been reached by the expansion, we need to construct all solutions of $T[s(x_0)/x_0, s(x_1)/x_1, \ldots]$, that is T, with all pre-state variables replaced by their actual values in s. Every solution s' gives rise to a new Kripke state with $L(s') = \{x_0 = s'(x_0), x_1 = s'(x_1), \ldots\}$.

Let's expand our initial state $\boxed{1,1,0,0,0}$: Replacing the prestate variables in T with these values results in formula

$$\begin{aligned} &T[1/p_0, 1/p_1, 0/s, 0/c_0, 0/c_1] \equiv \\ &\quad (p_0' = 2 \wedge p_1' = 1 \wedge s' = 0 \wedge c_0' = 0 \wedge c_1' = 0) \vee \\ &\quad (p_1' = 2 \wedge p_0' = 1 \wedge s' = 0 \wedge c_1' = 0 \wedge c_0' = 0) \end{aligned}$$

so initial state $\boxed{1,1,0,0,0}$ expands to $\boxed{2,1,0,0,0}$ and $\boxed{1,2,0,0,0}$. The resulting complete Kripke structure for the two interacting processes in this example is shown in Fig. 2.1. Observe that we can also represent the Kripke structure as an infinite tree which is called the *computation tree*. □

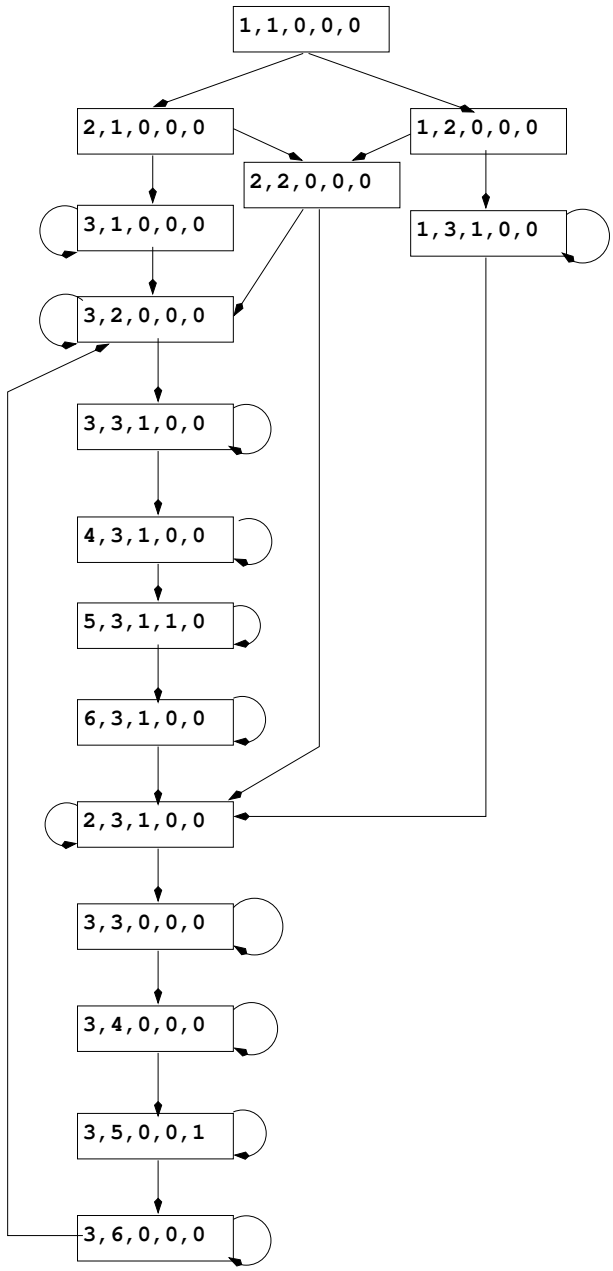


Figure 2.1: Kripke structure for the processes $P_0 \parallel P_1$ from Example 3.

Unwinding the Computation Tree. The following algorithm formalises an unwinding procedure for a finite section of the computation tree associated with a Kripke structure, as illustrated in Example 3. Since a state s may occur in more than one place of the computation tree we use tree nodes labelled by elements of set $N = S \times 2^{AP} \times \mathbb{N}$: $(s, P, n) \in N$ denotes a state $s \in S$ which is inserted as a tree node at level n and has valid atomic propositions $P = L(s)$. The computation tree to be constructed is a structure $TC = (N, \rho, \mathrm{succ}, \mathrm{pred})$ with

- $\rho \in N$ the *root* of the tree
- $\mathrm{succ} : N \to \mathbb{P}(N)$ the successor function mapping each tree node to the set of its children. If $\mathrm{succ}(z) = \varnothing$ then z is called a *leaf* of the tree.
- $\mathrm{pred} : N \to N \cup \{\bot\}$ the predecessor function mapping each node to its parent or – in case of the root node – to $\bot$

The algorithm is shown in Fig. 2.2. It unwinds the computation tree in a manner where a node becomes a leaf if it already occurs elsewhere *on the same path* on a higher level closer to the root. This representation is interesting in the context of test automation (to be discussed in later chapters) and suffices as a simplified model to prove or disprove assertions about the model with are of a certain restricted nature, to be discussed in the next section.

```
function computationTree(in (S, S_0, R, L, AP) : KripkeStructure) : (N, ρ, succ, pred)
begin
  n := 1; M := {(s, L(s), n) | s ∈ S_0}; N := {ρ} ∪ M;
  succ := {ρ ↦ M} ∪ {m ↦ ∅ | m ∈ M};
  pred := {m ↦ ρ | m ∈ M} ∪ {ρ ↦ ⊥}
  while M ≠ ∅ do
    M' := ∅;
    foreach (s, L(s), n) ∈ M do
      foreach s' ∈ S do
        if (s, s') ∈ R then
          N := N ∪ {(s', L(s'), n + 1)};
          succ(s, L(s), n) := succ(s, L(s), n) ∪ {(s', L(s'), n + 1)};
          succ(s', L(s'), n + 1) := ∅;
          pred(s', L(s'), n + 1) := (s, L(s), n);
          if (∀k ∈ {1, ..., n} : pr_1(pred^k(s', L(s'), n + 1)) ≠ s') then
            M' := M' ∪ {(s', L(s'), n + 1)}
          endif
        endif
      enddo
    enddo
    M := M'
    n := n + 1;
  enddo
  computationTree := (N, ρ, succ, pred);
end
```

Figure 2.2: Algorithm for generating a finite portion of the computation tree associated with a Kripke Structure (S, S_0, R, L, AP).

Exercise 2. Consider the specification model of component C in Fig. 2.3. C inputs $x \in \{0, 1, 2\}$ and outputs to $y \in \{-1, 0, 1, 2, \ldots\}$. Its behaviour is modelled in Statechart style: The rounded corner boxes denote *locations*, also called *control states*. Arrows between locations denote *transitions*; a transition arrow without source location marks the initial control state. Expressions in brackets (like `[x > y]`) specify *guard conditions*: The transition from location `l0` to `l1` can only be taken if `x > y` holds, which means, that the current valuation $s : V \not\to D$ results in $s(x) > s(y)$. Expressions after a slash, like `/ y = -1;`, denote *actions*, that is, assignments to internal variables (if any) or outputs. An action is executed if its associated transition is taken.

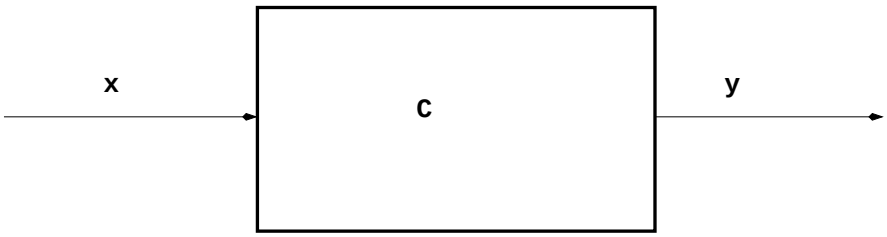
Applying the informal description of the behaviour of C in Example 3, specify the initial state and the transition relation as logical formulas.

□

Exercise 3. Following the algorithm described in Fig. 2.2, draw the initial part of the computation tree associated with the Kripke structure of C in Exercise 2. For the first 3 nodes in the tree, explain how they are derived from the transition relation. For this exercise assume `N = 2`. Use the GraphViz tool[2] to visualise the initial part of the computation tree.

□

[2]Program `dot`, see http://www.graphviz.org/Home.php



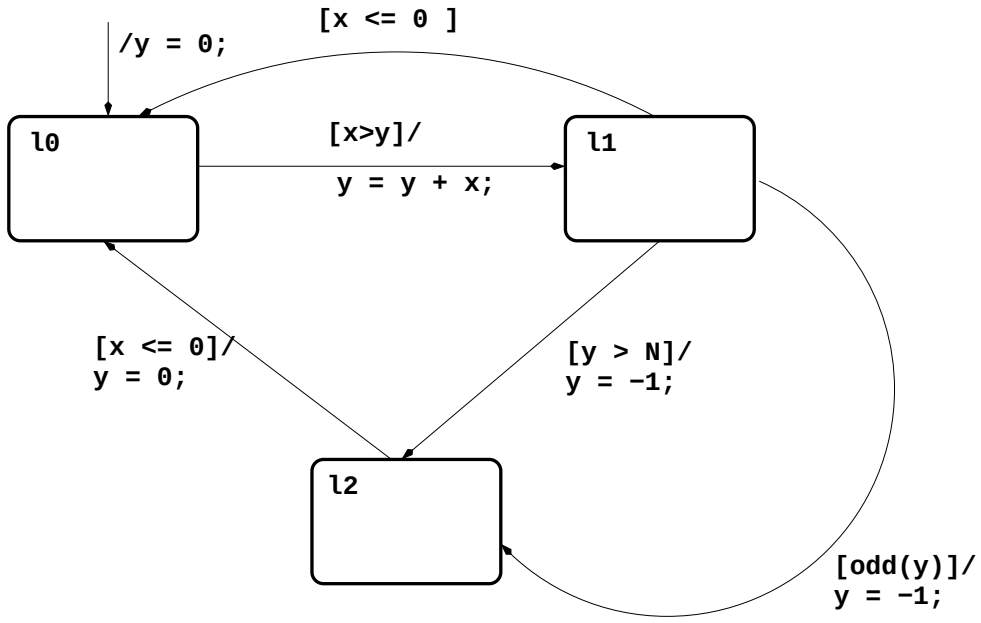


Figure 2.3: Model of component C.

Chapter 3

Property Specification With Temporal Logic

Temporal logic is a logical formalism aiming at the specification of *sequences of system states*, that is, computations. The invention of temporal logic is attributed to Amir Pnueli [11, 12, 15].

3.1 Linear Temporal Logic LTL

Given a Kripke structure $K = (S, S_0, R, L)$ with atomic propositions AP, *linear temporal logic (LTL)* has computations $\pi = s_0.s_1.s_2 \cdots \in S^\omega$ of K as models. An LTL formula φ expresses facts about the propositions that are valid on such a path π. To this end, LTL uses the usual *propositional operators* $\wedge, \vee, \neg, \ldots$ to model "ordinary" propositions and to create new LTL formulas from existing ones. In addition, LTL offers *path operators* (also called *temporal operators*) to express relationships between different states on the path.

The typical approach to defining a logic is to

- specify a syntax subset containing only some "core operators" of the logic,
- specify the semantics of formulas using these core operators, and

- introduce additional "convenience operators" by specifying syntactic equivalences of the formulas using these new operators with formulas using core operators only.

The LTL introduction presented here follows the approach of [17].

Core Operators of LTL. As core operators for propositional formulas we choose $\wedge, \neg$, and as core path operators $\mathbf{X}$ ("next") and $\mathbf{W}$ ("weak until"), whose semantics is explained below.

Core Syntax of LTL. The following syntax rules specify which LTL formulas are well-formed, if they use core operators only.

- Every Boolean constant `true`$(= 1)$ or `false`$(= 0)$ is an LTL formula.
- Every atomic proposition from a set AP is an LTL formula.
- If φ, ψ are well-formed LTL formulas, then

$$\varphi \wedge \psi, \quad \neg\varphi, \quad \mathbf{X}\varphi, \quad \varphi \mathbf{W} \psi$$

  are well-formed LTL formulas.

For displaying these rules more formally in grammar specification style, p denotes arbitrary atomic propositions from AP, and φ, ψ denote arbitrary LTL formulas. Then the syntax rules above can be written equivalently as

$$\text{LTL} ::= \ p \mid \varphi \wedge \psi \mid \neg\varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{W} \psi$$

Semantics of Core LTL Formulas. The semantics of logical formulas is specified by describing the models of each formula. This is typically done by structural induction over the formula syntax. For the core LTL, this is shown in Table 3.1. We use the following notation for paths $\pi = s_0.s_1.s_2 \ldots$: $\pi(i)$ denotes element s_i of π, and $\pi^i = s_i s_{i+1} \ldots$ the i^{th} suffix of π.

Table 3.1: Semantics of LTL formulas.

$\pi^i \models \texttt{true}$		for all $i \geq 0$
$\pi^i \not\models \texttt{false}$		for all $i \geq 0$
$\pi^i \models p$	iff	$p \in L(s_i)$ (This rule applies to all $p \in AP$.)
$\pi^i \models \neg\varphi$	iff	$\pi^i \not\models \varphi$
$\pi^i \models \varphi \wedge \psi$	iff	$\pi^i \models \varphi$ and $\pi^i \models \psi$
$\pi^i \models \mathbf{X}\varphi$	iff	$\pi^{i+1} \models \varphi$
$\pi^i \models \varphi \mathbf{W} \psi$	iff	either $\forall k \geq i : \pi^k \models \varphi$
		or $\exists j \geq i : \pi^j \models \psi$ and $\forall i \leq k < j : \pi^k \models \varphi$

Full LTL Syntax. Further propositional and temporal operators of LTL are introduced via syntactic equivalence. These operators and their equivalent core expressions are displayed in Table 3.2. The names of the new path operators are **G** ("globally"), **F** ("finally"), and **U** ("until").

Table 3.2: LTL operators defined by syntactic equivalence with core LTL expressions.

$\varphi \vee \psi$	$\equiv$	$\neg(\neg\varphi \wedge \neg\psi)$	$\varphi \Rightarrow \psi$	$\equiv$	$\neg\varphi \vee \psi$	$\varphi \Leftrightarrow \psi$	$\equiv$	$(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$
$\mathbf{G}\varphi$	$\equiv$	φ **W** $\texttt{false}$	$\mathbf{F}\varphi$	$\equiv$	$\neg\mathbf{G}\neg\varphi$	$\varphi\mathbf{U}\psi$	$\equiv$	$\varphi\mathbf{W}\psi \wedge \mathbf{F}\psi$

The following lemma is a simple consequence of the LTL semantics specified in Table 3.1.

Lemma 2 *Let* f *be a quantifier-free 1st-order expressions over the variables of a Kripke structure* K*. Then*

$$\pi^i \models f \quad \text{iff} \quad s_i \models f, \quad \text{where} \quad s_i = \pi^i(0) = \pi(i).$$

Proof. For Kripke structures with valuation functions s as states, atomic propositions p are atomic Boolean 1st order expressions over variables, using arithmetic operators and comparision operators. The labelling function fulfils

$$p \in L(s) \quad \text{iff} \quad s \models p,$$

which means that replacing all free variables v in p by their value $s(v)$ results in a true Boolean expression.

A quantifier-free 1st-order expression is a conjunction, disjunction or negation of atomic propositions. Applying the associated rules in Table 3.1 results in the statement of the lemma. □

LTL Formula Transformations. In every logic and every calculus it is important to know different, semantically equivalent, representations of the same formula. Lemma 3 shows a list of some useful equivalences, and the proof explains how these facts are typically derived, using the semantic definitions of the core operators.

Lemma 3 *Let φ, ψ be LTL formulas. Then*

$$\begin{aligned}
\neg(\varphi \wedge \psi) &\equiv \neg\varphi \vee \neg\psi \\
\neg(\varphi \vee \psi) &\equiv \neg\varphi \wedge \neg\psi \\
\neg\mathbf{X}\varphi &\equiv \mathbf{X}\neg\varphi \\
\neg\mathbf{G}\varphi &\equiv \mathbf{F}\neg\varphi \\
\neg(\varphi\mathbf{W}\psi) &\equiv \big(\neg\psi\mathbf{U}\neg(\varphi \vee \psi)\big) \\
\neg(\varphi\mathbf{U}\psi) &\equiv \big(\neg\psi\mathbf{U}\neg(\varphi \vee \psi)\big) \vee \mathbf{G}\neg\psi \\
\mathbf{F}(\varphi\mathbf{U}\psi) &\equiv \mathbf{F}\psi \\
\mathbf{G}(\varphi\mathbf{W}\psi) &\equiv \mathbf{G}(\varphi \vee \psi)
\end{aligned}$$

Proof. (1) We prove $\neg(\varphi \mathbf{W} \psi) \equiv \big(\neg\psi \mathbf{U} \neg(\varphi \vee \psi)\big)$ by transforming the left-hand side and right-hand side into their semantic representation.

$$\begin{aligned}
\pi^i \models \neg(\varphi \mathbf{W} \psi) \quad &\Leftrightarrow \quad \pi^i \not\models \varphi \mathbf{W} \psi \\
&\Leftrightarrow \quad \neg\big(\forall k \geq i : \pi^k \models \varphi\big) \wedge \\
&\qquad \neg\big(\exists j \geq i : (\pi^j \models \psi \wedge \forall i \leq k < j : \pi^k \models \varphi)\big) \\
&\Leftrightarrow \quad \big(\exists h \geq i : \pi^h \not\models \varphi\big) \wedge \\
&\qquad \big(\forall j \geq i : (\pi^j \not\models \psi \vee \exists i \leq k < j : \pi^k \not\models \varphi)\big) \\
&\Leftrightarrow \quad \big(\exists h \geq i : \pi^h \models \neg\varphi\big) \wedge \\
&\qquad \big(\forall j \geq i : (\pi^j \models \neg\psi \vee \exists i \leq k < j : \pi^k \models \neg\varphi)\big) \\
&\Leftrightarrow \quad \big((\exists h \geq i : \pi^h \models \neg\varphi) \wedge (\forall j \geq i : \pi^j \models \neg\psi)\big) \vee \\
&\qquad \big(\exists j \geq i : (\pi^j \models \psi \wedge \forall\, i \leq k < j : \pi^k \models \neg\psi \wedge \exists i \leq h < j : \pi^h \models \neg\varphi)\big) \\
&\Leftrightarrow \quad \big(\exists h \geq i : (\pi^h \models \neg\varphi \wedge \neg\psi \wedge \forall i \leq k < h : \pi^k \models \neg\psi)\big) \\
&\Leftrightarrow \quad \pi^i \models \big(\neg\psi \mathbf{U} \neg(\varphi \vee \psi)\big)
\end{aligned}$$

(2) Now we prove $\mathbf{F}(\varphi \mathbf{U} \psi) \equiv \mathbf{F}\psi$; the technique is the same as used in step (1).

$$\begin{aligned}
\pi^i \models \mathbf{F}(\varphi \mathbf{U} \psi) \quad &\Leftrightarrow \quad \big(\exists j \geq i : \pi^j \models \varphi \mathbf{U} \psi\big) \\
&\Leftrightarrow \quad \big(\exists k \geq j \geq i : (\pi^k \models \psi \wedge \forall\, j \leq h < k : \pi^h \models \varphi)\big) \\
&\Leftrightarrow \quad \big(\exists k \geq i : \pi^k \models \psi\big) \\
&\Leftrightarrow \quad \pi^i \models \mathbf{F}\psi
\end{aligned}$$

(3) Now we prove $\mathbf{G}(\varphi \mathbf{W} \psi) \equiv \mathbf{G}(\varphi \vee \psi)$ and exploit the other equivalences that are expressed in the lemma. This allows us to perform a proof on a purely syntactic level.

$$\begin{aligned}
\pi^i \models \mathbf{G}(\varphi \mathbf{W} \psi) \quad &\equiv \quad \pi^i \not\models \neg\mathbf{G}(\varphi \mathbf{W} \psi) \\
&\equiv \quad \pi^i \not\models \mathbf{F}\neg(\varphi \mathbf{W} \psi) \\
&\equiv \quad \pi^i \not\models \mathbf{F}(\neg\psi \mathbf{U} \neg(\varphi \vee \psi)) \\
&\equiv \quad \pi^i \not\models \mathbf{F}\neg(\varphi \vee \psi) \\
&\equiv \quad \pi^i \not\models \neg\mathbf{G}(\varphi \vee \psi) \\
&\equiv \quad \pi^i \models \mathbf{G}(\varphi \vee \psi)
\end{aligned}$$

□

The next lemma shows *recursive* equivalences. These are very useful in the context of bounded model checking which will be introduced in Chapter 5.

Lemma 4 *Let* φ, ψ *be LTL formulas. Then*

$$\begin{aligned} \mathbf{G}\varphi &\equiv \varphi \wedge \mathbf{XG}\varphi \\ \mathbf{F}\varphi &\equiv \varphi \vee \mathbf{XF}\varphi \\ \varphi \mathbf{U} \psi &\equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)) \end{aligned}$$

Proof. (1) We prove $\mathbf{G}\varphi \equiv \varphi \wedge \mathbf{XG}\varphi$ by transforming the formulas into LTL core syntax and then applying their semantic definitions. The left-hand side of the equivalence is transformed into a predicate of first order logic as follows.

$$\begin{aligned} \pi^i \models \mathbf{G}\varphi &\equiv \pi^i \models \varphi \mathbf{W} \mathtt{false} \\ &\equiv \forall k \geq i : \pi^k \models \varphi \end{aligned}$$

Now the right-hand side formula is transformed as follows.

$$\begin{aligned} \pi^i \models \varphi \wedge \mathbf{XG}\varphi &\equiv (\pi^i \models \varphi) \wedge (\pi^{i+1} \models \mathbf{G}\varphi) \\ &\equiv (\pi^i \models \varphi) \wedge (\pi^{i+1} \models \varphi \mathbf{W} \mathtt{false}) \\ &\equiv (\pi^i \models \varphi) \wedge (\forall k \geq i+1 : \pi^k \models \varphi) \\ &\equiv \forall k \geq i : \pi^k \models \varphi \end{aligned}$$

This proves the first equivalence, because both sides have been transformed into the same first order predicates. The other equivalences are handled in Exercise 4. □

Exercise 4. Prove the equivalences

$$\begin{aligned} \mathbf{F}\varphi &\equiv \varphi \vee \mathbf{XF}\varphi \\ \varphi \mathbf{U} \psi &\equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \psi)) \end{aligned}$$

specified in Lemma 4. □

Positive Normal Form. Every LTL formula can be equivalently represented by a formula in *postive normal form (PNF)*; the latter adhere to the syntax

$$\text{PNF} ::= \texttt{true} \mid \texttt{false} \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U} \varphi_2 \mid \varphi_1 \mathbf{W} \varphi_2$$

This is a direct consequence of Lemma 3.

Propositional LTL (pLTL). In some situations, it is useful to abstract from state valuation functions $s : V \not\to D$. Instead, states are just uninterpreted elements of some state space S. The "interesting" information about a Kripke structure over S is encoded alone by means of atomic propositions $p \in AP$ and the labelling function $L : S \to 2^{AP}$. As models of an LTL formula φ we now consider infinite sequences of sets of atomic propositions, that is

$$\pi \in \left(2^{AP}\right)^{\omega}$$

and say that π^i is a model for atomic proposition $p \in AP$ (again written as $\pi^i \models p$) if and only if $p \in \pi(i)$. Then the other semantic rules of Table 3.1 can be re-used without changes.

3.2 The Computation Tree Logic CTL*

The temporal logics discussed in the remainder of this chapter are distinguished from LTL by the fact that their models are *arbitrary* Kripke structures, whereas LTL uses linear models, that is, infinite computation paths, only. As a consequence, the new logics introduced below consider branching situations on computations paths, and these are modelled by *trees*. This motivates the name *Computation Tree Logic*, abbreviated by CTL, that is always part of the specific logic's name.

Operators. CTL* formulas are based on the following operators:

- The *path quantifiers* are
  - **A** ("on every path")

  - **E** ("there exists a path")

- The *temporal operators* are
  - **X** ("next time")
  - **G** ("globally" or "always")
  - **F** ("eventually" or "finally")
  - **U** ("until")
  - **R** ("release")

Apart from these new operators the conventional Boolean operators can be used, as will be specified in the syntax definition below.

Syntax of CTL* formulas. CTL* distinguishes between

- *state formulas* which refer to properties of a specific Kripke state
- *path formulas* which specify properties of a path in the computation tree.

State and path formulas refer recursively to each other. The set of all valid CTL* formulas is given by the *state* formulas generated according to the following inductive rules:

1. Every atomic proposition $p \in AP$ is a state formula.
2. If f and g are state formulas then $\neg f, f \wedge g, f \vee g$ are state formulas.
3. If f is a *path formula* then **E** f, **A** f are *state formulas*.

The path formulas are defined according to the following rules:

(iv) Every state formula is also a path formula.

(v) If f and g are path formulas, then $\neg f, f \wedge g, f \vee g$ are path formulas.

(vi) If f and g are path formulas, then **X** f, **F** f, **G** f, f **U** g, f **R** g are path formulas.

More formally, we can write these syntax rules in EBNF notation as follows, where $p \in AP$, ϕ denotes state formulas and ψ denotes path formulas

CTL*-formula $::= \phi$
$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{E}\ \psi \mid \mathbf{A}\ \psi$
$\psi ::= \phi \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\ \psi \mid \mathbf{F}\ \psi \mid \mathbf{G}\ \psi \mid \psi\ \mathbf{U}\ \psi \mid \psi\ \mathbf{R}\ \psi$

Semantics of CTL* formulas. The semantics of CTL* formulas is explained using a Kripke structure M, specific states s of M and paths π through the computation tree of M. We write

$$M, s \models \phi \ (\phi \text{ a state formula})$$

to express that ϕ holds in state s of M. We write

$$M, \pi \models \psi (\psi \text{ a path formula})$$

to express that ψ holds along path π through M. For CTL* formulas ϕ we say ϕ *holds in the Kripke model* M and write

$$M \models \phi$$

if and only if $\forall s_0 \in S_0 : M, s_0 \models \phi$. Recall that for paths $\pi = s_0.s_1.s_2\ldots$, $\pi(i)$ denotes element s_i of π, and $\pi^i = s_i.s_{i+1}\ldots$ the i^{th} suffix of π.

The inductive definition of $\models$ is given in Fig. 3.1, where p denotes atomic propositions from AP, ϕ, ϕ_i denote state formulas and ψ, ψ_j denote path formulas:

Exercise 5. Using the syntax rules of CTL* formulas and a syntax tree representation, prove or disprove that the following formulas conform to the CTL*-syntax ($a, b, c \in AP$):

1. $\mathbf{AG}(\mathbf{XF}a \wedge \neg(b\mathbf{U}\mathbf{G}c))$
2. $\mathbf{AXG}\neg a \wedge \mathbf{EFG}(a \vee \mathbf{A}(b\mathbf{U}a))$

□

$M, s \models p$	$\equiv$	$p \in L(s)$
$M, s \models \neg\phi$	$\equiv$	$M, s \not\models \phi$
$M, s \models \phi_1 \vee \phi_2$	$\equiv$	$M, s \models \phi_1$ or $M, s \models \phi_2$
$M, s \models \phi_1 \wedge \phi_2$	$\equiv$	$M, s \models \phi_1$ and $M, s \models \phi_2$
$M, s \models \mathbf{E}\,\psi$	$\equiv$	there is a path π from s such that $M, \pi \models \psi$
$M, s \models \mathbf{A}\,\psi$	$\equiv$	on every path π from s holds $M, \pi \models \psi$
$M, \pi \models \phi$	$\equiv$	$M, \pi(0) \models \phi$
$M, \pi \models \neg\psi$	$\equiv$	$M, \pi \not\models \psi$
$M, \pi \models \psi_1 \vee \psi_2$	$\equiv$	$M, \pi \models \psi_1$ or $M, \pi \models \psi_2$
$M, \pi \models \psi_1 \wedge \psi_2$	$\equiv$	$M, \pi \models \psi_1$ and $M, \pi \models \psi_2$
$M, \pi \models \mathbf{X}\,\psi$	$\equiv$	$M, \pi^1 \models \psi$
$M, \pi \models \mathbf{F}\,\psi$	$\equiv$	there exists $k \geq 0$ such that $M, \pi^k \models \psi$
$M, \pi \models \mathbf{G}\,\psi$	$\equiv$	For all $k \geq 0$ $M, \pi^k \models \psi$
$M, \pi \models \psi_1 \mathbf{U} \psi_2$	$\equiv$	there exists $k \geq 0$ such that $M, \pi^k \models \psi_2$ and for all $0 \leq j < k$ $M, \pi^j \models \psi_1$
$M, \pi \models \psi_1 \mathbf{R} \psi_2$	$\equiv$	for all $j \geq 0$ holds: if $M, \pi^i \not\models \psi_1$ for every $i < j$ then $M, \pi^j \models \psi_2$

Figure 3.1: Semantics of CTL* formulas.

Exercise 6. Using the Kripke structure displayed in Fig. 2.1 prove or disprove the following CTL*-assertions, using the semantic definition described in Fig. 3.1 in a step-by step manner. For each of the formulas, give a textual interpretation of their meaning.

1. $\mathbf{AG}\neg(c_0 \wedge c_1)$
2. $\mathbf{A}(\mathbf{F}c_0 \wedge \mathbf{G}(c_0 \Rightarrow \mathbf{F}(c_1 \wedge \mathbf{F}c_0)))$

Justify why the first assertion could be proved on the finite representation of the Kripke structure's computation tree as explained in algorithm 2.2 while this is not possible for the second assertion. □

3.3 The Computation Tree Logic CTL

A frequently used subset of CTL* is called CTL. It is defined by the following restricted syntactic rule (CTL.vi) for the path formulas (the other rules (i), (ii), (iii) for CTL* syntax apply in the same way to CTL):

(CTL.vi) If f and g are *state formulas* then $\mathbf{X}\ f, \mathbf{F}\ f, \mathbf{G}\ f, f\ \mathbf{U}\ g, f\ \mathbf{R}\ g$ are path formulas.

More formally, the CTL syntax is defined by (p denotes atomic propositions from AP)

$$\text{CTL-formula} ::= \phi$$
$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{E}\ \psi \mid \mathbf{A}\ \psi$$
$$\psi ::= \mathbf{X}\ \phi \mid \mathbf{F}\ \phi \mid \mathbf{G}\ \phi \mid \phi\ \mathbf{U}\ \phi \mid \phi\ \mathbf{R}\ \phi$$

As a consequence, the temporal operators $\mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}$ can never be prefixed by another temporal operator in CTL. Only pairs consisting of path quantifier and temporal operator can occur in a row.

Example 4. The CTL* formula $\mathbf{A}(\mathbf{FG}f)$ (*On every path, f will finally hold in all states*) has no equivalent in CTL. □

Theorem 1 *Every CTL formula can be expressed by means of the operators* $\neg, \vee, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}$.

Proof. Obviously $\phi_1 \wedge \phi_2$ can be expressed as $\neg(\neg\phi_1 \vee \neg\phi_2)$. The theorem now follows from the fact that the following equivalences hold for all CTL state formulas ϕ, ϕ_1, ϕ_2:

1. $\mathbf{AX}\phi \equiv \neg\mathbf{EX}(\neg\phi)$
2. $\mathbf{EF}\phi \equiv \mathbf{E}(\mathtt{true}\mathbf{U}\phi)$
3. $\mathbf{AG}\phi \equiv \neg\mathbf{EF}(\neg\phi)$
4. $\mathbf{AF}\phi \equiv \neg\mathbf{EG}(\neg\phi)$
5. $\mathbf{A}(\phi_1\mathbf{U}\phi_2) \equiv \neg\mathbf{E}(\neg\phi_2\mathbf{U}(\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg\mathbf{EG}\neg\phi_2$
6. $\mathbf{A}(\phi_1\mathbf{R}\phi_2) \equiv \neg\mathbf{E}(\neg\phi_1\mathbf{U}\neg\phi_2)$
7. $\mathbf{E}(\phi_1\mathbf{R}\phi_2) \equiv \neg\mathbf{A}(\neg\phi_1\mathbf{U}\neg\phi_2)$
8. $\mathbf{E}\phi \equiv \mathbf{E}(\mathtt{false}\mathbf{U}\phi)$ if ϕ does not contain $\mathbf{E}, \mathbf{A}, \mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}$
9. $\mathbf{A}\phi \equiv \neg\mathbf{E}(\mathtt{false}\mathbf{U}\neg\phi)$ if ϕ does not contain $\mathbf{E}, \mathbf{A}, \mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}$

The proof of these equivalences is performed using the semantic rules given in Fig. 3.1, to be performed by the reader in Exercise 7. □

Exercise 7. Prove the 9 semantic equivalences used in the proof of Theorem 1. □

3.4 The Computation Tree Logics ACTL* and ACTL

If we restrict CTL* formulas to universal quantification only, the resulting computation tree logic is called ACTL*. More precisely, ACTL* only admits CTL* formulas satisfying

- The formula is in *positive normal form*, that is, the negation operator $\neg$ is only applied to atomic propositions.
- The only occurring path quantifier is $\mathbf{A}$.

The corresponding restriction of CTL formulas to universal quantification is called ACTL.

Example 5. $\mathbf{AFAX}a$ is an ACTL formula, but $\mathbf{AGEF}a$ is not in ACTL*, since its $\mathbf{E}$-free representation $\mathbf{AG}\neg\mathbf{AG}\neg a$ is not in positive normal form. □

In Section 7.4 we will prove a theorem about simulation relations between Kripke structures, and the properties that may be transferred from an abstract Kripke structure to its associated concrete one. It will turn out that a sufficient condition for this implication from abstract to concrete level is for the formula to be in the subset of ACTL* or ACTL, respectively.

3.5 Safety Properties and Over-approximation of LTL Safety Violation Formulae by CTL

The material presented in this section is based on Peleska et al. [14].

3.5.1 Safety Properties

A *safety property* P is a set of computations $\pi \in S^\omega$, such that for every $\pi' \in S^\omega$ with $\pi' \notin P$, the fact that π' does *not* fulfil P can already be

decided on a finite prefix of π'. This is specified more formally in the following definition.

Definition 4 (Safety Property) *A* safety property $P \subseteq S^\omega$ *is a set of computations satisfying*

$$\forall \pi' \in S^\omega \setminus P \,.\, \big(\exists \pi'' \,.\, \pi'' \leq \pi' \wedge (\forall \tau \in S^\omega \,.\, \pi''.\tau \notin P)\big).$$

Intuitively speaking, this definition states that any computation π' *violating* the safety property P (that is, $\pi' \in S^\omega \setminus P$) has a finite prefix π'' such that no infinite continuation τ of π'' can ever be safe (that is, an element of P) again.

Sistla has shown that every safety property P can be characterised by a *Safety LTL* formula φ, so that the computations in P are exactly those fulfilling φ. The Safety LTL formulae are specified as follows [17, Theorem 3.1]:

1. Every unquantified first-order formula is a Safety LTL-formula.
2. If φ, ψ are Safety LTL-Formulae, then so are

$$\varphi \wedge \psi, \quad \varphi \vee \psi, \quad \mathbf{X}\varphi, \quad \varphi \mathbf{W} \psi, \quad \mathbf{G}\varphi.$$

Observe that in these safety formulae, the negation operator must only occur in first-order sub-formulae.

Suppose that a safety property P is specified by Safety LTL formula φ. When looking for a path π *violating* φ, the violation $\pi \models_{\mathrm{LTL}} \neg\varphi$ can be equivalently expressed by a formula containing only first-order expressions composed by the operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$. This is shown in the following theorem.

Theorem 2 *Let φ be a Safety LTL formula. Then* safety violation $\neg\varphi$ *can be equivalently expressed using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$.*

Proof. We use structural induction over the syntax of safety LTL formulae.

Base case. If φ is a first-order expression, then its negation is again a first-order expression.

Induction hypothesis. Suppose that the negation of Safety LTL formulae φ, ψ can be expressed using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ only.

Induction step. Since every Safety LTL formula can be expressed using operators $\wedge, \vee, \mathbf{X}, \mathbf{W}, \mathbf{G}$, we need to show that the negations of $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathbf{X}\varphi$, $\varphi \mathbf{W} \psi$, $\mathbf{G}\varphi$ can also be expressed using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$. To prove this, we use the equivalences for LTL formulae established in Lemma 3.

Case $\varphi \wedge \psi$. Since $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ and, according to the induction hypothesis, φ, ψ can be negated using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ only, the induction step holds for operator $\wedge$.

Case $\varphi \vee \psi$. Since $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ and φ, ψ can be negated using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ only, the induction step holds for operator $\vee$.

Case $\mathbf{X}\varphi$. Since $\neg\mathbf{X}\varphi \equiv \mathbf{X}\neg\varphi$ and φ can be negated using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ only, the induction step holds for operator $\mathbf{X}$.

Case $(\varphi \mathbf{W} \psi)$. Since $\neg(\varphi \mathbf{W} \psi) \equiv \big(\neg\psi \mathbf{U} \neg(\varphi \vee \psi)\big) \equiv \big(\neg\psi \mathbf{U} (\neg\varphi \wedge \neg\psi)\big)$ and φ, ψ can be negated using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ only, the induction step holds for operator $\mathbf{W}$.

Case $\mathbf{G}\varphi$. Since $\neg\mathbf{G}\varphi \equiv \mathbf{F}\neg\varphi \equiv (\mathtt{true}\mathbf{U}\neg\varphi)$ and φ can be negated using first-order expressions composed by operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ only, the induction step holds for operator $\mathbf{G}$. This completes the proof. □

As a consequence of Theorem 2, a model checker specialised on the detection of safety violations only needs to support the evaluation of first-order formulae and operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$.

Full LTL and CTL have different expressiveness, and neither one is able to express all formulae of the other with equivalent semantics [6]. In

this section, however, it will be shown that any safety violation specified by an LTL formula f on a path π can also be detected by applying CTL model checking to a translated formula $\Phi(f)$ on any Kripke structure K containing π as a computation. This is, however, an *over-approximation*, in the sense that witnesses for $\Phi(f)$ in K will not always correspond to "real" rule violations in the IXL configuration. An algorithm for identifying false alarms is presented in Section 3.5.4.

3.5.2 A Transformation from LTL Safety Violation Formulae to CTL

Recalling from Theorem 2 that any safety violation can be specified using first-order formulae and operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$, we specify a partial transformation function $\Phi : \text{LTL} \not\to \text{CTL}$ as follows.

$$\begin{aligned}
\Phi(f) &= f \text{ for all first-order expressions } f \\
\Phi(f \wedge g) &= \Phi(f) \wedge \Phi(g) \\
\Phi(f \vee g) &= \Phi(f) \vee \Phi(g) \\
\Phi(\mathbf{X}f) &= \mathbf{EX}(\Phi(f)) \\
\Phi(f\mathbf{U}g) &= \mathbf{E}(\Phi(f)\mathbf{U}\Phi(g))
\end{aligned}$$

3.5.3 Model Checking of LTL Safety Formulae by CTL Over-Approximation

Observe that Φ maps every LTL formula in its domain to a CTL state formula, since first-order expressions are state-formulae, and any LTL formula starting with a temporal operator is prefixed under Φ with the existential path quantifier $\mathbf{E}$. With this transformation at hand, the following theorem states that the absence of witnesses for $\Phi(f)$ in K guarantees that f cannot be fulfilled on π.

Theorem 3 *Let π be any finite path and f an LTL formula specifying a safety violation on π. Let K be a Kripke structure over state space S containing π as a computation. Then*

$$\pi \models_{\text{LTL}} f \quad \textit{implies} \quad K, \pi(0) \models_{\text{CTL}} \Phi(f).$$

Proof. The proof uses structural induction over the syntax of LTL formulae representing safety violations. These are expressed by first-order formulae and operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$ according to Theorem 2. Throughout the proof, let $k = |\pi| - 1$ be the last valid index of $\pi = \pi(0) \dots \pi(|\pi| - 1)$ and $\pi^i = \pi(i).\pi(i+1).\pi(i+2) \dots \pi(k)$ be an arbitrary path segment of π with $0 \leq i \leq k$.

Base case. Suppose that $\pi^i \models_{\text{LTL}} g$ for an arbitrary first-order expression g. According to Lemma 2, this is equivalent to $\pi(i) \models g$. Since π is a computation of K by assumption, π^i is a path segment of K. Since the evaluation rules for first-order expressions are the same in LTL and CTL, $K, \pi(i) \models_{\text{CTL}} g$ follows. This argument was independent on the value of $0 \leq i \leq k$. Therefore, we can conclude from $\pi = \pi^0$ that $\pi \models_{\text{LTL}} f$ implies $K, \pi(0) \models_{\text{CTL}} f$ for any first-order expression f, which concludes the base case.

Induction hypothesis. Suppose that $\pi^i \models_{\text{LTL}} f$ and $\pi^i \models_{\text{LTL}} g$ imply $K, \pi(i) \models_{\text{CTL}} \Phi(f)$ and $K, \pi(i) \models_{\text{CTL}} \Phi(g)$, respectively, for given LTL formulae f, g expressing safety violations and any path segment π^i with $0 \leq i \leq k$.

Induction step. Using the induction hypothesis, it has to be shown that $\pi^i \models_{\text{LTL}} f \wedge g$, $\pi^i \models_{\text{LTL}} f \vee g$, $\pi^i \models_{\text{LTL}} \mathbf{X}f$, and $\pi^i \models_{\text{LTL}} f\mathbf{U}g$ imply that $K, \pi(i) \models_{\text{CTL}} \Phi(f) \wedge \Phi(g)$, $K, \pi(i) \models_{\text{CTL}} \Phi(f) \vee \Phi(g)$, $K, \pi(i) \models_{\text{CTL}} \mathbf{EX}\Phi(f)$, and $K, \pi(i) \models_{\text{CTL}} \mathbf{E}(\Phi(f)\mathbf{U}\Phi(g))$, respectively.[1]

Case $\pi^i \models_{\text{LTL}} f \wedge g$. This case is equivalent to $\pi^i \models_{\text{LTL}} f$ and $\pi^i \models_{\text{LTL}} g$ according to the LTL semantics specified in Table 3.1. According to the induction hypothesis, this implies $K, \pi(i) \models_{\text{CTL}} \Phi(f)$ and $K, \pi(i) \models_{\text{CTL}} \Phi(g)$. According to the CTL semantics specified in Table 3.1, this is in turn equivalent to $K, \pi(i) \models_{\text{CTL}} \Phi(f) \wedge \Phi(g)$.

Case $\pi^i \models_{\text{LTL}} f \vee g$. This case is shown in analogy to the previous case.

Case $\pi^i \models_{\text{LTL}} \mathbf{X}f$. This case is equivalent to $\pi^{i+1} \models_{\text{LTL}} f$ according to the LTL semantics specified in Table 3.1. According to the induction

[1] Recall that we do not have to consider negation, since this only occurs inside first-order formulae.

hypothesis, this implies $K, \pi(i+1) \models_{CTL} \Phi(f)$. From the definition of Φ we know that $\Phi(f)$ is a state formula. Therefore, $\mathbf{EX}\Phi(f)$ is again a CTL state formula. From the CTL semantics in Table 3.1 and from the fact that $K, \pi(i+1) \models_{CTL} \Phi(f)$ has been established, we can derive $K, \pi(i) \models \mathbf{EX}\Phi(f)$. Now, by definition of Φ, we have $\mathbf{EX}\Phi(f) = \Phi(\mathbf{X}f)$, and this proves the current case.

Case $\pi^i \models_{LTL} f\mathbf{U}g$. This case is equivalent to

$$\exists 0 \leq j \,.\, \left(\pi^{i+j} \models_{LTL} g \wedge \forall 0 \leq \ell < j \,.\, \pi^{i+\ell} \models_{LTL} f\right)$$

according to the LTL semantics specified in Table 3.1. The induction hypothesis implies

$$\exists 0 \leq j \,.\, \left(K, \pi(i+j) \models_{CTL} \Phi(g) \wedge \forall 0 \leq \ell < j \,.\, K, \pi(i+\ell) \models_{CTL} \Phi(f)\right) \qquad (*)$$

according to the induction hypothesis. Since $\Phi(f), \Phi(g)$ are state formulae, $\Phi(f)\mathbf{U}\Phi(g)$ is a path formula, and the CTL semantics specified in Table 3.1 shows that (*) implies $K, \pi^i \models_{CTL} \Phi(f)\mathbf{U}\Phi(g)$. As a consequence, $K, \pi(i) \models_{CTL} \mathbf{E}(\Phi(f)\mathbf{U}\Phi(g))$ holds as well. This completes the induction step and the proof of Theorem 3. □

Now Theorem 3 can be applied to perform fast CTL-based checks to prove that a violation of a given LTL safety property cannot exist. This is performed according to the following steps.

1. Let $\varphi \in$ LTL be a safety property. We wish to show that $K \models_{LTL} \varphi$. This means that every computation π of K fulfils $\pi \models_{LTL} \varphi$.

2. $K \models_{LTL} \varphi$ is equivalent to the fact that *no* computation π of K satisfies $\pi \models_{LTL} \neg\varphi$.

3. From Theorem 2 we know that $\neg\varphi$ can be expressed by means of operators $\wedge, \vee, \mathbf{X}, \mathbf{U}$.

4. From Theorem 3 we know that $\pi \models_{LTL} \neg\varphi$ for any computation π of K implies $K, \pi(0) \models_{CTL} \phi(\neg\varphi)$.

5. Now we apply CTL model checking on K for property $\phi(\neg\varphi)$.

6. If $\phi(\neg\varphi)$ does *not* hold for K, we can conclude that *no* path π violating φ can exist in K, so K fulfils LTL safety property φ.

7. If $\phi(\neg\varphi)$ holds in K, we need to check whether the solution is also a witness for LTL formula $\neg\varphi$. If this is the case, we have found a path proving that the model violates the safety formula. If this is not the case, we apply an LTL model checker (which requires significantly more effort than the CTL checker) and check whether φ holds.

3.5.4 False Alarms

The following example shows how the CTL over-approximation for checking LTL formulae on non-linear models may lead to false alarms.

Example 6. Consider the transition graph of a Kripke structure $K(s_0)$ sketched in Fig. 3.2 with root node s_0 and atomic propositions p, q. It is fictitious, but this graph pattern might well occur in an IXL sub-model with driving direction $s_0 \longrightarrow s_1$, where node s_2 represents a point.

Each node in Fig. 3.2 is annotated with the propositions fulfilled in the corresponding Kripke-state. For example, s_1 satisfies p but not q, s_4 fulfils p and q, and s_3 satisfies neither p nor q.

Suppose we wish to prove the absence of a witness for LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$. Applying the checking approach described above, the formula is translated to CTL as $\Phi((\mathbf{X}p)\ \mathbf{U}\ q) = \mathbf{E}((\mathbf{EX}p)\ \mathbf{U}\ q)$.

The model fulfils $K(s_0) \models_{CTL} \mathbf{E}((\mathbf{EX}p)\ \mathbf{U}\ q)$, because the path $\pi = s_0.s_1.s_2.s_3.s_4\ldots$ fulfils $(\mathbf{EX}p)\ \mathbf{U}\ q$. This is true because the states s_0, s_1, s_2, s_3 each fulfil $\mathbf{EX}p$, and in s_4, proposition q is fulfilled. Note that in state s_2, formula $\mathbf{EX}p$ holds because the outgoing path $s_2.s_5.s_6\ldots$ fulfils $\mathbf{X}p$. Path π, however, is not a witness for the LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$, since $s_2.s_3.s_4\cdots \not\models_{LTL} \mathbf{X}p$. Also for path $\pi' = s_0.s_1.s_2.s_5.s_6\ldots$, the LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$ is not fulfilled, because s_6 neither fulfils p nor q.

Summarising, the CTL-based model checking approach yields a false alarm when trying to prove the absence of a witness for LTL formula $(\mathbf{X}p)\ \mathbf{U}\ q$. □

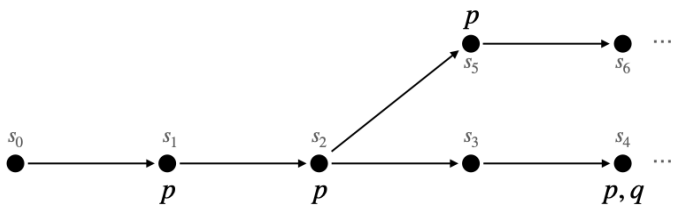


Figure 3.2: Model fulfilling $\mathbf{E}((\mathbf{EX}\mathrm{p})\ \mathbf{U}\ \mathrm{q})$ but not $(\mathbf{X}\mathrm{p})\ \mathbf{U}\ \mathrm{q}$.

Chapter 4

CTL Model Checking

Variants of model checking. Model checking distinguishes between

- *Equivalence checking.* Two models (these are usually given in state transition system or labelled transition system representation) are compared with respect to semantic equivalence.
- *Refinement checking.* Two models are compared by means of a (usually transitive) relation which is weaker than equivalence.
- *Property checking.* A model is checked with respect to *an (implicit) specification*: The specification is given by a logical formula stating some desired property of the model. The model is usually represented as a transition system or as a Kripke structure $K = (S, S_0, R, L, AP)$. The specification is most frequently expressed by a temporal logic formula ϕ; an alternative specification formalism is *trace logic*.

In the most general case of property checking, we wish to identify all states $s \in S$ where ϕ holds, i. e., $s \models \phi$. In most practical applications the objective is to prove that ϕ holds in every initial state of the model; the notation for this fact has been introduced before as $K \models \phi$, defined by

$$\forall \phi \in CTL : (K \models \phi \Leftrightarrow (\forall s \in S_0 : s \models \phi))$$

Model checking techniques. The technique which is introduced here is called *explicit model checking* because it requires to represent the Kripke

structure's state space in an explicit way, so that all the necessary atomic propositions of the form $x = v$ can be directly derived from each state's representation. This is the oldest form of model checking which is only applicable if state spaces are sufficiently small to be enumerated explicitly.

Explicit model checking is a variant of *global model checking*; the latter term is used when *every* state of a model is checked with respect to fulfilment of a given property. In contrast to that, *local model checking* only investigates whether a property holds in a specific state. Instead of explicit model representations in memory, it is also possible to present model states by means of logical formulas; the most popular variant of this technique uses *Ordered Binary Decision Diagrams (OBDDs)* [5, Chapter 5]. This variant of global model checking is called *symbolic model checking* [5, Chapter 6], because of its Boolean formula representation of the state space.

Another variant of symbolic model checking is *bounded model checking (BMC)*. Its purpose is to perform local model checking, but BMC investigates the validity of a formula in the neighbourhood of a given state only. As a consequence, no global model representations (whether explicit or symbolic) are required, but results are only partial: if a solution is found, this result would also have been found when applying global explicit model checking; if, however, no solution is found, there might still exist a solution beyond the state's neighbourhood which has been investigated. BMC is studied in more detail in Chapter 5.

The basic idea of the property checking algorithm. The property checking algorithm introduced formally below is based on the following concept:

- The CTL specification formula is decomposed into its (binary) syntax tree.
- Starting at the leaves of the syntax tree (the leaves represent atomic propositions), the algorithm processes a sequence of sub-formulas ϕ_i in bottom-up manner. This is implemented by means of a recursive in-order traversal of the syntax tree.

- The goal of each processing step is to annotate all states s satisfying $s \models \phi_i$ with the new sub-formula ϕ_i. To this end, a labelling function $L_\phi : S \to 2^{CTL}$ is used.
- The algorithm stops when the last formula ϕ_i having been processed coincides with the specification ϕ.
- The result of the algorithm is the set $S_\phi =_{def} \{s \in S \mid \phi \in L_\phi(s)\}$.
- The Kripke model (S, S_0, R, L, AP) satisfies ϕ if its initial states are part of S_ϕ, that is,

$$(S, S_0, R, L, AP) \models \phi \equiv S_0 \subseteq \{s \in S \mid \phi \in L_\phi(s)\}$$

Syntax tree representation of CTL formulas. From Section 3.3 we know that every CTL formula can be represented by means of the operators $\neg, \vee, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}$ alone. The binary syntax tree representation of such a formula can be defined recursively using the tree notation

- ε: empty tree
- $T(t_0, n, t_1)$: tree with root n and left sub-tree t_0 and right sub-tree t_1.

The recursive syntax tree definition $t(\phi)$ for a given CTL formula ϕ is as follows:

1. If $\phi \in AP$ then $t(\phi) = T(\varepsilon, \phi, \varepsilon)$.
2. If $\phi = \neg\phi_1$ then $t(\phi) = T(\varepsilon, \neg, t(\phi_1))$.
3. If $\phi = \phi_0 \vee \phi_1$ then $t(\phi) = T(t(\phi_0), \vee, t(\phi_1))$.
4. If $\phi = \mathbf{EX}\phi_1$ then $t(\phi) = T(\varepsilon, \mathbf{EX}, t(\phi_1))$.
5. If $\phi = \mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ then $t(\phi) = T(t(\phi_0), \mathbf{EU}, t(\phi_1))$ [1].

[1] We regard $\mathbf{EU}$ as a binary operator, so that formulas $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ could be equivalently written as $(\phi_0(\mathbf{EU})\phi_1)$. As a consequence its tree representation is $T(t(\phi_0), \mathbf{EU}, t(\phi_1))$

6. If $\phi = \mathbf{EG}\phi_1$ then $t(\phi) = T(\varepsilon, \mathbf{EG}, t(\phi_1))$.

Given a tree representation $t(\phi)$ of a formula ϕ, its leaves (i. e. its atomic propositions) can be extracted by means of the function leaves : Tree $\rightarrow 2^{AP}$ by means of the following recursive definition:

1. $\text{leaves}(T(\varepsilon, \phi, \varepsilon)) = \{\phi\}$
2. $\text{leaves}(T(\varepsilon, \neg, t(\phi_1))) = \text{leaves}(t(\phi_1))$
3. $\text{leaves}(T(t(\phi_0), \vee, t(\phi_1))) = \text{leaves}(t(\phi_0)) \cup \text{leaves}(t(\phi_1))$
4. $\text{leaves}(T(\varepsilon, \mathbf{EX}, t(\phi_1))) = \text{leaves}(t(\phi_1))$
5. $\text{leaves}(T(t(\phi_0), \mathbf{EU}, t(\phi_1))) = \text{leaves}(t(\phi_0)) \cup \text{leaves}(t(\phi_1))$
6. $\text{leaves}(T(\varepsilon, \mathbf{EG}, t(\phi_1))) = \text{leaves}(t(\phi_1))$

Overview over the algorithm. In Fig. 4.1, the entry function of the recursive algorithm is shown. *checkCTL* returns the set $\{s \in S \mid \phi \in \text{label}(s)\}$ of all states satisfying the given formula ψ. It remains to check whether the initial states S_0 of the Kripke Structure K form a subset of $\{s \in S \mid \phi \in \text{label}(s)\}$.

---

function *checkCTL*(**in** (S, S_0, R, L, AP) : KripkeStructure; **in** ϕ : CTL) : $\mathbb{P}(S)$
begin
  label : $S \rightarrow 2^{CTL}$;
  label := $\{s \mapsto \{\texttt{true}\} \mid s \in S\}$;
  calcLabel$((S, S_0, R, L, AP), \phi, \text{label})$;
  checkCTL := $\{s \in S \mid \phi \in \text{label}(s)\}$;
end

Figure 4.1: Main algorithm for CTL property checking against Kripke structures.

In Fig. 4.2, the main function *calcLabel* of the algorithm is shown. It traverses the syntax tree representation of the formula ψ to be checked and calls recursively itself or special sub-functions for processing sub-formulas. Atomic propositions, negation, disjunction, and $\mathbf{EX}$-formulas are handled directly by *calcLabel*; formulas containing operators $\mathbf{EU}$ and $\mathbf{EG}$ are processed by sub-functions specialised for this purpose.

In the algorithm of Fig. 4.8 SCC denotes a set of *strongly connected components*, that is, maximal subgraphs C of S' such that every node in C is reachable from every other node in C by a path contained entirely in C. We require that every C is *nontrivial*, that is, C contains either more than one node or it contains one node with a self-loop. The classical algorithm for identifying SCCs in a given graph has been developed by Tarjan [18].

```
procedure calcLabel(in (S, S_0, R, L, AP) : KripkeStructure;
                    in φ : CTL;
                    inout label : S → 2^CTL)
begin
  if φ ∈ AP then
    calcLabelAP((S, S_0, R, L, AP), φ, label);
  elseif t(φ) = T(ε, ¬, t(φ_1)) then
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelNE((S, S_0, R, L, AP), φ_1, label);
  elseif t(φ) = T(t(φ_0), ∨, t(φ_1)) then
    calcLabel((S, S_0, R, L, AP), φ_0, label);
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelOR((S, S_0, R, L, AP), φ_0, φ_1, label);
  elseif t(φ) = T(ε, EX, t(φ_1)) then
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelEX((S, S_0, R, L, AP), φ_1, label);
  elseif t(φ) = T(t(φ_0), EU, t(φ_1)) then
    calcLabel((S, S_0, R, L, AP), φ_0, label);
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelEU((S, S_0, R, L, AP), φ_0, φ_1, label);
  elseif t(φ) = T(ε, EG, t(φ_1)) then
    calcLabel((S, S_0, R, L, AP), φ_1, label);
    calcLabelEG((S, S_0, R, L, AP), φ_1, label);
  endif
end
```

Figure 4.2: Label calculation – control algorithm driven by formula syntax.

```
procedure calcLabelAP(in (S, S_0, R, L, AP) : KripkeStructure;
                      in p : AP;
                      inout label : S → 2^CTL)
begin
  foreach s ∈ S do
      if p ∈ L(s) then
        label(s) := label(s) ∪ {p};
      endif
  enddo
end
```

Figure 4.3: Algorithm for labelling states with atomic propositions.

```
procedure calcLabelNE(in (S, S_0, R, L, AP) : KripkeStructure;
                      in φ_1 : CTL;
                      inout label : S → 2^CTL)
begin
  foreach s ∈ S do
      if φ_1 ∉ label(s) then
        label(s) := label(s) ∪ {¬φ_1};
      endif
  enddo
end
```

Figure 4.4: Algorithm for labelling states with negated formulas $\neg\phi_1$.

```
procedure calcLabelOR(in (S, S_0, R, L, AP) : KripkeStructure;
                      in φ_0 : CTL; in φ_1 : CTL;
                      inout label : S → 2^CTL)
begin
  foreach s ∈ S do
      if φ_0 ∈ label(s) ∨ φ_1 ∈ label(s) then
        label(s) := label(s) ∪ {φ_0 ∨ φ_1};
      endif
    enddo
end
```

Figure 4.5: Algorithm for labelling states with $\phi_0 \vee \phi_1$ formulas.

```
procedure calcLabelEX(in (S, S_0, R, L, AP) : KripkeStructure;
                      in φ_1 : CTL;
                      inout label : S → 2^CTL)
begin
  foreach s ∈ S do
      if ∃s' ∈ S : R(s, s') ∧ φ_1 ∈ label(s') then
        label(s) := label(s) ∪ {EXφ_1};
      endif
  enddo
end
```

Figure 4.6: Algorithm for labelling states with $\mathbf{EX}\phi_1$ formulas.

```
procedure calcLabelEU (in (S, S_0, R, L, AP) : KripkeStructure;
                       in φ_0 : CTL; in φ_1 : CTL;
                       inout label : S → 2^CTL)
begin
  // Define T as sequence of states (in arbitrary order) fulfilling φ_1
  T := ⟨s ∈ S | φ_1 ∈ label(s)⟩;
  foreach s ∈ T do
    label(s) := label(s) ∪ {E(φ_0 U φ_1)};
  enddo
  while T ≠ ε do
    s := head(T);
    T := tail(T);
    foreach u ∈ {v ∈ S | R(v, s)} do
      if E(φ_0 U φ_1) ∉ label(u) ∧ φ_0 ∈ label(u) then
        label(u) := label(u) ∪ {E(φ_0 U φ_1)};
        // Append state u to sequence T
        T := T.u;
      endif
    enddo
  enddo
end
```

Figure 4.7: Algorithm for labelling states with $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$ formulas.

```
procedure calcLabelEG(in (S, S_0, R, L, AP) : KripkeStructure;
                      in φ_1 : CTL;
                      inout label : S → 2^CTL)
begin
  S' := {s ∈ S | φ_1 ∈ label(s)};
  SCC := {C | C is a nontrivial SCC of S'}
  T := ⟨s | ∃C ∈ SCC : s ∈ C⟩;
  foreach s ∈ T do
    label(s) := label(s) ∪ {EGφ_1};
  enddo
  while T ≠ ε do
    s := head(T);
    T := tail(T);
    foreach u ∈ {v ∈ S' | R(v, s)} do
      if EGφ_1 ∉ label(u) then
        label(u) := label(u) ∪ {EGφ_1};
        // Append state u to sequence T
        T := T.u;
      endif
    enddo
  enddo
end
```

Figure 4.8: Algorithm for labelling states with $\mathbf{EG}\phi_1$ formulas.

Theorem 4 *The time complexity of CTL model checking is*

$$O((|S| + |R|) \cdot |\Phi|),$$

where $|S|$ *is the number of states in the underlying Kripke structure* $K = (S, S_0, R, L, AP)$, $|R|$ *is the number of transitions, and* $|\Phi|$ *is the size of the CTL formula* Φ *to bechecked, that is, the number of operators contained in* Φ.

Exercise 8. Give an semi-formal argument for the validity of Theorem 4. □

Exercise 9. Consider again the Kripke structure specified by the tran-

sition graph shown in Figure 4.9. It is based on the processes $P_0 \parallel P_1$ from Example 3 with the same variables, but now it is assumed that the scheduler is fair so that starvation cannot occur. The objective of this exercise is to explain by means of a manual exercise how the classical CTL model checking algorithm introduced above works. To this end, analyse the following CTL formulas

- $\mathbf{AGAF}c_0$
- $\mathbf{AGAF}c_1$
- $\mathbf{EG}(c_1 = 0 \vee (s = 0 \wedge c_0 = 0))$

and perform the following tasks.

1. Explain the meaning of each formula in natural language.
2. For the first and third formula, produce a manual illustration of the model checking algorithm as follows.
   - Transform the formula into the standard form according to Theorem 1, which is accepted by the model checking algorithm.
   - Draw the formula tree.
   - Explain how the model checking algorithm traverses the formula tree.
     - Explain which function is called in each step, including the recursions.
     - Whenever a function call terminates (so all of its recursive sub-calls have terminated), draw a new version of the graph and annotate the nodes with the new (atomic or non-atomic) formulas that have found to be valid (if any).

   The last drawing of the graph should mark each node where the complete formula holds.

□

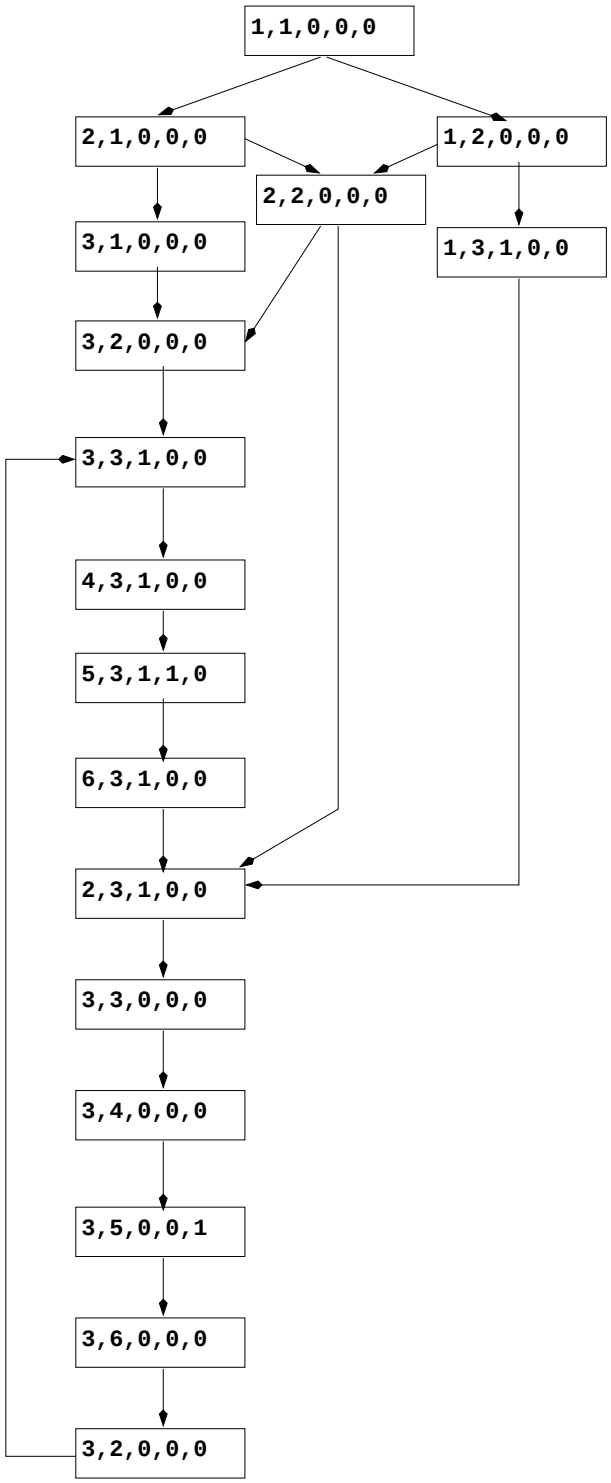


Figure 4.9: Kripke structure for Exercise 9.

Chapter 5

Bounded Model Checking

5.1 Motivation

The applicability and efficiency of property checking techniques as the one described in Chapter 4 depends on the possibility to create an *explicit, global* representation of the Kripke structure in memory. As a consequence, models with very large state spaces cannot be handled by explicit model checking, and it is *a priori* impossible to check models with infinite state spaces.

In contrast to that, *bounded model checking (BMC)* is a *symbolic, local* model checking technique that aims at finding a witness for the validity of some property $s \models \varphi$ in the vicinity of a state s only. States are represented by propositions, and possible transitions between states are represented by the model's transition relation in propositional form. As a consequence, also infinite models can be represented symbolically by means of finite propositions.

The introduction of BMC is due to [2, 3].

5.2 BMC Instances

Definition of BMC instances. Bounded model checking investigates solutions of propositions of the form

$$\mathrm{bmc} \equiv J(s_0) \wedge \bigwedge_{i=1}^{k} \Phi(s_{i-1}, s_i) \wedge G(s_0, \ldots, s_k) \tag{5.1}$$

These are called *BMC instances*. For solving the BMC instance in formula (5.1), one has to find model states $s_0, \ldots, s_k$ making bmc evaluate to `true`. Since every state s_i is a mapping from variable symbols $v \in V$ to current values $s_i(v) \in D_v$, the BMC instance (5.1) represents a formula with

$$(k+1) \cdot \#V \quad \text{unknown quantities} \quad s_i(v), i = 0, \ldots, k, v \in V$$

to be fixed when solving bmc. The formula components of bmc are interpreted as follows.

The conjunct $J(s_0)$ specifies possible start states s_0 in whose neighbourhood solutions of formula (5.1) should be found. In this chapter, s_0 does not necessarily denote an initial state of the underlying Kripke structure $K = (S, S_0, R, L)$, but any state that is suitable for starting the search for a solution. J is a proposition with free variables in V, the variable set of K's state space $S \subseteq (V \to D)$. The notation $J(s_0)$ is short for

$$J(s_0) \equiv J[s_0(v)/v \mid v \in V]$$

that is, proposition J with every free variable v exchanged by its value $s_0(v)$ in state s_0. Note that $J(s_0)$ may identify more than one state s_0 as a suitable start state, if predicate J admits more than one solution for variables $v \in V$ and their domains D_v.

Conjunct $\bigwedge_{i=1}^{k} \Phi(s_{i-1}, s_i)$ specifies the *unrolling of the transition relation* Φ. Proposition Φ represents K's transition relation as described in Chapter 1. As described there, Φ is a proposition with free variables in V and V', the unprimed variables denoting pre-states of a transition specified by Φ and the primed variables $v' \in V'$ denoting the post-states. Notation $\Phi(s_{i-1}, s_i)$ specifies the application of Φ to pre-state s_{i-1} and post-state s_i.

$$\Phi(s_{i-1}, s_i) \equiv \Phi[s_{i-1}(v)/v, s_i(v)/v' \mid v \in V]$$

denotes the proposition ϕ, with every unprimed variable v replaced by its value in state s_{i-1} and every primed variable replaced by its value in state s_i. Intuitively speaking, the conjunct $\bigwedge_{i=1}^{k} \Phi(s_{i-1}, s_i)$ ensures that every solution $s_0 \ldots s_k$ of a BMC instance (5.1) is a valid computation fragment of the model K: each pair of consecutive states s_{i-1}, s_i is connected by the transition relation.

Conjunct $G(s_0, \ldots, s_k)$ is the *verification goal* of the BMC instance (5.1). It is a proposition over all unknown quantities $s_i(v), i = 0, \ldots, k, v \in V$. It describes a desired or an unwanted property of the model in the vicinity of states satisfying $J(s_0)$, to be fulfilled by a computation fragment $s_0 \ldots s_k$ of model K.

Summarising, a BMC instance (5.1) specifies a Boolean problem over unknown quantities $s_i(v), i = 0, \ldots, k, v \in V$ which can be informally expressed as follows.

> *Find a valid computation fragment* $s_0 \ldots s_k$ *of model* K *that starts in a state satisfying* J *and satisfies goal* G.

Desired and undesired model properties. If G represents an unwanted model property, a solution of BMC instance (5.1) uncovers a modelling error. Conversely, if we have a property P that should always be fulfilled by states satisfying J, we try to solve a BMC instance with $G \equiv \neg P$. Again, a solution of this instance uncovers a modelling error, namely the violation of P. Therefore BMC is very suitable for *bug finding*. The global verification of assertions, however, is more difficult, since not finding a solution for a given k does not necessarily mean that we cannot find one for $k' > k$. Global solution techniques will be discussed later in this chapter.

In model-based testing, BMC instances may specify *test cases*, that is, valid computation fragments of the model where a specific test objective can be investigated [13]. For these types of applications G expresses a desired model property.

Any solution $s_0, \ldots, s_k$ of a BMC instance bmc is called a *witness* of bmc.

False alarms and related terms. If G represents an unwanted model property, a solution of BMC instance (5.1) may represent a *false alarm*, if the start state s_0 of the solution is unreachable from any initial state $\bar{s} \in S_0$. False alarms are also called *false positive*; this term is borrowed from the field of medicine, where a "positive" outcome of a medical examination means that the patient is affected by the disease under consideration. Conversely, a "negative" result of a medical examination indicates that the patient is not affected by the disease. This leads to the term *false negative*, if finding no solution of a BMC instance (5.1) with unwanted property G indicates that G is never fulfilled when starting from states fulfilling J, but in truth solutions could be found for higher values of k.

Solution techniques. Solutions of BMC instances (5.1) can be generated by SAT solvers if all model variables $v \in V$ are typed as Booleans. Otherwise, an SMT solver is required to handle other data types and associated operators for the transformation of variable values. Today's SMT-solvers typically support Booleans, integers with bit vector or integer arithmetic, and arrays thereof. Some solvers already support floating point arithmetic with associated operations and transcendent functions.[1]

Complexity considerations. Finding solutions of BMC instances (5.1) has worst-case complexity $O(2^k)$, because every new unrolling step k of the transition relation introduces a whole new set of unknown quantities $s_k(v), v \in V$. Various SMT-solving techniques, however, ensure that this worst case is not encountered too often.

5.3 LTL Property Specifications on Finite Traces

While the previous chapter described explicit global model checking against CTL properties, bounded model checking is typically performed against LTL properties. This is because BMC investigates computation fragments

[1] See, for example, http://www.informatik.uni-bremen.de/agbs/florian/sonolar/

in the vicinity of a given start state. The semantics of LTL formulas as specified in Section 3.1 has (infinite) computations as models. In BMC, however, only finite computation fragments are investigated. Therefore an alternative semantic description of LTL is required that

- allows to decide the validity of $s \models \varphi$ on a *finite* computation fragment, and
- is consistent with the original semantics introduced on infinite computations.

This semantics has been introduced in [3] by defining the *bounded semantics* of formulas φ with *fixpoint evaluation encoding* $|[M, \varphi, k]|$.

An encoding $|[M, \varphi, k]|$ consists of three parts:

- *Model constraints* $|[M]|_k$,
- *Loop constraints* $|[\text{LoopConstraints}]|_k$, and
- LTL formula translations $|[\varphi]|_0^k$ to propositions G.

It is defined by

$$|[M, \varphi, k]| \equiv |[M]|_k \wedge |[\text{LoopConstraints}]|_k \wedge |[\varphi]|_0^k$$

Model constraints $|[M]|_k$ encode legal initialised finite traces of the model M with length k:

$$|[M]|_k \equiv J(s_0) \wedge \bigwedge_{i=1}^{k} \Phi(s_{i-1}, s_i)$$

The

$$|[\text{LoopConstraints}]|_k \wedge |[\varphi]|_0^k$$

specifies the propositional encoding $G(s_0, \ldots, s_k)$ of the verification goal which has originally been defined by an LTL formula. The encoding rules are explained below in Section 5.4. As a result, $|[M, \varphi, k]|$ corresponds to a BMC instance as specified in formula (5.1).

5.4 Finite Trace Semantics for LTL Formulas – the Fixpoint Evaluation Encoding

Loop constraints. As will become apparent below, the bounded semantics of LTL formulas evaluated on a path segment $\pi = s_0 \dots s_k$ depends on the fact whether s_k is a *lasso state*, meaning that $s_k = s_{j-1}$ holds for some $0 < j \leq k$ (see Fig. 5.1). This consideration induces the *loop constraints* specified in Table 5.1, where $\ell_j = 1$ for some $0 < j \leq k$ if and only if $s_k = s_{j-1}$. Note that $\ell_k = 1$ denotes the situation where $s_k = s_{k-1}$, that is, path segment π ends in a self loop emanating from state s_k. InLoop_i is true if s_i is in the loop part of the trace. The loop selectors $\ell_0, \dots, \ell_k$ determine where the path loops, if ℓ_j is true then the path has the loop part $s_j \dots . s_k$. At most one loop selector is allowed to be true. When no ℓ_i is true, then the trace is a no-loop case. In the k loop case, LoopExists will be true and in the no-loop case it will be false.

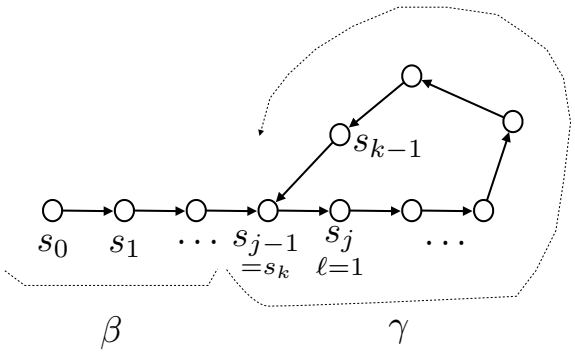


Figure 5.1: Lasso-shaped computation fragment $s_0 \dots s_k$.

Table 5.1: Loop Constraints

Base	$\ell_0 \Leftrightarrow 0$ $\text{InLoop}_0 \Leftrightarrow 0$
$1 \leq i \leq k$	$\ell_i \Rightarrow (s_{i-1} = s_k)$ $\text{InLoop}_i \Leftrightarrow \text{InLoop}_{i-1} \vee \ell_i$ $\text{InLoop}_{i-1} \Rightarrow \neg\ell_i$ $\text{LoopExists} \Leftrightarrow \text{InLoop}_k$

Proposition $|[\text{LoopConstraints}]|_k$ denotes the conjunction of the constraints listed in Table 5.1, such that $|[\text{LoopConstraints}]|_k$ always evaluates to `true`. This enforces consistent assignments of ℓ_i, InLoop_i, and LoopExists both in presence and in absence of loops.

Transformation rules. The following tables specify the translation rules $|[\varphi]|_0$ of LTL formulas φ into propositions G. Table 5.2 shows the translation of LTL formulas that are propositions without any temporal operators. Consistent with our intuition, propositions evaluate to `true` in the computation segment $s_i \ldots s_k$, if and only if they evaluate to `true` in state s_i.

Table 5.2: Translation of propositional LTL formulas.

$\|[\varphi]\|_i$	$0 \leq i \leq k$
$\|[p]\|_i$	$p \in L(s_i)$
$\|[\neg p]\|_i$	$p \notin L(s_i)$
$\|[\psi_1 \wedge \psi_2]\|_i$	$\|[\psi_1]\|_i \wedge \|[\psi_2]\|_i$
$\|[\psi_1 \vee \psi_2]\|_i$	$\|[\psi_1]\|_i \vee \|[\psi_2]\|_i$

For formulas φ containing path operators $\mathbf{X}, \mathbf{U}, \mathbf{R}$, the translation of $|[\varphi]|_i$ depends on the cases $0 \leq i < k$ and $i = k$, as specified in Ta-

ble 5.3. The auxiliary translation operators $\langle\langle \psi_1 \mathbf{U} \psi_2 \rangle\rangle_j$ and $\langle\langle \psi_1 \mathbf{R} \psi_2 \rangle\rangle_j$ are needed to terminate the translation process in the case where $|[\psi_1 \mathbf{U} \psi_2]|_i$ or $|[\psi_1 \mathbf{R} \psi_2]|_i$ do *not* hold on $s_0 \dots s_k$: as defined in Table 5.4, the auxiliary operators terminate with result `false`, if $i = k$ is reached and $|[\psi_2]|_k$ does not hold.

Table 5.3: Translation rules for path operators $\mathbf{X}, \mathbf{U}, \mathbf{R}$.

$\lvert[\varphi]\rvert_i$	$0 \leq i < k$	$i = k$
$\lvert[\mathbf{X}\psi]\rvert_i$	$\lvert[\psi]\rvert_{i+1}$	$\bigvee_{j=1}^{k} (\ell_j \wedge \lvert[\psi]\rvert_j)$
$\lvert[\psi_1 \mathbf{U} \psi_2]\rvert_i$	$\lvert[\psi_2]\rvert_i \vee (\lvert[\psi_1]\rvert_i \wedge \lvert[\psi_1 \mathbf{U} \psi_2]\rvert_{i+1})$	$\lvert[\psi_2]\rvert_i \vee (\lvert[\psi_1]\rvert_i \wedge (\bigvee_{j=1}^{k} (\ell_j \wedge \langle\langle \psi_1 \mathbf{U} \psi_2 \rangle\rangle_j)))$
$\lvert[\psi_1 \mathbf{R} \psi_2]\rvert_i$	$\lvert[\psi_2]\rvert_i \wedge (\lvert[\psi_1]\rvert_i \vee \lvert[\psi_1 \mathbf{R} \psi_2]\rvert_{i+1})$	$\lvert[\psi_2]\rvert_i \wedge (\lvert[\psi_1]\rvert_i \vee (\bigvee_{j=1}^{k} (\ell_j \wedge \langle\langle \psi_1 \mathbf{R} \psi_2 \rangle\rangle_j)))$

Table 5.4: Specification of the auxiliary translation operators $\langle\langle \psi_1 \mathbf{U} \psi_2 \rangle\rangle_j$ and $\langle\langle \psi_1 \mathbf{R} \psi_2 \rangle\rangle_j$.

$\langle\langle \varphi \rangle\rangle_i$	$0 \leq i < k$	$i = k$
$\langle\langle \psi_1 \mathbf{U} \psi_2 \rangle\rangle_i$	$\lvert[\psi_2]\rvert_i \vee (\lvert[\psi_1]\rvert_i \wedge \langle\langle \psi_1 \mathbf{U} \psi_2 \rangle\rangle_{i+1})$	$\lvert[\psi_2]\rvert_k$
$\langle\langle \psi_1 \mathbf{R} \psi_2 \rangle\rangle_i$	$\lvert[\psi_2]\rvert_i \wedge (\lvert[\psi_1]\rvert_i \vee \langle\langle \psi_1 \mathbf{R} \psi_2 \rangle\rangle_{i+1})$	$\lvert[\psi_2]\rvert_k$

Exercise 10. Give a definition of the bounded semantics of $\psi_1 \mathbf{W} \psi_2$ in analogy to the specifications for $\mathbf{U}, \mathbf{R}$ in Table 5.3 and Table 5.4 □

Exercise 11. In Example 3, apply the bounded semantics of LTL to prove the existence of a finite computation segment $s_0 \dots s_k$ starting in the initial state and satisfying

$$\varphi \equiv \big((p_0 < 3) \mathbf{U} (\mathbf{G}(2 \leq p_0 \leq 6 \wedge 2 \leq p_1 \leq 6))\big)$$

To achieve this, proceed as follows.

1. Construct a computation segment $s_0 \dots s_k$ satisfying the formula by hand, using the transition graph representation of the interaction of parallel processes `P0`, `P1` in Fig. 2.1.

2. Now *prove* that your path satisfies the formula by creating the BMC instance according to Formula (5.1). To this end,

   - Prove that each pair of consecutive states in your path are related by the transition relation specified in Example 3. Do this for the first 5 states of your segment only.
   - Translate φ according to the bounded semantics for the appropriate k which is needed to find the solution. Since the resulting formula $|[\varphi]|_0$ is quite long for the value of k which is needed, you may reduce $|[\varphi]|_0$ by dropping disjuncts that will not be fulfilled by the solution. This leads to a sub-formula which implies $|[\varphi]|_0$, and which is solved by the solution $s_0 \dots s_k$.

□

Biere et al. [3, Theorem 3.1] have stated and proven the relationship between witnesses of LTL formulae in finite fixpoint encoding and witnesses of LTL formulae in the "usual" interpretation on infinite paths as follows.

Theorem 5 *Given a Kripke structure* K *and an LTL formula* ψ*. Then the following statements are equivalent.*

1. *There exists a computation (i.e. an initialised infinite path)* π *of* K *such that* $\pi \models \psi$ *in the LTL semantics specified in Table 3.1.*

2. *There exists* $k \in \mathbb{N}$ *such that* $|[K, \psi, k]|$ *is satisfiable.*

Moreover, $|[K, \psi, k]|$ *is satisfiable if* $|[\psi]|_0$ *holds on the finite computation prefix* $\pi(0) \dots \pi(k)$ *of some computation* π *of* K. □

5.5 Verifying Global Properties With BMC

BMC verifies local properties in the neighbourhood of given states, investigating all states that are reachable from there by means of a bounded number of transitions. We will now investigate the question how to use BMC to *global* model properties in models with potentially large, but finite state spaces.

5.5.1 k-Induction

An alternative to exploring the vicinity of a state until the recurrence diameter has been reached is the so-called k-*induction* originally introduced in [16]. The crucial algorithm presented there is specified in Fig. 5.2. It is specialised on proving safety conditions of the form

$$\varphi \equiv \mathbf{G}P \quad \text{where} \quad P \quad \text{is a first order predicate}$$

This algorithm uses the following notation. Proposition P is the safety property whose invariant validity is to be proved or disproved by the algorithm for every path starting in an initial state. For predicate α, $\mathrm{Sat}(\alpha)$ is the Boolean return value of a SAT or an SMT solver. If the return value is `true`, the solver has found a solution, and as a side effect, the solution is stored in the the sequence $c_{[0..i]} = c_0 \dots c_i$ of state vectors. Every vector c_i is indexed over all variables in $V = \{v_1, \dots, v_n\}$, and its value c_{ij} corresponds to the i^{th} state valuation function value $s_i(v_j)$ for variable v_j. $\mathrm{Taut}(\alpha)$ returns `true` if and only if α is a tautology, that is, it evaluates to `true` for all possible assignments of its free variables. With an ordinary SAT or SMT solver, $\mathrm{Taut}(\alpha)$ is evaluated by proving that $\mathrm{Sat}(\neg\alpha)$ does not have any solution.

Proposition $I(s_0)$ is a proposition characterising the initial states, as explained in the previous chapters. Proposition $\mathrm{path}(s_{[0..i]})$ states that state sequence $s_{[0..i]} = s_0 \dots s_i$ consists of neighbouring states, each pair connected by the transition relation.

$$\mathrm{path}(s_{[0..i]}) \equiv \bigwedge_{0 \le j < i} \Phi(s_j, s_{j+1})$$

```
function kInd(k : ℕ0; I : InitialCondition; Φ : TransitionRelation;
              out tr : Trace) : 𝔹
begin
  i = k;
  while true do
    if Sat(¬(I(s0) ∧ path(s[0..i]) ⇒ all.P(s[0..i]))) then
      tr = Trace(c[0..i]);
      return false;
    endif
    if Taut((all.¬I(s[1..i+1]) ∧ loopFree(s[0..i+1])) ⇒ ¬I(s0)) ∨
       Taut(loopFree(s[0..i+1]) ∧ all.P(s[0..i]) ⇒ P(si+1)) then
      return true;
    endif
    i = i + 1;
  enddo
end
```

Figure 5.2: k-induction algorithm.

Predicate $\text{all.}\alpha(s_{[0..i]})$ states that α holds in every state $s_0 \dots s_i$,

$$\text{all.}\alpha(s_{[0..i]}) \equiv \bigwedge_{0 \le j \le i} \alpha(s_j)$$

Predicate $\text{loopFree}(s_{[0..i+1]})$ states that the sequence $s_0 \dots s_{i+1}$ is a cycle free path.

$$\text{loopFree}(s_{[0..i+1]}) \equiv \text{path}(s_{[0..i+1]}) \wedge \bigwedge_{0 \le j < r \le i+1} s_j \neq s_r$$

Algorithm kInd() operates as follows.

1. The while-loop terminates if
   (a) a path of length i exists which begins in an initial state and violates the invariant P in at least one state (this is the case where the Sat-condition of the first if-command evaluates to `true`), or
   (b) all loop-free paths starting in an initial state are shorter than $i + 1$ (this is the first disjunct of the second if-condition), or
   (c) if any extension of any loop free path segment of length i which satisfies P in every state $s_0 \dots s_i$ will also satisfy P in s_{i+1} (this is the second disjunct of the second if-condition).
2. In termination case (a) a violation of the safety property has been detected: there exists a path of length i, starting in an initial state, such that P is violated somewhere on this path. Observe that if $k > 0$, $Sat(\neg(I(s_0) \wedge \text{path}(s_{[0..i]}) \Rightarrow \text{all.P}(s_{[0..i=k]}))$ is also fulfilled if P is violated in some s_j with $s < k$, because
$$\neg\text{all.P}(s_{[0..i=k]}) \equiv \bigvee_{j=0}^{i} \neg P(s_j)$$
3. Since i is incremented in every cycle of the while-loop, any violation of P in a reachable state s will finally be found, as soon as the value of i equals the shortest reachable path from an initial state to s. This statement is true, provided that the loop's second if-condition only evaluates to `true` if no reachable state can violate P.

4. In case (b) the condition evaluates to `true` if there is no path starting in an initial state which is loop free, never returns to an initial state, and has a length of $i + 1$. Since the first if-condition has already checked *all* initialised paths of length i and did not detect any violation of P so far, we can terminate and confirm the validity of P in every reachable state.

5. In case (c) the condition evaluates to `true` if all loop-free traces of length $i + 1$ satisfy $P(s_{i+1})$ if $P(s_j)$ already holds for $j = 0, 1, \ldots, i$. Observe that the traces under consideration do *not* necessarily start in an initial state, but may begin in arbitrary system states. This is necessary for the induction step: it allows us to "move forward" on any initialised trace $s_{[0..i]}$ satisfying P in every state to a trace $s_{[1..i+1]}$ which also satisfies P everywhere. This principle can be continued ad infinitum, so validity of P is proven on reachable states in any distance from an initial state, and the termination is justified.

6. If both disjuncts of if-condition 2 evaluate to `false`, two cases may occur.

   (a) A trace $s_{[0..i+1]}$ where P holds from s_0 to s_i, but not anymore in s_{i+1} and which starts in an initial state has been found. This will lead to termination in the first if-condition of the next loop cycle, and the violation of the safety invariant will be indicated by means of a solution of $(\text{loopFree}(s_{[0..i+1]}) \wedge \text{all.P}(s_{[0..i]}) \wedge \neg P(s_{i+1}))$.

   (b) A loop-free trace $s_{[0..i+1]}$ where P holds from s_0 to s_i, but not anymore in s_{i+1} has been found, but this trace does not begin in an initial state. Then it will be checked in the next cycle, first if-condition, whether an *initialised* trace of the same length exists, also violating P. If this is the case, a safety violation has been detected. Otherwise the loop is continued. In the worst case situation, much longer loop-free initialised traces can be found, and at the same time longer traces satisfying $(\text{loopFree}(s_{[0..i+1]}) \wedge \text{all.P}(s_{[0..i]}) \wedge \neg P(s_{i+1}))$ can be found, but these traces are not reachable from an initial state. This may delay termination of

the algorithm in a considerable way.

To mitigate this problem, it is advisable to extend the safety property P by a conjunct P' characterising reachable states, and use the algorithm to prove global validity of $P \wedge P'$. If P' Is a good approximation of reachable states, termination condition (3) will only be violated if a real violation of P occurred in a reachable state s_{i+1}. This technique is called *strengthening of the invariant* P. In any case, care must be taken that P' is always an *over-approximation* of reachable states, so that no reachable states can ever be forgotten.

In [3] it is shown how the k-induction principle described in this section can be extended to general LTL formulas.

Exercise 12. Using Example 3, illustrate how the k-induction algorithm operates in order to prove $\mathbf{G}\neg(c_0 \wedge c_1)$. □

Exercise 13. Construct a Kripke structure and a safety invariant $\mathbf{G}P$, such that the k-induction proof for $\mathbf{G}P$ is best performed with a value $k \geq 1$. □

Chapter 6

Model Checking With nuXmv

Currently, one of the most popular model checkers that is freely licensed in binary form for non-commercial or academic purposes, is nuXmv[1] [4]. Its modelling language and checking capabilities are described in the nuXmv user manual.[2] The tool supports several variants of model checking, such as BMC, k-induction, and classical algorithms based on binary decision diagrams (BDDs).

In this chapter we introduce the nuXmv modelling language and the specification of CTL and LTL assertions, as far as needed in this lecture.

6.1 Encoding Simple Kripke Structures in nuXmv

Consider a very basic type of Kripke structures, where states are just uninterpreted elements represented by enumeration values or integers, and atomic propositions are just elementary statements without references to variables. An example of such a Kripke structure called Oven is given in Fig. 6.1. Model Oven describes a simple cooking oven with six states.

[1] https://nuxmv.fbk.eu

[2] https://nuxmv.fbk.eu/pmwiki.php?n=Documentation.Home

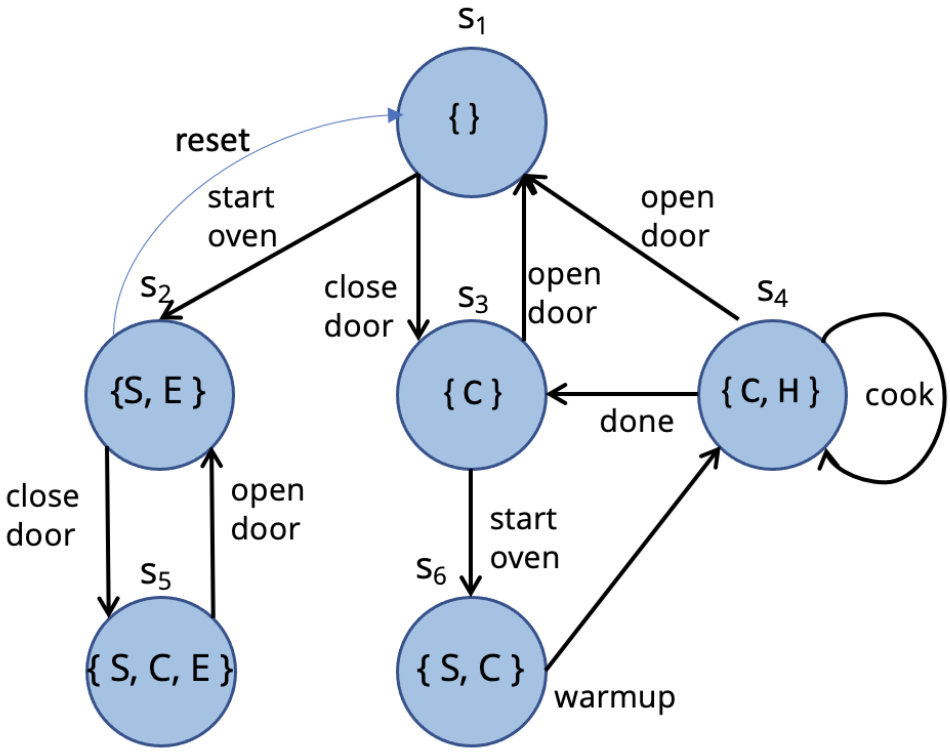


Figure 6.1: Kripke structure presenting a kitchen oven.

The atomic propositions labelling the states have the meaning

S The oven has been started

E The oven is in an error state

C The oven door is closed

H The oven is hot

The normal operation of the oven is to start from s_1 by closing the door, this leads to state s_3. After that the oven is started, this leads to s_6. After

having been started the oven heats up (state s_4). It is possible to stay in s_4 forever, so the food in the oven can be overcooked and burnt. It is also possible to open the door at any time (transit to state s_1), so that the oven is switch off. The best way to enjoy the cooked food is to wait until its done and then stop the heating (transit to s_3). There, the oven door can be opened (transit to s_1) and the perfectly cooked food can be relished.

If the oven is started before the door is closed (transition $s_1 \longrightarrow s_2$), error states are visited where the oven is never heated (s_2, s_5). From there, only a reset leading to transition $s_2 \longrightarrow s_1$ enables us to start cooking again.

Note that the transition labels in Fig. 6.1 are not part of the Kripke structure, they just have illustrative purposes. There is no concept of inputs and outputs: all model information is encoded in the transitions and the labelling function.

The representation in Fig. 6.1 corresponds to the mathematical representation

$$
\begin{aligned}
\textsc{Oven} &= (S, S_0, R, AP, L) \\
S &= \{s_1, s_2, s_3, s_4, s_5, s_6\} \\
S_0 &= \{s_1\} \\
R &= \{(s_1, s_2), (s_1, s_3), (s_2, s_1), (s_2, s_5), (s_3, s_2), (s_3, s_6), \\
&\quad (s_4, s_1), (s_4, s_3)(s_4, s_4), (s_5, s_2)(s_6, s_4)\} \\
AP &= \{S, E, C, H\} \\
L &= \{s_1 \mapsto \{\}, s_2 \mapsto \{S, E\}, s_3 \mapsto \{C\}, s_4 \mapsto \{C, H\}, s_5 \mapsto \{S, C, E\}, s_6 \mapsto \{S, C\}\}
\end{aligned}
$$

Listing 6.1 shows a practical way to encode such a simple Kripke structure in the syntax of the nuXmv input language.

1. The states are represented by one variable declared as enumeration or as a fixed integer range (variable `state` in line 3, here we use the encoding by an enumeration).

2. The initial state is set by an ASSIGN-directive (lines 8 — 9).

3. For each atomic proposition, a Boolean variable is declared (lines 4 — 7).

4. The atomic propositions that are valid in a state are specified by means of invariant statements (INVAR, lines 24—29), structured by the possible state values. These lines specify the labelling function

$$L : \{s_1, s_2, s_3, s_4, s_5, s_6\} \rightarrow 2^{\{S,E,C,H\}}.$$

5. The transition relation is specified by a next-directive, as shown in lines 12 — 20. The case statement is interpreted like an if-else statement. The if-conditions are specified before the colon ":". The post-state value of the variable (variable `state` in our example) that should hold after a certain condition is fulfilled is specified after the colon. Using set enumerations as shown in lines 14—19, nondeterministic assignments are realised.

Listing 6.1: nuXmv model of the Kripke structure shown in Fig. 6.1

```
1  MODULE main
2    VAR
3      state : {s1,s2,s3,s4,s5,s6};
4      ps : boolean; -- atomic proposition S
5      pe : boolean; -- atomic proposition E
6      pc : boolean; -- atomic proposition C
7      ph : boolean; -- atomic proposition H
8    ASSIGN
9      init(state) := s1;
10
11     -- next states depend on the selected transition
12     next(state) :=
13       case
14         state = s1 : { s2, s3 };
15         state = s2 : { s1, s5 };
16         state = s3 : { s1, s6 };
17         state = s4 : { s1, s3, s4 };
18         state = s5 : { s2 };
19         state = s6 : { s4 };
20       esac;
21
22
23   -- Atomic propositions only depend on the state
24   INVAR state = s1 -> !ps & !pe & !pc & !ph;
```

```
25    INVAR state = s2 -> ps & pe & !pc & !ph;
26    INVAR state = s3 -> !ps & !pe & pc & !ph;
27    INVAR state = s4 -> !ps & !pe & pc & ph;
28    INVAR state = s5 -> ps & pe & pc & !ph;
29    INVAR state = s6 -> ps & !pe & pc & !ph;
```

6.2 Specifying Assertions

For the purposes of this lecture, LTL and CTL property specifications are associated with a model. This can be done in the same file, after the model specification. Of the Oven model introduced above, useful assertions to be checked by nuXmv are shown in Listing 6.2. CTL properties are specified after keyword SPEC, LTL specification after keyword LTLSPEC. Since programming errors can occur when coding a model in the nuXmv input language, it is always useful to specify properties checking whether the model has been properly constructed. If this is the case, further formulae are specified to deal with application-specific properties the model should have.

Listing 6.2: CTL and LTL specifications that are fulfilled by the Oven model from Listing 6.1.

```
1
2  -- Validate that transitions have been properly defined
3  -- These properties show that no transition has the wrong target state
4  SPEC AG (state = s1 -> AX (state in {s2,s3}))
5  SPEC AG (state = s2 -> AX (state in {s1,s5}))
6  SPEC AG (state = s3 -> AX (state in {s1,s6}))
7  SPEC AG (state = s4 -> AX (state in {s1,s3,s4}))
8  SPEC AG (state = s5 -> AX (state in {s2}))
9  SPEC AG (state = s6 -> AX (state in {s4}))
10
11 -- Validate that transitions have been properly defined
12 -- These properties show that no transition has been forgotten
13 SPEC AG (state = s1 -> EX (state in {s3}))
14 SPEC AG (state = s1 -> EX (state in {s2}))
15 SPEC AG (state = s2 -> EX (state in {s1}))
16 SPEC AG (state = s2 -> EX (state in {s5}))
17 -- ...
```

```
18
19 -- Check that we can always go back to the initial state
20 SPEC AG ( state != s1 -> EF ( state = s1 ) )
21
22 -- Check that we can always finally get our food hot
23 SPEC AG ( EF ph )
24
25 -- Check that the door is always closed when the oven is heated
26 LTLSPEC G(ph -> pc)
27
28 -- Check that the oven can never be hot if an error occurred.
29 LTLSPEC G(ph -> X !pe)
30
31 -- Check that we can always recover from errors
32 SPEC AG (pe -> EF !pe)
33
34 -- Check that starting the oven will always result in heat,
35 -- unless an error occurred before
36 LTLSPEC G( (ps & !pe) -> X ph)
```

6.3 Encoding Complex Kripke Structures in nuXmv

As discussed in the previous chapters, more complex Kripke structures usually involve variables, and the state spaces are defined by means of valuation functions $s : V \to D$ mapping variable values into some value domain. For these structures, it is advisable to encode nuXmv models by means of the transition relation. This is shown in Listing 6.3 for the two processes applying the strict alternation protocol in Example 3. Observe that the transition relation encoded after keyword TRANS in this listing is represented exactly as shown on page 25 for this example.

1. We declare nuXmv variables in one-to-one correspondence to the variables of the Kripke structure (lines 2 — 8).

2. The primed variables v' occurring in our transition relations and denoting the post-state of the variable, after the transition has been performed, are encoded by next(v) in the nuXmv model.

Listing 6.3: nuXmv-model for the two processes from Example 3.

```
MODULE main
  VAR
    s : { 0, 1 };
    c0 : boolean;
    c1 : boolean;

    p0 : { 1, 2, 3, 4, 5, 6, 7, 8 };
    p1 : { 1, 2, 3, 4, 5, 6, 7, 8 };

  ASSIGN

    init(s) := 0;
    init(c0) := FALSE;
    init(c1) := FALSE;
    init(p0) := 1;
    init(p1) := 1;

  TRANS
    (p0 = 1 & next(p0) = 2 & next(p1) = p1 & next(s) = s
                                    & next(c0) = c0 & next(c1) = c1)
  | (p0 = 2 & next(p0) = 3 & next(p1) = p1 & next(s) = 0
                                    & next(c0) = c0 & next(c1) = c1)
  | (p0 = 3 & s = 0 & next(p0) = 3 & next(p1) = p1
                                & next(s) = s & next(c0) = c0 & next(c1) = c1)
  | (p0 = 3 & s != 0 & next(p0) = 4 & next(p1) = p1
                                & next(s) = s & next(c0) = c0 & next(c1) = c1)
  | (p0 = 4 & next(p0) = 5 & next(p1) = p1 & next(s) = s
                                    & next(c0) = TRUE & next(c1) = c1)
  | (p0 = 5 & next(p0) = 6 & next(p1) = p1 & next(s) = s
                                    & next(c0) = FALSE & next(c1) = c1)
  | (p0 = 6 & next(p0) = 2 & next(p1) = p1 & next(s) = s
                                    & next(c0) = c0 & next(c1) = c1)

  | (p1 = 1 & next(p1) = 2 & next(p0) = p0 & next(s) = s
                                    & next(c0) = c0 & next(c1) = c1)
  | (p1 = 2 & next(p1) = 3 & next(p0) = p0 & next(s) = 1
                                    & next(c0) = c0 & next(c1) = c1)
  | (p1 = 3 & s = 1 & next(p1) = 3 & next(p0) = p0 & next(s) = s
                                    & next(c0) = c0 & next(c1) = c1)
  | (p1 = 3 & s != 1 & next(p1) = 4 & next(p0) = p0 & next(s) = s
```

```
41                                          & next(c0) = c0 & next(c1) = c1)
42    | (p1 = 4 & next(p1) = 5 & next(p0) = p0 & next(s) = s
43                                          & next(c0) = c0 & next(c1) = TRUE)
44    | (p1 = 5 & next(p1) = 6 & next(p0) = p0 & next(s) = s
45                                          & next(c0) = c0 & next(c1) = FALSE)
46    | (p1 = 6 & next(p1) = 2 & next(p0) = p0 & next(s) = s
47                                          & next(c0) = c0 & next(c1) = c1)
```

Useful verification and validation formulae for this model are shown in Listing 6.4.

Listing 6.4: CTL formulae for validating the nuXmv-model shown in Listing 6.3.

```
1  -- Validate that processes can finally reach the last line of the code.
2  SPEC EF (p0 = 6)
3  SPEC EF (p1 = 6)
4
5  -- Verify that the strict alternation protocol never
6  -- allows that P0 and P1 are in their critical sections
7  -- at the same time.
8  SPEC AG ( !(c0 & c1) )
9
10 -- Validate that it is possible for each process to
11 -- enter their critical sections
12 SPEC EF ( c0 )
13 SPEC EF ( c1 )
14
15 -- Scheduler allows for starvation
16 SPEC EF ( EG p0 = 3 )
17 SPEC EF ( EG p1 = 3 )
18
19 -- We don't have fairness in the scheduler, so there should be
20 -- infinite paths where either P0 or P1 never reach
21 -- the critical section (this is just another way of looking
22 -- at the starvation problem)
23 --
24 SPEC EG( ! c0 )
25 SPEC EG( ! c1 )
```

Chapter 7

Data Abstraction

This section deals with state space reduction by means of data abstraction.

7.1 Equivalence Classes and Factorisation of Transition Systems

Let $TS = (S, S_0, R)$ a transition system and $\sim \subseteq S \times S$ an equivalence relation on S, that is,

- $\forall s \in S : s \sim s$ (reflexivity)
- $\forall s, s' \in S : s \sim s' \Rightarrow s' \sim s$ (symmetry)
- $\forall s, s', s'' \in S : s \sim s' \wedge s' \sim s'' \Rightarrow s \sim s''$ (transitivity)

Let $S/_\sim$ denote the set of equivalence classes; each class is written in the form $[s] \in S/_\sim$, $[s] =_{\text{def}} \{u \mid s \sim u\}$. An equivalence relation gives rise to a transition system *factorised by* $\sim$ which is defined by

$$\begin{array}{l} TS/_\sim =_{\text{def}} (S/_\sim, S_0/_\sim, R/_\sim) \\ S_0/_\sim =_{\text{def}} \{[s_0] \mid s_0 \in S_0 \wedge [s_0] \in S/_\sim\} \\ R/_\sim =_{\text{def}} \{([s],[s']) \mid \exists u \in [s], u' \in [s'] \,.\, R(u,u')\} \end{array} \tag{7.1}$$

7.2 Auxiliary Variables and Associated Equivalence Classes

Let us consider now again only state spaces S whose elements are variable valuations $s : V \nrightarrow D, V = \{x_1, x_2, \ldots\}$. Let $AUX = \{a_1, a_2, \ldots\}$ a set of fresh variables such that $V \cap AUX = \varnothing$. Let $e_i(x_1^i, x_2^i, \ldots)$ be expressions associated with each $a_i \in AUX$. For a fixed set of auxiliary variables a_i and expressions e_i, extend valuation functions by

$$
\begin{array}{l}
s_e : V \cup AUX \nrightarrow D \\
\text{dom } s_e = \text{dom } s \cup \{a_i \in AUX \mid x_1^i, x_2^i, \ldots \in \text{dom } s\} \\
s_e|_V = s \text{ that is, } \forall x \in V \cap \text{dom } s_e : s_e(x) = s(x) \\
\forall a_i \in AUX \cap \text{dom } s_e : s_e(a_i) = e_i(s(x_1^i), s(x_2^i), \ldots)
\end{array}
$$

Observe that the expressions $e_i(x_1^i, x_2^i, \ldots)$ induce a type D_{a_i} on the corresponding auxiliary variables a_i. We denote the transition system extended by the variables from AUX and the extended valuations s_e by $TS_e = (S_e, S_{0e}, R_e)$, where the transition relation is defined by

$$R_e =_{\text{def}} \{(s_e, s'_e) \mid (s_e|_V, s'_e|_V) \in R\}$$

A collection of auxiliary variables induces an equivalence relation $\sim$ on $TS_e = (S_e, S_{oe}, R_e)$ by defining

$$\forall s, s' \in S : s \sim s' \equiv_{\text{def}} (\forall a \in AUX : s_e(a) = s'_e(a))$$

$TS_e/_\sim$ is called the factorisation of TS by means of the *data abstraction*

$$a_i = e_i(x_1^i, x_2^i, \ldots), \ i = 1, 2, \ldots$$

Observe that, given a valuation $(s : V \nrightarrow D) \in S$, its equivalence class $[s]$ may also be regarded as a valuation function on the variables from AUX by setting

$$\forall a_i \in AUX : [s](a_i) =_{\text{def}} e_i(s(x_1), s(x_2), \ldots)$$

The definition of $\sim$ guarantees that this valuation function is well-defined, since all members $s' \in [s]$ fulfill

$$\forall i : e_i(s(x_1), s(x_2), \ldots) = e_i(s'(x_1), s'(x_2), \ldots)$$

Lemma 5 *Suppose that the initial state S_0 is characterised by first-order predicate $\mathcal{I}$ with free variables in $V = \{x_1, x_2, \ldots\}$, and that the transition relation $R \subseteq S \times S$ is characterised by predicate $\mathcal{R}$ with free variables in V and $V' =_{def} \{x'_1, x'_2, \ldots\}$. Then the respective predicates for $TS_e/_\sim$ are given by*

$$\mathcal{I}/_\sim(a_1, a_2, \ldots) =_{def} \exists \xi_1, \xi_2, \ldots . (\forall i : a_i = e_i(\xi_1, \xi_2, \ldots)) \wedge \mathcal{I}[\xi_1/x_1, \xi_2/x_2, \ldots] \tag{7.2}$$

$$\begin{aligned} \mathcal{R}/_\sim(a_1, a_2, \ldots, a'_1, a'_2, \ldots) =_{def} \exists \xi_1, \xi_2, \ldots, \xi'_1, \xi'_2, \ldots . \\ \forall i : (a_i = e_i(\xi_1, \xi_2, \ldots) \wedge a'_i = e_i(\xi'_1, \xi'_2, \ldots)) \wedge \\ \mathcal{R}[\xi_1/x_1, \xi_2/x_2, \ldots, \xi'_1/x'_1, \xi'_2/x'_2, \ldots] \end{aligned} \tag{7.3}$$

Proof. From (7.1) and the fact that $\mathcal{I}$ characterises S_0 we conclude that

$$S_{0e}/_\sim = \{[s_0] : AUX \nrightarrow D \mid s_0 : V \cup AUX \nrightarrow D \wedge \mathcal{I}[s_0(x_1)/x_1, s_0(x_2)/x_2, \ldots]\}$$

Therefore, in order to prove correctness of $\mathcal{I}/_\sim$, it has to be shown that

$$\begin{aligned} \overline{S} =_{def} \{s_a : AUX \nrightarrow D \mid \mathcal{I}/_\sim[s_a(a_1)/a_1, s_a(x_a)/a_2, \ldots]\} = \\ \{s_a : AUX \nrightarrow D \mid \exists \xi_1, \xi_2, \ldots . (\forall i : s_a(a_i) = e_i(\xi_1, \xi_2, \ldots)) \wedge \mathcal{I}[\xi_1/x_1, \xi_2/x_2, \ldots]\} \end{aligned}$$

equals $S_{0e}/_\sim$.

We show first that $S_{0e}/_\sim \subseteq \overline{S}$: Let $[s_0] \in S_{0e}/_\sim$. Define $\xi_i =_{def} s_0(x_i), i = 1, 2, \ldots$. Then, because $\mathcal{I}[s_0(x_1)/x_1, s_0(x_2)/x_2, \ldots]$ holds, this implies $\mathcal{I}[\xi_1/x_1, \xi_2/x_2, \ldots]$. Furthermore, $[s_0](a_i) = e_i(s_0(x_1), s_0(x_2), \ldots)$ by definition of $[\cdot]$, so $(\forall i : a_i = e_i(\xi_1, \xi_2, \ldots))$. As a consequence, $\mathcal{I}/_\sim[[s_0](a_1)/a_1, [s_0](a_2)/a_2, \ldots]$ holds which shows that $[s_0] \in \overline{S}$.

Now we show $\overline{S} \subseteq S_{0e}/_\sim$: Let $s_a \in \overline{S}$, then there exist $\xi_1, \xi_2, \ldots$ such that $(\forall i : s_a(a_i) = e_i(\xi_1, \xi_2, \ldots)) \wedge \mathcal{I}[\xi_1/x_1, \xi_2/x_2, \ldots]$. Now define a valuation $s_0 : V \nrightarrow D$ by $s_0(x_i) =_{def} \xi_i, i = 1, 2, \ldots$. This s_0 is contained in S_0 and therefore $[s_0] \in S_{0e}/_\sim$, since $\mathcal{I}[\xi_1/x_1, \xi_2/x_2, \ldots]$ and therefore $\mathcal{I}[s_0(x_1)/x_1, s_0(x_2)/x_2, \ldots]$ holds. Since $s_a(a_i) = e_i(\xi_1, \xi_2, \ldots) = e_i(s_0(x_1), s_0(x_2), \ldots)$, the construction of s_0 implies $s_a = [s_0]$, so $s_a \in S_{0e}/_\sim$, and this shows $\overline{S} \subseteq S_{0e}/_\sim$ and proves (7.2).

For proving (7.3), recall from (7.1) that the transition relation of the factorised transition system $TS_e/_\sim$ is defined by

$$R/_\sim =_{def} \{([s], [s']) \mid \exists u \in [s], u' \in [s'] . R(u, u')\}$$

We define

$$\overline{R} =_{\text{def}} \{(s_a, s'_a) \mid \mathcal{R}/_\sim[s_a(a_1)/a_1, s_a(a_2)/a_2, \ldots, s'_a(a_1)/a'_1, s'_a(a_2)/a_2, \ldots]\}$$

and show that $R/_\sim$ equals $\overline{R}$.

To show that $R/_\sim \subseteq \overline{R}$, suppose that $([s], [s']) \in R/_\sim$. By definition of $[\cdot]$, $R/_\sim$ and $\mathcal{R}$ there exists $u, u' : V \nrightarrow D$ such that

$$\begin{aligned}\forall i : (&e_i(s(x_1), s(x_2), \ldots) = e_i(u(x_1), u(x_2), \ldots) \wedge \\ &e_i(s'(x_1), s'(x_2), \ldots) = e_i(u'(x_1), u'(x_2), \ldots)) \wedge \\ &\mathcal{R}[u(x_1)/x_1, u(x_2)/x_2, \ldots, u'(x_1)/x'_1, u'(x_2)/x'_2, \ldots]\end{aligned}$$

holds. Setting $\xi_i = u(x_i), \xi'_i = u'(x_i), i = 1, 2, \ldots$ yields

$$\forall i : (a_i = e_i(\xi_1, \xi_2, \ldots) \wedge a'_i = e_i(\xi'_1, \xi'_2, \ldots)) \wedge \mathcal{R}[\xi_1/x_1, \xi_2/x_2, \ldots, \xi'_1/x'_1, \xi'_2/x'_2, \ldots]$$

and, since $e_i(s(x_1), s(x_2), \ldots)$ equals $e_i(\xi_1, \xi_2, \ldots)$ and $e_i(s'(x_1), s'(x_2), \ldots)$ equals $e_i(\xi'_1, \xi'_2, \ldots)$, this implies that

$$\mathcal{R}/_\sim[[s](a_1)/a_1, [s](a_2)/a_2, \ldots, [s'](a_1)/a'_1, [s'](a_2)/a'_2, \ldots]$$

holds. This proves $([s], [s']) \in \overline{R}$.

It remains to show that $\overline{R} \subseteq R/_\sim$. To this end, assume that $(s_a, s'_a) \in \overline{R}$. By definition of $\overline{R}$ and $\mathcal{R}/_\sim$ this implies the existence of $\xi_i, \xi'_i, i = 1, 2, \ldots$ such that

$$\begin{aligned}\forall i : (&s_a(a_i) = e_i(\xi_1, \xi_2, \ldots) \wedge s'_a(a'_i) = e_i(\xi'_1, \xi'_2, \ldots)) \wedge \\ &\mathcal{R}[\xi_1/x_1, \xi_2/x_2, \ldots, \xi'_1/x'_1, \xi'_2/x'_2, \ldots]\end{aligned}$$

Now define

$$s : V \nrightarrow D; s(x_i) \mapsto \xi_i, \quad s' : V \nrightarrow D; s'(x_i) \mapsto \xi'_i, i = 1, 2, \ldots$$

Then $[s] = s_a$ and $[s'] = s'_a$ and $\mathcal{R}[s(x_1)/x_1, s(x_2)/x_2, \ldots, s'(x_1)/x'_1, s'(x_2)/x'_2, \ldots]$ by construction and this implies $R(s, s')$ and finally yields $([s], [s']) \in R/_\sim$. This shows $(s_a, s'_a) \in R/_\sim$ and completes the proof. □

7.3 Data Abstraction on Kripke Structures

Given a Kripke structure $K = (S, S_0, R, L)$ and a set AUX of auxiliary variables a_i with associated expressions $e_i(x_1^i, x_2^i, \ldots)$ we can extend K to a Kripke structure $K_e =_{\text{def}} (S_e, S_{oe}, R_e, L_e)$ by defining its set of atomic propositions and the labelling function as

$$\begin{aligned}
&AP_e =_{\text{def}} AP \cup AP_{AUX} \\
&AP_{AUX} =_{\text{def}} \{a_i = \alpha \mid a_i \in AUX \wedge \alpha \in D_{a_i}\} \\
&L_e : S_e \to 2^{AP_e} \\
&L_e(s) = L(s) \cup \{a_i = e_i(s(x_1^i), s(x_2^i), \ldots) \mid a_i \in AUX\}
\end{aligned}$$

If we now factorise K_e's transition system (S_e, S_{oe}, R_e) by the equivalence relation $\sim$ introduced by AUX then we can extend the abstracted transition system to a Kripke structure by "forgetting" about the original variables in V and considering only the propositions on abstraction variables of AUX. This is done in the obvious way by defining a labelling function

$$L_e/_\sim : S_e/_\sim \to 2^{AP_{AUX}};\ [s] \mapsto \{a_i = e_i(s(x_1^i), s(x_2^i), \ldots) \mid a_i \in AUX\}$$

Note that $L_e/_\sim$ is well-defined since all members of $[s]$ induce the same valuations for all $a_i \in AUX$. As a consequence

$$K_e/_\sim = (S_e/_\sim, S_{0e}/_\sim, R_e/_\sim, L_e/_\sim)$$

is a well-defined Kripke structure, and the explicit model checking algorithms introduced in Section 4 can be applied to $K_e/_\sim$, as long as we only consider CTL formulas φ over the auxiliary variables from AUX, without any reference to the variables from V. Such a formula would also be applicable to the unfactorised Kripke structure K_e. Therefore we would like to know when a formula φ proven to be valid in $K_e/_\sim$ is also valid in K_e.

Example 7. Consider the Kripke Structure depicted in Fig. 7.1, which is associated with a specification model of a traffic light controller. As is well known to every law-abiding citizen, we always stop our cars on red *and* on yellow. Therefore, if we are only interested in knowing when cars are in a

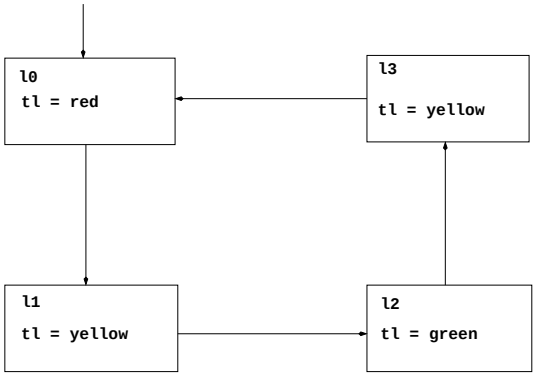


Figure 7.1: Kripke structure of traffic light controller from Example 7.

halt-state in front of the traffic light, it makes sense to introduce a Boolean auxiliary variable

$$\texttt{stops} =_{\text{def}} (\texttt{tl} = \texttt{red} \vee \texttt{tl} = \texttt{yellow})$$

Factorisation against the equivalence relation introduced by stops leads to the abstracted Kripke structure shown in Fig. 7.2.

Now suppose we wish to prove that $\mathbf{EF}(\texttt{tl} = \texttt{green})$ holds for the Kripke structure of the original model in Fig. 7.1. The assertion can be readily expressed on abstract level as $\mathbf{EF}(\neg\texttt{stops})$ which obviously holds on abstract level, since every path in Fig. 7.2 visits $(\texttt{m1}, \neg\texttt{stops})$. Similarly, the concrete condition $\mathbf{AF}(\texttt{tl} = \texttt{red} \vee \texttt{tl} = \texttt{yellow})$ can be expressed in an abstract way as $\mathbf{AF}\texttt{stops}$. It is easy to see that it holds on abstract level.

In these special cases, the assertions also hold on concrete level, but this is not always the case: On abstracted level we can also prove the formula $\mathbf{EG}(\texttt{stops})$ which obviously does not hold in the concrete model with its concrete formula representation $\mathbf{EG}(\texttt{tl} = \texttt{red} \vee \texttt{tl} = \texttt{yellow})$. Conversely, the concrete model satisfies $\mathbf{AF}(\texttt{tl} = \texttt{green})$, while the corresponding formula $\mathbf{AF}(\neg\texttt{stop})$ is not fulfilled on abstract level. □

Exercise 14. Consider the slightly modified specification model from Exercise 2, now shown in Fig. 7.3. Assume now that x and y have unbounded range $D_x = D_y = \mathbb{Z}$, so that explicit model checking becomes infeasible. Chose suitable abstraction variables and construct the corresponding factorisation of the model's Kripke structure such that the following assertion

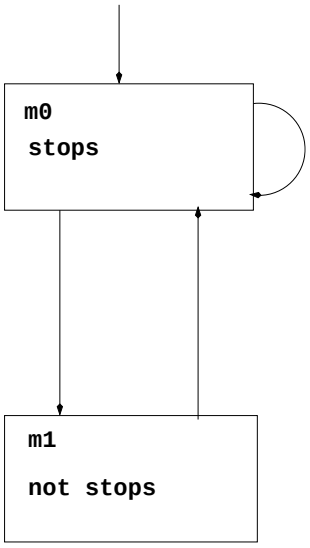


Figure 7.2: Abstracted Kripke structure induced by auxiliary variable `stops` in Example 7.

can be proved using the explicit CTL model checking algorithms on the abstracted Kripke structure:

$$\neg\mathbf{EF}(\texttt{10} \wedge \texttt{odd(y)})$$

Give informal justifications for

- the completeness and correctness of your abstracted Kripke structure (since you do not want to enumerate the concrete (infinite!) Kripke structure of the model),
- the fact that the proof for the abstracted model implies that the assertion also holds for the concrete model.

□

7.4 Simulations

In order to investigate the situations where assertions on auxiliary variables proven on abstract level also hold for the concrete level we introduce the concept of *simulations*:

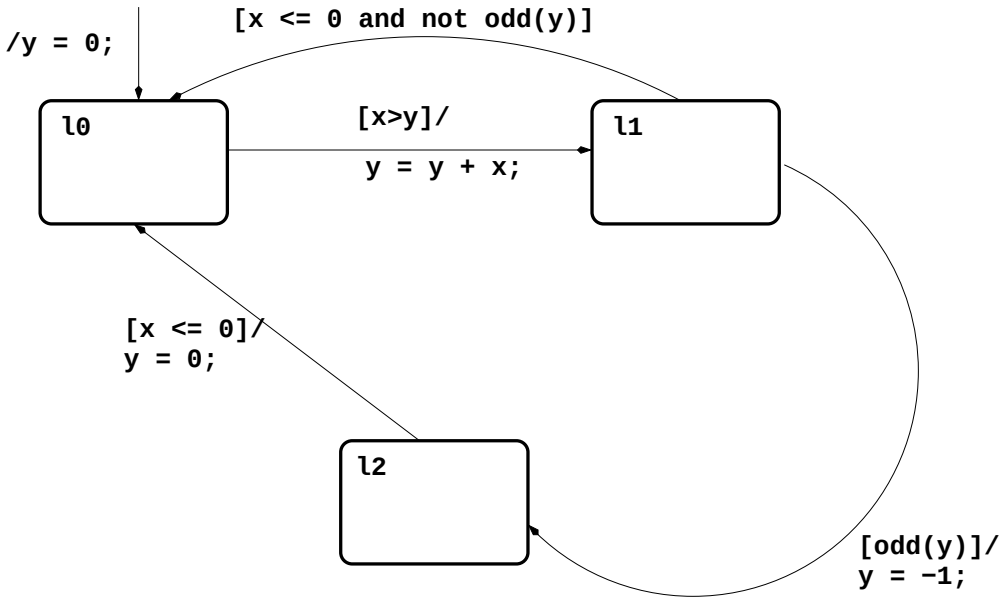


Figure 7.3: Model for Exercise 7.

Definition 5 (Simulation) *Given two Kripke structures* $K = (S, S_0, R, L), K' = (S', S_0', R', L')$ *such that* K *refers to atomic propositions* AP *and* K' *refers to atomic propositions* AP' *and* $AP' \subseteq AP$. *The relation* $H \subseteq S \times S'$ *is called a* simulation, *if the following conditions hold for all* $(s, s') \in H$*:*

1. $L(s) \cap AP' = L'(s')$
2. $\forall s_1 \in S : R(s, s_1) \Rightarrow \exists s_1' \in S' \,.\, R'(s', s_1') \wedge H(s_1, s_1')$

We write $K \preccurlyeq K'$ (K is simulated by K') *if such a simulation* H *exists and*

$$\forall s_0 \in S_0 \,.\, \exists s_0' \in S_0' : H(s_0, s_0')$$

□

Example 8. Consider again the traffic light example introduced in Example 7. In a more formal way, its Kripke structure, extended by auxiliary variable stops can be introlduced as follows.

$$
\begin{aligned}
\mathrm{TL}_e &= (S, S_0, R, AP, L) \\
S &= \{l_0, l_1, l_2, l_3\} \\
S_0 &= \{l_0\} \\
R &= \{(l_0, l_1), (l_1, l_2), (l_2, l_3), (l_3, l_0)\} \\
AP &= \{\mathrm{tl} = \mathrm{red}, \mathrm{tl} = \mathrm{yellow}, \mathrm{tl} = \mathrm{green}, \mathrm{stops}\} \\
L &= \{l_0 \mapsto \{\mathrm{tl} = \mathrm{red}, \mathrm{stops}\}, l_1 \mapsto \{\mathrm{tl} = \mathrm{yellow}, \mathrm{stops}\}, \\
&\quad\; l_2 \mapsto \{\mathrm{tl} = \mathrm{green}\}, l_3 \mapsto \{\mathrm{tl} = \mathrm{yellow}, \mathrm{stops}\}\}
\end{aligned}
$$

Now we "guess" an abstracted Kripke structure and show that this is indeed a simulation of TL_e. To this end, we invent an equivalence relation on the state space S by defining

$$
\begin{aligned}
\sim &\subseteq S \times S \\
\sim &= \{(l_0, l_0), (l_1, l_1), (l_2, l_2), (l_3, l_3), \\
&\quad\; (l_0, l_1), (l_0, l_3), (l_1, l_0), (l_3, l_0), (l_1, l_3), (l_3, l_1)\}
\end{aligned}
$$

This induces two equivalence classes

$$[l_0] = \{l_0, l_1, l_3\} \quad \text{and} \quad [l_2] = \{l_2\}$$

Applying the construction rules of Formula (7.1), this induces the Kripke structure

$$
\begin{aligned}
\mathrm{TL}_e/_\sim &= (S', S_0', R', AP', L') \\
S' &= \{[l_0], [l_2]\} \\
S_0' &= \{[l_0]\} \\
R' &= \{([l_0], [l_0]), ([l_0], [l_2]), ([l_2], [l_0])\} \\
AP' &= \{\mathrm{stops}\} \\
L' &= \{[l_0] \mapsto \{\mathrm{stops}\}, [l_2] \mapsto \{\,\}\}
\end{aligned}
$$

Now we guess the simulation relation $H \subseteq S \times S'$ as

$$H = \{(l, [l]) \mid l \in S\},$$

which can be explicitly enumerated as (note that $[l_0] = [l_1] = [l_3]$)

$$H = \{(l_0, [l_0]), (l_1, [l_0]), (l_2, [l_2]), (l_3, [l_0])\}.$$

It will turn out below in Theorem 6 that this is the standard construction for creating simulation relations with factorised Kripe structures. Let's

prove that H is indeed a simulation relation. Following Definition 5, we first have to show that $L(s) \cap AP' = L'(s')$ for all states s related by the simulation H. This holds, since

$$\begin{aligned} L(l_0) \cap AP' &= \{\text{stops}\} = L'([l_0]) \\ L(l_1) \cap AP' &= \{\text{stops}\} = L'([l_0]) \\ L(l_2) \cap AP' &= \{\,\} = L'([l_2]) \\ L(l_3) \cap AP' &= \{\text{stops}\} = L'([l_0]) \end{aligned}$$

Next, we have to show that Condition 2 of Definition 5 holds; this follows from the facts that

$$\begin{aligned} &(l_0, l_1) \in R \wedge ([l_0], [l_0]) \in R' \wedge (l_0, [l_0]) \in H \wedge (l_1, [l_0]) \in H, \\ &(l_1, l_2) \in R \wedge ([l_0], [l_2]) \in R' \wedge (l_1, [l_0]) \in H \wedge (l_2, [l_2]) \in H, \\ &(l_2, l_3) \in R \wedge ([l_2], [l_0]) \in R' \wedge (l_2, [l_2]) \in H \wedge (l_3, [l_0]) \in H, \\ &(l_3, l_0) \in R \wedge ([l_0], [l_0]) \in R' \wedge (l_3, [l_0]) \in H \wedge (l_0, [l_0]) \in H. \end{aligned}$$

Finally, we note that $(l_0, [l_0]) \in H$, so the initial state of TL_e is mapped to the initial state of $TL_e/_\sim$. This concludes the proof that $TL_e \preccurlyeq TL_e/_\sim$. □

Before exploiting the simulation concept in Theorem 7 below, it is necessary to show that the equivalence relation $\sim$ induced by auxiliary variables as introduced above establishes a simulation relation between original Kripke structure K_e and its factorisation $K_e/_\sim$[1]:

Theorem 6 *Given $\sim$, equivalence classes $[s]$, AP_e, L_e, K_e, $K_e/_\sim$ as introduced in Section 7.3 above, define*

$$H =_{def} \{(s, [s]) \mid s \in S_e\} \subseteq S_e \times S_e/_\sim$$

Then H is a simulation between K_e and $K_e/_\sim$ and $K_e \preccurlyeq K_e/\sim$ holds.

Proof. Let H be defined according to the precondition of the theorem and $s \in S_e$, so that $(s, [s]) \in H$. By the construction rules given in Section 7.3, the states of K_e are labelled with atomic propositions from $AP \cup AP_{AUX}$, and the states (i. e., equivalence classes) of $K_e/_\sim$ are labelled with atomic propositions from AP_{AUX}. As a consequence, the construction of the labelling functions L_e on K_e and $L_e/_\sim$ on $K_e/_\sim$ implies

$$L_e(s) \cap AP_{AUX} = \{a_i = e_i(s(x_1^i), s(x_2^i), \ldots) \mid a_i \in AUX\} = L_e/_\sim([s])$$

[1] So Example 8 is just a confirmation of Theorem 6 by example.

Therefore condition (i) of Definition 5 holds.

Now let $s_1 \in S_e$ such that $R(s, s_1)$. By construction of $R/_\sim$ in Section 7.1 this implies $R/_\sim([s], [s_1])$ and by construction of H this also implies $H(s_1, [s_1])$. Therefore condition (ii) of Definition 5 is also fulfilled.

Finally, we note that $\forall s_0 \in S_0 : H(s_0, [s_0])$ holds by construction of H, and $[s_0] \in S_{0e}/_\sim$ by construction of $K_e/_\sim$. As a consequence, $K_e \preccurlyeq K_e/\sim$, and this completes the proof. □

Definition 6 *Let* $K \preccurlyeq K'$ *with simulation relation* $H \subset S \times S'$ *and* $H(s, s')$. *Suppose* π *is a path in* K *starting at* s *and* π' *a path starting at* s' *in* K'. *We say that* π and π' correspond to each other *if*

$$\forall i \geq 0 : H(\pi(i), \pi'(i))$$

□

Lemma 6 *Let* $K \preccurlyeq K'$ *with simulation relation* $H \subset S \times S'$ *and* $H(s, s')$. *Then for every path* π *in* K *starting at* s *there is a corresponding path* π' *in* K' *starting at* s'.

Proof. Since π is a path starting at s,

$$\pi(0) = s \wedge (\forall i \geq 0 : R(\pi(i), \pi(i+1)))$$

follows. Since $s = \pi(0)$ and $H(s, s')$, this implies $H(\pi(0), s')$. Applying condition (ii) of Definition 5 successively on $\pi(0), \pi(1), \pi(2), \ldots$ this yields the existence of states $\pi'(i) \in S', i \geq 0$, such that

$$\pi'(0) = s' \wedge (\forall i \geq 0 : R'(\pi'(i), \pi'(i+1)) \wedge H(\pi(i+1), \pi'(i+1))),$$

so π' is a path in K', and it corresponds to π by construction. □

Theorem 7 *Assume* $K \preccurlyeq K'$. *Then for every ACTL* formula* ϕ *with atomic propositions in* AP'

$$(K' \models \phi) \textit{ implies } (K \models \phi)$$

Proof. Let ϕ an ACTL* formula as defined in Section 3.4. Suppose $K' \models \phi$, which is equivalent to $\forall s'_0 \in S'_0 : (K', s'_0) \models \phi$. We have to show that for any $s_0 \in S_0$, $(K, s_0) \models \phi$ holds. This is achieved by proving the more general fact that

$$\forall (s, s') \in H : ((K', s') \models \phi) \Rightarrow ((K, s) \models \phi) \qquad (*)$$

which implies our original proof goal. The proof of (*) is performed by structural induction over the formula ϕ. Assume $(s, s') \in H$ and $(K', s') \models \phi$ for the rest of this proof.

(1) If ϕ is an atomic proposition, then $(K, s) \models \phi$ if and only if $\phi \in L(s)$. Since $(K', s') \models \phi$ by assumption, ϕ must be contained in AP'. Since K' simulates K, we can conclude $L(s) \cap AP' = L'(s')$ (condition (i) of Definition 5). Now $K' \models \phi$, and therefore $\phi \in L'(s')$ and $L'(s') = L(s) \cap AP'$, so $\phi \in L(s)$ follows.

(2) Let $\phi = \neg\phi_1$ and suppose $(K', s') \models \phi$. Since ϕ is an ACTL* formula ϕ_1 must be an atomic proposition. This implies that $\phi_1 \notin L'(s')$ and, since $L'(s') = L(s) \cap AP'$ and $\phi_1 \in AP'$ also $\phi_1 \notin L(s)$. This means $K, s \not\models \phi_1$ and therefore $K, s \models \neg\phi_1$ which is equivalent to $K, s \models \phi$.

(3) Let $\phi = \phi_1 \vee \phi_2$ such that ϕ_i are state formulas for $i = 1, 2$ and $(K, s) \models \phi_i$ whenever $(K', s') \models \phi_i$. Since $(K', s') \models \phi$, $(K', s') \models \phi_1$ or $(K', s') \models \phi_2$ follows. If $(K', s') \models \phi_1$ then we know already that $(K, s) \models \phi_1$ follows, and this implies $(K, s) \models \phi_1 \vee \phi_2$. The same argument applies if $(K', s') \models \phi_2$. As a consequence $(K, s) \models \phi_1$ or $(K, s) \models \phi_2$ holds, which proves $(K, s) \models \phi_1 \vee \phi_2$.

(4) Let $\phi = \phi_1 \wedge \phi_2$ such that ϕ_i are state formulas for $i = 1, 2$ and $(K, s) \models \phi_i$ whenever $(K', s') \models \phi_i$. This case is handled in analogy to (3).

(5) Let ϕ a state formula, such that $(K, s) \models \phi$ whenever $(K', s') \models \phi$. Let π a path with $\pi(0) = s$, and π' its corresponding path in K', starting at $s' = \pi'(0)$ (this path exists according to Lemma 14). Suppose that $K', \pi' \models \phi$ (remember that every state formula is also a path formula). This is equivalent to $K', \pi'(0) \models \phi$, so by our assumption $K, \pi(0) \models \phi$. This implies that $K, \pi \models \phi$. Now we have shown that $K, \pi \models \phi$ whenever $K', \pi' \models \phi$ on a path π' corresponding to π.

(6) Let $\phi = \mathbf{A}\psi$ such that ψ is a path formula and $K, \pi \models \psi$ whenever

$K', \pi' \models \psi$, where π, π' are corresponding paths starting in s and s', respectively. Now $K, s \models \mathbf{A}\psi$ is equivalent to the condition that every path π emanating from s satisfies $K, \pi \models \psi$. Since $K', s' \models \mathbf{A}\psi$ we know that $K', \pi'' \models \psi$ for *every* π'' starting at s', so this holds in particular for the path π' corresponding to π. Therefore also $K, \pi \models \psi$ holds, and this implies $K, s \models \mathbf{A}\psi$ since π was an arbitrary path starting at s.

(7) Let $\phi = \psi_1 \vee \psi_2$, such that ψ_i are path formulas where $K, \pi \models \psi_i$ whenever $K', \pi' \models \psi_i$ for $i = 1, 2$ on a path π' corresponding to π. Suppose $K', \pi' \models \psi_1 \vee \psi_2$. This means that $K', \pi' \models \psi_1$ or $K', \pi' \models \psi_2$. By (5) we can deduce that $K, \pi \models \psi_1$ or $K, \pi \models \psi_2$, and we have shown that $K, \pi \models \psi_1 \vee \psi_2$ whenever $K', \pi' \models \psi_1 \vee \psi_2$ on a path π' corresponding to π.

(8) Let $\phi = \psi_1 \wedge \psi_2$, such that ψ_i are path formulas where $K, \pi \models \psi_i$ whenever $K', \pi' \models \psi_i$ for $i = 1, 2$ on a path π' corresponding to π. With an argument analogous to (7) it is shown that $K, \pi \models \psi_1 \wedge \psi_2$ whenever $K', \pi' \models \psi_1 \wedge \psi_2$ on a path π' corresponding to π.

(9) Let $\phi = \mathbf{X}\psi$ and ψ a path formula such that $K, \pi \models \psi$ holds whenever $K', \pi' \models \psi$ holds on a path π' corresponding to π. Now $K', \pi' \models \mathbf{X}\psi$ is equivalent to $K', \pi'^1 \models \psi$. Since π'^1 corresponds to π^1 we know already that $K', \pi'^1 \models \psi$ implies $K, \pi^1 \models \psi$. As a consequence $K, \pi \models \mathbf{X}\psi$ also holds.

(10) The cases $\phi = \mathbf{F}\psi, \phi = \mathbf{G}\psi, \phi = \psi_1 \mathbf{U} \psi_2, \phi = \psi_1 \mathbf{R} \psi_2$ are shown in analogy to (9), and this completes the proof. □

Exercise 15. Give the following explanations regarding the proof of Theorem 7:

1. Give a detailed formal explanation why the theorem follows from (*).
2. Give a formal syntax specification for ACTL* similar to EBNF notation introduced for CTL* formulas in Section 3.2.
3. Explain how ACTL* is inductively defined according to Definition 11:
   (a) What might be a suitable universe U?
   (b) What is the base set B?
   (c) Which are the constructors $r \in K$?

4. Explain how the proof of Theorem 7 applies the principle of structural induction.

□

Theorem 8 *Let* $K = (S, S_0, R, L)$ *and* $K' = (S, S'_0, R', L)$ *Kripke structures with variable symbols from* V *and atomic propositions* AP*, using the same set of states* S *and the same labelling function* $L : S \to 2^{AP}$*. Let* $\mathcal{I}$*,* $\mathcal{I}'$ *be the first order predicates characterising the initial states* S_0 *and* S'_0*, respectively, and* $\mathcal{R}$*,* $\mathcal{R}'$ *the first order predicates characterising the transition relations* R *and* R'*, respectively. Suppose that*

- $\mathcal{I} \Rightarrow \mathcal{I}'$
- $\mathcal{R} \Rightarrow \mathcal{R}'$

Then $K \preccurlyeq K'$*.*

Proof. See Exercise 16. □

Exercise 16. Prove Theorem 8, using the facts on first order representations given in Section 2. □

7.5 Bisimulations

Having studied simulations it is natural to ask how much we have to strengthen the simulation definition in order to be sure that *all* CTL* formulas valid in one Kripke structure are also valid in the other one and vice versa. This leads us to the concept of *bisimulation*.

Definition 7 (Bisimulation) *Given two Kripke structures* $K = (S, S_0, R, L), K' = (S', S'_0, R', L')$ *such that* K, K' *refer to the same set of atomic propositions* AP*. A relation* $B \subseteq S \times S'$ *is called* bisimulation *(relation) between* K *and* K'*, if and only if the following conditions hold for all* $s \in S, s' \in S'$ *with* $B(s, s')$*:*

1. $L(s) = L'(s')$

2. $\forall s_1 \in S : R(s, s_1) \Rightarrow \exists s'_1 \in S' \,.\, R'(s', s'_1) \wedge B(s_1, s'_1)$

3. $\forall s'_1 \in S' : R'(s', s'_1) \Rightarrow \exists s_1 \in S \,.\, R(s, s_1) \wedge B(s_1, s'_1)$

We write $K \equiv K'$ *if there exists a bisimulation* B *between* K *and* K' *such that*

$$(\forall s_0 \in S_0 : \exists s'_0 \in S'_0 \,.\, B(s_0, s'_0)) \wedge (\forall s'_0 \in S'_0 : \exists s_0 \in S_0 \,.\, B(s_0, s'_0))$$

□

Bisimilar Kripke structures satisfy the same CTL* formulas[2]:

Theorem 9 *If* $K \equiv K'$ *and* $\phi \in CTL^*$*, then*

$$(K \models \phi) \text{ if and only if } (K' \models \phi)$$

□

7.6 Predicate Abstraction

With the knowledge of Section 7.3 alone we could construct abstractions only from the original Kripke structure $K = (S, S_0, R, L)$. This is unsatisfactory, since the very objective of abstraction is to help in situations where the original Kripke structure is too large to be represented in an explicit way. Fortunately there is an alternative for constructing abstractions: Having defined auxiliary variables a_i and associated expressions $a_i = e_i(x_1^i, x_2^i, \ldots)$ we can lift the original predicates $\mathcal{I}, \mathcal{R}$ over $x_j \in V$ specifying initial state and transition relation of K to predicates over a_i specifying initial state and transition relation of the abstracted Kripke structure $K' = (S', S'_0, R', L')$. In the next section we will see that this relation can be further approximated by simpler predicates that still preserve the simulation relation but are coarser and therefore even simpler to compute.

[2] For a proof, see [5, pp. 171].

Definition 8 *Let* $K = (S, S_0, R, L)$ *a Kripke structure with variables from* $V = \{x_1, \ldots, x_n\}$ *and* ϕ *a predicate with free variables over* V*. Let* $AUX = \{a_1, \ldots, a_k\}$ *a set of auxiliary variables defining an abstraction relation via expressions* $a_i = e_i(x_1^i, x_2^i, \ldots), i = 1, \ldots, k$*. Then the* lifting of ϕ *with respect to this abstraction is denoted by* $[\phi]$ *and defined as*

$$[\phi] \equiv_{def} \exists \xi_1, \ldots, \xi_n . (\forall i = 1, \ldots, k : a_i = e_i(\xi_1^i, \ldots, \xi_n^i)) \wedge \phi[\xi_1/x_1, \ldots, \xi_n/x_n]$$

□

Theorem 10 *Let* $K = (S, S_0, R, L)$ *a Kripke structure with variables from* $V = \{x_1, \ldots, x_n\}$ *and* ϕ *a predicate with free variables over* V*. Let* $AUX = \{a_1, \ldots, a_k\}$ *a set of auxiliary variables defining an abstraction relation via expressions* $a_i = e_i(x_1^i, x_2^i, \ldots), i = 1, \ldots, k$*. Let* $K' = (S', S_0', R', L')$ *denote the abstracted Kripke structure obtained by factorisation with* $\sim$ *as described in Section 7.3. Let* $\mathcal{I}, \mathcal{R}$ *denote initial condition and transition relation of* K*.*

Then initial condition and transition relation of K' *are given by the lifted predicates*

$$[\mathcal{I}] \text{ and } [\mathcal{R}]$$

Proof. Applying Definition 8 on $\mathcal{I}$ and $\mathcal{R}$ yields

$$\begin{aligned}
[\mathcal{I}] &\equiv \exists \xi_1, \ldots, \xi_n \,.\, (\forall i = 1, \ldots, k \,.\, a_i = e_i(\xi_1, \ldots, \xi_n)) \wedge \mathcal{I}[\xi_1/x_1, \ldots, \xi_n/x_n] \\
[\mathcal{R}] &\equiv \exists \xi_1, \ldots, \xi_n \,.\, \exists \xi_1', \ldots, \xi_n' \,.\, (\forall i = 1, \ldots, k \,.\, a_i = e_i(\xi_1, \ldots, \xi_n)) \wedge \\
&\qquad (\forall i = 1, \ldots, k \,.\, a_i' = e_i(\xi_1', \ldots, \xi_n')) \wedge \\
&\qquad \mathcal{R}[\xi_1/x_1, \ldots, \xi_n/x_n, \xi_1'/x_1', \ldots, \xi_n'/x_n']
\end{aligned}$$

According to Lemma 5 these formulas represent initial condition $\mathcal{I}/_\sim$ and transition relation $\mathcal{R}/_\sim$ of K'. □

The essential contribution of Theorem 10 is that for constructing a suitable simulation for proving the validity of an ACTL formula, it is *not* necessary to enumerate states and simulation relation in an explicit way. Instead, we can lift initial condition and transition relation of the original

Kripke structure, thereby obtaining the initial condition and transition relation of the simulation.

Example 9. Consider again the traffic light example introduced in Example 7 and extended further in Example 8. In contrast to the latter example, we will now construct the simulation TL/ $\sim$ by means of predicate abstraction according to Theorem 10, *without* explicitly enumerating states, equivalence classes, and the elements of the simulation relation. Moreover, we do not need to consider the extended Kripke structure TL_e, but the original structure TL without auxiliary variables.

Representing TL as a Kripke structure over variables and with initial condition and transition relation results in the definition

$$
\begin{array}{rcl}
\mathrm{TL} & & \\
V & = & \{\ell, \mathrm{tl}\} \\
D_\ell & = & \{l_0, l_1, l_2, l_3\} \quad \text{the } l_i \text{ are now enum values for state variable } \ell \\
D_{\mathrm{tl}} & = & \{\mathrm{red}, \mathrm{yellow}, \mathrm{green}\} \\
D & = & D_\ell \cup D_{\mathrm{tl}} \\
S & \subseteq & V \to D \quad \text{we don't need to enumerate the states explicitly} \\
\mathcal{I} & \equiv & \ell = l_0 \wedge \mathrm{tl} = \mathrm{red} \\
\mathcal{R} & \equiv & (\ell = l_0 \wedge \mathrm{tl} = \mathrm{red} \wedge \ell' = l_1 \wedge \mathrm{tl}' = \mathrm{yellow}) \vee \\
 & & (\ell = l_1 \wedge \mathrm{tl} = \mathrm{yellow} \wedge \ell' = l_2 \wedge \mathrm{tl}' = \mathrm{green}) \vee \\
 & & (\ell = l_2 \wedge \mathrm{tl} = \mathrm{green} \wedge \ell' = l_3 \wedge \mathrm{tl}' = \mathrm{yellow}) \vee \\
 & & (\ell = l_3 \wedge \mathrm{tl} = \mathrm{yellow} \wedge \ell' = l_0 \wedge \mathrm{tl}' = \mathrm{red}) \\
S_0 & = & \{s : V \to D \mid s \models \mathcal{I}\} = \{\{\ell \mapsto l_0, \mathrm{tl} \mapsto \mathrm{red}\}\} \\
AP & = & \{\ell = l_0, \ell = l_1, \ell = l_2, \ell = l_3, \mathrm{tl} = \mathrm{red}, \mathrm{tl} = \mathrm{yellow}, \mathrm{tl} = \mathrm{green}\} \\
L & = & \{s \mapsto \{\ell = s(\ell), \mathrm{tl} = s(\mathrm{tl})\} \mid s \in S\}
\end{array}
$$

As explained before, we introduce auxiliary variable

$$\mathrm{stops} \equiv (\mathrm{tl} = \mathrm{red} \vee \mathrm{tl} = \mathrm{yellow}),$$

so we have just one abstraction variable $a_1 = \mathrm{stops}$. Now we lift the initial condition $\mathcal{I}$ according to Definition 8; this results in

$$
\begin{array}{rcl}
[\mathcal{I}] & \equiv & \exists \xi_\ell, \xi_{\mathrm{tl}} \,.\, \big(\mathrm{stops} = (\xi_{\mathrm{tl}} = \mathrm{red} \vee \xi_{\mathrm{tl}} = \mathrm{yellow})\big) \wedge \mathcal{I}[\xi_\ell/\ell, \xi_{\mathrm{tl}}/\mathrm{tl}] \\
 & \equiv & \exists \xi_\ell, \xi_{\mathrm{tl}} \,.\, \big(\mathrm{stops} = (\xi_{\mathrm{tl}} = \mathrm{red} \vee \xi_{\mathrm{tl}} = \mathrm{yellow})\big) \wedge \big(\xi_\ell = l_0 \wedge \xi_{\mathrm{tl}} = \mathrm{red}\big) \\
 & \equiv & \mathrm{stops}
\end{array}
$$

For lifting the transition relation, we need bound variable ξ for both the unprimed and the primed variable symbols. This results in

$$\begin{aligned}
[\mathcal{R}] \quad &\equiv \quad \exists \xi_\ell, \xi_{tl}, \xi'_\ell, \xi'_{tl} \,.\, \big(\text{stops} = (\xi_{tl} = \text{red} \vee \xi_{tl} = \text{yellow})\big) \wedge \\
&\qquad \big(\text{stops}' = (\xi'_{tl} = \text{red} \vee \xi'_{tl} = \text{yellow})\big) \wedge \mathcal{R}[\xi_\ell/\ell, \xi_{tl}/tl, \xi'_\ell/\ell', \xi'_{tl}/tl'] \\
&\equiv \quad \exists \xi_\ell, \xi_{tl}, \xi'_\ell, \xi'_{tl} \,.\, \big(\text{stops} = (\xi_{tl} = \text{red} \vee \xi_{tl} = \text{yellow})\big) \wedge \\
&\qquad \big(\text{stops}' = (\xi'_{tl} = \text{red} \vee \xi'_{tl} = \text{yellow})\big) \wedge \\
&\qquad \big((\xi_\ell = l_0 \wedge \xi_{tl} = \text{red} \wedge \xi'_\ell = l_1 \wedge \xi'_{tl} = \text{yellow}) \vee \\
&\qquad (\xi_\ell = l_1 \wedge \xi_{tl} = \text{yellow} \wedge \xi'_\ell = l_2 \wedge \xi'_{tl} = \text{green}) \vee \\
&\qquad (\xi_\ell = l_2 \wedge \xi_{tl} = \text{green} \wedge \xi'_\ell = l_3 \wedge \xi'_{tl} = \text{yellow}) \vee \\
&\qquad (\xi_\ell = l_3 \wedge \xi_{tl} = \text{yellow} \wedge \xi'_\ell = l_0 \wedge \xi'_{tl} = \text{red})\big) \\
&\equiv \quad (\text{stops} \wedge \text{stops}') \vee \\
&\qquad (\text{stops} \wedge \neg\text{stops}') \vee \\
&\qquad (\neg\text{stops} \wedge \text{stops}') \vee \\
&\qquad (\text{stops} \wedge \text{stops}') \\
&\equiv \quad (\text{stops} \vee \text{stops}')
\end{aligned}$$

The graphical structure of this Kripke structure is again the one already presented in Fig 7.2. □

Example 10. Consider again the model displayed in Fig. 7.3 with integer variables x, y having unbounded range. With the knowledge about simulations and predicate abstraction it is now possible to give a rigorous proof for the formula $\neg\mathbf{EF}(10 \wedge \text{odd}(y))$. First we observe that

$$\neg\mathbf{EF}(10 \wedge \text{odd}(y)) \equiv \mathbf{AG}(\neg 10 \vee \neg\text{odd}(y))$$

so our proof objective is an ACTL formula. As a possible abstraction for this objective consider

$$\begin{aligned}
a_0 &= 10 \\
a_1 &= 11 \\
a_2 &= 12 \\
a_3 &= \text{odd}(y)
\end{aligned} \tag{7.4}$$

Note, that $a_0, \ldots, a_3$ form not the simplest abstraction possible to show the required property - indeed, abstraction by a_0 and a_3 would suffice. The

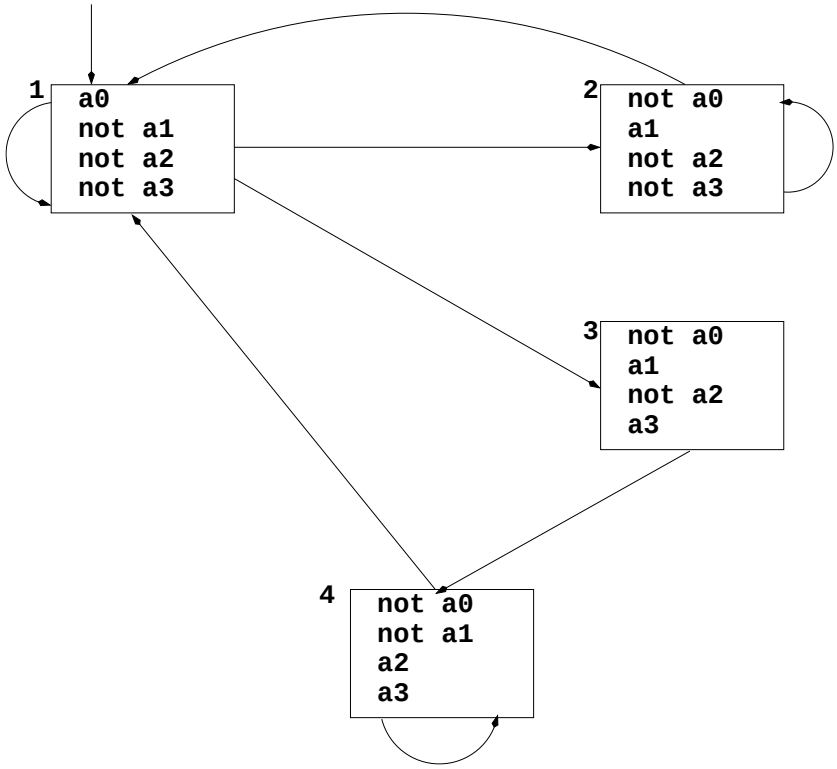


Figure 7.4: Kripke structure for abstracted model from Example 10.

effect of the coarser abstraction would be, that proving several other formulas like $\mathbf{AG}(\neg \mathtt{l2} \vee \mathtt{odd}(\mathtt{y}))$ becomes impossible in the resulting abstracted Kripke structure.

We proceed now to construct the resulting abstracted Kripke structure without first unfolding the one of the concrete system, but exploiting instead its predicates for initial state and transition relation.

Step. 1. Specify initial condition of the concrete system: From Fig. 7.3 we derive

$$\mathcal{I}(\mathtt{l0}, \mathtt{l1}, \mathtt{l2}, \mathtt{x}, \mathtt{y}) \equiv \mathtt{l0} \wedge \neg \mathtt{l1} \wedge \neg \mathtt{l2} \wedge \mathtt{y} = 0$$

Step. 2. Specify formula for the transition relation of the concrete system: Evaluating Fig. 7.3 again, we derive

$$\begin{aligned}
&\mathcal{R}(\mathtt{l0}, \mathtt{l1}, \mathtt{l2}, \mathtt{x}, \mathtt{y}, \mathtt{l0}', \mathtt{l1}', \mathtt{l2}', \mathtt{x}', \mathtt{y}') \equiv \\
&\quad ((\mathtt{l0} \wedge \mathtt{x} \leq \mathtt{y} \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{l0}') \vee \\
&\quad (\mathtt{l0} \wedge \mathtt{x} > \mathtt{y} \wedge \mathtt{y}' = \mathtt{y} + \mathtt{x} \wedge \mathtt{l1}') \vee \\
&\quad (\mathtt{l1} \wedge \mathtt{x} \leq 0 \wedge \neg \mathtt{odd}(\mathtt{y}) \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{l0}') \vee \\
&\quad (\mathtt{l1} \wedge \mathtt{odd}(\mathtt{y}) \wedge \mathtt{y}' = -1 \wedge \mathtt{l2}') \vee \\
&\quad (\mathtt{l1} \wedge \mathtt{x} > 0 \wedge \neg \mathtt{odd}(\mathtt{y}) \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{l1}') \vee \\
&\quad (\mathtt{l2} \wedge \mathtt{x} \leq 0 \wedge \mathtt{y}' = 0 \wedge \mathtt{l0}') \vee \\
&\quad (\mathtt{l2} \wedge \mathtt{x} > 0 \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{l2}')) \wedge \\
&\quad ((\mathtt{l0} \wedge \neg \mathtt{l1} \wedge \neg \mathtt{l2}) \vee (\neg \mathtt{l0} \wedge \mathtt{l1} \wedge \neg \mathtt{l2}) \vee (\neg \mathtt{l0} \wedge \neg \mathtt{l1} \wedge \mathtt{l2})) \wedge \\
&\quad ((\mathtt{l0}' \wedge \neg \mathtt{l1}' \wedge \neg \mathtt{l2}') \vee (\neg \mathtt{l0}' \wedge \mathtt{l1}' \wedge \neg \mathtt{l2}') \vee (\neg \mathtt{l0}' \wedge \neg \mathtt{l1}' \wedge \mathtt{l2}'))
\end{aligned}$$

Step. 3. Compute the abstracted initial condition $\mathcal{I}/_{\sim} = [\mathcal{I}]$: Applying Definition 8 on $[\mathcal{I}]$ for the given abstraction (7.4) results in

$$\begin{aligned}
[\mathcal{I}](a_0, a_1, a_2, a_3) \quad &\equiv \quad \exists \xi_0, \xi_1, \xi_2, \xi_3, \xi_4 \,. \\
&\qquad\qquad a_0 = \xi_0 \wedge a_1 = \xi_1 \wedge a_2 = \xi_2 \wedge a_3 = \mathtt{odd}(\xi_4) \wedge \\
&\qquad\qquad \xi_0 \wedge \neg \xi_1 \wedge \neg \xi_2 \wedge \xi_4 = 0 \\
&\equiv \quad a_0 \wedge \neg a_1 \wedge \neg a_2 \wedge \neg a_3
\end{aligned}$$

Step. 4. Compute the abstracted transition relation $\mathcal{R}/_{\sim} = [\mathcal{R}]$: Applying

Definition 8 on $[\mathcal{R}]$ for the given abstraction (7.4) results in

$$
\begin{aligned}
&[\mathcal{R}](a_0, a_1, a_2, a_3, a_0', a_1', a_2', a_3') \equiv \\
&\quad \exists \xi_0, \xi_1, \xi_2, \xi_3, \xi_4, \xi_0', \xi_1', \xi_2', \xi_3', \xi_4' . \\
&\qquad a_0 = \xi_0 \wedge a_1 = \xi_1 \wedge a_2 = \xi_2 \wedge a_3 = \mathtt{odd}(\xi_4) \wedge \\
&\qquad a_0' = \xi_0' \wedge a_1' = \xi_1' \wedge a_2' = \xi_2' \wedge a_3' = \mathtt{odd}(\xi_4') \wedge \\
&\qquad ((\xi_0 \wedge \xi_3 \leq \xi_4 \wedge \xi_4' = \xi_4 \wedge \xi_0') \vee \\
&\qquad (\xi_0 \wedge \xi_3 > \xi_4 \wedge \xi_4' = \xi_4 + \xi_3 \wedge \xi_1') \vee \\
&\qquad (\xi_1 \wedge \xi_3 \leq 0 \wedge \neg \mathtt{odd}(\xi_4) \wedge \xi_4' = \xi_4 \wedge \xi_0') \vee \\
&\qquad (\xi_1 \wedge \mathtt{odd}(\xi_4) \wedge \xi_4' = -1 \wedge \xi_2') \vee \\
&\qquad (\xi_1 \wedge \xi_3 > 0 \wedge \neg \mathtt{odd}(\xi_4) \wedge \xi_4' = \xi_4 \wedge \xi_1') \vee \\
&\qquad (\xi_2 \wedge \xi_3 \leq 0 \wedge \xi_4' = 0 \wedge \xi_0') \vee \\
&\qquad (\xi_2 \wedge \xi_3 > 0 \wedge \xi_4' = \xi_4 \wedge \xi_2')) \wedge \\
&\qquad ((\xi_0 \wedge \neg \xi_1 \wedge \neg \xi_2) \vee (\neg \xi_0 \wedge \xi_1 \wedge \neg \xi_2) \vee (\neg \xi_0 \wedge \neg \xi_1 \wedge \xi_2)) \wedge \\
&\qquad ((\xi_0' \wedge \neg \xi_1' \wedge \neg \xi_2') \vee (\neg \xi_0' \wedge \xi_1' \wedge \neg \xi_2') \vee (\neg \xi_0' \wedge \neg \xi_1' \wedge \xi_2')) \equiv \\
&\quad ((a_0 \wedge a_3' = a_3 \wedge a_0') \vee (a_0 \wedge a_3' \wedge a_1') \vee (a_0 \wedge \neg a_3' \wedge a_1') \vee \\
&\qquad (a_1 \wedge \neg a_3 \wedge a_3' = a_3 \wedge a_0') \vee (a_1 \wedge \neg a_3 \wedge a_3' = a_3 \wedge a_1') \vee \\
&\qquad (a_1 \wedge a_3 \wedge a_3' \wedge a_2') \vee (a_2 \wedge \neg a_3' \wedge a_0') \vee (a_2 \wedge a_3' = a_3 \wedge a_2')) \wedge \\
&\qquad ((a_0 \wedge \neg a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge \neg a_1 \wedge a_2)) \wedge \\
&\qquad ((a_0' \wedge \neg a_1' \wedge \neg a_2') \vee (\neg a_0' \wedge a_1' \wedge \neg a_2') \vee (\neg a_0' \wedge \neg a_1' \wedge a_2')) \equiv \\
&\qquad ((a_0 \wedge a_3' = a_3 \wedge a_0') \vee (a_0 \wedge a_1') \vee \\
&\qquad (a_1 \wedge \neg a_3 \wedge a_3' = a_3 \wedge (a_0' \vee a_1')) \vee (a_1 \wedge a_3 \wedge a_3' \wedge a_2') \vee \\
&\qquad (a_2 \wedge \neg a_3' \wedge a_0') \vee (a_2 \wedge a_3' = a_3 \wedge a_2')) \wedge \\
&\qquad ((a_0 \wedge \neg a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge \neg a_1 \wedge a_2)) \wedge \\
&\qquad ((a_0' \wedge \neg a_1' \wedge \neg a_2') \vee (\neg a_0' \wedge a_1' \wedge \neg a_2') \vee (\neg a_0' \wedge \neg a_1' \wedge a_2'))
\end{aligned}
$$

The resulting abstracted Kripke structure is displayed in Fig. 7.4, and it is trivial to see from the graphic representation that $\mathbf{AG}(\neg \mathtt{10} \vee \neg \mathtt{odd}(y))$ holds, because this formula is equivalent to $\mathbf{AG}(\neg a_0 \vee \neg a_3)$ and the Kripke structure in Fig. 7.4 simulates the concrete system from Fig. 7.3 by construction. □

Exercise 17. Check whether the following C program fragment terminates:

```
1     uint32_t x,y;
2     y = 1;
3     while ( y < 256 ) {
4      x = input(); // Assume 0 <= x <= 15
5      if ( x > y ) {
6        y = y * x;
7      }
8     }
9     exit();
```

Perform this check by means of an abstraction function α that calculates the minimal number of bits needed to represent an integral number:

$$\alpha : \mathbb{N}_0 \to \mathbb{N}_0; \quad x \mapsto \begin{cases} 1, & \text{if } x = 0 \\ \lfloor \log_2 x \rfloor + 1, & \text{if } x > 0 \end{cases}$$

Observe that, since $\log_b x \cdot y = \log_b x + \log_b y$, the following estimates hold:

$$\begin{aligned} &\alpha(x \cdot y) \leq \alpha(x) + \alpha(y) \\ &N \leq \alpha(x) + \alpha(y) \Rightarrow N - 1 \leq \alpha(x \cdot y) \\ &\alpha(x) + \alpha(y) \leq N \Rightarrow \alpha(x \cdot y) \leq N \end{aligned}$$

Prove termination or non-termination along the following lines:

1. Specify initial condition $\mathcal{I}$ and transition formula $\mathcal{R}$ of the concrete program fragment above.

2. Now use the abstraction $a_1 = \alpha(x), a_2 = \alpha(y)$. and calculate the abstracted formulas $[\mathcal{I}]$ and $[\mathcal{R}]$.

3. Unfold the Kripke structure of the abstracted system given by $[\mathcal{I}]$ and $[\mathcal{R}]$ and sketch how the model checking algorithms introduced in Section 4 come to a conclusion about termination or non-termination.

□

Example 11. We present an alternative solution for Exercise 17 which uses another abstraction and motivates the concept of *abstract interpretation*.

The initial condition of the program from Exercise 17 is

$$\mathcal{I}(p, x, y) \equiv p = 1$$

The transition relation is specified by the predicate

$$\begin{aligned}
&\mathcal{R}(p, x, y, p', x', y') \equiv \\
&\quad (p = 1 \wedge p' = 2 \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 2 \wedge p' = 3 \wedge x' = x \wedge y' = 1) \vee \\
&\quad (p = 3 \wedge p' = 9 \wedge y \geq 256 \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 3 \wedge p' = 4 \wedge y < 256 \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 4 \wedge p' = 5 \wedge 0 \leq x' \leq 15 \wedge y' = y) \vee \\
&\quad (p = 5 \wedge p' = 3 \wedge x \leq y \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 5 \wedge p' = 6 \wedge x > y \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 6 \wedge p' = 3 \wedge x' = x \wedge y' = y \cdot x)
\end{aligned}$$

We choose the following abstraction functions – they are induced by a scan of "relevant" decisions in the program:

$$\begin{aligned}
a_0(p, x, y) &= p \\
a_1(p, x, y) &= (x \in [0, 15]) \\
a_2(p, x, y) &= (y < 256) \\
a_3(p, x, y) &= (x > y)
\end{aligned}$$

In order to prove that the program never terminates we try to prove ACTL formula

$$\mathbf{AG}(a_0 \neq 9)$$

which exactly expresses non-termination.

Applying the predicate abstraction principle on abstraction functions $a_0, \ldots, a_3$ results in

$$[\mathcal{I}] \equiv a_0 = 1$$

for the initial condition; for the abstracted transition relation we get[3]

$$
\begin{aligned}
&[\mathcal{R}] \equiv \exists p, x, y, p', x', y' : \\
&\quad a_0 = p \wedge a_1 = (x \in [0, 15]) \wedge a_2 = (y < 256) \wedge a_3 = (x > y) \wedge \\
&\quad a_0' = p' \wedge a_1' = (x' \in [0, 15]) \wedge a_2' = (y' < 256) \wedge a_3' = (x' > y') \wedge \\
&\quad ((p = 1 \wedge p' = 2 \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 2 \wedge p' = 3 \wedge x' = x \wedge y' = 1) \vee \\
&\quad (p = 3 \wedge p' = 9 \wedge y \geq 256 \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 3 \wedge p' = 4 \wedge y < 256 \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 4 \wedge p' = 5 \wedge 0 \leq x' \leq 15 \wedge y' = y) \vee \\
&\quad (p = 5 \wedge p' = 3 \wedge x \leq y \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 5 \wedge p' = 6 \wedge x > y \wedge x' = x \wedge y' = y) \vee \\
&\quad (p = 6 \wedge p' = 3 \wedge x' = x \wedge y' = y \cdot x))
\end{aligned}
$$

Replacing terms which may be directly expressed by a_i or $\neg a_i$ due to equality or direct implication results in the fact that $[\mathcal{R}]$ implies

$$
\begin{aligned}
&R_1 \equiv \exists x, y, x', y' : \\
&\quad a_1 = (x \in [0, 15]) \wedge a_2 = (y < 256) \wedge a_3 = (x > y) \wedge \\
&\quad a_1' = (x' \in [0, 15]) \wedge a_2' = (y' < 256) \wedge a_3' = (x' > y') \wedge \\
&\quad ((a_0 = 1 \wedge a_0' = 2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
&\quad (a_0 = 2 \wedge a_0' = 3 \wedge a_1' = a_1 \wedge a_2') \vee \\
&\quad (a_0 = 3 \wedge a_0' = 9 \wedge \neg a_2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
&\quad (a_0 = 3 \wedge a_0' = 4 \wedge a_2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
&\quad (a_0 = 4 \wedge a_0' = 5 \wedge a_1' \wedge a_2' = a_2) \vee \\
&\quad (a_0 = 5 \wedge a_0' = 3 \wedge \neg a_3 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
&\quad (a_0 = 5 \wedge a_0' = 6 \wedge a_3 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
&\quad (a_0 = 6 \wedge a_0' = 3 \wedge a_1' = a_1 \wedge y' = y \cdot x))
\end{aligned}
$$

We use the following observation.

$$
\begin{aligned}
&a_1 \wedge a_2 \wedge a_3 \wedge y' = y \cdot x \Rightarrow \\
&\quad (x \in [0, 15]) \wedge (y < 256) \wedge (x > y) \wedge y' = y \cdot x \Rightarrow \\
&\quad (x \in [0, 15]) \wedge (y < 15) \wedge (x > y) \wedge y' = y \cdot x \Rightarrow \\
&\quad (y' \leq 210) \Rightarrow \\
&\quad a_2'
\end{aligned}
$$

[3] Observe that we still use p, x, y as in the original transition relation above, but now these symbols are bound to the existential quantifier.

Therefore $R_1 \Rightarrow R_2$ with

$$
\begin{aligned}
R_2 \equiv\ & \exists x, y, x', y' : \\
& a_1 = (x \in [0,15]) \wedge a_2 = (y < 256) \wedge a_3 = (x > y) \wedge \\
& a_1' = (x' \in [0,15]) \wedge a_2' = (y' < 256) \wedge a_3' = (x' > y') \wedge \\
& ((a_0 = 1 \wedge a_0' = 2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 2 \wedge a_0' = 3 \wedge a_1' = a_1 \wedge a_2') \vee \\
& (a_0 = 3 \wedge a_0' = 9 \wedge \neg a_2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 3 \wedge a_0' = 4 \wedge a_2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 4 \wedge a_0' = 5 \wedge a_1' \wedge a_2' = a_2) \vee \\
& (a_0 = 5 \wedge a_0' = 3 \wedge \neg a_3 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 5 \wedge a_0' = 6 \wedge a_3 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 6 \wedge a_0' = 3 \wedge a_1 \wedge a_2 \wedge a_3 \wedge a_2' \wedge a_1' = a_1) \vee \\
& (a_0 = 6 \wedge a_0' = 3 \wedge \neg(a_1 \wedge a_2 \wedge a_3) \wedge a_1' = a_1))
\end{aligned}
$$

Finally $R_2 \Rightarrow R_3$ with

$$
\begin{aligned}
R_3 \equiv\ & (a_0 = 1 \wedge a_0' = 2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 2 \wedge a_0' = 3 \wedge a_1' = a_1 \wedge a_2') \vee \\
& (a_0 = 3 \wedge a_0' = 9 \wedge \neg a_2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 3 \wedge a_0' = 4 \wedge a_2 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 4 \wedge a_0' = 5 \wedge a_1' \wedge a_2' = a_2) \vee \\
& (a_0 = 5 \wedge a_0' = 3 \wedge \neg a_3 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 5 \wedge a_0' = 6 \wedge a_3 \wedge a_1' = a_1 \wedge a_2' = a_2 \wedge a_3' = a_3) \vee \\
& (a_0 = 6 \wedge a_0' = 3 \wedge a_1 \wedge a_2 \wedge a_3 \wedge a_2' \wedge a_1' = a_1) \vee \\
& (a_0 = 6 \wedge a_0' = 3 \wedge \neg(a_1 \wedge a_2 \wedge a_3) \wedge a_1' = a_1)
\end{aligned}
$$

Applying Theorem 8 we conclude that if the Kripke structure associated with R_3 fulfills $\mathbf{AG}(a_0 \neq 9)$, the same holds for the structure associated with $[\mathcal{R}]$, and therefore the same holds for the concrete structure associated with $\mathcal{R}$ (Theorem 10). For $([\mathcal{I}], R_3)$, the Kripke structure looks as shown in Fig. 7.5, and obviously every reachable Kripke state fulfills $a_0 \neq 9$. This proves non-termination of our sample program. □

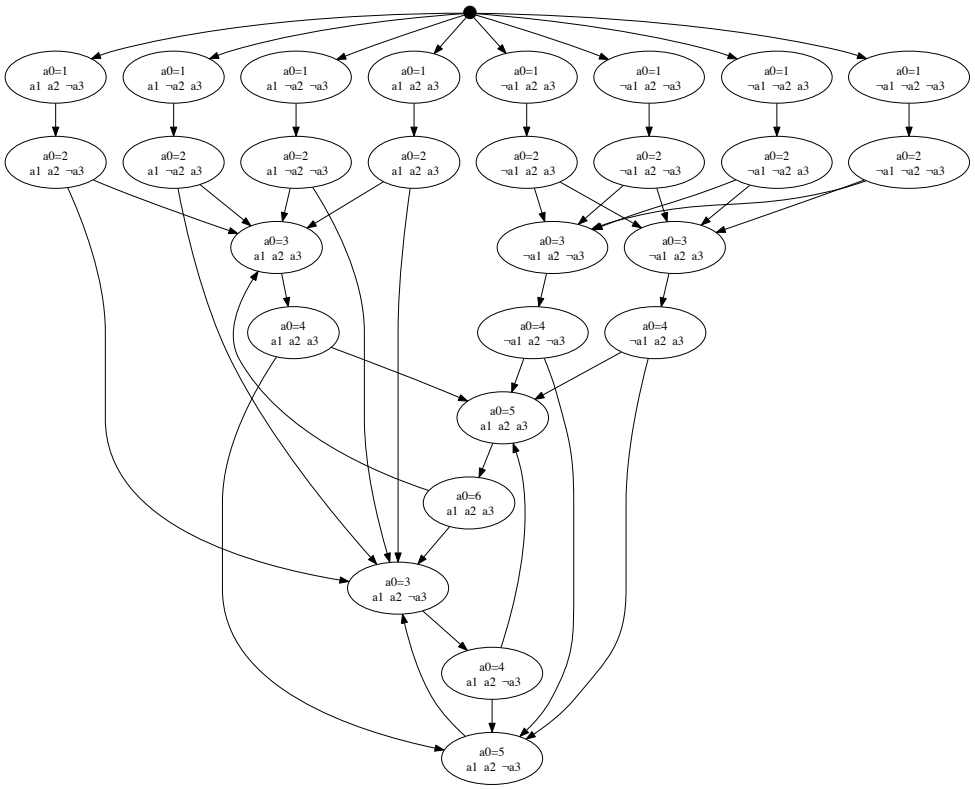


Figure 7.5: Kripke structure associated with $([\mathcal{I}], \mathsf{R}_3)$ from Example 11.

7.7 Predicate Approximation

Depending on the complexity of initial conditions $\mathcal{I}$ and transition relations $\mathcal{R}$ it may be quite hard to compute $[\mathcal{I}]$ and $[\mathcal{R}]$. It is therefore useful to have a technique at hand for further simplifying this computation, at the cost of not arriving exactly at $[\mathcal{I}]$ and $[\mathcal{R}]$, but at *approximations* of these predicates, denoted by $\mathcal{A}(\mathcal{I})$ and $\mathcal{A}(\mathcal{R})$, respectively. We say that predicate ϕ' *approximates* ϕ if $\phi \Rightarrow \phi'$.

Definition 9 *Let ϕ a predicate in negation normal form with free variables in $V = \{x_1, x_2, \ldots\}$. Given an abstraction $a_i = e_i(x_1, x_2, \ldots), i = 1, 2, \ldots$, the* approximation of ϕ *is denoted by $\mathcal{A}(\phi)$. $\mathcal{A}(\phi)$ has free variables in $\{a_1, a_2, \ldots\}$ and is defined inductively by the following rules:*

1. *If ϕ is an atomic proposition*[4]*, then $\mathcal{A}(\phi) =_{def} [\phi]$.*
2. *If $\neg\phi$ is a negated atomic proposition, then $\mathcal{A}(\neg\phi) =_{def} [\neg\phi]$.*
3. $\mathcal{A}(\phi_1 \wedge \phi_2) =_{def} \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$
4. $\mathcal{A}(\phi_1 \vee \phi_2) =_{def} \mathcal{A}(\phi_1) \vee \mathcal{A}(\phi_2)$
5. $\mathcal{A}(\exists x : \phi) =_{def} \exists a : \mathcal{A}(\phi)$
6. $\mathcal{A}(\forall x : \phi) =_{def} \forall a : \mathcal{A}(\phi)$

□

Theorem 11 *Let ϕ a predicate in negation normal form with free variables in $V = \{x_1, x_2, \ldots\}$. Given an abstraction $a_i = e_i(x_1, x_2, \ldots), i = 1, 2, \ldots$, the lifted version of ϕ implies its approximated version, i. e.,*

$$[\phi](a_1, a_2, \ldots) \Rightarrow \mathcal{A}(a_1, a_2, \ldots)$$

Proof. The proof is performed by structural induction over the formula ϕ.
Step 1. If ϕ is atomic or the negation of an atom, $\mathcal{A}(\phi) = [\phi]$, so there is nothing to prove.

[4] Observe that this includes all primitive relations such as $x < y$, $x = f(y, z)$.

Step 2. Suppose $\phi \equiv \phi_1 \wedge \phi_2$ and $[\phi_j] \Rightarrow \mathcal{A}(\phi_j)$, $j = 1, 2$. From the definition of $[\cdot]$ we calculate

$$\begin{aligned}
[\phi_1 \wedge \phi_2] &\equiv \exists \xi_1, \xi_2, \ldots : (\forall i : a_i = e_i(\xi_1, \xi_2, \ldots)) \wedge \\
&\qquad \phi_1(\xi_1/x_1, \xi_2/x_2, \ldots) \wedge \phi_2(\xi_1/x_1, \xi_2/x_2, \ldots) \\
&\Rightarrow (\exists \xi_1, \xi_2, \ldots : (\forall i : a_i = e_i(\xi_1, \xi_2, \ldots)) \wedge \phi_1(\xi_1/x_1, \xi_2/x_2, \ldots)) \wedge \\
&\quad (\exists \xi_1, \xi_2, \ldots : (\forall i : a_i = e_i(\xi_1, \xi_2, \ldots)) \wedge \phi_2(\xi_1/x_1, \xi_2/x_2, \ldots)) \\
&\Rightarrow \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)
\end{aligned}$$

Step 3. Suppose $\phi \equiv \phi_1 \vee \phi_2$ and $[\phi_j] \Rightarrow \mathcal{A}(\phi_j)$, $j = 1, 2$. This case is handled in analogy to Step. 2.

Step 4. Suppose $\phi \equiv \exists x : \phi_1$ and $[\phi_1] \Rightarrow \mathcal{A}(\phi_1)$. Assume without loss of generality that $x \neq x_i$ for all $i = 1, 2, \ldots$ and that $\phi = \phi(x, x_1, x_2, \ldots)$. Then

$$\begin{aligned}
[\exists x : \phi_1] &\equiv \exists \xi_1, \xi_2, \ldots : (\forall i : a_i = e_i(\xi_1, \xi_2, \ldots)) \wedge (\exists \xi : \phi_1(\xi/x, \xi_1/x_1, \xi_2/x_2, \ldots)) \\
&\Rightarrow \exists \xi, \xi_1, \xi_2, \ldots : (\forall i : a_i = e_i(\xi_1, \xi_2, \ldots)) \wedge \phi_1(\xi/x, \xi_1/x_1, \xi_2/x_2, \ldots) \\
&\Rightarrow \exists \xi : (\exists \xi_1, \xi_2, \ldots : (\forall i : a_i = e_i(\xi_1, \xi_2, \ldots)) \wedge \phi_1(\xi/x, \xi_1/x_1, \xi_2/x_2, \ldots)) \\
&\Rightarrow \exists a : \mathcal{A}(\phi_1)
\end{aligned}$$

Step 5. Suppose $\phi \equiv \forall x : \phi_1$ and $[\phi_1] \Rightarrow \mathcal{A}(\phi_1)$. This step is handled in analogy to Step 4. □

Theorem 12 *Given a Kripke structure $K = (S, S_0, R, L)$ with variables in $V = \{x_1, x_2, \ldots\}$, initial condition $\mathcal{I}$ and transition formula $\mathcal{R}$. Given an abstraction $a_i = e_i(x_1, x_2, \ldots), i = 1, 2, \ldots$. Let $K' = (S', S'_0, R', L')$ denote the Kripke structure with variables $\{a_1, a_2, \ldots\}$, initial condition $\mathcal{A}(\mathcal{I})$ and transition relation $\mathcal{A}(\mathcal{R})$. Then*

$$K \preccurlyeq K'$$

Proof. Let K'' denote the abstracted Kripke structure with variables $\{a_1, a_2, \ldots\}$, initial condition $[\mathcal{I}]$ and transition formula $[\mathcal{R}]$. From Theorem 10 and Theorem 6 we know that K'' simulates K. From Theorem 11 we know that $[\mathcal{I}] \Rightarrow \mathcal{A}(\mathcal{I})$ and $[\mathcal{R}] \Rightarrow \mathcal{A}(\mathcal{R})$. Now Theorem 8 implies that K' simulates K''. Since $\preccurlyeq$ is transitive, the theorem follows. □

Exercise 18. Given a Kripke structure $K = (S, S_0, R, L)$ we use the following notation:

- $K_s =_{\text{def}} (S, \{s\}, R, L)$ for $s \in S$
- $s_0 \preccurlyeq s_1 \equiv_{\text{def}}$ there exists a simulation relation $H \subseteq S \times S$ such that $H(s_0, s_1)$

Consider the following algorithm:

$H := \{(s_0, s_1) \mid L(s_0) = L(s_1)\}$;
while H is not a simulation relation **do**
  Choose (s_0, s_1) such that
    $\exists s_0' \in S : R(s_0, s_0') \wedge (\forall s_1' \in S : R(s_1, s_1') \Rightarrow (s_0', s_1') \notin H)$;
  $H := H - \{(s_0, s_1)\}$;
enddo

1. Justify informally why H, as computed by this algorithm, is a simulation relation.
2. Explain the relation between H as computed by this algorithm, $s_0 \preccurlyeq s_1$, K_{s_0} and K_{s_1}.

□

Chapter 8

Abstract Interpretation

8.1 Lattice Abstractions of Primitive Datatypes

For the simplest form of abstract interpretation which is introduced in this section, concrete data types `int`, `float`, `bool` will be abstracted to their interval lattice counterparts as described in the example above. It is also possible to lift concrete n-ary functions

$$f : t_1 \times \ldots \times t_n \rightarrow t_0$$

with $t_i \in \{\texttt{int}, \texttt{float}, \texttt{bool}\}$ to n-ary functions over their concrete data types' lattice counterparts,

$$[f] : L(t_1) \times \ldots \times L(t_n) \rightarrow L(t_0)$$

This lifting operation is performed according to the following construction (arguments a_i in the following definition are intervals over the concrete data types t_i).

$$[f](a_1, \ldots, a_n) =_{\text{def}} \bigsqcup\{[f(x_1, \ldots, x_n), f(x_1, \ldots, x_n)] \mid x_i \in a_i, i = 1, \ldots, n\} \quad (8.1)$$

$$= [\inf\{f(x_1, \ldots, x_n) \mid x_i \in a_i, i = 1, \ldots, n\}, \quad (8.2)$$

$$\sup\{f(x_1, \ldots, x_n) \mid x_i \in a_i, i = 1, \ldots, n\}] \quad (8.3)$$

Intuitively speaking, function value $[f](a_1, \ldots, a_n)$ is constructed as follows:

1. Calculate each concrete function value $f(x_1, \ldots, x_n)$ over arguments x_i from intervals a_i supplied as lattice element arguments to $[f]$.

2. Represent every concrete function value $f(x_1, \ldots, x_n)$ as a single-point interval $[f(x_1, \ldots, x_n), f(x_1, \ldots, x_n)]$ of the interval latice over t_0.

3. The function value $[f](a_1, \ldots, a_n)$ is now determined by calculating the supremum over all of the single-point intervals constructed in step (2); this may be expressed in the simpler form $[\inf\{f(x_1, \ldots, x_n) \mid x_i \in a_i, i = 1, \ldots, n\}, \sup\{f(x_1, \ldots, x_n) \mid x_i \in a_i, i = 1, \ldots, n\}]$.

Observe that for datatype `float` which is a finite subset of $\mathbb{Q}$ it is possible that the infimum and/or supremum used in the construction of $[f](a_1, \ldots, a_n)$ does not exist:

- The infimum or supremum may be an irrational number.
- The infimum or supremum may be a rational number q, but q cannot be represented as a floating point number.

This problem can be addressed by *widening* the theoretically precise interval function value $[\underline{u}, \overline{u}]$ to the closest lower and upper bounds $\underline{v}, \overline{v}$ representable in datatype `float`. The widening operation ensures $[\underline{u}, \overline{u}] \subseteq [\underline{v}, \overline{v}]$, so we know that the exact result is conservatively approximated by the representable interval $[\underline{v}, \overline{v}]$.

Applying the general lifting construction (8.1) to the arithmetic operations $+, -, \cdot, /$ results in the following interval counterparts:

$$
\begin{aligned}
&[\underline{x}, \overline{x}][+][\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
&[\underline{x}, \overline{x}][-][\underline{y}, \overline{y}] = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
&[\underline{x}, \overline{x}][\cdot][\underline{y}, \overline{y}] = [\min S, \max S], \quad S = \{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\} \\
&[\underline{x}, \overline{x}][/][\underline{y}, \overline{y}] = [\underline{x}, \overline{x}][\cdot]1/[\underline{y}, \overline{y}] \\
&1/[0, 0] = \bot \\
&1/[\underline{y}, \overline{y}] = [1/\overline{y}, 1/\underline{y}] \quad \text{if } 0 \notin [\underline{y}, \overline{y}] \\
&1/[\underline{y}, \overline{y}] = [1/\overline{y}, \infty[\quad \text{if } \underline{y} = 0 \wedge 0 < \overline{y} \\
&1/[\underline{y}, \overline{y}] =]-\infty, 1/\underline{y}] \quad \text{if } \underline{y} < 0 \wedge \overline{y} = 0 \\
&1/[\underline{y}, \overline{y}] =]-\infty, \infty[\quad \text{if } \underline{y} < 0 \wedge \overline{y} > 0
\end{aligned}
$$

Boolean expressions and operations are evaluated in $L(\mathbb{B})$ introduced above. In an interval context, the lattice elements are expressed as

$$\begin{aligned} \bot &= [\,] \quad \text{(the empty interval)} \\ \top &= [0,1] \\ \text{true} &= [1,1] \\ \text{false} &= [0,0] \end{aligned}$$

Boolean operations $b(x_1,\ldots,x_n)$ are lifted to $L(\mathbb{B})$-valued operations

$$[b](a_1,\ldots,a_n) = \begin{cases} [0,0] & \text{if} \quad \forall x_i \in a_i, i = 1,\ldots,n : b(x_1,\ldots,x_n) = 0 \\ [1,1] & \text{if} \quad \forall x_i \in a_i, i = 1,\ldots,n : b(x_1,\ldots,x_n) = 1 \\ [0,1] & \text{otherwise} \end{cases}$$

Applying this to the Boolean comparisons $<, \leq, >, \geq, =, \neq$ yields the following lattice counterparts.

$$[\underline{x},\overline{x}]\ [<]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \overline{y} \leq \underline{x} \\ [1,1] & \text{if} \quad \overline{x} < \underline{y} \\ [0,1] & \text{otherwise} \end{cases}$$

$$[\underline{x},\overline{x}]\ [\leq]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \overline{y} < \underline{x} \\ [1,1] & \text{if} \quad \overline{x} \leq \underline{y} \\ [0,1] & \text{otherwise} \end{cases}$$

$$[\underline{x},\overline{x}]\ [>]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \overline{x} \leq \underline{y} \\ [1,1] & \text{if} \quad \overline{y} < \underline{x} \\ [0,1] & \text{otherwise} \end{cases}$$

$$[\underline{x},\overline{x}]\ [\geq]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \overline{x} < \underline{y} \\ [1,1] & \text{if} \quad \overline{y} \leq \underline{x} \\ [0,1] & \text{otherwise} \end{cases}$$

$$[\underline{x},\overline{x}]\ [=]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \overline{x} < \underline{y} \vee \overline{y} < \underline{x} \\ [1,1] & \text{if} \quad \underline{x} = \overline{x} = \underline{y} = \overline{y} \\ [0,1] & \text{otherwise} \end{cases}$$

$$[\underline{x},\overline{x}]\ [\neq]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \underline{x} = \overline{x} = \underline{y} = \overline{y} \\ [1,1] & \text{if} \quad \overline{x} < \underline{y} \vee \overline{y} < \underline{x} \\ [0,1] & \text{otherwise} \end{cases}$$

Boolean operators $\wedge, \vee, \neg$ are lifted to interval counterparts well-known from 3-valued logic:

$$[\underline{x},\overline{x}]\ [\wedge]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \underline{x} = \overline{x} = 0 \vee \underline{y} = \overline{y} = 0 \\ [1,1] & \text{if} \quad \overline{x} = \underline{x} = 1 \wedge \underline{y} = \overline{y} = 1 \\ [0,1] & \text{otherwise} \end{cases}$$

$$[\underline{x},\overline{x}]\ [\vee]\ [\underline{y},\overline{y}] = \begin{cases} [0,0] & \text{if} \quad \underline{x} = \overline{x} = 0 \wedge \underline{y} = \overline{y} = 0 \\ [1,1] & \text{if} \quad \underline{x} = \overline{x} = 1 \vee \underline{y} = \overline{y} = 1 \\ [0,1] & \text{otherwise} \end{cases}$$

$$[\neg][\underline{x},\overline{x}] \ = \begin{cases} [0,0] & \text{if} \quad \underline{x} = \overline{x} = 1 \\ [1,1] & \text{if} \quad \underline{x} = \overline{x} = 0 \\ [0,1] & \text{otherwise} \end{cases}$$

8.2 Abstract Interpretation Concepts

The objective of abstract interpretation is to associate a single abstract computation sequence

$$a = \alpha_0.\alpha_1.\alpha_2 \ldots$$

with a program, function or method. Each element of a is an abstract valuation function α mapping each variable symbol to its current lattice valuation (which is an interval valuation in the simplest case considered here). The basic principles for obtaining such an abstract interpretation computation are as follows:

Assignments. An assignment $x_0 = f(x_1, \ldots, x_n)$; performed in program state α_i maps to a new state α_{i+1} which differs from α_i in two arguments only:

- The program counter p (evaluated as a concrete natural number and not as an interval for the simplest form of abstract interpretation) is incremented by one,

$$\alpha_{i+1}(p) = \alpha_i(p) + 1$$

- The new interval valuation of x_0 is equal to the interval valuation of f with argument valuations taken from state α_i:

$$\alpha_{i+1}(x_0) = [f](\alpha_i(x_1), \ldots, \alpha_i(x_n))$$

This may be expressed equivalently as

$$\alpha_{i+1} = \alpha_i \oplus \{p \mapsto \alpha_i(p) + 1, x_0 \mapsto [f](\alpha_i(x_1), \ldots, \alpha_i(x_n))\}$$

or, using the semantic brackets notation and an arbitrary abstract pre-state α,

$$[\![x_0 = f(x_1, \ldots, x_n);]\!]_A(\alpha) = \alpha \oplus \{p \mapsto \alpha(p) + 1, x_0 \mapsto [f](\alpha(x_1), \ldots, \alpha(x_n))\}$$

Conditional statements. A conditional statement

```
if ( BooleanCondition ) {
     ifBlock
}
else {
     elseBlock
}
```

evaluates to

- the valuation of the if-block if the interval valuation of $[BooleanCondition]$ results in $[1, 1]$,
- the valuation of the else-block if the interval valuation of $[BooleanCondition]$ results in $[0, 0]$,
- the join of the if-block and else-block valuations otherwise.

More formally,

$$\begin{aligned}
&[\![\textbf{if } (b)\ S_1 \textbf{ else } S_2]\!]_A(\alpha) = \\
&\quad (\textbf{if } [\![b]\!]_A(\alpha) = [1,1] \textbf{ then } [\![S_1]\!]_A(\alpha) \\
&\quad \textbf{elseif } [\![b]\!]_A(\alpha) = [0,0] \textbf{ then } [\![S_2]\!]_A(\alpha) \\
&\quad \textbf{else } [\![S_1]\!]_A(\alpha) \sqcup [\![S_2]\!]_A(\alpha) \\
&\quad \textbf{endif}) \oplus \{p \mapsto p'\}
\end{aligned}$$

where p' is the program counter value of the next statement following the if statement. For abstract valuation functions α_0, α_1 we define their join by joining each of their function values, that is,

$$\alpha_0 \sqcup \alpha_1 : V \to L(D);\ x \mapsto \alpha_0(x) \sqcup \alpha_1(x)$$

Observe that the set of abstract state valuation functions α becomes a lattice by means of this join definition and by defining the meet in the analogous way as

$$\alpha_0 \sqcap \alpha_1 : V \to L(D);\ x \mapsto \alpha_0(x) \sqcap \alpha_1(x)$$

Loops. While loops of the form

```
while ( BooleanCondition ) {
     whileBlock
 }
```

are interpreted as (potentially infinite) if-else sequences

```
if ( BooleanCondition ) {
     whileBlock;
     if ( BooleanCondition ) {
        whileBlock;
        if ( BooleanCondition ) {
            whileBlock;
            if ( ....
            ....
        }
     }
}
```

The properties of complete lattices (for incomplete ones widening has to be applied) guarantee that repetitive application of the if-else rules to this expanded loop representation results in a fixpoint, where no interval valuations change any further. Therefore we can define the abstract interpretation of a while loop by building the supremum

$$[\![\textbf{while } (b)\ S;]\!]_A(\alpha) = \left(\bigsqcup_{i \geq 0} (\mathcal{F}^i(\alpha)) \right) \oplus \{p \mapsto p'\}$$

where $\mathcal{F}$ is defined by

$$\begin{aligned} \mathcal{F}(\alpha) = \ & \textbf{if } [\![b]\!]_A(\alpha) = [1,1] \textbf{ then } [\![S]\!]_A(\alpha) \\ & \textbf{elseif } [\![b]\!]_A(\alpha) = [0,0] \textbf{ then } \alpha \\ & \textbf{else } [\![S]\!]_A(\alpha) \sqcup \alpha \\ & \textbf{endif} \end{aligned}$$

Expression $\mathcal{F}^i(\alpha)$ denotes i-fold functional composition of $\mathcal{F}$ applied to α, that is,

$$\mathcal{F}^i(\alpha) = \underbrace{\mathcal{F} \circ \ldots \circ \mathcal{F}}_{i \text{ times}}(\alpha)$$

8.3 Abstract Interpretation Examples

Example 12. Consider the following C fragment consisting of a while loop which terminates after having received an input `b = 0` in the body of the loop. We assume that the input can only assume values 0 or 1.

```
1   int b = 1; int x = 0;
2   while (b) {
3     x = 1 - x;
4     b = input(); // b in [0,1]
5   }
```

We are interested in the possible valuations of `b` and `x` in situations where the loop terminates. We apply abstract interpretation rules for assignment and sequential composition and get

```
int b = 1; int x = 0;
// b in [1,1], x in [0,0]
while (b) {
  x = 1 - x;
  b = input();
}
// Due to fix point calculation below:
// b in [0,1], x in [0,1]          (*)
```

To prove the abstract post-state (*), we apply the while-rule given above with fix point function

$$\begin{aligned}\mathcal{F}(\alpha) = \;& \textbf{if } \alpha(b) = [1,1] \textbf{ then } [\![x = 1 - x; b = \text{input}()]\!]_A(\alpha)\\ & \textbf{elseif } \alpha(b) = [0,0] \textbf{ then } \alpha\\ & \textbf{else } [\![x = 1 - x; b = \text{input}()]\!]_A(\alpha) \sqcup \alpha\\ & \textbf{endif}\end{aligned}$$

Now we calculate

$$\begin{aligned}\mathcal{F}(\{b \mapsto [1,1], x \mapsto [0,0]\}) &= \{b \mapsto [0,1], x \mapsto [1,1]\}\\ \mathcal{F}^2(\{b \mapsto [1,1], x \mapsto [0,0]\}) &= \mathcal{F}(\{b \mapsto [0,1], x \mapsto [1,1]\})\\ &= \{b \mapsto [0,1], x \mapsto [0,0]\} \sqcup \{b \mapsto [0,1], x \mapsto [1,1]\}\\ &= \{b \mapsto [0,1], x \mapsto [0,1]\}\\ \mathcal{F}^3(\{b \mapsto [1,1], x \mapsto [0,0]\}) &= \mathcal{F}(\{b \mapsto [0,1], x \mapsto [0,1]\})\\ &= \{b \mapsto [0,1], x \mapsto [0,1]\} \sqcup \{b \mapsto [0,1], x \mapsto [0,1]\}\\ &= \{b \mapsto [0,1], x \mapsto [0,1]\}\end{aligned}$$

Therefore $\{b \mapsto [0,1], x \mapsto [0,1]\}$ is the supremum calculated according to the while-rule. □

Example 13. Consider the following C-function which inputs x, y, z and returns a computed value.

```
1   /**
2    *  @pre x in [0,100] and y in [0,100] and z in [-2000,-1001]
3    */
4   int f(int x, int y, int z) {
5      int w = 10;
6      if ( x > w && w > x + y )
7      {
8         w = w*x + y - 1000;
9      }
10     else
11     {
12       w = x*y;
13     }
14     return 1000 / ( z - w );
15  }
```

We wish to explore whether a divide-by-zero runtime error may occur, provided that the pre-condition of the function is met. Since the only devision in this function occurs in line 14, the verification goal can be expressed as usual as a CTL* formula which is indeed an ACTL formula (we use p to denote the "program counter" indicating the current line number of the execution):

$$\mathbf{AG}(\mathrm{p} = 13 \Rightarrow (z - w) \neq 0)$$

Performing the simplest form of abstract interpretation over integer intervals without using contractors gives us the following interpretation results which are marked as comments in the listing:

```
1   /**
2    *  @pre x in [0,100] and y in [0,100] and z in [-2000,-1001]
3    */
4   int f(int x, int y, int z) {
```

```
5       int w = 10;  // w in [10,10]
6       if ( x > w && w > x + y )
7       // ([0,100] > [10,10] && [10,10] > [0,100] + [0,100])
8       // = [0,1]   (top)
9       {
10         w = w*x + y - 1000; // w in [-1000,100]
11      }
12      else
13      {
14        w = x*y; // w in [0,10000]
15      }
16      // join of if-else branches: w in [-1000,10000] ;
17      // this implies (z-w) in [-12000,-1]
18      return 1000 / ( z - w );
19      // return in [-1000,0] (rules for integer division)
20  }
```

As a consequence, the function will not produce divide-by-zero runtime errors as long as the pre-condition is observed, because the verification goal $\mathbf{AG}(p = 13 \Rightarrow (z - w) \neq 0)$ is a direct consequence of the stricter assertion

$$\mathbf{AG}(p = 14 \Rightarrow (z - w) \in [-12000, -1])$$

obtained from the abstract interpretation. □

Exercise 19. For the code fragment given below, apply abstract interpretation rules introduced earlier in this section in order to compute the sequence $\alpha_0, \alpha_1, \ldots$ of abstract states. As pre-state, $\alpha_0 = \{p \mapsto 1\}$ can be assumed. The possible range for the input is defined as $[0, 10]$.

```
1 int x = input();
2 int y = x/2;
3 while ( x > 0 ) {
4   if ( y < 3 )
5     y = y + 1;
6   x = x - y;
7 }
```

Using the abstract interpretation's result, please answer the following questions (and do not forget to also provide a justification):

1. Does the while loop always terminate?

2. Is it ever possible to reach a state where $x < 0$?

□

In the remainder of this section we will justify, using the abstraction concepts introduced in Section 7, why abstract interpretation is a sound abstraction concept. Indeed, it will become apparent that abstract interpretation induces a Boolean simulation of the concrete program, and the interval valuations obtained in the abstract interpretation each lead to one Boolean abstraction variable expressing *"The concrete variable valuation at this program execution point lies within the range indicated by its interval valuation"*. The justification will be performed using the function from the example above, so it does not represent a comprehensive proof. The procedure we use, however, can be easily seen to apply to abstract interpretations of any program.

Initial condition and transition relation of the concrete system. As usual, we start by associating the C function with its predicates specifying intial state and transition relation. In addition to program variables x, y, z, w we use p to denote the "program counter" indicating the current line of the program execution (line numbering as indicated in the first listing of Example 13).

$$
\begin{aligned}
&I(p,x,y,z,w) \equiv_{\text{def}} \\
&\quad p = 5 \wedge x \in [0,100] \wedge y \in [0,100] \wedge z \in [-2000,-1001] \\
&R(p,x,y,z,w,p',x',y',z',w',\text{return}') \equiv_{\text{def}} \\
&\quad (p = 5 \wedge p' = 6 \wedge w' = 10 \wedge x' = x \wedge y' = y \wedge z' = z) \vee \\
&\quad (p = 6 \wedge x > w \wedge w > x + y \wedge p' = 8 \wedge x' = x \wedge y' = y \wedge z' = z \wedge w' = w) \vee \\
&\quad (p = 6 \wedge (x \leq w \vee w \leq x + y) \wedge p' = 11 \wedge x' = x \wedge y' = y \wedge z' = z \wedge w' = w) \vee \\
&\quad (p = 8 \wedge p' = 14 \wedge w' = w \cdot x + y - 1000 \wedge x' = x \wedge y' = y \wedge z' = z) \vee \\
&\quad (p = 11 \wedge p' = 14 \wedge w' = x \cdot y \wedge x' = x \wedge y' = y \wedge z' = z) \vee \\
&\quad (p = 14 \wedge \text{return}' = 1000/(z - w) \wedge p' = 14)
\end{aligned}
$$

Identification of abstraction variables. The next step of the justification introduces one Boolean abstraction variable for every interval valuation

obtained in the abstract interpretation for any expression of interest.

$$
\begin{aligned}
a_0 &= p && (8.4)\\
a_1 &= w \in [10, 10] && (8.5)\\
a_2 &= x \in [0, 100] && (8.6)\\
a_3 &= y \in [0, 100] && (8.7)\\
a_4 &= z \in [-2000, -1001] && (8.8)\\
a_5 &= w \in [-1000, 100] && (8.9)\\
a_6 &= w \in [0, 10000] && (8.10)\\
a_7 &= w \in [-1000, 10000] && (8.11)\\
a_8 &= (z - w) \in [-12000, -1] && (8.12)
\end{aligned}
$$

The intuition for selection $a_1, \ldots, a_7$ is obvious: one Boolean abstraction variable for each concrete variable and associated interval valuation encountered during abstract interpretation; $a_i = \texttt{true}$ indicates that the variable is in the range specified by the interval involved. Variable a_8 has been introduced because the interval valuation of $(z - w)$ can be used to prove that a divide-by-zero runtime error does not occur.

In the current example only a finite number of interval valuations exist. An abstraction constructed as the a_i above only works if this number is *always* finite. For terminating programs only containing bounded loops this is quite obvious, for non-terminating programs or programs containing unbounded while-loops an additional argument is required: the result of each loop execution can be recorded in an interval valuation per variable. For two consecutive loop executions, the join of each valuation results again in a single valuation per variable. For complete lattices this continued join operation will result in a fixpoint which is again an element of the lattice. Since intervals over integral numbers form a complete lattice, we can rest assured that application of the fixpoint technique will result in one valuation result per variable for each loop. Since program text is finite, the finiteness of interval valuations follows.

Predicate abstraction of initial condition and transition relation. Using the predicate abstraction techniques introduced in Section 7, the

intial condition and transition relation of the abstracted Kripke structure constructed via the abstraction variables $a_0 \ldots a_8$ look as follows.

$$
\begin{array}{l}
[\mathrm{I}](a_0, \ldots, a_8) \equiv_{\mathsf{def}} \\
\quad \exists \xi_0, \ldots, \xi_4 : (a_0 = \xi_0 \wedge a_1 = \xi_4 \in [10,10] \wedge a_2 = \xi_1 \in [0,100] \wedge \\
\quad a_3 = \xi_2 \in [0,100] \wedge a_4 = \xi_3 \in [-2000,-1001] \wedge a_5 = \xi_4 \in [-1000,100] \wedge \\
\quad a_6 = \xi_4 \in [0,10000] \wedge a_7 = \xi_4 \in [-1000,10000] \wedge a_8 = (\xi_3 - \xi_4) \in [-12000,-1]) \wedge \\
\quad (\xi_0 = 5 \wedge \xi_1 \in [0,100] \wedge \xi_2 \in [0,100] \wedge \xi_3 \in [-2000,-1001])
\end{array}
$$

Dropping binding information about $a_1, a_5, \ldots, a_8$ not needed in the initial state leads to the fact that

$$[\mathrm{I}](a_0, \ldots, a_8) \Rightarrow A(\mathrm{I}) \text{ with } A(\mathrm{I}) =_{\mathsf{def}} (a_0 = 5 \wedge a_2 \wedge a_3 \wedge a_4)$$

For the transition relation, predicate abstraction results in (we have already performed term replacement of a_0 for p or ξ_0, respectively)

$$
\begin{array}{l}
[\mathrm{R}](a_0, \ldots, a_8, a_0', \ldots, a_8') \equiv \exists \xi_1, \ldots, \xi_4, \xi_1', \ldots, \xi_4' : \\
\quad a_1 = \xi_4 \in [10,10] \wedge a_2 = \xi_1 \in [0,100] \wedge \\
\quad a_3 = \xi_2 \in [0,100] \wedge a_4 = \xi_3 \in [-2000,-1001] \wedge a_5 = \xi_4 \in [-1000,100] \wedge \\
\quad a_6 = \xi_4 \in [0,10000] \wedge a_7 = \xi_4 \in [-1000,10000] \wedge a_8 = (\xi_3 - \xi_4) \in [-12000,-1] \wedge \\
\quad a_1' = \xi_4' \in [10,10] \wedge a_2' = \xi_1' \in [0,100] \wedge \\
\quad a_3' = \xi_2' \in [0,100] \wedge a_4' = \xi_3' \in [-2000,-1001] \wedge a_5' = \xi_4' \in [-1000,100] \wedge \\
\quad a_6' = \xi_4' \in [0,10000] \wedge a_7' = \xi_4' \in [-1000,10000] \wedge a_8' = (\xi_3' - \xi_4') \in [-12000,-1]) \wedge \\
\quad ((a_0 = 5 \wedge a_0' = 6 \wedge \xi_4' = 10 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3) \vee \\
\quad (a_0 = 6 \wedge \xi_1 > \xi_4 \wedge \xi_4 > \xi_1 + \xi_2 \wedge a_0' = 8 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3 \wedge \xi_4' = \xi_4) \vee \\
\quad (a_0 = 6 \wedge (\xi_1 \leq \xi_4 \vee \xi_4 \leq \xi_1 + \xi_2) \wedge a_0' = 11 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3 \wedge \xi_4' = \xi_4) \vee \\
\quad (a_0 = 8 \wedge a_0' = 14 \wedge \xi_4' = \xi_4 \cdot \xi_1 + \xi_2 - 1000 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3) \vee \\
\quad (a_0 = 11 \wedge a_0' = 14 \wedge \xi_4' = \xi_1 \cdot \xi_2 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3) \vee \\
\quad (a_0 = 14 \wedge \mathrm{return}' = 1000/(\xi_3 - \xi_4) \wedge a_0' = 14))
\end{array}
$$

For the next step we use the abbreviations

$$
\begin{array}{l}
\vec{a} =_{\mathsf{def}} (a_1, \ldots, a_8) \\
\vec{a}' =_{\mathsf{def}} (a_1', \ldots, a_8') \\
\vec{\xi} =_{\mathsf{def}} (\xi_1, \ldots, \xi_4) \\
\vec{\xi}' =_{\mathsf{def}} (\xi_1', \ldots, \xi_4') \\
\mathrm{B}(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \equiv_{\mathsf{def}} \\
\quad a_1 = \xi_4 \in [10,10] \wedge a_2 = \xi_1 \in [0,100] \wedge \\
\quad a_3 = \xi_2 \in [0,100] \wedge a_4 = \xi_3 \in [-2000,-1001] \wedge a_5 = \xi_4 \in [-1000,100] \wedge \\
\quad a_6 = \xi_4 \in [0,10000] \wedge a_7 = \xi_4 \in [-1000,10000] \wedge a_8 = (\xi_3 - \xi_4) \in [-12000,-1] \wedge \\
\quad a_1' = \xi_4' \in [10,10] \wedge a_2' = \xi_1' \in [0,100] \wedge \\
\quad a_3' = \xi_2' \in [0,100] \wedge a_4' = \xi_3' \in [-2000,-1001] \wedge a_5' = \xi_4' \in [-1000,100] \wedge \\
\quad a_6' = \xi_4' \in [0,10000] \wedge a_7' = \xi_4' \in [-1000,10000] \wedge a_8' = (\xi_3' - \xi_4') \in [-12000,-1]
\end{array}
$$

Applying predicate approximation we get

$$[R](a_0,\ldots,a_8,a_0',\ldots,a_8') \Rightarrow A(R)(a_0,\ldots,a_8,a_0',\ldots,a_8')$$

with

$$\begin{aligned}
&A(R)(a_0,\ldots,a_8,a_0',\ldots,a_8') \equiv\\
&\quad((a_0 = 5 \wedge a_0' = 6 \wedge a_1' \wedge a_2' = a_2 \wedge a_3' = a_3 \wedge a_4' = a_4 \wedge a_5' \wedge a_6' \wedge a_7') \vee\\
&\quad(a_0 = 6 \wedge a_0' = 8 \wedge\\
&\quad(\exists \vec{\xi}, \vec{\xi}' : B(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \wedge \xi_1 > \xi_4 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3 \wedge \xi_4' = \xi_4) \wedge\\
&\quad(\exists \vec{\xi}, \vec{\xi}' : B(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \wedge \xi_4 > \xi_1 + \xi_2 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3 \wedge \xi_4' = \xi_4)) \vee\\
&\quad(\exists \vec{\xi}, \vec{\xi}' : B(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \wedge\\
&\qquad a_0 = 6 \wedge \xi_1 \leq \xi_4 \wedge a_0' = 11 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3 \wedge \xi_4' = \xi_4) \vee\\
&\quad(\exists \vec{\xi}, \vec{\xi}' : B(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \wedge\\
&\qquad a_0 = 6 \wedge \xi_4 \leq \xi_1 + \xi_2 \wedge a_0' = 11 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3 \wedge \xi_4' = \xi_4) \vee\\
&\quad(\exists \vec{\xi}, \vec{\xi}' : B(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \wedge\\
&\qquad a_0 = 8 \wedge a_0' = 14 \wedge \xi_4' = \xi_4 \cdot \xi_1 + \xi_2 - 1000 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3) \vee\\
&\quad(\exists \vec{\xi}, \vec{\xi}' : B(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \wedge\\
&\qquad a_0 = 11 \wedge a_0' = 14 \wedge \xi_4' = \xi_1 \cdot \xi_2 \wedge \xi_1' = \xi_1 \wedge \xi_2' = \xi_2 \wedge \xi_3' = \xi_3) \vee\\
&\quad(\exists \vec{\xi}, \vec{\xi}' : B(\vec{a}, \vec{\xi}, \vec{a}', \vec{\xi}') \wedge a_0 = 14 \wedge \mathrm{return}' = 1000/(\xi_3 - \xi_4) \wedge a_0' = 14))
\end{aligned}$$

Construction of abstracted and approximated Kripke structure. The Kripke structure resulting from the abstraction and approximation $(A(I), A(R))$ of the concrete C function's initial condition and transition relation is depicted in Fig. 8.1; it is derived from constructing all possible solutions of $(A(I), A(R))$. We have adopted a 3-valued valuation of atomic propositions for this Kripke structure, where each predicate a may be true (a), false $(\mathrm{not}\ a)$ or undecided $(a = \top)$. This allows us to omit branches and additional nodes in the Kripke graph if we are not interested in the current valuation of predicates.

Construction of the final linear Kripke structure. Abstract interpretation in its most simple form which is discussed in this section does not perform any branching: by taking join operations for the resulting valuations of if-, else- and while-blocks we achieve one linear abstracted computation. This process can be repeated on the level of the Kripke structure by introducing additional "undecided"-valuations or weaker predicates for some atomic propositions: observe that nodes $n3$ and $n4$ only differ in the

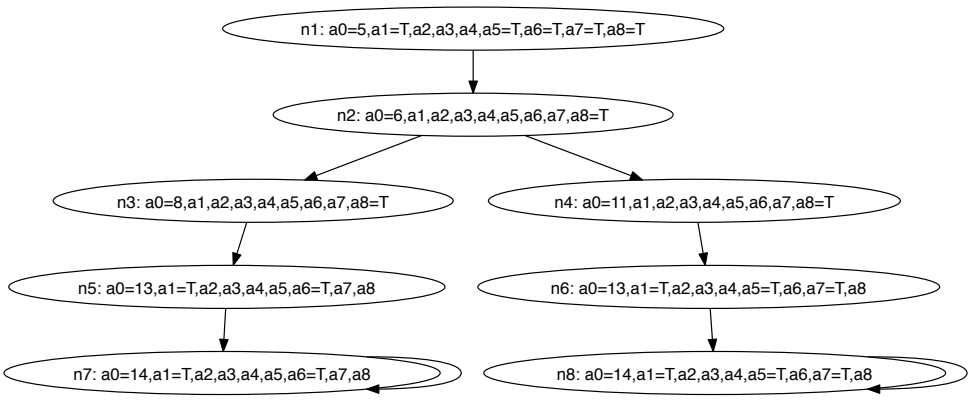


Figure 8.1: Kripke structure associated with abstracted and approximated initial condition and transition relation $(A(I), A(R))$.

program counter value a_0. We my collapse these two nodes into a single one by choosing a weaker predicate $a_0 = 8 \vee a_0 = 11$, which results in a Kripke structure as shown in Fig. 8.2.

Finally we observe that – since the truth value of a_8 alone decides about absence of devide-by-zero errors – the actual valuations of a_6, a_7 are not relevant as long as a_8 holds. This leads us to the final linear Kripke structure shown in Fig. 8.3.

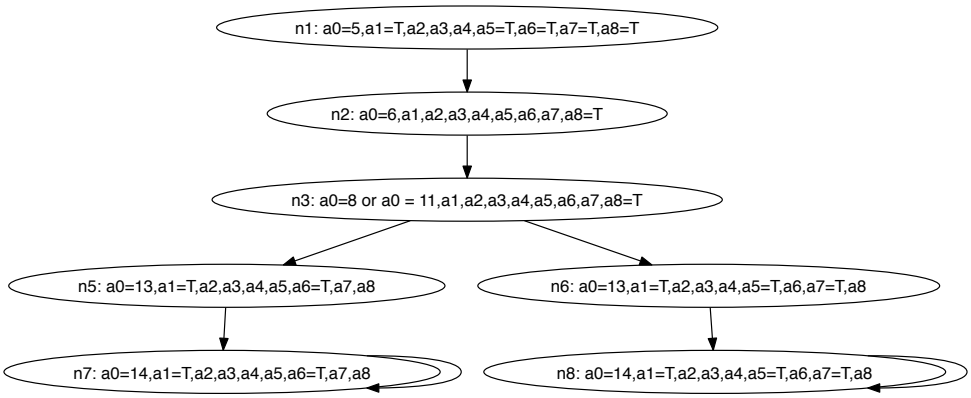


Figure 8.2: Kripke structure of Fig. 8.1 with collapsed nodes n3 and n4.

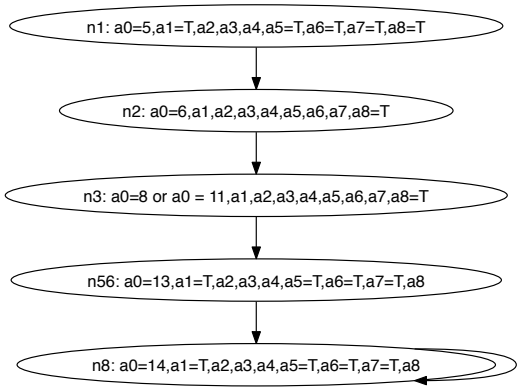


Figure 8.3: Final linear Kripke structure which is in one-one-correspondence with the abstract interpretation.

Chapter 9

Real-Time Formalisms Based on State-Transition Systems and Shared Variables

In this section we introduce a description formalism incorporating the notion of real time: Time is captured in a new model variable $\hat{t}$ typed over $\mathbb{R}_+ = [0, \infty)$. This allows us to describe time-continuous evolutions as needed in the description of physical models. Real-time formalisms supporting a notion of time in $\mathbb{R}_+$ are called *dense-time* formalisms, in contrast to *discrete-time* formalisms, where time is described by a counter recording the number of discrete clock ticks that occurred since the start of a computation. Variables are taken as usual from a set V which is now partitioned into five disjoint subsets $I, O, V_L, T, \{\hat{t}\}$ denoting input variables, outputs, local variables, timer variables and the current time, respectively.

9.1 The Passage of Time

This section has been created for Issue 3.3.

Recall from Chapter 1 that we require that timed computations fulfil the following realistic assumptions for timed computations

$$c = (t_0, s_0).(t_1, s_1)....$$

1. *Time is monotonic.* This means that consecutive time stamps in a computation never decrease.

2. *Finite variability.* The computation can perform only finitely many state changes in any finite time interval.

3. *Bounded variability.* There exists global upper bounds β, δ, such that the computation can perform at most β state changes in any time interval of a duration less or equal to δ. This is a stronger alternative to finite variability: it states that the state changes inside the computation cannot be accelerated.

4. *Absence of time convergence.* The sequence of t_i does not converge to a finite time value.

5. *Absence of time lock.* A time lock exists if the computation is time convergent and a computation suffix exists where no state changes occur anymore. The system is "locked" in this state for infinitely many steps, while the time converges to a finite value.

6. *Non zenoness condition.* Infinitely many state changes in the computation require infinite time. This means that systems cannot be infinitely fast. Zenoness complements time lock: a Zeno computation is time convergent and performs *infinitely many state changes* on a computation suffix.

Analogous well-formedness conditions apply to traces of events. These intuitive definitions are formalised as follows.

Definition 10 *Let* $c = (t_0, s_0).(t_1, s_1)....$ *be a computation.*

1. Time is monotonic *in* c *if and only if*

$$\forall i, j \in \mathbb{N}_0 \bullet i \leq j \Rightarrow t_i \leq t_j$$

2. *Computation* c *fulfils the* finite variability *property if and only if*

$$\forall t_{start}, t_{end} \in \mathbb{R}_{\geq 0} \bullet |\{i \in \mathbb{N}_0 \mid t_i, t_{i+1} \in [t_{start}, t_{end}] \wedge s_i \neq s_{i+1}\}| < \infty$$

3. *Computation* c *fulfils the* bounded variability *property if and only if*

$$\exists \beta \in \mathbb{N},\ \delta \in \mathbb{R}_{>0} \,.\, \forall t_{start} \leq t_{end} \in \mathbb{R}_{\geq 0} \,.\, t_{end} - t_{start} \leq \delta \Rightarrow |\{i \in \mathbb{N}_0 \mid t_i, t_{i+1} \in [t_{start}, t_{end}] \wedge s_i \neq s_{i+1}\}| \leq \beta$$

4. *The computation* c *is* time divergent *if and only if* $\lim_{i\to\infty} t_i = \infty$. *If* c *is not time divergent, it is called* time convergent.

5. *Computation* c *has a* time lock *if and only if*

$$\exists \tau \in \mathbb{R}_{\geq 0},\ \ell \in \mathbb{N} \,.\, \lim_{i\to\infty} t_i = \tau \wedge \forall j \geq \ell \,.\, s_j = s_\ell$$

6. *Computation* c *is a* Zeno sequence *if and only if*

$$\exists \tau \in \mathbb{R}_{\geq 0} \,.\, \lim_{i\to\infty} t_i = \tau \wedge |\{j \in \mathbb{N}_0 \mid s_j \neq s_{j+1}\}| = \infty$$

□

In the following example, we illustrate the concepts introduced in Definition 10 and the differences between them.

Example 14. Let $S = \{a_0, a_1\}$ be a state space with two states. The computation

$$c_1 = (0, a_0).(1 - \frac{1}{2}, a_1).(1 - \frac{1}{2^2}, a_0).(1 - \frac{1}{2^3}, a_1) \ldots (1 - \frac{1}{2^i}, a_{i\%2}) \ldots$$

is time convergent to $\tau = 1$, because $\lim_{i\to\infty} 2^{-i} = 0$. Moreover, it is a Zeno sequence, since infinitely many state changes $a_0 \longrightarrow a_1 \longrightarrow a_0 \ldots$ are performed in time interval $[0, 1]$.

In contrast to this, computation

$$c_2 = (0, a_0).(1 - \frac{1}{2}, a_1).(1 - \frac{1}{2^2}, a_0).(1 - \frac{1}{2^3}, a_1) \ldots$$
$$(1 - \frac{1}{2^{1000}}, a_0).(1 - \frac{1}{2^{1001}}, a_0).(1 - \frac{1}{2^{1002}}, a_0) \ldots$$

is time convergent and a time lock, since beyond element 1000, the state a_0 will never be left again.

Obviously, c_1 violates both the finite variability and the bounded variability property, since infinitely many state changes are performed in time interval $[0,1]$. Computation c_2, is time convergent, but is fulfils the bounded variability property, and, therefore, also the finite variability property.

The following computation is an example of a computation that fulfils the finite variability property but not the bounded variability, and it is time-divergent.

$$c_3 = (0,a_0).(1,a_1).(\sum_{j=1}^{2}\frac{1}{j},a_0).(\sum_{j=1}^{3}\frac{1}{j},a_1)\ldots(\sum_{j=1}^{i}\frac{1}{j},a_{i\%2})\ldots$$

The time divergence follows from the fact that the time stamps $t_1,t_2,\ldots$ in c_3 represent the *harmonic series*

$$t_i = 1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{i},\quad i=1,2,3,\longrightarrow\infty$$

which is known to diverge. Moreover, since the sum diverges, there can only be finitely many time stamps fitting into any given time interval $[t_{start},t_{end}]$ with $t_{start}\leq t_{end}\in\mathbb{R}_{\geq 0}$.

This computation, however, does not fulfil the bounded variability property: given any bound $\beta\in\mathbb{N}$ and any time interval width $\delta\in\mathbb{R}_{\geq 0}$, we find a computation segment covering a time span less or equal to δ, but containing *more* than β time stamps. To understand this, recall that the harmonic function

$$H_n=\sum_{j=0}^{n}\frac{1}{j}$$

grows only logarithmically. In the time interval $[H_m,H_{m+\beta}]$, computation c_3 has $(\beta+1)$ computation elements with time stamps $H_m,H_{m+1},\ldots,H_{m+\beta}$. The width of the interval is $H_{m+\beta}-H_m$ which converges (slowly) towards 0 with $m\longrightarrow\infty$.

As an example, assume that the number of state changes occurring in intervals with width less or equal to $\delta=4$ were bounded by $\beta=10000$. This is easily disproved by analysing the function graph of $f(m)=H_{m+10000}-H_m$, as depicted in Fig. 9.1. For $m=200$, $f(m)$ is already smaller than

$\delta = 4$ (Mathematica will tell you, it's approximately 3.929376735), but the interval $[H_{200}, H_{10200}]$ contains $10001 > \beta$ computation elements. For a smaller δ, we just increase m: for $m = 100000$, for example, the interval width $H_{m+10000} - H_m$ is only 0.09530972526. □

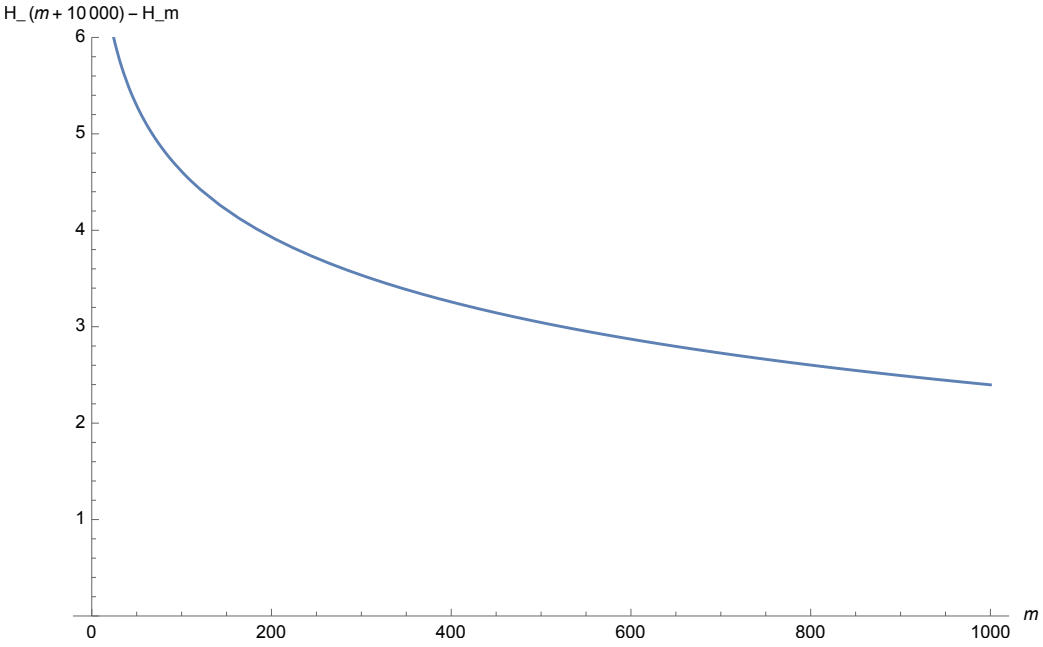


Figure 9.1: Difference $H_{m+10000} - H_m$ for $m \in (0, 1000]$.

9.2 Abstract Syntax of Timed State Machines

Timed State Machines m consist of *locations* $\ell \in \mathrm{Loc}(m)$ (also called *control states*) and *transitions*

$$\tau = (\ell, p, g, \alpha, \ell') \in \Sigma(m) \subseteq \mathrm{Loc}(m) \times P \times G \times A \times \mathrm{Loc}(m)$$

connecting source and target locations ℓ and ℓ', respectively. Value $p \in P = \mathbb{N}_0$ denotes the priority of the transition (0 is the best priority) and is used to enforce determinism for state machines. Transition component $g \in$

$\mathrm{Bexpr}(V)$ denotes the guard condition of τ which is a Boolean expression over symbols from V. For timer symbols $t \in T$ occurring in g we only allow Boolean conditions $\mathrm{elapsed}(t, c)$ with constants c. Intuitively speaking, $\mathrm{elapsed}(t, c)$ evaluates to `true` if at least c time units have passed since t's most recent reset.

Transition component $\alpha \in A = \mathbb{P}(V \times \mathrm{Expr}(V))$ denotes a set of value assignments to variables in V, according to expressions from $\mathrm{Expr}(V)$. For a pair $a = (v, e) \in A$, $var(a) =_{\mathrm{def}} v$ and $expr(a) =_{\mathrm{def}} e$ denote the projections on variable and expression, respectively. For timer symbols $t \in T$ only resets (t, reset) are allowed. A transition without accompanying assignments is associated with an empty set $\alpha = \varnothing$. Function

$$\omega : \mathrm{Loc}(m) \to \mathbb{P}(\Sigma(m));\ \ell \mapsto \{(\bar{\ell}, p, g, \alpha, \ell') \in \Sigma(m) \mid \bar{\ell} = \ell\}$$

maps locations to their outgoing transitions. Each state machine m has a specific *start location* $\mathrm{start}(m)$. Exactly one transition must leave $\mathrm{start}(m)$, and the guard of this transition has to be `true`.

The *parallel composition* of timed state machines $m_1, \ldots, m_n$ operating over the same set V of variables is denoted by

$$\|_{i=1,\ldots,n}\ m_i$$

If more than one machine write to the same variables from $V_L \cup O$ then these are called *shared variables*. Timer variables must never be shared, and inputs must never be written to.

Example 15. Fig. 9.2 shows an example of a simple switching mechanism involving a timer t: The start location is marked by the black bullet. Initially, the device controlled by this mechanism is switched off by setting the control output out to 0. If the switch sw is set to 1 then the device is switched on by means of output $\mathrm{out} = 1$. A timer is set, so that the device is automatically switched off after 100 time units. In that case, the input switch sw has to be reset first, before the device can be switched on again. Otherwise, if the switch sw is reset to 0 before the timer elapses, the device is switched off at once and switched on again as soon as $\mathrm{sw} = 1$. □

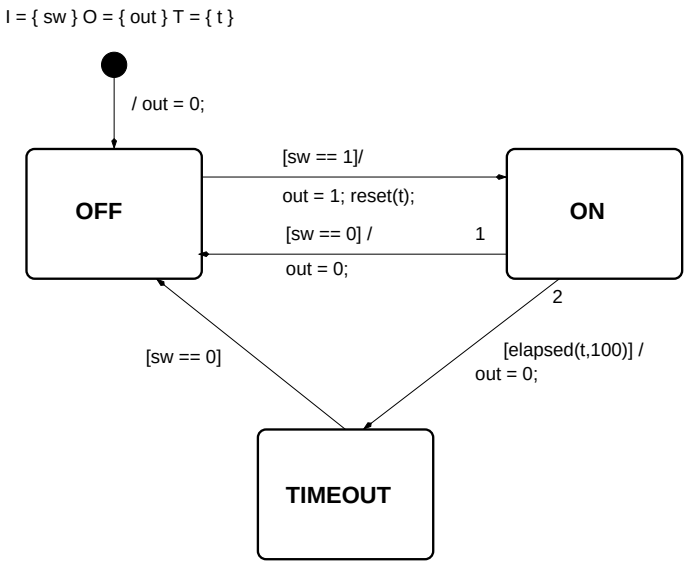


Figure 9.2: Timed state machine s for switch with timeout.

9.3 Semantics of Timed State Machines

The semantics of timed state machines is based on *timed state transition systems* $\mathrm{TSTS} = (S, S_0, R)$: The state space S consists of valuation functions $s : \mathrm{Loc}(m) \cup V \to D$ satisfying $s(\hat{t}), s(t) \in \mathbb{R}_+$ for valuation of global time $\hat{t}$ and timer variables t. As a consequence, S has uncountable cardinality. For locations ℓ, $s(\ell) \in \mathbb{B}$, $s(\ell) = \mathtt{true}$ signifying that the state machine is currently in this location. Initial states reside in the start location and have current time $\hat{t} = 0$, but may be associated with arbitrary input values. Also, local variables, outputs and timer have arbitrary values which are typically reset during the first transition from the start location to its target.

Current time $\hat{t}$ changes over physical time z like an ideal clock: if the model execution starts at physical point in time z_0, then the current time always fulfils

$$\hat{t} = z - z_0$$

or, equivalently,

$$\frac{d\hat{t}}{dz} = 1$$

which will occur in the invariants introduced below, which are part of the transition relation.

Example 16. For the example from Fig. 9.2, this results in the following initial state:

$$
\begin{aligned}
&S_0 = \{s \in S \mid s \models \mathcal{I}\} \\
&\mathcal{I} \equiv \mathrm{start}(m) \wedge \hat{t} = 0 \wedge \mathrm{INV} \\
&\mathrm{INV} \equiv (\mathrm{start}(m) \vee \mathrm{OFF} \vee \mathrm{ON} \vee \mathrm{TIMEOUT}) \wedge \\
&\qquad \neg(\mathrm{start}(m) \wedge \mathrm{OFF}) \wedge \neg(\mathrm{start}(m) \wedge \mathrm{ON}) \wedge \neg(\mathrm{start}(m) \wedge \mathrm{TIMEOUT})) \wedge \\
&\qquad \neg(\mathrm{OFF} \wedge \mathrm{ON}) \wedge \neg(\mathrm{OFF} \wedge \mathrm{TIMEOUT}) \wedge \neg(\mathrm{ON} \wedge \mathrm{TIMEOUT}) \wedge \tfrac{d\hat{t}}{dz} = 1
\end{aligned}
$$

□

Transitions are classified as

- *discrete transitions* and
- *delay transitions*,

which is the canonical approach for dense-time formalisms: Discrete transitions take place in zero time; they may change outputs, local variables, timers and locations, while inputs and current time $\hat{t}$ remain stable. Delay transitions can only happen when no discrete transition is enabled. In that case, the current time is advanced by a positive value, but only as far as the point in time where the next timers elapse, because this might enable another discrete transition. Obviously, TSTS contains uncountably many transitions, since time may proceed in infinitesimally small units, each unit inducing a delay transition.

More formally, the *effect* of an action $\alpha = \{a_1, \ldots, a_k\}$ is defined as

$$
\begin{aligned}
\epsilon(\alpha) \quad \equiv_{\mathsf{def}} \quad & \Big(\bigwedge_{a \in \alpha \wedge \mathrm{var}(a) \in V - T} \mathrm{var}(a)' = \mathrm{expr}(a) \Big) \wedge \\
& \Big(\bigwedge_{a \in \alpha \wedge \mathrm{var}(a) \in T} \mathrm{var}(a)' = \hat{t} \Big)
\end{aligned}
$$

A state machine transition $\tau = (\ell_0, p, g, \alpha, \ell_1)$ may be *triggered* (or, synonymously, it may *fire*) if

$$
\mathrm{trigger}_m(\ell_0, p, g, \alpha, \ell_1) \equiv_{\mathsf{def}} \ell_0 \wedge g \wedge (\forall (\ell_0, \overline{p}, \overline{g}, \overline{\alpha}, \overline{\ell_1}) \in \omega_m(\ell_0) : \overline{p} \geq p \vee \neg \overline{g})
$$

holds. This means that for τ to fire, s must reside in location ℓ_0, τ's guard condition has to evaluate to `true` and no higher-priority transition emanating from ℓ_0 can be triggered. The *effect* of a state machine transition $\tau = (\ell_0, p, g, \alpha, \ell_1)$ that can be triggered is specified as

$$\epsilon(\ell_0, p, g, \alpha, \ell_1) \equiv_{\text{def}} \epsilon(\alpha) \wedge \ell_1'$$

The *write set* of an action α is defined by the set of left-hand side variables and timers that are changed by this action:

$$W(\alpha) =_{\text{def}} \{var(a) \mid a \in \alpha\}$$

The *write set* of a transition $\tau = (\ell_0, p, g, \alpha, \ell_1)$ is defined by the write set of its action:

$$W(\tau) =_{\text{def}} W(\alpha)$$

The complete transition relation of a parallel system $\|_{i=1,\ldots,n}\ s_i$ is defined by

$$\Phi \equiv_{\text{def}} ((\text{trigger}_D \wedge \Phi_D) \vee (\neg\text{trigger}_D \wedge \Phi_T)) \wedge \text{Inv}'$$

where predicate trigger_D is defined as follows:

$$\text{trigger}_D \equiv_{\text{def}} (\bigvee_{i=1}^{n} \bigvee_{\tau \in \Sigma(s_i)} \text{trigger}_{m_i}(\tau))$$

The *invariant* Inv states that

- every state machine may be in at most one location at time,
- every variable only takes values in its specified domain, and
- the current time behaves like an ideal clock.

$$\begin{aligned}&\text{Inv} \equiv_{\text{def}}\\ &\quad(\forall i \in \{1,\ldots,n\}, \ell_0, \ell_1 \in \text{Loc}(m_i) : \ell_0 \wedge \ell_1 \Rightarrow \ell_0 = \ell_1) \wedge\\ &\quad(\forall v \in V : v \in D_v) \wedge\\ &\quad\tfrac{d\hat{t}}{dz} = 1\end{aligned}$$

Components Φ_D and Φ_T denote the discrete and delay transition aspects of the complete transition relation Φ, respectively: if trigger_D evaluates to

`true` we get the effect of a discrete transition, and if it evaluates to `false`, a delay transition is performed. For discrete transitions we define

$$\begin{aligned}\Phi_D \equiv_{\text{def}} \; & (\hat{t}' = \hat{t}) \wedge (\forall v \in I : v' = v) \wedge \\ & (\forall i \in \{1, \ldots, n\}, \tau \in \Sigma(m_i) : \text{trigger}(\tau) \Rightarrow \epsilon(\tau)) \wedge \\ & (\forall v \in V - I : \text{written}(v) \vee v' = v)\end{aligned}$$

The current time and the inputs remain unchanged during a discrete transition; all transitions of state machines s_i that may fire are performed simultaneously, and variables that are not written to by any transition remain unchanged. Formally, $\text{written}(v)$ is defined as

$$\text{written}(v) \equiv_{\text{def}} (\exists i \in \{1, \ldots, n\}, \tau \in \Sigma(m_i) : \text{trigger}(\tau) \wedge v \in W(\tau))$$

The delay component Φ_T formalises the following rules:

- The current time has to be advanced.
- All locations, local variables and outputs remain unchanged.
- The current time may be advanced at most up to the point in time where the next timer will elapse.
- Timers which are already elapsed do *not* restrict the amount of time $\hat{t}$ is advanced.

$$\begin{aligned}\Phi_T \equiv_{\text{def}} \; & (\hat{t}' > \hat{t}) \wedge \\ & (\forall i \in \{1, \ldots, n\}, \ell \in \text{Loc}(m_i) : \ell' \Leftrightarrow \ell) \wedge \\ & (\forall v \in V - I : v' = v) \wedge \\ & (\forall i \in \{1, \ldots, n\}, (\ell_0, p, g, \alpha, \ell_1) \in \Sigma(m_i) : \\ & \quad (\exists \overline{g} \in \text{Bexpr}, t \in T, c \in \mathbb{N} : g \equiv \overline{g} \wedge \text{elapsed}(t, c)) \Rightarrow \\ & \qquad (\hat{t}' \leq c + t \vee \hat{t} \geq c + t))\end{aligned}$$

Example 17. For the example from Fig. 9.2, this results in the following transition relation:

$$
\begin{aligned}
R \equiv\ & \mathrm{INV} \wedge \mathrm{INV}' \wedge ((\mathrm{start}(m) \wedge sw' = sw \wedge t' = t \wedge \hat{t}' = \hat{t} \wedge out' = 0 \wedge \mathrm{OFF}') \vee \\
& (\mathrm{OFF} \wedge sw = 0 \wedge \hat{t}' > \hat{t} \wedge out' = out \wedge t' = t \wedge \mathrm{OFF}') \vee \\
& (\mathrm{OFF} \wedge sw = 1 \wedge sw' = sw \wedge \hat{t}' = \hat{t} \wedge out' = 1 \wedge t' = \hat{t} \wedge \mathrm{ON}') \vee \\
& (\mathrm{ON} \wedge sw = 1 \wedge \hat{t}' > \hat{t} \wedge (\hat{t} - t) < 100 \wedge (\hat{t}' - t) \leq 100 \wedge out' = out \wedge t' = t \wedge \mathrm{ON}') \vee \\
& (\mathrm{ON} \wedge sw = 1 \wedge (\hat{t} - t) \geq 100 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge out' = 0 \wedge t' = t \wedge \mathrm{TIMEOUT}') \vee \\
& (\mathrm{ON} \wedge sw = 0 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge out' = 0 \wedge t' = t \wedge \mathrm{OFF}') \vee \\
& (\mathrm{TIMEOUT} \wedge sw = 1 \wedge \hat{t}' > \hat{t} \wedge t' = t \wedge out' = out \wedge \mathrm{TIMEOUT}') \vee \\
& (\mathrm{TIMEOUT} \wedge sw = 0 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge out' = out \wedge t' = t \wedge \mathrm{OFF}'))
\end{aligned}
$$

□

Exercise 20.

1. Give an intuitive natural-language explanation why R in Example 17 represents the transition relation of the example from Fig. 9.2 in a correct way.

2. Trace back every component in predicate R to the general predicate constructions Φ, Φ_D, Φ_T introduced above.

□

Example 18. We apply the concept of predicate abstraction introduced in Section 7.6 in order to prove that the sample model displayed in Fig. 9.2 satisfies the property

$$\Phi_1 \equiv \mathbf{AG}(\neg \mathrm{ON} \vee (\hat{t} - t \leq 100))$$

The KS from Fig. 9.2 is K with $V = \{sw, \mathrm{out}, t, \hat{t}, \mathrm{ON}, \mathrm{OFF}, \mathrm{TIMEOUT}\}$, initial condition $\mathcal{I}$ as defined in Example 16, and transition relation as specified in Example 17.

Applying the recipe from Example 10, we proceed as follows.

1. A useful heuristic is to create an abstraction from the atomic propositions occurring in the formula to be proven. Φ_1 has atoms ON and $(\hat{t} - t \leq 100)$, so we define a variable set $V_a = \{\mathrm{ON}, a\}$ for the simulation space, with abstracting expression

   $$a \equiv (\hat{t} - t \leq 100)$$

   For ON we do not need an abstracting expression, because it will be used with the same meaning as in the original Kripke Structure.

2. The state space of the simulation is $S_a = V_a \to \mathbb{B}$, the set of Boolean functions over V_a.

3. The simulation relation is defined by

   $$H = \{(s, s_a) \in S \times S_a \mid s_a(\mathrm{ON}) = s(\mathrm{ON}), s_a(a) = s(\hat{t} - t \leq 100)\}$$

4. The initial condition $\mathcal{I}_a$ is constructed below, using Theorem 10 and the associated Lemma 5.

5. The transition relation $\mathcal{R}_a$ is constructed below, using Theorem 10.

6. The atomic propositions of the simulation space are

$$AP_a = \{ON, a\}$$

7. The labelling function of the simulation space is specified by

$$\forall s_a \in S_a : L_a(s_a) = \{p \in AP_a \mid s_a(p)\}$$

Using Theorem 10, we calculate the initial condition as follows (we write ss short for start(s) and TO short for TIMEOUT). This calculation is based on the original initial condition of the concrete system from Fig. 9.2, as elaborated in Example 16.

$$\begin{aligned} \mathcal{I}_a &\equiv \exists ss, sw, out, OFF, TO, t, \hat{t} : \mathcal{I} \wedge a = (\hat{t} - t \leq 100) \\ &\equiv \exists ss, sw, out, OFF, TO, t, \hat{t} : ss \wedge \hat{t} = 0 \wedge INV \wedge a = (\hat{t} - t \leq 100) \end{aligned}$$

Since $\hat{t}, t$ have domain $\mathbb{R}_+$, $(\hat{t} - t \leq 100)$ must hold in any initial state of K; this implies

$$\mathcal{I}_a \equiv \neg ON \wedge a$$

Now we calculate $\mathcal{R}_a$, using again Theorem 16 and the formula for $\mathcal{R}$

developed in Example 17.

$$
\begin{aligned}
\mathcal{R}_a \quad &\equiv \quad \exists ss, sw, \mathrm{out}, \mathrm{OFF}, \mathrm{TO}, t, \hat{t}, ss', sw', \mathrm{out}', \mathrm{OFF}', \mathrm{TO}', t', \hat{t}' : \\
&\qquad \big(\mathcal{R} \wedge a = (\hat{t} - t \leq 100) \wedge a' = (\hat{t}' - t' \leq 100)\big) \\
&\equiv \quad \exists ss, sw, \mathrm{out}, \mathrm{OFF}, \mathrm{TO}, t, \hat{t}, ss', sw', \mathrm{out}', \mathrm{OFF}', \mathrm{TO}', t', \hat{t}' : \\
&\qquad \big(\mathrm{INV} \wedge \mathrm{INV}' \wedge a = (\hat{t} - t \leq 100) \wedge a' = (\hat{t}' - t' \leq 100) \wedge \\
&\qquad \big((ss \wedge sw' = sw \wedge t' = t \wedge \hat{t}' = \hat{t} \wedge \mathrm{out}' = 0 \wedge \mathrm{OFF}') \vee \\
&\qquad (\mathrm{OFF} \wedge sw = 0 \wedge \hat{t}' > \hat{t} \wedge \mathrm{out}' = \mathrm{out} \wedge t' = t \wedge \mathrm{OFF}') \vee \\
&\qquad (\mathrm{OFF} \wedge sw = 1 \wedge sw' = sw \wedge \hat{t}' = \hat{t} \wedge \mathrm{out}' = 1 \wedge t' = \hat{t} \wedge \mathrm{ON}') \vee \\
&\qquad (\mathrm{ON} \wedge sw = 1 \wedge \hat{t}' > \hat{t} \wedge (\hat{t} - t) < 100 \wedge (\hat{t}' - t) \leq 100 \wedge \mathrm{out}' = \mathrm{out} \wedge t' = t \wedge \mathrm{ON}') \vee \\
&\qquad (\mathrm{ON} \wedge sw = 1 \wedge (\hat{t} - t) \geq 100 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge \mathrm{out}' = 0 \wedge t' = t \wedge \mathrm{TO}') \vee \\
&\qquad (\mathrm{ON} \wedge sw = 0 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge \mathrm{out}' = 0 \wedge t' = t \wedge \mathrm{OFF}') \vee \\
&\qquad (\mathrm{TO} \wedge sw = 1 \wedge \hat{t}' > \hat{t} \wedge t' = t \wedge \mathrm{out}' = \mathrm{out} \wedge \mathrm{TO}') \vee \\
&\qquad (\mathrm{TO} \wedge sw = 0 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge \mathrm{out}' = \mathrm{out} \wedge t' = t \wedge \mathrm{OFF}'))\big) \\
&\equiv \quad (\neg \mathrm{ON} \wedge a \wedge \neg \mathrm{ON}' \wedge a') \vee \\
&\qquad (\neg \mathrm{ON} \wedge \neg \mathrm{ON}' \wedge \neg a') \vee \\
&\qquad (\neg \mathrm{ON} \wedge \mathrm{ON}' \wedge a') \vee \\
&\qquad (\mathrm{ON} \wedge \mathrm{ON}' \wedge a') \vee \\
&\qquad (\mathrm{ON} \wedge \neg \mathrm{ON}')
\end{aligned}
$$

In this calculation we have used several simplification rules for propositional formulas; most importantly

1. If $p(\vec{x})$ is a proposition with free variables in $\vec{x} = (x_1, \ldots, x_n)$ (a "vector" of one or more free variables), and q is a proposition with free variables outside $\{x_1, \ldots, x_n\}$, then

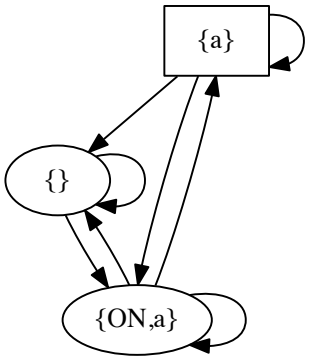
$$\exists \vec{x} : (p(\vec{x}) \wedge q) \equiv (\exists \vec{x} : p(\vec{x})) \wedge q$$

2. $((a \wedge b) \vee (a \wedge \neg b)) \equiv a$

The transition relation of the simulation KS is shown in Fig. 9.3. It is obvious that Φ_1. □

Example 19. In the same context as in Example 18, we will now show that the sample model displayed in Fig. 9.2 satisfies the property

$$\mathbf{A}(\mathbf{G}(\mathrm{sw}) \Rightarrow \mathbf{F}(\mathrm{ON} \wedge (\hat{t} - t) > 50))$$



Initial states are drawn as boxes.

Figure 9.3: Transition graph of the simulation KS specified in Example 18.

We handle sw as a Boolean variable so that $(sw = 1) \equiv sw$. Proceeding in analogy to the previous example, we construct the simulation K_a as shown below. As in the previous example, we use short-hand ss for the predicate $start(s)$. Here it will turn out that in order to prove the assertion, we need ss as additional variable in the simulation space, because otherwise it is impossible to prove the assertion.

1. $V_a = \{ss, sw, ON, b\}$, where ss, sw, ON are used in K_a just as in the concrete model, and
$$b \equiv ((\hat{t} - t) > 50)$$
is used to abstract the Boolean variable b.

2. The state space of the simulation is $S_a = V_a \to \mathbb{B}$, the set of Boolean functions over V_a.

3. The simulation relation is defined by
$$\begin{aligned} H \;=\; \{(s, s_a) \in S \times S_a \mid\; & s_a(sw) = s(sw), \\ & s_a(ss) = s(ss), s_a(ON) = s(ON), s_a(b) = s(\hat{t} - t > 50)\} \end{aligned}$$

4. The initial condition $\mathcal{I}_a$ is constructed below, using Theorem 10.
5. The transition relation $\mathcal{R}_a$ is constructed below, using Theorem 10.
6. The atomic propositions of the simulation space are

$$AP_a = \{sw, ON, b\}$$

7. The labelling function of the simulation space is specified by

$$\forall s_a \in S_a : L_a(s_a) = \{p \in AP_a \mid s_a(p)\}$$

The initial condition is calculated as follows.

$$\begin{aligned}
\mathcal{I}_a &\equiv \exists ss, out, OFF, TO, t, \hat{t} : \mathcal{I} \wedge b = (\hat{t} - t > 50) \\
&\equiv \exists out, OFF, TO, t, \hat{t} : ss \wedge \hat{t} = 0 \wedge INV \wedge b = (\hat{t} - t > 50) \\
&\equiv ss \wedge \neg ON \wedge \neg b
\end{aligned}$$

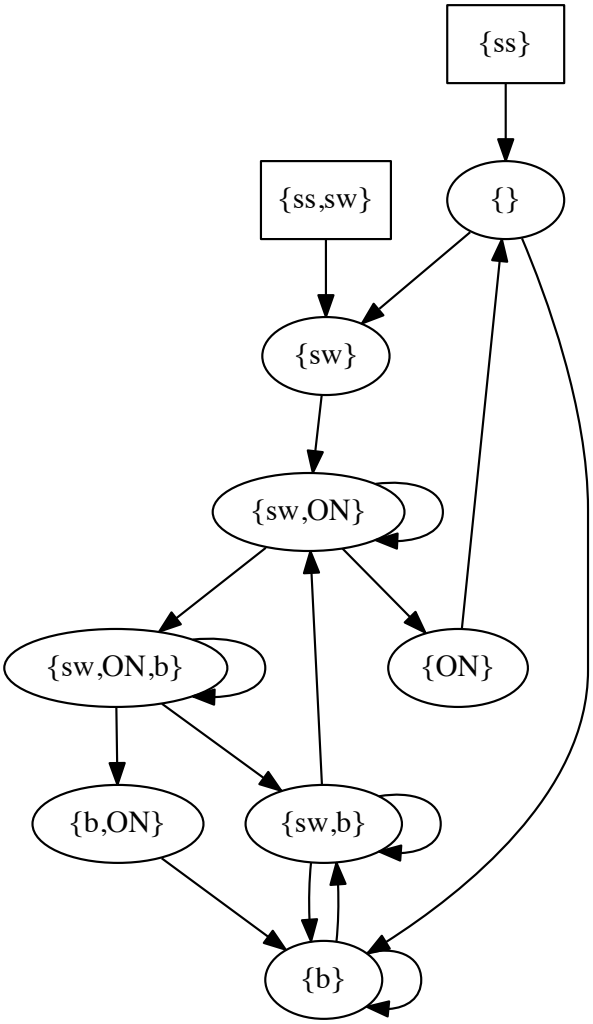
The transition relation is calculated as follows.

$$
\begin{aligned}
\mathcal{R}_a \quad \equiv \quad & \exists \mathrm{out}, \mathrm{OFF}, \mathrm{TO}, t, \hat{t}, \mathrm{out}', \mathrm{OFF}', \mathrm{TO}', t', \hat{t}' : \\
& \big(\mathcal{R} \wedge b = (\hat{t} - t > 50) \wedge b' = (\hat{t}' - t' > 50)\big) \\
\equiv \quad & \exists \mathrm{out}, \mathrm{OFF}, \mathrm{TO}, t, \hat{t}, \mathrm{out}', \mathrm{OFF}', \mathrm{TO}', t', \hat{t}' : \\
& \big(\mathrm{INV} \wedge \mathrm{INV}' \wedge b = (\hat{t} - t > 50) \wedge b' = (\hat{t}' - t' > 50) \wedge \\
& \big((ss \wedge sw' = sw \wedge t' = t \wedge \hat{t}' = \hat{t} \wedge \mathrm{out}' = 0 \wedge \mathrm{OFF}') \vee \\
& (\mathrm{OFF} \wedge \neg sw \wedge \hat{t}' > \hat{t} \wedge \mathrm{out}' = \mathrm{out} \wedge t' = t \wedge \mathrm{OFF}') \vee \\
& (\mathrm{OFF} \wedge sw \wedge sw' = sw \wedge \hat{t}' = \hat{t} \wedge \mathrm{out}' = 1 \wedge t' = \hat{t} \wedge \mathrm{ON}') \vee \\
& (\mathrm{ON} \wedge sw \wedge \hat{t}' > \hat{t} \wedge (\hat{t} - t) < 100 \wedge (\hat{t}' - t) \leq 100 \wedge \mathrm{out}' = \mathrm{out} \wedge t' = t \wedge \mathrm{ON}') \vee \\
& (\mathrm{ON} \wedge sw \wedge (\hat{t} - t) \geq 100 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge \mathrm{out}' = 0 \wedge t' = t \wedge \mathrm{TO}') \vee \\
& (\mathrm{ON} \wedge \neg sw \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge \mathrm{out}' = 0 \wedge t' = t \wedge \mathrm{OFF}') \vee \\
& (\mathrm{TO} \wedge sw \wedge \hat{t}' > \hat{t} \wedge t' = t \wedge \mathrm{out}' = \mathrm{out} \wedge \mathrm{TO}') \vee \\
& (\mathrm{TO} \wedge \neg sw \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge \mathrm{out}' = \mathrm{out} \wedge t' = t \wedge \mathrm{OFF}')\big)\big) \\
\equiv \quad & (ss \wedge \neg \mathrm{ON} \wedge sw' = sw \wedge b' = b \wedge \neg ss' \wedge \neg \mathrm{ON}') \vee \\
& (\neg ss \wedge \neg \mathrm{ON} \wedge \neg sw \wedge (\neg b \vee (b \wedge b')) \wedge \neg ss' \wedge \neg \mathrm{ON}') \vee \\
& (\neg ss \wedge \neg \mathrm{ON} \wedge sw \wedge sw' \wedge \neg b' \wedge \neg ss' \wedge \mathrm{ON}) \vee \\
& (\neg ss \wedge \mathrm{ON} \wedge sw \wedge (\neg b \vee (b \wedge b')) \wedge \neg ss' \wedge \mathrm{ON}') \vee \\
& (\neg ss \wedge \mathrm{ON} \wedge \neg sw \wedge \neg sw' \wedge b' = b \wedge \neg ss' \wedge \neg \mathrm{ON}') \vee \\
& (\neg ss \wedge \mathrm{ON} \wedge sw \wedge b \wedge b' \wedge \neg ss' \wedge \neg \mathrm{ON}') \vee \\
& (\neg ss \wedge \neg \mathrm{ON} \wedge sw \wedge b \wedge b' \wedge \neg ss' \wedge \neg \mathrm{ON}') \vee \\
& (\neg ss \wedge \neg \mathrm{ON} \wedge \neg sw \wedge \neg sw' \wedge b' = b \wedge \neg ss' \wedge \neg \mathrm{ON}')
\end{aligned}
$$

In Figure 9.4, the transition graph of the simulation KS is shown. To prove the assertion, it is necessary to refer to the invariant $\frac{d\hat{t}}{dz} = 1$: this implies that the node labelled by $\{sw, \mathrm{ON}\}$ cannot have a path performing an infinite number of self loops, because the side condition $\neg b \equiv \hat{t} - t \leq 50$ can only hold for a finite time interval. With this in mind, all infinite paths satisfying $\mathbf{G}sw$ must pass through the state with label $\{sw, \mathrm{ON}, b\}$. □

9.4 Discussion

Modelling formalisms where all parallel components fire transitions simultaneously in zero time, as soon as their trigger conditions are fulfilled are called *synchronous*; it is also said that they implement the *true paral-*



Initial states are drawn as boxes.

Figure 9.4: Transition graph of the simulation KS specified in Example 19.

lelism paradigm. They are appropriate for modelling multi-core systems or distributed systems where different tasks can perform computation steps in a truly simultaneous way. Since parallelism is basically expressed by logical conjunction, the model deadlocks as soon as *racing conditions* occur: If one action or several actions executed by simultaneous transitions try to write different values to the same variable, say $\alpha = \{(x, 5), (x, 6)\}$, this leads to a logical contradiction, such as $x' = 5 \wedge x' = 6$. As a consequence, the transition relation predicate has no solution, and the system is blocked. As a consequence, models containing racing conditions are not allowed.

In contrast to true parallelism, formalisms using *interleaving semantics* do not block in presence of racing conditions: These semantics stipulate that no two events – say $e_1 =_{\text{def}} x := 5;, e_2 =_{\text{def}} x := 6;$ may happen simultaneously, but are always causally related. So either e_1 happens before e_2 or vice versa, and you get the result of the event that has been executed last. This paradigm corresponds to quasi-parallel execution of events. It is only applicable if it can be assured that events are atomic. This is not the case, for example, if assignments to wide integers or floats are made, which need two memory bus transfers for one assignment: as consequence, two "interleaved" assignments may lead to a result where the upper word contains the value of the first assignment while the lower word contains the value of the second assignment or the other way round. If these situations have to be taken into account, it is better to use synchronous semantics and disallow racing conditions, because the atomicity assumption of interleaving semantics is not justified.

The transition relation specified above is *non-compositional* in the sense that it is not just defined by the conjunction of local transition relations for isolated state machines, but additional predicates specify the conditions when variables remain unchanged. This is the price to pay for being allowed to use shared variables in $V_L \cup O$, which can be written to by more than one state machine.

The most important dense-time formalisms with interleaving semantics is called *Timed Automata* [5, Chapter 17]. In contrast to the Timed State Machines investigated in this chapter, Timed Automata (TA) have the following distinguishing properties, apart from the fact that they are based

on interleaving semantics.

- Synchronisation events for concurrent TA
- Atomic operations involving multiple assignments
- Non-urgency. Transitions without synchronisation events, whose guard conditions are enabled, do not need to perform the transition immediately. Instead, a TA can "linger" in a control state as long as a *state invariant* still evaluates to true.
- Nondeterminism. Several transitions leaving a control state may be enabled at the same time. A nondeterministic choice is performed which one (if any) fires. There is no prioritisiation.

9.5 Clock Abstraction

In order to perform finite-state model checking of timed state machine properties we introduce *clock variables*, applying the well-known abstraction techniques introduced in Section 7. Given a timed state machine s with timers $t_i \in T$ and current time $\hat{t}$ the auxiliary variables

$$x_i(\hat{t}, t_i) =_{\text{def}} (\hat{t} - t_i),\ t_i \in T$$

are called clock variables; let C denote the set of all these x_i. Observe that, since $\hat{t}$ is an ideal clock, x_i satisfies

$$\frac{dx_i}{dz} = 1$$

where z denotes physical time.

Now we take AUX to be the set of all these clock variables together with all original variables used in s with exception of the timers, that is,

$$AUX =_{\text{def}} C \cup (V - T)$$

Let $\sim$ denote the equivalence relation induced by AUX according to the factorisation principle described in Section 7.2. Then, if K denotes the

Kripke structure associated with s, it is easy to see that $K/_\sim$ is bisimilar to K.

Since the original expressions involving timers t_i and model execution time $\hat{t}$ were assignments $t_i = \hat{t}$ and conditions $(\hat{t} - t_i) \geq c$, the only operations of interest on clock variables x_i are assignments $x_i = 0$ and conditions of the form $x_i \geq c$; the latter are called *atomic clock constraints*. The set $ACC(C)$ denotes the set of all atomic clock constraints. Just as timer conditions $(\hat{t} - t_i) \geq c$ may be combined by conjunction, atomic clock constraints can be connected by $\wedge$. If σ is a state of $K/_\sim$ then the valuation of (atomic and non atomic) clock constraints g is defined in the obvious way by

$$\begin{aligned}
\sigma &\models x < c \text{ iff } \sigma(x) < c \\
\sigma &\models x \leq c \text{ iff } \sigma(x) \leq c \\
\sigma &\models x > c \text{ iff } \sigma(x) > c \\
\sigma &\models x \geq c \text{ iff } \sigma(x) \geq c \\
\sigma &\models \neg g \text{ iff } \sigma \not\models g \\
\sigma &\models g \wedge g' \text{ iff } \sigma \models g \text{ and } \sigma \models g' \\
\sigma &\models g \vee g' \text{ iff } \sigma \models g \text{ or } \sigma \models g'
\end{aligned}$$

With these valuation rules at hand, a labelling function

$$L_C : S \to \mathbb{P}(ACC)$$

can be defined which maps every state σ to the set of atomic clock constraints valid in σ.

Example 20. The timed state machine shown in Fig. 9.2 and described in Example 17 can be modelled with clocks instead of timer variables as shown in Fig. 9.5: instead of timer variable $t \in T$ we introduce a clock x. The reset(t) command is transformed into a reset of the clock to zero. The elapsed(t,c) guard condition is changed into a guard $x \geq c$. The initial condition and transition relation for the new model is easily derived from the original predicates shown in Example 17:

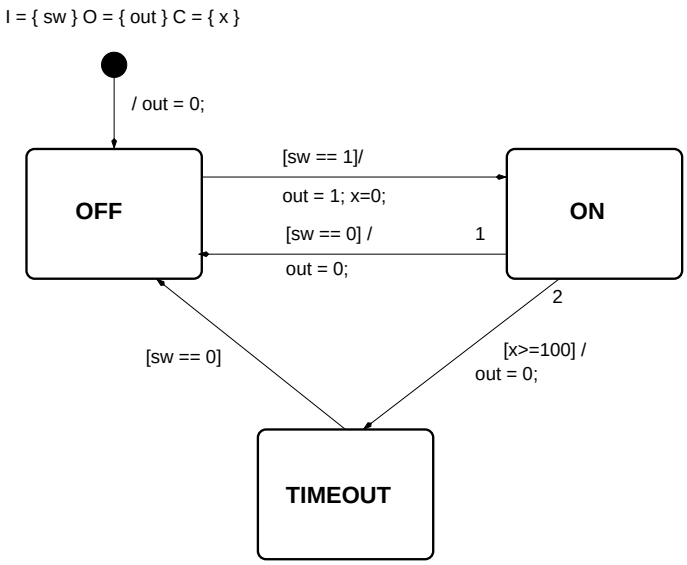


Figure 9.5: Timed state machine s with clock instead of timer variable.

$$
\begin{aligned}
&S_0/_\sim = \{s \in S/_\sim \mid s \models I/_\sim\}\\
&I/_\sim \equiv \mathrm{start}(s) \wedge \hat{t} = 0 \wedge \mathrm{INV}/_\sim\\
&\mathrm{INV}/_\sim \equiv (\mathrm{start}(s) \vee \mathrm{OFF} \vee \mathrm{ON} \vee \mathrm{TIMEOUT}) \wedge\\
&\qquad \neg(\mathrm{start}(s) \wedge \mathrm{OFF}) \wedge \neg(\mathrm{start}(s) \wedge \mathrm{ON}) \wedge \neg(\mathrm{start}(s) \wedge \mathrm{TIMEOUT})) \wedge\\
&\qquad \neg(\mathrm{OFF} \wedge \mathrm{ON}) \wedge \neg(\mathrm{OFF} \wedge \mathrm{TIMEOUT}) \wedge \neg(\mathrm{ON} \wedge \mathrm{TIMEOUT}) \wedge\\
&\qquad \frac{d\hat{t}}{dz} = 1 \wedge \frac{dx}{dz} = 1
\end{aligned}
$$

$$
\begin{aligned}
R/_\sim \equiv\ & \mathrm{INV}/_\sim \wedge \mathrm{INV}/'_\sim \wedge ((\mathrm{start}(s) \wedge sw' = sw \wedge x' = x \wedge \hat{t}' = \hat{t} \wedge out' = 0 \wedge \mathrm{OFF}') \vee\\
& (\mathrm{OFF} \wedge sw = 0 \wedge \hat{t}' > \hat{t} \wedge out' = out \wedge x' = x + \hat{t}' - \hat{t} \wedge \mathrm{OFF}') \vee\\
& (\mathrm{OFF} \wedge sw = 1 \wedge sw' = sw \wedge \hat{t}' = \hat{t} \wedge out' = 1 \wedge x' = x \wedge \mathrm{ON}') \vee\\
& (\mathrm{ON} \wedge sw = 1 \wedge \hat{t}' > \hat{t} \wedge x' = x + \hat{t}' - \hat{t} \wedge x < 100 \wedge x' \leq 100 \wedge out' = out \wedge \mathrm{ON}') \vee\\
& (\mathrm{ON} \wedge sw = 1 \wedge x \geq 100 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge out' = 0 \wedge x' = x \wedge \mathrm{TIMEOUT}') \vee\\
& (\mathrm{ON} \wedge sw = 0 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge out' = 0 \wedge x' = x \wedge \mathrm{OFF}') \vee\\
& (\mathrm{TIMEOUT} \wedge sw = 1 \wedge \hat{t}' > \hat{t} \wedge x' = x + \hat{t}' - \hat{t} \wedge out' = out \wedge \mathrm{TIMEOUT}') \vee\\
& (\mathrm{TIMEOUT} \wedge sw = 0 \wedge \hat{t}' = \hat{t} \wedge sw' = sw \wedge out' = out \wedge t' = t \wedge \mathrm{OFF}'))
\end{aligned}
$$

Note that in the definition of $R/_\sim$ we could drop the conjuncts $x' = x + \hat{t}' - \hat{t}$ because this is already implied by $\frac{d\hat{t}}{dz} = 1 \wedge \frac{dx}{dz} = 1$ which is part of the invariant. □

9.6 Property Specifications for Timed State Machines

As variants of CTL have been introduced to describe properties of reactive systems without timing aspects, we will now define *TCTLX (Timed CTL With Next Operator)* for property specification of timed state machines. Observe that TCTLX has been derived from TCTL which was introduced for reasoning about timed automata [1]. Since timed automata are non-deterministic and allow non-urgent execution of discrete transitions, a Next-operator has no meaning in this context, because uncountably many delays may occur in most situations before a discrete transition fires. In contrast to this, TCTLX has a well-defined meaning for the Next-operator:

$\mathbf{X}\phi \equiv_{\text{def}}$ the next transition is a discrete one and its post-state satisfies ϕ

Just as in TCTL, TCTLX defines timing properties by means of a timed variant of the Until-operator:

$$\phi \mathbf{U}^J \psi$$

asserts that property ψ will be fulfilled within $t \in J$ time units, where t is taken from some interval $J \subseteq \mathbb{R}_{\geq 0}$, and until then ϕ holds. Any time interval $J \subseteq \mathbb{R}_{\geq 0}$ with open or closed boundaries is admissible; in particular unbounded restrictions like $J = [u, \infty), u \geq 0$ is allowed. Timed variants of the Globally and Finally operators are defined as syntactic abbreviations of constructs involving the timed Until-operator:

$$\begin{aligned} \mathbf{F}^J\phi &\equiv_{\text{def}} \texttt{true}\mathbf{U}^J\phi \\ \mathbf{EG}^J\phi &\equiv_{\text{def}} \neg\mathbf{A}\mathbf{F}^J\neg\phi \\ \mathbf{AG}^J\phi &\equiv_{\text{def}} \neg\mathbf{E}\mathbf{F}^J\neg\phi \end{aligned}$$

Observe that these definitions are quite intuitive: $\mathbf{AG}^J\phi$, for example, asserts that ϕ holds on every path at least for the time period $t \in J$.

More formally, TCTLX syntax is defined as follows.

TCTLX-formula ::= ϕ
ϕ ::= p | g | $\neg\phi$ | $\phi \vee \phi$ | $\phi \wedge \phi$ | $\mathbf{E}\ \psi$ | $\mathbf{A}\ \psi$
ψ ::= ϕ | $\neg\psi$ | $\psi \vee \psi$ | $\psi \wedge \psi$ | $\mathbf{X}\ \phi$ | $\mathbf{F}^J\ \phi$ | $\mathbf{G}^J\ \phi$ | $\phi\ \mathbf{U}^J\ \phi$

In this syntax definition, $p \in AP$ denotes an "ordinary" atomic proposition, and $g \in ACC(C)$ an atomic clock constraint.

Given a system model M of concurrent timed state machines whose initial condition is defined by predicate I and whose transition relation is given by Φ as introduced above, the semantics of a TCTLX formula is defined in Fig. 9.6. All paths π referenced in this definition are assumed to be time-divergent. If

$$\pi = \sigma_0.\sigma_1.\sigma_2 \ldots$$

then $d_i, i \geq 0$ are defined as the delays between consecutive states, that is,

$$d_i =_{def} (\sigma_{i+1}(\hat{t}) - \sigma_i(\hat{t}))$$

For $d \in \mathbb{R}_{\geq 0}$ a *time shift* $\sigma + d$ is defined on states σ by setting

$$(\sigma + d)(v) =_{def} \begin{cases} \sigma(v) & \text{if } v \in V - (C \cup \{\hat{t}\}) \\ \sigma(\hat{t}) + d & \text{if } v = \hat{t} \\ \sigma(v) + d & \text{if } v \in C \end{cases}$$

9.7 Property Checking of Concurrent Timed State Machines

The fundamental idea for TCTLX property checking time state machines has been adopted from TCTL property checking of Timed Automata [1, 5]. We follow, however, the general abstraction approach for Kripke Structures introduced in Section 7 and show that our usual construction technique is applicable to use classical model checking on timed state machines:

- A first abstraction is introduced by "forgetting" about all atomic propositions of the concrete Kripke structure referring to explicit model execution time $\hat{t}$ and confine ourselves to atomic clock constraints only.
- Since both TCTLX formulas and timed state machine guard conditions refer to atomic clock constraints only, every property expressed in TCTLX can be verified on this first abstraction of the original Kripke structure.

$$
\begin{array}{lcl}
M, s \models p & \equiv & p \in L(s) \\
M, s \models g & \equiv & p \in L_C(s) \\
M, s \models \neg\phi & \equiv & M, s \not\models \phi \\
M, s \models \phi_1 \vee \phi_2 & \equiv & M, s \models \phi_1 \text{ or } M, s \models \phi_2 \\
M, s \models \phi_1 \wedge \phi_2 & \equiv & M, s \models \phi_1 \text{ and } M, s \models \phi_2 \\
M, s \models \mathbf{E}\ \psi & \equiv & \text{there is a time-divergent path } \pi \text{ from } s \text{ such that } M, \pi \models \psi \\
M, s \models \mathbf{A}\ \psi & \equiv & \text{on every time-divergent path } \pi \text{ from } s \text{ holds } M, \pi \models \psi \\
M, \pi \models \phi & \equiv & M, \pi(0) \models \phi \\
M, \pi \models \neg\psi & \equiv & M, \pi \not\models \psi \\
M, \pi \models \psi_1 \vee \psi_2 & \equiv & M, \pi \models \psi_1 \text{ or } M, \pi \models \psi_2 \\
M, \pi \models \psi_1 \wedge \psi_2 & \equiv & M, \pi \models \psi_1 \text{ and } M, \pi \models \psi_2 \\
M, \pi \models \mathbf{X}\ \psi & \equiv & M, \pi(0) \models \text{trigger}_D \text{ and } M, \pi^1 \models \psi \\
M, \pi \models \psi_1 \mathbf{U}^J \psi_2 & \equiv & (1) \text{ there exists } i \geq 0, d \in \mathbb{R}_{\geq 0} \text{ such that } d \in [0, d_i], \\
 & & d + \Sigma_{k=0}^{i-1} d_k \in J \text{ and } M, (\pi(i) + d).\pi^{i+1} \models \psi_2 \\
 & & \text{and} \\
 & & (2) \text{ for all } 0 \leq j < i, \text{ for all } d' \in [0, d_j] \text{ satisfying} \\
 & & d' + \Sigma_{k=0}^{j-1} d_k \leq d + \Sigma_{k=0}^{i-1} d_k\ M, (\pi(j) + d').\pi^{j+1} \models \psi_1 \vee \psi_2
\end{array}
$$

Figure 9.6: Semantics of TCTLX formulas.

- The originally uncountable state space is abstracted to a countable state space by collapsing all concrete system states whose clock valuations lie in the same clock region (a concept to be introduced in the next section) into a single equivalence class.
- By collapsing all clock regions referring to clock values no longer "relevant" for the verification goal under consideration, the countable collection of clock regions is reduced to a finite one.
- The finite collection of remaining "relevant" clock regions is specified by a finite number of abstractions $a_i = e_i(x_1, \ldots, x_n)$ as introduced in Section 7.
- On the resulting finite Kripke Structure CTL property checking may be performed with the algorithms introduced in Section 4.
- It is shown that TCTLX formulas over the original system can be expressed as CTL formulas over the finite abstration.
- It is shown that the abstracted Kripke Structure is bi-similar to the original one. Therefore *every* CTL formula (an not only ACTL properties) which holds for the abstracted system hold for the original one.

We introduce the concepts for TCTLX property checking of timed state machines by means of Example 21.

Example 21. The control mechanism from Fig. 9.5 is extended to a concurrent controller as depicted in Fig. 9.7. The original control state machine from Fig.9.5 is still present as state machine `s1`, but has been modified in the following way:

- The time scale has been changed so that the timeout occurs a time 1 instead of 100. This has only been done to reduce the number of clock regions which are introduced below.
- Whenever the machine is switched off due to the timeout `x >= 1` used as trigger in the transition `l1`→`l2`, a counter is increment in order

to record the number of timeouts which had to be handled since the system has been activated.

- As soon as an internal shutdown command `off = 1;` is given by state machine `s2`, state machine `s1` performs a transition into control state `shutdown`, stops the machine by setting `out = 0;` and remains passive.

State machine `s1` has been augmented by a new state machine `s2` which resets a clock `y` as soon as the switch `sw` has been activated for the first time. After two time units have elapsed, machine `s2` shuts down the controller by setting `off = 1;`.

Observe that the number of transitions `l1`$\rightarrow$`l0` is unbounded because the amount of time spending in location `l1` before switching `sw` manually back to 0 may be infinitesimally small. For the transition `l1`$\rightarrow$`l2` to occur, however, one time unit has to pass. We wish to prove via model checking whether our intuition is right that the counter `ctr` can never become greater than 2. A closer look shows that even the value 2 may never be reached: Incrementing `ctr` to 2 requires 2 transitions from `l1` to `l2`, each transition requiring `s1` to linger in `l1` for 1 time unit. Transitions `l2` $\rightarrow$ `l0` $\rightarrow$ `l1` require a value change $0 \rightarrow 1$ for input `sw`, and this requires at least one delay transition of duration $\varepsilon > 0$. As a consequence `s1` needs more than 2 time units to increment the counter to 2, while `s2` sets the shutdown signal exactly after 2 time units have passed. Formally speaking, we wish to check the TCTLX formula

$$\mathbf{AG}(\mathrm{ctr} < 2)$$

□

9.8 Clock Regions

Clock regions are constructed to identify vectors of clock valuations, each vector component for one clock, for which the system will behave in an equivalent way. The construction "recipe" for clock regions is as follows.

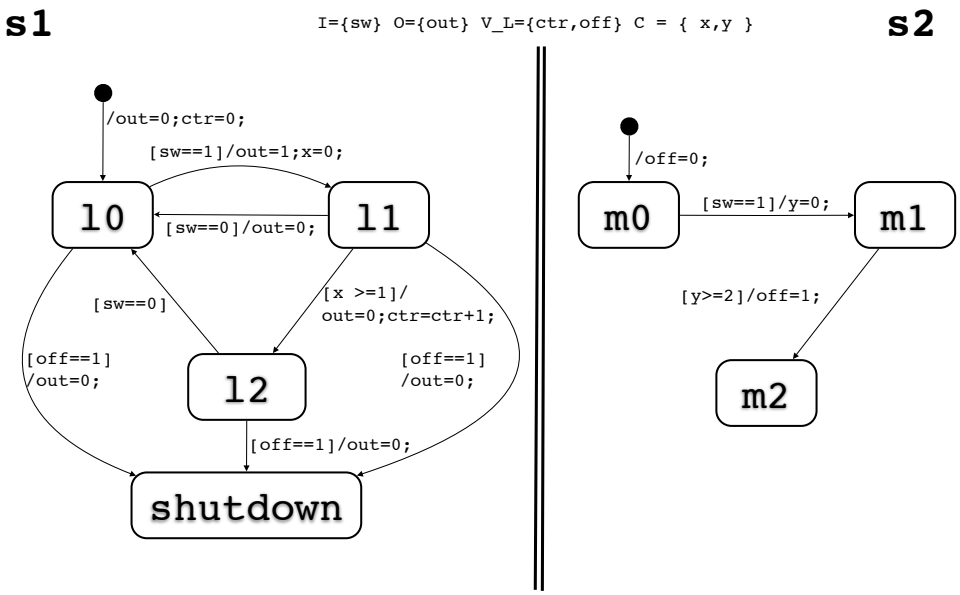


Figure 9.7: Two concurrent timed state machines for controlling a machine via interface out with switch-off clock and a final-shutdown clock.

Step 1. For each clock $x \in C$, let $c_x \in \mathbb{N}$ the largest integer c occurring in an atomic clock constraint $x \geq c, x > c, x \leq c, x < c, x = c$, either in a guard condition or in the TCTLX property.

Step 2. For each clock $x \in C$, define elementary regions by the following atomic clock constraints.

$$\begin{array}{l} x = 0 \\ x \in (0,1) \\ x = 1 \\ x \in (1,2) \\ \ldots \\ x \in (c_x - 1, c_x) \\ x = c_x \\ x \in (c_x, \infty) \end{array}$$

This defines $2 \cdot (c_x + 1)$ clock constraints, and we use function

$$\alpha : C \times \mathbb{N}_0 \not\longrightarrow ACC$$

as abbreviation for these constraints. For example, if $c_x = 5$, $\alpha(x,n)$ is defined for $n = 0, 1, \ldots, 9$, and $\alpha(x,7) \equiv x \in (3,4)$. More general,

$$\alpha(x,n) =_{\text{def}} \begin{cases} x = n \texttt{ div } 2 & n \texttt{ mod } 2 = 0 \\ x \in (n \texttt{ div } 2, (n \texttt{ div } 2) + 1) & n \texttt{ mod } 2 = 1 \end{cases}$$

Step 4. For different clocks whose current valuation is inside some open interval of length 1, it is necessary to know the ordering of their fractional parts $frac(x)$, because the clock whose valuation has the largest fractional part will be the next to meet an integer threshold $x \geq c$, so that a discrete transition might become enabled. Let

$$\beta : \{0, \ldots, |C| - 1\} \longrightarrow C$$

a permutation signifying the predicate

$$frac(\beta(0)) \leq frac(\beta(1)) \leq frac(\beta(|C| - 1))$$

Since the valuations of some clocks may have the same fractional part we need another function

$$\gamma : \{1, \ldots, |C| - 1\} \longrightarrow \mathbb{B}$$

signifying whether $frac(\beta(i-1))\ \omega\ frac(\beta(i))$ holds with $\omega = '<'$ ($\gamma(i) = 1$) or $\omega = '='$ ($\gamma(i) = 0$). Let

$$ord(\beta, \gamma)$$

denote the predicate stating the order of fractional parts of all clocks according to a given β, γ.

Step 5. A clock region is a conjunction

$$\bigwedge_{x \in C} \alpha(x, n_x) \wedge ord(\beta, \gamma)$$

such that each (x, n_x) is in the domain of α and β, γ are defined as explained in Step 4.

9.9 Abstraction by Clock Regions

Given the full collection of constraints defining clock regions as described in the section above we can introduce abstractions using all atomic constraints created during this process.

Example 22. The clock regions associated with Example 21 induce the following abstraction functions (observe that $c_x = 1$ and $c_y = 2$):

$$\begin{array}{lcllcllcl}
a_0 & = & (x = 0) & b_0 & = & (y = 0) & f_0 & = & (frac(x) < frac(y)) \\
a_1 & = & (x \in (0,1)) & b_1 & = & (y \in (0,1)) & f_1 & = & (frac(x) = frac(y)) \\
a_2 & = & (x = 1) & b_2 & = & (y = 1) & f_2 & = & (frac(y) < frac(x)) \\
a_3 & = & (x \in (1,\infty)) & b_3 & = & (y \in (1,2)) & & & \\
 & & & b_4 & = & (y = 2) & & & \\
 & & & b_5 & = & (y \in (2,\infty)) & & &
\end{array}$$

Applying the usual construction of initial condition and transition relation (I, R) for the concrete system and abstracting to $([I], [R])$ as explained in Section 7, yields the abstracted finite Kripke Structure depicted in Fig. 9.8. Evaluation of all graph nodes of the abstracted Kripke Structure immediately shows that the desired property $\mathbf{AG}(ctr < 2)$ holds. □

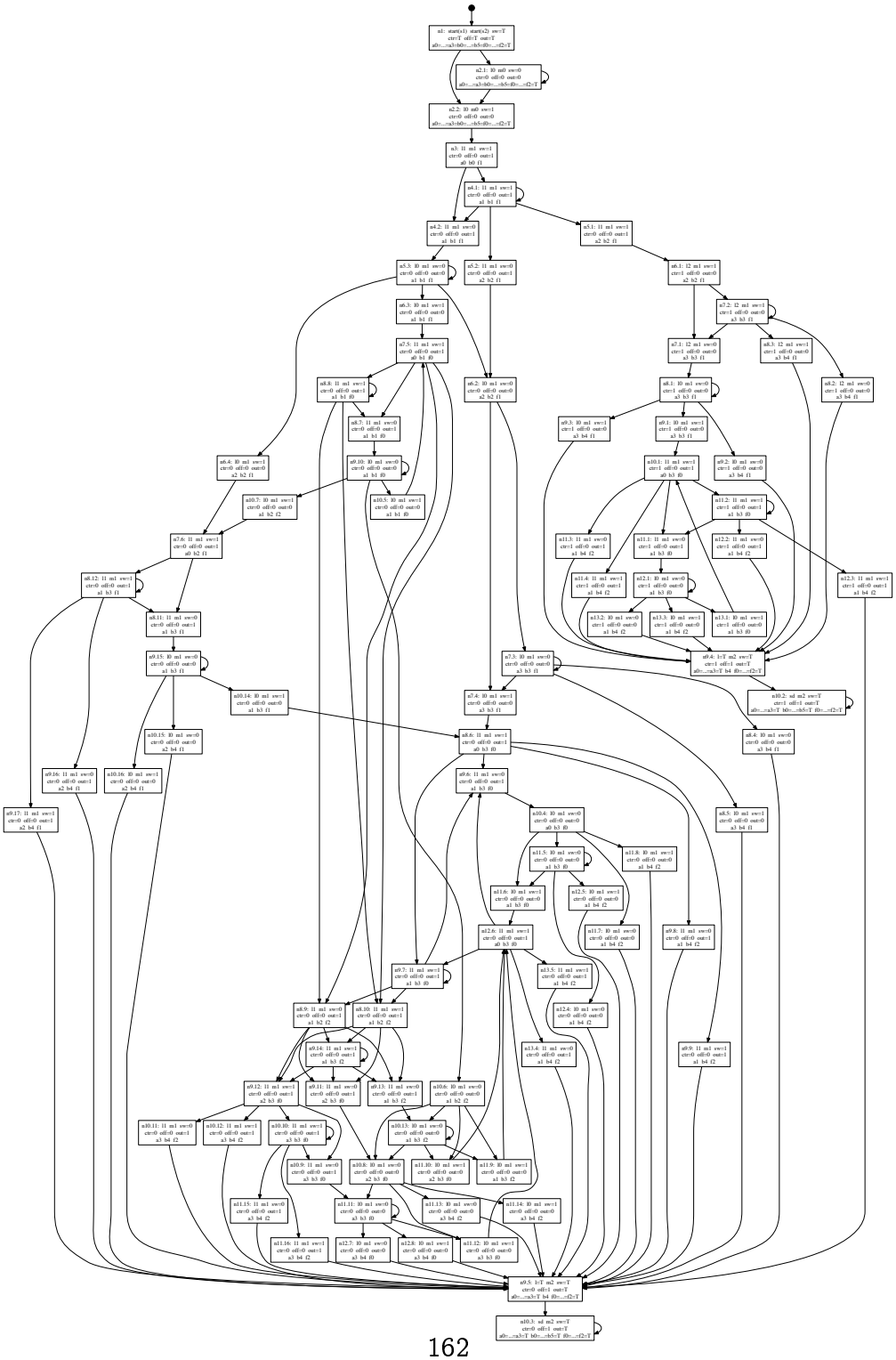


Figure 9.8: Abstracted Kripke Structure for system from Example 21. (Best viewed with PDF reader, magnification.)

Bibliography

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, Massachusetts, 2008.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. ISBN 3-540-65703-7. doi: 10.1007/3-540-49059-0_14. URL http://dx.doi.org/10.1007/3-540-49059-0_14.

[3] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded ltl model checking. *Logical Methods in Computer Science*, 2(5):1–64, 2006.

[4] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014. doi: 10.1007/978-3-319-08867-9_22. URL https://doi.org/10.1007/978-3-319-08867-9_22.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. ISBN 0262032708.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. ISBN 0262032708.

[7] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[8] T.A. Henzinger. The theory of hybrid automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, July 1996. doi: 10.1109/LICS.1996.561342. ISSN: 1043-6871.

[9] C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International, Englewood Cliffs NJ, 1985.

[10] L. Loeckx and K. Sieber. *The Foundations of Program Verification*. Series in Computer Science. Wiley–Teubner, 1984.

[11] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992. ISBN 978-3-540-97664-6.

[12] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995. ISBN 978-0-387-94459-3.

[13] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. In Alexander K. Petrenko and Holger Schlingloff, editors, Proceedings Eighth Workshop on *Model-Based Testing,* Rome, Italy, 17th March 2013, volume 111 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–28. Open Publishing Association, 2013. doi: 10.4204/EPTCS.111.1.

[14] Jan Peleska, Niklas Krafczyk, Anne E. Haxthausen, and Ralf Pinger. Efficient data validation for geographical interlocking systems. *Formal Aspects Comput.*, 33(6):925–955, 2021.

doi: 10.1007/s00165-021-00551-6. URL https://doi.org/10.1007/s00165-021-00551-6.

[15] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi: 10.1109/SFCS.1977.32. URL http://dx.doi.org/10.1109/SFCS.1977.32.

[16] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000. ISBN 3-540-41219-0. doi: 10.1007/3-540-40922-X_8. URL http://dx.doi.org/10.1007/3-540-40922-X_8.

[17] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511, September 1994. ISSN 0934-5043, 1433-299X. doi: 10.1007/BF01211865. URL http://link.springer.com/article/10.1007/BF01211865.

[18] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

Appendix A

Structural Induction

In this section the principle of *structural induction* is introduced. The material is taken from [10, pp. 8].

Definition 11 (Inductive Definition of Sets) *Let* U *be a set called* universe *and* $B \subseteq U$, *called the* base set. *Let* K *a set of relations* $r \subseteq U^n \times U$, *where* $n \in \mathbb{N}$ *depends on* r. K *is called the* constructor set *and each* $r \in K$ *a* constructor. *A set* $A \subseteq U$ *is called* inductively defined *by* B *and* K, *if* A *is the smallest subset of* U *satisfying*

1. $B \subseteq A$
2. *If* $a_1, \ldots, a_n \in A$ *and* $((a_1, \ldots, a_n), a) \in r$ *for some constructor* $r \in K$, *then* $a \in A$.

Theorem 13 (Principle of Structural Induction) *let* $A \subseteq U$ *be inductively defined by base set* B *and constructor set* K, *and* $P(x)$ *a property on elements of* $x \in A$. *Suppose that*

1. Induction basis. $P(x)$ *holds for all* $x \in B$.
2. Induction step. *If* $P(a_i), i = 1, \ldots, n$ *holds for* $a_1, \ldots, a_n \in A$ (induction hypothesis) *and* $((a_1, \ldots, a_n), a) \in r$ *for some constructor* $r \in K$, *then* $P(a)$ *also holds.*

Then $P(a)$ *holds for all* $a \in A$. □

Appendix B

Lattices, Galois Connections, and Kripke Structures

In this Chapter we introduce lattices and Galois connections first as abstract concepts, independent on their application to Kripke Structures, model checking, and test automation. The definitions and results recapped here are based on the detailed exposition on lattices and order presented in [7].

B.1 Lattices

Recall that a binary relation $\leq$ on a set L is called a *(partial) order* if $\leq$ is reflexive, transitive and anti-symmetric. An element $y \in L$ is called an *upper bound of* $X \subseteq L$ if $x \leq y$ holds for all $x \in X$. The lower bound of a set is defined dually. An upper bound y' of X is called a *least upper bound of* X and denoted by $y' = \bigvee X$ if $y' \leq y$ holds for all upper bounds y of X. Dually, the *greatest lower bound* $\bigwedge X$ of a set X is defined, that is, $z \leq \bigwedge X \leq x$ for all lower bounds z of X and for all $x \in X$.

An ordered set $(L, \leq)$ is called a *complete lattice*, if $\bigwedge X$ and $\bigvee X$ exist as elements of L for all subsets $X \subseteq L$. Lattice L has a *largest element* (or *top*) denoted by $\top = \bigvee L$ and a *smallest element* (or *bottom*) denoted by $\bot = \bigwedge L$. Least upper bounds and greatest lower bounds induce binary operations $\vee, \wedge : L \times L \to L$ by defining $x \vee y = \bigvee\{x, y\}$ (the *join* of x and

y) and $x \wedge y = \bigwedge\{x, y\}$ (the *meet* of x and y), respectively. If the join and meet are well-defined for an ordered set $(L, \leq)$ but $\bigvee X, \bigwedge X$ do not exist for all $X \subseteq L$ then $(L, \leq)$ is called an *(incomplete) lattice*.

Mappings $\phi : (L_1, \leq_1) \to (L_2, \leq_2)$ between ordered sets are called *monotonic* if $x \leq_1 y$ implies $\phi(x) \leq_2 \phi(y)$ for all $x, y \in L$. Mappings $\phi : (L_1, \leq_1) \to (L_2, \leq_2)$ between lattices are called *homomorphisms* if they respect meets and joins, that is, $\phi(x \vee_1 y) = \phi(x) \vee_2 \phi(y)$ and $\phi(x \wedge_1 y) = \phi(x) \wedge_2 \phi(y)$ for all $x, y \in (L_1, \leq_1)$. Since $x \leq_1 y$ implies $x \vee_1 y = y$ and $x \wedge_1 y = x$, homomorphisms are monotonic. Lattice homomorphisms map $\top$ to $\top$ and $\bot$ to $\bot$.

Example 23.

1. For every set M the *power set lattice* is defined by $(\mathbb{P}(M), \subseteq)$. The join is defined by $m \vee m' =_{\text{def}} m \cup m'$, the meet by $m \wedge m' =_{\text{def}} m \cap m'$. Top and bottom elements are $\top = M$, $\bot = \varnothing$, respectively.

2. For every set M we can introduce a nearly trivial ordering $\sqsubseteq$ by adding two new elements $\top, \bot \notin M$ and defining a lattice $(M \cup \{\top, \bot\}, \sqsubseteq)$ such that all $m \neq m' \in M$ are incomparable and $\forall m \in M : \bot \sqsubseteq m \sqsubseteq \top$.

3. Applying the above construction to Booleans $\mathbb{B} = \{\texttt{false}, \texttt{true}\}$ results in the lattice $(L(\mathbb{B}), \sqsubseteq)$ with $L(\mathbb{B}) =_{\text{def}} \{\bot, \texttt{false}, \texttt{true}, \top\}$, $\bot \sqsubseteq \texttt{false} \sqsubseteq \top, \bot \sqsubseteq \texttt{true} \sqsubseteq \top$ and $\texttt{true}, \texttt{false}$ incomparable. The top element $\top$ has the intuitive interpretation "undecided – maybe true or false".

4. $(\mathbb{Q}, \leq)$ is an *incomplete* lattice: Take any infinite set $S \subseteq \mathbb{Q}$ whose elements are converging towards a transcendent number, say $\sqrt{2}$, from below. Then $\bigvee S \notin \mathbb{Q}$.

5. The lattice of *intervals* over reals including $\pm\infty$ is defined as $(\mathbb{IR}, \subseteq)$ with $[\underline{a}, \overline{a}] \wedge [\underline{b}, \overline{b}] =_{\text{def}} [\underline{a}, \overline{a}] \cap [\underline{b}, \overline{b}]$ and $[\underline{a}, \overline{a}] \vee [\underline{b}, \overline{b}] =_{\text{def}} [\min\{\underline{a}, \underline{b}\}, \max\{\overline{a}, \overline{b}\}]$. The join of $[\underline{a}, \overline{a}]$ and $[\underline{b}, \overline{b}]$ is also called the *interval hull* of $[\underline{a}, \overline{a}]$ and $[\underline{b}, \overline{b}]$. The maximal element is $\top = [-\infty, +\infty]$, and $\bot = [\,] = \varnothing$.

6. Interval lattices may be introduced over integral numbers from $\mathbb{Z}$ or $\mathbb{N}$ and over rational numbers $\mathbb{Q}$ in analogy to (5). Interval lattices over $\mathbb{Z}$ and $\mathbb{N}$ are complete. The interval lattice over $\mathbb{Q}$ is not complete, because an infinite sequence of intervals may have a supremum which is an interval of $(\mathbb{IR}, \subseteq)$, but is not an interval of $\mathbb{Q}$, since its boundaries are irrational numbers.

7. Let Pred be a set of first order predicates over a given set V of typed variable symbols. For $\varphi, \psi \in \mathrm{Pred}$, define

$$[\varphi \Rightarrow \psi] \equiv \forall d_\nu \in D_\nu : \varphi[d_\nu/\nu | \nu \in V] \Rightarrow \psi[d_\nu/\nu | \nu \in V]$$

$[\varphi \Rightarrow \psi]$ indicates that φ implies ψ for every admissible valuation of free variables (observe that some $\nu \in V$ might not occur in φ or ψ or both propositions), that is, $\varphi \Rightarrow \psi$ is a tautology. Then $(\mathrm{Pred}, \Rightarrow)$ is a lattice with join $\vee$ (logical or) and meet $\wedge$ (logical and), top-element `true` and bottom element `false`. For finite sets $P \subseteq \mathrm{Pred}$ the least upper bound is the disjunction

$$\bigvee_{\varphi \in P} \varphi$$

and the greatest lower bound is the conjunction

$$\bigwedge_{\varphi \in P} \varphi$$

because obviously

$$\forall \psi \in P : \left[\psi \Rightarrow \bigvee_{\varphi \in P} \varphi\right] \quad \text{and} \quad \left[\bigwedge_{\varphi \in P} \varphi \Rightarrow \psi\right]$$

□

Example 24. We show for the implication lattice (Item 7 of the previous example) that $\bigvee_{\varphi \in P} \varphi$ is indeed a *lowest* upper bound of a finite set P of predicates. To this end, let $\xi \in \mathrm{Pred}$ be another upper bound of P, that is,

$$\forall \psi \in P : [\psi \Rightarrow \xi]$$

Then this can be equivalently re-written as

$$\left[\bigvee_{\varphi \in P} \varphi \Rightarrow \xi\right]$$

which proves the lowest upper bound property for $\bigvee_{\varphi \in P} \varphi$. □

B.2 Galois Connections

A *Galois connection (GC)* between $(L_1, \leq_1), (L_2, \leq_2)$ is a tuple of mappings $F : (L_1, \leq_1) \to (L_2, \leq_2)$ (called *lower adjoint*) and $F^* : (L_2, \leq_2) \to (L_1, \leq_1)$ (called *upper adjoint*) such that

$$F(a) \leq_2 b \Leftrightarrow a \leq_1 F^*(b) \quad \text{for all } a \in L_1, b \in L_2$$

This defining characteristic implies additional properties [7] which are listed in the following lemma.

Lemma 7 *Let* $F : (L_1, \leq_1) \to (L_2, \leq_2)$, $F^* : (L_2, \leq_2) \to (L_1, \leq_1)$ *be a GC. Then the following properties are fulfilled by* F *and* F^*.

1. $\forall a \in L_1 : a \leq_1 F^*(F(a))$
2. $\forall b \in L_2 : F(F^*(b)) \leq_2 b$
3. F *and* F^* *are monotonic.*
4. F *preserves joins:* $\forall p_1, p_2 \in L_1 : F(p_1 \vee p_2) = F(p_1) \vee F(P2)$.
5. F^* *preserves meets:* $\forall q_1, q_2 \in L_2 : F^*(q_1 \wedge q_2) = F^*(q_1) \wedge F^*(q_2)$.
6. *Given* F, *if* $(L_1, \leq_1)$ *is complete, the left mapping* F^* *is fully determined by*

$$\forall b \in L_2 : F^*(b) = \bigvee \{a \in L_1 \mid F(a) \leq_2 b\}$$

7. *Given* F^*, *if* $(L_2, \leq_2)$ *is complete, the right mapping* F *is fully determined by*

$$\forall a \in L_1 : F(a) = \bigwedge \{b \in L_2 \mid a \leq_1 F^*(b)\}$$

Proof. **Proof of property 1.** Set $b = F(a) \in L_2$. Now we derive step by

step

$$
\begin{aligned}
F(a) \leq_2 F(a) \quad & [\leq_2 \text{ is reflexive}] \\
\Rightarrow \quad & F(a) \leq_2 b \\
& [\text{Definition of } b] \\
\Rightarrow \quad & a \leq_1 F^*(b) \\
& [\text{Defining property of GCs}] \\
\Rightarrow \quad & a \leq_1 F^*(F(a)) \\
& [\text{Definition of } b]
\end{aligned}
$$

Proof of property 2. Set $a = F^*(b) \in L_1$. Then

$$
\begin{aligned}
F^*(b) \leq_1 F^*(b) \quad & [\leq_1 \text{ is reflexive}] \\
\Rightarrow \quad & a \leq_1 F^*(b) \\
& [\text{Definition of } a] \\
\Rightarrow \quad & F(a) \leq_2 b \\
& [\text{Defining property of GCs}] \\
\Rightarrow \quad & F(F^*(b)) \leq_2 b \\
& [\text{Definition of } a]
\end{aligned}
$$

Proof of property 3. Let $a_1, a_2 \in L_1$. Then

$$
\begin{aligned}
a_1 \leq_1 a_2 \quad \Rightarrow \quad & a_1 \leq_1 F^*F(a_2) \\
& [\text{Property 1 implies } a_2 \leq_1 F^*F(a_2) \text{ and } \leq_1 \text{ is transitive}] \\
\Rightarrow \quad & F(a_1) \leq_2 F(a_2) \\
& [\text{Defining property of GCs with } b = F(a_2)]
\end{aligned}
$$

This proves that F is monotonic. Now let $b_1, b_2 \in L_2$. Then

$$
\begin{aligned}
b_1 \leq_2 b_2 \quad \Rightarrow \quad & FF^*(b_1) \leq_2 b_2 \\
& [\text{Property 2 implies } FF^*(b_1) \leq_2 b_1 \text{ and } \leq_2 \text{ is transitive}] \\
\Rightarrow \quad & F^*(b_1) \leq_1 F^*(b_2) \\
& [\text{Defining property of GCs with } a = F^*(b_1)]
\end{aligned}
$$

This proves that F^* is monotonic.

Proof of property 4. By definition of the join function, we have $p_1 \leq_1 p_1 \vee p_2$ and $p_2 \leq_1 p_1 \vee p_2$. Monotonicity of F (Property 3) implies

$$F(p_1) \leq_2 F(p_1 \vee p_2) \wedge F(p_2) \leq_2 F(p_1 \vee p_2)$$

Therefore $F(p_1 \vee p_2)$ is an upper bound of $F(p_1)$ and of $F(p_2)$, so the least upper bound of $F(p_1)$ and of $F(p_2)$ is less or equal to $F(p_1 \vee p_2)$ This implies $F(p_1) \vee F(p_2) \leq_2 F(p_1 \vee p_2)$.

Now we show that $F(p_1 \vee p_2) \leq_2 F(p_1) \vee F(p_2)$ holds, too. Since $\leq_2$ is anti-symmetric, this implies $F(p_1 \vee p_2) = F(p_1) \vee F(p_2)$. To this end, assume that $z \in L_2$ is an arbitrary upper bound of $\{F(p_1), F(p_2)\}$. Then

$$\begin{aligned}
F(p_1) \leq_2 z \wedge F(p_2) \leq_2 z \quad &\Rightarrow \quad p_1 \leq_1 F^*(z) \wedge p_2 \leq_1 F^*(z) \\
&\quad\ [\text{Defining property of GCs}] \\
&\Rightarrow \quad p_1 \vee p_2 \leq_1 F^*(z) \\
&\quad\ [p_1 \vee p_2 \text{ is least upper bound of } \{p_1, p_2\}\ ] \\
&\Rightarrow \quad F(p_1 \vee p_2) \leq_2 z \\
&\quad\ [\text{Defining property of GCs}] \\
&\Rightarrow \quad F(p_1 \vee p_2) \leq_2 F(p_1) \vee F(p_2) \\
&\quad\ [z \text{ was an arbitrary upper bound, so we can} \\
&\quad\ \text{select } z = F(p_1) \vee F(p_2)]
\end{aligned}$$

Proof of property 5. By definition of the meet function, we have $q_1 \wedge_2 q_2 \leq_2 q_1$ and $q_1 \wedge_2 q_2 \leq_2 q_2$. Monotonicity of F^* (Property 3) implies

$$F^*(q_1 \wedge_2 q_2) \leq_1 F^*(q_1) \quad \text{and} \quad F^*(q_1 \wedge_2 q_2) \leq_1 F^*(q_2)$$

Therefore $F^*(q_1 \wedge_2 q_2)$ is a lower bound of $F^*(q_1)$ and of $F^*(q_2)$, so the greatest lower bound of $F^*(q_1)$ and of $F^*(q_2)$ is greater or equal to $F^*(q_1 \wedge_2 q_2)$ This implies $F^*(q_1 \wedge_2 q_2) \leq_1 F^*(q_1) \wedge_1 F^*(q_2)$.

Now we show that $F^*(q_1) \wedge_1 F^*(q_2) \leq_1 F^*(q_1 \wedge_2 q_2)$ holds, too. Since $\leq_1$ is anti-symmetric, this implies $F^*(q_1 \wedge_2 q_2) = F^*(q_1) \wedge_1 F^*(q_2)$. To this end,

assume that $w \in L_1$ is an arbitrary lower bound of $\{F^*(q_1), F^*(q_2)\}$. Then

$$\begin{aligned}
w \leq_1 F^*(q_1) \wedge w \leq_1 F^*(q_2) \quad &\Rightarrow \quad F(w) \leq_2 q_1 \wedge F(w) \leq_2 q_2 \\
&\quad [\text{Defining property of GCs}] \\
&\Rightarrow \quad F(w) \leq_2 q_1 \wedge_2 q_2 \\
&\quad [F(w) \text{ is a lower bound of } \{q_1, q_2\} \text{ and} \\
&\quad\ q_1 \wedge_2 q_2 \text{ is the greatest lower bound}] \\
&\Rightarrow \quad w \leq_2 F^*(q_1 \wedge_2 q_2) \\
&\quad [\text{Defining property of GCs}] \\
&\Rightarrow \quad F^*(q_1) \wedge_1 F^*(q_2) \leq_2 F^*(q_1 \wedge_2 q_2) \\
&\quad [w \text{ is a lower bound of } \{F^*(q_1), F^*(q_2)\} \text{ and} \\
&\quad\ F^*(q_1) \wedge_1 F^*(q_2) \text{ is the greatest lower bound}]
\end{aligned}$$

Proof of property 6. Since $(L_1, \leq_1)$ is a complete lattice, the greatest lower bound $\bigvee\{a \in L_1 \mid F(a) \leq_2 b\}$ is an element of L_1, and this holds for arbitrary $b \in L_2$. Therefore the expression

$$\forall b \in L_2 : G(b) = \bigvee\{a \in L_1 \mid F(a) \leq_2 b\}$$

introduces a well-defined function. To show that $G = F^*$, we observe that trivially, $F^*(b) \leq_1 F^*(b)$, and therefore $F^*(b) \in \{a \in L_1 \mid a \leq_1 F^*(b)\}$ which implies $F^*(b) \leq_1 \bigvee\{a \in L_1 \mid a \leq_1 F^*(b)\} = \bigvee\{a \in L_1 \mid F(a) \leq_2 b\} = G(b)$. Conversely, we calculate

$$\begin{aligned}
G(b) \quad &= \quad \bigvee\{a \in L_1 \mid F(a) \leq_2 b\} \\
&\quad [\text{Definition of } G] \\
&= \quad \bigvee\{a \in L_1 \mid a \leq_1 F^*(b)\} \\
&\quad [\text{Defining property of GCs}] \\
&\leq_1 \quad F^*(b) \\
&\quad [\text{Property of supremum}]
\end{aligned}$$

This proves that $G(b) = F^*(b)$. Since b had been an arbitrary element of L_2, $G = F^*$ follows.

Proof of property 7. Since $(L_2, \leq_2)$ is a complete lattice, the function

$$\forall a \in L_1 : H(a) = \bigwedge \{b \in L_2 \mid a \leq_1 F^*(b)\}$$

is well-defined, and it remains to show that $F = H$. Given $a \in L_1$, $F(a) \leq_2 F(a)$ implies $F(a) \in \{b \in L_2 \mid F(a) \leq_2 b\}$, and therefore $\bigwedge\{b \in L_2 \mid F(a) \leq_2 b\} \leq_2 F(a)$ which implies $H(a) = \bigwedge\{b \in L_2 \mid a \leq_1 F^*(b)\} = \bigwedge\{b \in L_2 \mid F(a) \leq_2 b\} \leq_2 F(a)$. Conversely,

$$\begin{aligned} H(a) &= \bigwedge \{b \in L_2 \mid a \leq_1 F^*(b)\} \\ &\quad [\text{Definition of } H] \\ &= \bigwedge \{b \in L_2 \mid F(a) \leq_2 b\} \\ &\quad [\text{Defining property of GCs}] \\ &\geq_2 F(a) \\ &\quad [\text{Property of infimum}] \end{aligned}$$

□

Example 25. Let $F : \mathbb{P}(\mathbb{R}) \to \mathbb{IR}$, $F^* : \mathbb{IR} \to \mathbb{P}(\mathbb{R})$ be the GC between the power set lattice of real numbers and the lattice of intervals over the real numbers. Then (recall that the join function in $\mathbb{IR}$ is equal to the interval hull $\sqcup$)

$$\begin{aligned} F(\{1,4,6\} \cup \{2,3.5,4\}) &= F(\{1,2,3.5,4,6\}) \\ &= [1,6] \\ &= [1,6] \sqcup [2,4] \\ &= F(\{1,4,6\}) \sqcup F(\{2,3.5,4\}) \end{aligned}$$

as expected according to Lemma 7 (4). However, when calculating the meet of these sets (recall that the meet function $\sqcap$ in $\mathbb{IR}$ is equal to the set intersection) results in

$$\begin{aligned} F(\{1,4,6\} \cap \{2,3.5,4\}) &= F(\{4\}) \\ &= [4,4] \\ &\neq [2,4] \\ &= [1,6] \sqcap [2,4] \\ &= F(\{1,4,6\}) \sqcap F(\{2,3.5,4\}) \end{aligned}$$

□

Any surjective mapping $f : X \to Y$ induces a *natural GC* between its associated power set lattices $(\mathbb{P}(X), \subseteq) \underset{F}{\overset{F^*}{\leftrightarrows}} (\mathbb{P}(Y), \subseteq)$ by defining

$$F : \mathbb{P}(X) \to \mathbb{P}(Y); \quad F(A) = \{f(a) \mid a \in A\} \tag{B.1}$$

and using property 5 above to construct

$$F^* : \mathbb{P}(Y) \to \mathbb{P}(X); \quad F^*(B) = \bigcup \{A \in \mathbb{P}(X) \mid F(A) \subseteq B\}$$

which is simplified to

$$F^*(B) = \{a \in X \mid f(a) \in B\} \tag{B.2}$$

Lemma 8 *For any surjective mappings* $f : X \to Y$, $g : Y \to Z$, $h : X \to Z$ *with* $h = gf$, *the induced natural GCs satisfy*

$$H = GF, H^* = F^*G^*$$

Proof. Let $U \subseteq X$. $GF(U) = G(\{f(u) \mid u \in U\}) = \{g(f(u)) \mid u \in U\} = \{h(u) \mid u \in U\} = H(U)$. Now let $V \subseteq Z$. $F^*G^*(V) = \{x \in X \mid f(x) \in G^*(V)\} = \{x \in X \mid gf(x) \in V\} = \{x \in X \mid h(x) \in V\} = H^*(V)$. □

The following technical property of GC between power set lattices will be needed later for characterising simulation relations.

Lemma 9 *Any GC* $(\mathbb{P}(X), \subseteq) \underset{F}{\overset{F^*}{\leftrightarrows}} (\mathbb{P}(Y), \subseteq)$ *between power set lattices maps singleton sets to singleton sets.*

Proof. Let $A = \{a\} \in \mathbb{P}(X)$ a singleton set and suppose that $F(A) = B \in \mathbb{P}(Y)$ with $|B| > 1$. Then we can find a proper subset $\varnothing \neq B' \subset B$ of B.

Suppose that there existed an $A' \in \mathbb{P}(X)$ such that $F(A') = B'$. Then $A' \neq \varnothing$, because otherwise $G(A') = \varnothing$, since G is a homomorphism, and this contradicted $B' \neq \varnothing$. The lattice homomorphism properties imply further that $F(A \cap A') = F(A) \cap F(A') = B \cap B' = B'$. Since A is a singleton

set we either have $A \cap A' = \varnothing$ or $A \cap A' = A$. In the first case this would imply $F(A \cap A') = F(\varnothing) = \varnothing$, a contradiction to $F(A \cap A') = B' \neq \varnothing$. In the second case $A \cap A' = A$ we calculate $F(A \cap A') = F(A) = B$, which is a contradiction to $F(A \cap A') = B' \neq B$. From these contradictions we conclude that there exists no $A' \in \mathbb{P}(X)$ such that $F(A') = B'$. As a consequence, $F^*(B') = \varnothing$ for any proper subset $B' \subset B$.

According to our assumption $|B| > 1$, we can write $B = B_1 \cup B_2$ such that $B_1, B_2 \neq \varnothing \wedge B_1, B_2 \neq B \wedge B_1 \neq B_2$. Then we conclude using the characteristic GC property that $A \subseteq F^*(B) = F^*(B_1) \cup F^*(B_2) = \varnothing$.

As a consequence, the original assumption $|B| > 1$ leads to a contradiction, and this proves the theorem. □

Exercise 21. Construct a GC between the power set lattice over $\mathbb{R}$ and the interval lattice over $\mathbb{R}$. Using this example, explain intuitively what it means that the left-hand side lattice (domain of the lower adjoint) contains more fine-grained information than the right-hand side lattice (domain of the upper adjoint), so that it is justified to call the right-hand side lattice an abstraction of the left-hand side lattice. □

B.3 Kripke Structures and Galois Connections

A KS $K = (S, S_0, R, L, AP)$ is associated with two power set lattices in a natural way.

- The set of states S induces $(\mathbb{P}(S), \subseteq)$.
- The atomic propositions AP induce $(\mathcal{L}(AP), \subseteq)$ with[1] $\mathcal{L}(AP) = \mathbb{P}(L(S))$.

An element e of $\mathcal{L}(AP)$ is a set of sets of atomic propositions: by definition, we can find states $s_1, \ldots, s_n$ such that $e = \{L(s_1), \ldots, L(s_n)\}$

The labelling function $L : S \to L(S);\ s \mapsto L(s)$ induces the natural GC

$$(\mathbb{P}(S), \subseteq) \underset{L}{\overset{L^*}{\leftrightarrows}} (\mathcal{L}(AP), \subseteq)$$

[1] $L(S)$ denotes the image of the labelling function, that is, $L(S) = \{L(s) \mid s \in S\}$

as described above, so for $U \subseteq S$ and $Z_1, \ldots, Z_k \in L(S)$

$$
\begin{aligned}
&L : \mathbb{P}(S) \to \mathcal{L}(AP) \\
&L(U) = \{L(s) \mid s \in U\} \\
&L^* : \mathcal{L}(AP) \to \mathbb{P}(S) \\
&L^*(\{Z_1, \ldots, Z_k\}) = \textstyle\bigcup_{i=1}^{k} \{s \in S \mid L(s) = Z_i\}
\end{aligned}
$$

Note that with a slight abuse of notation, the lower adjoint of this GC is again denoted by L.

Remarks. Lattice $(\mathcal{L}(AP), \subseteq)$ is isomorphic to the lattice $(\mathcal{B}(AP), \Rightarrow)$ of Boolean propositions over atomic propositions from $AP = \{p_1, \ldots, p_{|AP|}\}$, where all formulas that do not have a model $s : V \to D$ in S are identified with $\bot = \texttt{false}$. Every formula $\varphi \in \mathcal{B}(AP)$ can be represented in disjunctive normal form (DNF) as $\varphi = \bigvee_{i=1}^{k} (\varepsilon_1^i(p_1) \wedge \cdots \wedge \varepsilon_{|AP|}^i(p_{|AP|}))$, where k is the number of minterms needed for representing φ in DNF, and each $(\varepsilon_i^1(p_1) \wedge \cdots \wedge \varepsilon_i^{|AP|}(p_{|AP|}))$ denotes such a minterm. The p_ℓ are the atomic propositions in AP, and $\varepsilon_\ell^i(p_\ell)$ is equal to p_ℓ or its complement $\neg p_\ell$. Each minterm can be encoded by a label $Z = L(s)$ for some $s \in S$, such that $\varepsilon_\ell^i(p_\ell) = p_\ell$ if $p \in Z$ and $\varepsilon_\ell^i(p_\ell) = \neg p_\ell$ if $p_\ell \in AP - Z$. State s is a model for φ, since minterm number i of φ evaluates to $\texttt{true}$ in s. The complete set of models for φ is the set of states $L^*(\{Z_1, \ldots, Z_k\})$, where each Z_i encodes the minterm $(\varepsilon_1^i(p_1) \wedge \cdots \wedge \varepsilon_{|AP|}^i(p_{|AP|}))$. Lattice $(\mathcal{L}(AP), \subseteq)$ is also isomorphic to a sub-lattice of the power set lattice $(\mathbb{P}(2^{|AP|}), \subseteq) = (2^{2^{|AP|}}, \subseteq)$. The isomorphism maps each singleton set $\{Z\} \in \mathcal{L}(AP)$ to $\{z\}$, where the i^{th} bit $z(i)$ of bit vector $z \in 2^{|AP|}$ equals 1 if and only if $p_i \in Z$.

Appendix C

Data Abstraction

C.1 Abstractions and Refinements of Kripke Structures Without Change of Variables

Let us now consider two KS with identical state space, initial states, and transition relation, but with different sets of atomic propositions.

Definition 12 *KS* $K = (S, S_0, R, L, AP)$ *is called a* refinement *of* $K' = (S, S_0, R, L', AP')$*, and* K' *an* abstraction *of* K*, if and only if the function*

$$g : L(S) \to L'(S);\ g(L(s)) = L'(s)$$

is well-defined.

Observe that g is well-defined if and only if $L(s) = L(r)$ implies $L'(s) = L'(r)$ for all $s, r \in S$. Hence, if K is a refinement of K', this implies that for any $s \in S$: $\{r \in S \mid L(r) = L(s)\} \subseteq \{r \in S \mid L'(r) = L'(s)\}$. Moreover, if g is well-defined, it is also surjective by construction, and $gL = L'$.

The following lemma shows that the refinement property specified in Definition 12 is equivalent to the existence of a GC completing the commutative triangle diagram shown in Figure C.1.

Lemma 10 *Let* $K = (S, S_0, R, L, AP)$ *and* $K' = (S, S_0, R, L', AP')$ *be KS with natural GCs*

$$\mathbf{P}(S) \underset{L}{\overset{L^*}{\leftrightarrows}} \mathcal{L}(AP) \quad \textit{and} \quad \mathbf{P}(S) \underset{L'}{\overset{L'^*}{\leftrightarrows}} \mathcal{L}(AP')$$

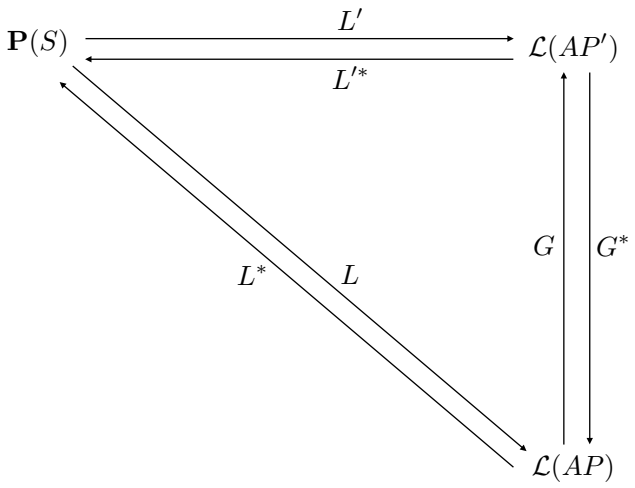


Figure C.1: GC $\mathcal{L}(\mathrm{AP}) \underset{\mathrm{G}}{\overset{\mathrm{G}^*}{\leftrightarrows}} \mathcal{L}(\mathrm{AP}')$ completing the commutative triangle.

as introduced above. Then the following statements are equivalent.

1. K *is a refinement of* K'.
2. *There exists a GC*
$$\mathcal{L}(\mathrm{AP}) \underset{\mathrm{G}}{\overset{\mathrm{G}^*}{\leftrightarrows}} \mathcal{L}(\mathrm{AP}')$$
satisfying $\mathrm{L}' = \mathrm{GL}$ *and* $\mathrm{L}'^* = \mathrm{L}^*\mathrm{G}^*$.

Proof. Let K be a refinement of K' with associated function g according to Definition 12. By Definition 12, g is surjective. Moreover, the labelling functions $\mathrm{L} : \mathrm{S} \to \mathrm{L}(\mathrm{S})$ and $\mathrm{L}' : \mathrm{S} \to \mathrm{L}(\mathrm{S}')$ are surjective. By the specification of g given in Definition 12, $\mathrm{gL} = \mathrm{L}'$. Choose $\mathcal{L}(\mathrm{AP}) \underset{\mathrm{G}}{\overset{\mathrm{G}^*}{\leftrightarrows}} \mathcal{L}(\mathrm{AP}')$ to be the natural GC associated with g. Then we can apply Lemma 8 to conclude that $\mathrm{L}' = \mathrm{GL}$ and $\mathrm{L}'^* = \mathrm{L}^*\mathrm{G}^*$.

Now assume that an *arbitrary* GC $\mathcal{L}(\mathrm{AP}) \underset{\mathrm{G}}{\overset{\mathrm{G}^*}{\leftrightarrows}} \mathcal{L}(\mathrm{AP}')$ fulfils $\mathrm{L}' = \mathrm{GL}$ and $\mathrm{L}'^* = \mathrm{L}^*\mathrm{G}^*$. Define

$$\begin{aligned} &\mathrm{g} : \mathrm{L}(\mathrm{S}) \to \mathrm{L}'(\mathrm{S}); \\ &\mathrm{g}(\mathrm{L}(\mathrm{s})) = \mathrm{L}'(\mathrm{s}) \Leftrightarrow \mathrm{G}(\{\mathrm{L}(\mathrm{s})\}) = \{\mathrm{L}'(\mathrm{s})\} \end{aligned}$$

It remains to show that $G(\{L(s)\}) = \{L'(s)\}$ for all $s \in S$. To this end, we calculate

$$\begin{aligned} \{L'(s)\} &= L'(\{s\}) \qquad [\text{Property of GC } \mathbf{P}(S) \underset{L'}{\overset{L'^*}{\leftrightarrows}} \mathcal{L}(AP')] \\ &= G(L(\{s\})) \quad [\text{Assumption } L' = GL] \\ &= G(\{L(s)\}) \quad [\text{Definition of } L] \end{aligned}$$

This shows that g is well-defined and that the GC fulfilling assumption 2 of the lemma is in fact the natural GC associated with g. This completes the proof of the lemma. □

A simple form of refinement for an KS consists in successively adding atomic propositions with free variables in V.

Lemma 11 *Given a KS* $K' = (S, S_0, R', L', AP')$*, Let* $AP = AP' \cup \{p\}$ *such that* p *is a new atomic proposition with free variables from* V *which is not contained in* AP*. Then* $K = (S, S_0, R', L, AP)$ *refines* $K' = (S, S_0, R', L', AP')$.

Proof. Function

$$g : L(S) \to L'(S);\ g(L(s)) = L'(s)$$

is well-defined, because $L'(s) = L(s) \setminus \{p\}$ for all $s \in S$. □

C.2 Refinements of Kripke Structures With Change of Variables

Let us now consider the more general case where two KS with *different* state spaces are related to each other:

- $K = (S, S_0, R, L, AP)$ with $S = V \to D$, such that AP consists of atomic propositions over free variables from V.
- $K' = (S', S'_0, R', L', AP')$ with $S' = V' \to D'$, such that AP' consists of atomic propositions over free variables from V'.

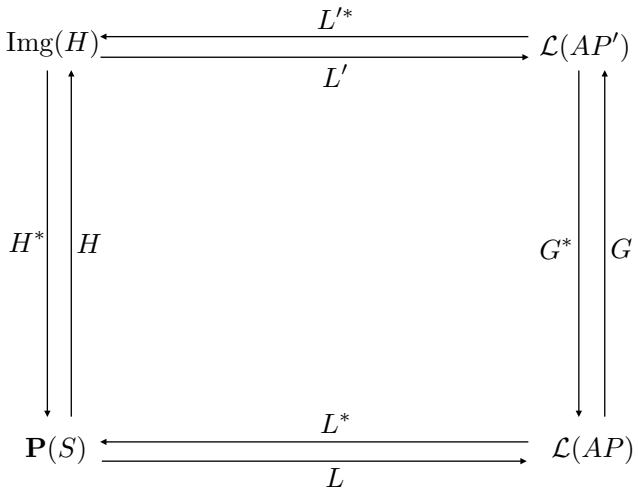


Figure C.2: GC $\mathcal{L}(AP) \underset{G}{\overset{G^*}{\leftrightarrows}} \mathcal{L}(AP')$ completing the commutative rectangle.

Definition 13 (Simulation) *With the notation introduced above, we say that* K' *is a* simulation *of* K*, or* K *is a* refinement *of* K'*, and write* $K \preccurlyeq K'$*, if and only if there exists a relation* $H \subseteq S \times S'$ *and a mapping* $g : L(S) \to L'(S')$*, such the following conditions hold.*

1. $\forall s \in S_0 : \exists s' \in S'_0 : H(s, s')$
2. $\forall (s_1, s'_1) \in H, s_2 \in S : R(s_1, s_2) \Rightarrow \exists s'_2 \in S' : R'(s'_1, s'_2) \wedge H(s_2, s'_2)$
3. $\forall (s, s') \in H : g(L(s)) = L'(s')$

Lemma 12 *If* K *is simulated by* K' *with simulation relation* H *and all elements of the state space* S *are reachable, then* H *is* total *on* S *in the sense that*

$$\forall s \in S : \exists s' \in S' : H(s, s')$$

Proof. Let $s \in S$. Since s is reachable, there exists a trace $\pi \in S^*$ and $i \geq 0$ such that $\pi(0) \in S_0$ and $\pi(i) = s$ and $R(\pi(j), \pi(j+1))$ for $j = 0, \ldots, i-1$. From simulation condition (i) in Definition 13 we know that there exists a $\pi'(0) \in S'_0$ such that $H(\pi(0), \pi'(0))$. Applying simulation

condition (ii) successively to $\pi(0), \pi(1), \ldots, \pi(i)$ we conclude that there exists states $\pi'(0), \pi'(1), \ldots, \pi'(i)$ such that $H(\pi(j), \pi'(j))$ for $j = 0, \ldots, i$. This shows $H(s, \pi'(i))$ and completes the proof. □

A simulation relation $H \subseteq S \times S'$ induces a *function* between the power set lattices. This function is again denoted by H and defined by

$$H : \mathbb{P}(S) \to \mathbb{P}(S'); \quad X \mapsto \{s' \in S' \mid \exists s \in X : H(s, s')\}$$

Let

$$\mathrm{Img}(H) = \{X' \in \mathbb{P}(S') \mid \exists X \in \mathbb{P}(S) : H(X) = X'\} \subseteq \mathbb{P}(S')$$

denote the image of the function H. The dual function H^* is defined by

$$H^* : \mathrm{Img}(H) \to \mathbb{P}(S); \quad X' \mapsto \{s \in S \mid \exists s' \in X' : H(s, s')\}$$

In analogy to Fig. C.1, simulations introduce a commuting diagram of Galois connections, as shown in Fig. C.2.

Lemma 13 *Let* $K = (S, S_0, R, L, AP)$ $K' = (S', S'_0, R', L', AP')$ *be KS with natural GCs*

$$\mathbf{P}(S) \underset{L}{\overset{L^*}{\leftrightarrows}} \mathcal{L}(AP) \quad \textit{and} \quad \mathbf{P}(S) \underset{L'}{\overset{L'^*}{\leftrightarrows}} \mathcal{L}(AP')$$

as introduced above, such that all states in S *are reachable. Then the following statements are equivalent.*

1. K *is simulated by* K' *with simulation relation* H *and mapping* $g : L(S) \to L'(S')$ *as given in Definition 13.*

2. *There exists a relation* $H \subseteq S \times S'$ *satisfying (i) and (ii) from Definition 13 and a GC*

$$\mathcal{L}(AP) \underset{G}{\overset{G^*}{\leftrightarrows}} \mathcal{L}(AP')$$

such that the rectangle presented in Fig. C.2 commutes in the sense that

(a) $GL = L'H$

(b) $H^*L'^* = L^*G^*$

Proof. Suppose that (i) holds. Let $\mathcal{L}(AP) \underset{G}{\overset{G^*}{\leftrightarrows}} \mathcal{L}(AP')$ be the natural GC associated with g. Given $X \in \mathbb{P}(S)$, we calculate

$$\begin{aligned} GL(X) &= G(\{L(s) \mid s \in X\}) \\ &= \{g(L(s)) \mid s \in X\} \\ &= \{L'(s') \mid s' \in H(X)\} \\ &= L'(H(X)) \end{aligned}$$

This proves property (a). Now let $Z' = \{L'(s'_1), \ldots, L'(s'_n)\} \in \mathcal{L}'(AP')$, and suppose that $H(s_1, s'_1), \ldots, H(s_n, s'_n)$. Then

$$\begin{aligned} H^*L'^*(Z') &= H^*L'^*(\{L'(s'_1), \ldots, L'(s'_n)\}) \\ &= H^*(\{s' \in S' \mid L'(s') \in \{L'(s'_1), \ldots, L'(s'_n)\}\}) \\ &= H^*(\{s' \in S' \mid L'(s') \in \{g(L(s_1)), \ldots, g(L(s_n))\}\}) \\ &= \{s \in S \mid \exists s' \in S' : H(s, s') \wedge g(L(s)) = L'(s') \wedge \\ &\qquad\qquad L'(s') \in \{g(L(s_1)), \ldots, g(L(s_n))\}\} \\ &= \{s \in S \mid g(L(s)) \in \{g(L(s_1)), \ldots, g(L(s_n))\}\} \\ &= \{s \in S \mid L(s) \in G^*(\{g(L(s_1)), \ldots, g(L(s_n))\})\} \\ &= \{s \in S \mid L(s) \in G^*(Z')\})\} \\ &= L^*G^*(Z') \end{aligned}$$

This proves property (b).

Now suppose that (ii) holds. Define

$$g : L(S) \to L'(S'); \quad g(L(s)) = Z' \text{ if and only if } G(\{L(s)\}) = \{Z'\}$$

Then g is well-defined because G is well-defined and maps singleton sets to singleton sets (Lemma 9). To prove that g fulfils the requirement (iii) for a simulation according to Definition 13, we prove that $H(s, s'_1) \wedge H(s, s'_2)$

implies $L'(s'_1) = L'(s'_2)$. To this end, we calculate

$$\begin{aligned} L'(\{s' \in S' \mid H(s, s')\}) &= L'H(\{s\}) \\ &= GL(\{s\}) \\ &= G(\{L(s)\}) \\ &= \{g(L(s))\} \end{aligned}$$

This completes the proof. □

Exercise 22. In the proof of Lemma 13, annotate each equality (like $GL(X) = G(\{L(s) \mid s \in X\}) = \dots$) with the reason why this equality holds. □

C.3 Translation of Temporal Formulas Between Kripke Structures and Their Simulations

The commutativity of the diagram from Fig. C.2 allows us to translate temporal logic formulas over K to formulas over K' and vice versa in a natural way. Given a temporal logic formula φ over K, use the following recursive translation algorithm F.

1. $F(\texttt{true}) = \texttt{true}$, $F(\texttt{false}) = \texttt{false}$
2. If φ is a state formula that does not contain any temporal operators or path quantifiers, then $F(\varphi) = G(DNF(\varphi))$
3. If $\varphi = \mathbf{A}\psi$ then $F(\varphi) = \mathbf{A}F(\psi)$.
4. If $\varphi = \mathbf{E}\psi$ then $F(\varphi) = \mathbf{E}F(\psi)$.
5. If $\varphi = \mathbf{F}\psi$ then $F(\varphi) = \mathbf{F}F(\psi)$.
6. If $\varphi = \mathbf{G}\psi$ then $F(\varphi) = \mathbf{G}F(\psi)$.
7. If $\varphi = \mathbf{X}\psi$ then $F(\varphi) = \mathbf{X}F(\psi)$.

8. If $\varphi = \varphi_1 \mathbf{U} \varphi_2$ then $F(\varphi) = F(\varphi_1)\mathbf{U}F(\varphi_2)$.
9. If $\varphi = \varphi_1 \vee \varphi_2$ then $F(\varphi) = F(\varphi_1) \vee F(\varphi_2)$.
10. If $\varphi = \varphi_1 \wedge \varphi_2$ then $F(\varphi) = F(\varphi_1) \wedge F(\varphi_2)$.
11. If $\varphi = \neg\psi$ then $F(\varphi) = \neg F(\psi)$.

Rule 2 means that φ is first transformed into disjunctive normal form, and then each of its minterms is represented by the corresponding set of atomic propositions that occur with positive sign in the minterm. The φ can be expressed as an element of $\mathcal{L}(AP)$, so that it can be transformed by G into an element of $\mathcal{L}'(AP')$, which can in turn be replaced by a formula φ' in DNF.

When translating temporal formulas from K' to K, we proceed in an analogous fashion with the following algorithm F'.

Observe that being able to translate formulas between K and K' and vice versa does not necessarily imply that $F(\varphi)$ holds in K', if φ holds in K and vice versa. This is further clarified by the theorem below.

1. $F'(\texttt{true}) = \texttt{true}$, $F'(\texttt{false}) = \texttt{false}$
2. If φ is a state formula that does not contain any temporal operators or path quantifiers, then $F'(\varphi) = G^*(\mathrm{DNF}(\varphi))$
3. If $\varphi = \mathbf{A}\psi$ then $F'(\varphi) = \mathbf{A}F'(\psi)$.
4. If $\varphi = \mathbf{E}\psi$ then $F'(\varphi) = \mathbf{E}F'(\psi)$.
5. If $\varphi = \mathbf{F}\psi$ then $F'(\varphi) = \mathbf{F}F'(\psi)$.
6. If $\varphi = \mathbf{G}\psi$ then $F'(\varphi) = \mathbf{G}F'(\psi)$.
7. If $\varphi = \mathbf{X}\psi$ then $F'(\varphi) = \mathbf{X}F'(\psi)$.
8. If $\varphi = \varphi_1 \mathbf{U} \varphi_2$ then $F'(\varphi) = F'(\varphi_1)\mathbf{U}F'(\varphi_2)$.
9. If $\varphi = \varphi_1 \vee \varphi_2$ then $F'(\varphi) = F'(\varphi_1) \vee F'(\varphi_2)$.
10. If $\varphi = \varphi_1 \wedge \varphi_2$ then $F'(\varphi) = F'(\varphi_1) \wedge F'(\varphi_2)$.
11. If $\varphi = \neg\psi$ then $F'(\varphi) = \neg F'(\psi)$.

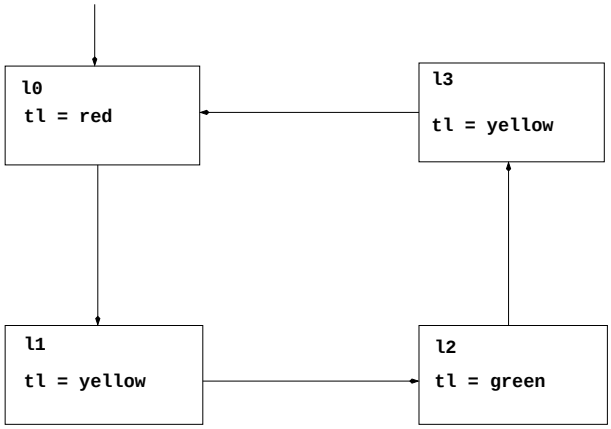


Figure C.3: Kripke structure of traffic light controller from Example 26.

Example 26. Consider the Kripke Structure depicted in Fig. C.3, which is associated with a specification model of a traffic light controller. Formalising this as a KS results in

$$
\begin{aligned}
K &= (S, S_0, R, L, AP) \\
V &= \{c, tl\} \\
S &= \{\ell_i : V \to D \mid i = 0, 1, 2, 3\} \\
\ell_0 &= \{c \mapsto 0, tl \mapsto \text{red}\} \\
\ell_1 &= \{c \mapsto 1, tl \mapsto \text{yellow}\} \\
\ell_2 &= \{c \mapsto 2, tl \mapsto \text{green}\} \\
\ell_3 &= \{c \mapsto 3, tl \mapsto \text{yellow}\} \\
S_0 &= \{\ell_0\} \\
D &= \{0, 1, 2, 3\} \cup \{\text{red}, \text{yellow}, \text{green}\} \\
R &= \{(\ell_0, \ell_1), (\ell_1, \ell_2), (\ell_2, \ell_3), (\ell_3, \ell_0)\} \\
AP &= \{tl = \text{red}, tl = \text{yellow}, tl = \text{green}\} \\
L &= \{\ell_0 \mapsto \{tl = \text{red}\}, \ell_1 \mapsto \{tl = \text{yellow}\}, \\
&\qquad \ell_2 \mapsto \{tl = \text{green}\}, \ell_3 \mapsto \{tl = \text{yellow}\}\}
\end{aligned}
$$

As is well known to every law-abiding citizen, we always stop our cars

on red *and* on yellow. Therefore, if we are only interested in knowing when cars are in a halt-state in front of the traffic light, it makes sense to introduce an abstracted KS as the one depicted in Figure C.4. Formally, this is described by

$$\begin{aligned}
K' &= (S', S'_0, R', L', AP') \\
S' &= \{m_i : V' \to D' \mid i = 0, 1\} \\
S'_0 &= \{m_0\} \\
V' &= \{\text{stops}\} \\
D' &= \mathbb{B} = \{0, 1\} \\
R' &= \{(m_0, m_0), (m_0, m_1), (m_1, m_0)\} \\
AP' &= \{\text{stops}\} \\
L' &= \{m_0 \mapsto \{\text{stops}\}, m_1 \mapsto \varnothing\}
\end{aligned}$$

In order to show that K' simulates K according to Definition 13 we proceed as follows. As simulation relation we choose

$$H = \{(\ell_0, m_0), (\ell_1, m_0), (\ell_3, m_0), (\ell_2, m_1), \}$$

and define the mapping g by

$$g : L(S) \to L'(S'); \ \{tl = \text{red}\} \mapsto \{\text{stops}\}, \{tl = \text{yellow}\} \mapsto \{\text{stops}\}, \{tl = \text{green}\} \mapsto \varnothing$$

It is easy to check that H, g fulfil the conditions (i — iii) of Definition 13.

Now suppose we wish to prove that $\mathbf{EF}(tl = \text{green})$ holds for the Kripke structure of the original model in Fig. C.3. The assertion can be readily expressed on abstract level using the algorithm F given above:

$$F(\mathbf{EF}(tl = \text{green})) = \mathbf{EF}\neg\text{stops}$$

Formula $\mathbf{EF}(\neg\text{stops})$ obviously holds on abstract level, since there exists a path in Fig. C.4 that starts in m_0 and visits m_1. Similarly, the concrete condition $\mathbf{AF}(tl = \text{red} \vee tl = \text{yellow})$ can be expressed in an abstract way as

$$F(\mathbf{AF}(tl = \text{red} \vee tl = \text{yellow})) = \mathbf{AF}\text{stops}$$

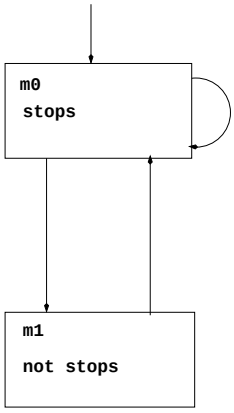


Figure C.4: Abstracted Kripke structure induced by auxiliary variable stops in Example 26.

It is easy to see that it holds on abstract level.

In these special cases, the assertions also hold on concrete level, but this is not always the case: On abstracted level we can also prove the formula $\mathbf{EG}(\text{stops})$ which has the concrete complement

$$F'(\mathbf{EG}(\text{stops})) = \mathbf{EG}(\text{tl} = \text{red} \vee \text{tl} = \text{yellow})$$

The latter formula does obviously not hold in the concrete model.

Conversely, the concrete model satisfies $\mathbf{AF}(\text{tl} = \text{green})$, while the corresponding formula

$$F(\mathbf{AF}(\text{tl} = \text{green})) = \mathbf{AF}(\neg\text{stop})$$

is not fulfilled on abstract level. □

Exercise 23. In [5] the concept of simulations is defined as follows.

Definition 14 (Simulation according to [5]) *Given two Kripke structures* $K = (S, S_0, R, L), K' = (S', S_0', R', L')$ *such that* K *refers to atomic propositions* AP *and* K' *refers to atomic propositions* AP' *and* $AP' \subseteq AP$. *The relation* $H \subseteq S \times S'$ *is called a* simulation, *if the following conditions hold for all* $(s, s') \in H$*:*

1. $L(s) \cap AP' = L'(s')$

2. $\forall s_1 \in S : R(s, s_1) \Rightarrow \exists s_1' \in S' : R'(s', s_1') \wedge H(s_1, s_1')$

We write $K \preccurlyeq K'$ (K is simulated by K') *if such a simulation* H *exists and*

$$\forall s_0 \in S_0 : \exists s_0' \in S_0' : H(s_0, s_0')$$

□

1. Explain informally why the definition from [5] is less general than our Definition 13 for the case that S, S' contain variable valuations $s : V \to D$, $s' : V' \to D'$, respectively. (Hint: think of the nature of the atomic propositions, if states are variable valuations.)

2. Prove that if K is simulated by K' according to Definition 14, it is also simulated by K' in the sense of our Definition 13.

□

C.4 Property Preservation for ACTL* Formulas

Definition 15 *Let* $K \preccurlyeq K'$ *with simulation relation* $H \subset S \times S'$ *and* $H(s, s')$. *Suppose* π *is a path in* K *starting at* s *and* π' *a path starting at* s' *in* K'. *We say that* π and π' correspond to each other *if*

$$\forall i \geq 0 : H(\pi(i), \pi'(i))$$

□

Lemma 14 *Let* $K \preccurlyeq K'$ *with simulation relation* $H \subset S \times S'$ *and* $H(s, s')$. *Then for every path* π *in* K *starting at* s *there is a corresponding path* π' *in* K' *starting at* s'.

Proof. Since π is a path starting at s,

$$\pi(0) = s \wedge (\forall i \geq 0 : R(\pi(i), \pi(i+1)))$$

follows. Since $s = \pi(0)$ and $H(s, s')$, this implies $H(\pi(0), s')$. Applying condition (ii) of Definition 13 successively on $\pi(0), \pi(1), \pi(2), \ldots$ this yields the existence of states $\pi'(i) \in S', i \geq 0$, such that

$$\pi'(0) = s' \wedge (\forall i \geq 0 : R'(\pi'(i), \pi'(i+1)) \wedge H(\pi(i+1), \pi'(i+1))),$$

so π' is a path in K', and it corresponds to π by construction. □

Lemma 15 *Assume* $K \preccurlyeq K'$ *according to Definition 13. Let* π, π' *be corresponding paths in* K *and* K'*, respectively, emanating from* $s = \pi(0)$ *and* $s' = \pi'(0)$ *with* $H(s, s')$*. Let* ψ' *be a CTL* path formula without path quantifiers, such that* $\pi' \models \psi'$*. Then* $\pi \models F'(\psi')$*.*

Proof. The proof is performed by structural induction over the formula ψ'. Since ψ' is a path formula not containing any path quantifiers, we can assume that it is represented in positive normal form, as specified for LTL formulas in Section 3.1. As a consequence, it suffices to perform the structural induction over $\vee, \wedge, \mathbf{X}, \mathbf{U}, \mathbf{W}$.

Step 1. Let ψ' be a proposition in negation normal form, that is, a state formula without path operators and without path quantifiers, where negation only occurs in front of atomic propositions. Then $\pi' \models \psi'$ is equivalent to $s' \models \psi$. This implies that $L'(\{s'\}) \Rightarrow \psi'$. Since $H(s, s')$, Lemma 13 yields $L(\{s\}) \Rightarrow G^*(DNF(\psi')) = F'(\psi')$. As a consequence, $s \models F'(\psi')$, and, since $F'(\psi')$ is also a state formula without path operators, this implies $\pi \models F'(\psi')$.

Step 2. Suppose that ψ'_0 is *any* path formula, such that $F'(\psi'_0)$ holds on any path π if ψ'_0 holds on a corresponding path π'. Suppose additionally that $\mathbf{X}\psi'_0$ is fulfilled on π'. This means that $\pi'^1 \models \psi'_0$. Since we are dealing with corresponding paths, $H(\pi(1), \pi'(1))$ is fulfilled, and also π^1 and π'^1 are corresponding paths. Applying the premise about ψ'_0, we can conclude that $\pi^1 \models F'(\psi'_0)$. Therefore $\pi \models \mathbf{X}F'(\psi'_0) = F'(\mathbf{X}\psi'_0)$.

Step 3. Suppose that ψ'_0, ψ'_1 are any path formulas, such that $F'(\psi'_i), i = 0, 1$ holds on any path π if $\psi'_i, i = 0, 1$ holds on a corresponding path π'. Suppose additionally that $\psi'_0 \mathbf{U} \psi'_1$ is fulfilled on π'. This is equivalent to

$$\exists i \geq 0 : \left(\pi'^i \models \psi'_1 \wedge (\forall 0 \leq j < i : \pi'^j \models \psi'_0)\right)$$

Since all π^i, π'^i and π^j, π'^j are corresponding paths, our assumptions imply

$$\exists i \geq 0 : \left(\pi^i \models F'(\psi_1') \wedge (\forall 0 \leq j < i : \pi^j \models F'(\psi_0'))\right)$$

and this is equivalent to $\pi \models F'(\psi_0')\mathbf{U}F'(\psi_1') = F'(\psi_0'\mathbf{U}\psi_1')$.

In analogy, one shows the induction steps for $\mathbf{W}, \wedge, \vee$, and this completes the proof. □

Theorem 14 *Assume* $K \preccurlyeq K'$ *according to Definition 13. Then for every ACTL* formula* ϕ' *with atomic propositions in* AP'

$$(K' \models \phi') \textit{ implies } (K \models F'(\phi'))$$

Proof. The proof is performed by structural induction over the usage of the **A**-quantifier. We show a slightly stronger property than the one required according to the theorem:

$$\forall (s, s') \in H : \left(s' \models \phi' \Rightarrow s \models F'(\phi')\right)$$

Step 1. Suppose that $\phi' = \mathbf{A}\phi_0'$, such that ϕ_0' is a quantifier-free path formula, that is, it does not contain the **A** quantifier again. Suppose that $\phi' = \mathbf{A}\phi_0'$ is fulfilled in $s' \in S'$. Assume that s is related to s' by $H(s, s')$. Take any path π starting in $\pi(0) = s$, and let π' be a corresponding path starting in s'. Since $\phi' = \mathbf{A}\phi_0'$ and is fulfilled in s', we conclude that $\pi' \models \phi_0$. According to our premise, ϕ_0 does not contain **A**, so it is a path formula. Then Lemma 15 implies that $\pi \models F'(\phi_0')$. Since π was an arbitrary path starting in s, this implies $s \models \mathbf{A}F'(\phi_0') = F'(\mathbf{A}\phi_0')$.

Step 2. Suppose that ϕ_0' is an ACTL* formula such that $s' \models \phi_0'$ implies $s \models F'(\phi_0')$ for any s, s' related by H. Suppose further that $s' \models \mathbf{AX}\phi_0'$. This is equivalent to $\pi' \models \mathbf{X}\phi_0'$ on every path emanating from s', which is in turn equivalent to

$$\pi'^1 \models \phi_0' \qquad (*)$$

Now take an arbitrary path π starting in s and let π' be a corresponding path starting in s'. This π' also fulfils (*), and, since π^1, π'^1 are corresponding paths as well, $\pi^1 \models F'(\phi_0')$ follows according to the assumptions. This implies $s \models \mathbf{AX}F'(\phi_0') = F'(\mathbf{AX}\phi_0')$.

In analogy, the structural induction is completed for **AW**, **AU**, $\vee$, and $\wedge$. This completes the proof. □

Exercise 24. Add the missing proof steps for the structural induction in the proof of Lemma 15. □

Exercise 25. Add the missing proof steps for the structural induction in the proof of Theorem 14. □

C.5 Construction of Simulations by Predicate Abstraction

In the previous sections we have introduced the definition of simulations and shown their most important property, the preservation of ACTL formulas in the sense of Theorem 14. In the present section we investigate how simulations can be systematically constructed, so that it is not required to perform a proof of the properties of Definition 13. This construction principle is called *predicate abstraction*.

Let E denote the set of well-typed expressions over variable symbols from V, such as $x < y + z$ for $x, y, z \in V$ and $D_x = D_y = D_z = \mathbb{N}$ and $D_{(x<y+z)} = \mathbb{B}$. In general, we denote expressions over variables $v_1, \ldots, v_k \in V$ by $e(v_1, \ldots, v_k)$, and the type of such an expression e is denoted by D_e.

Given a KS $K = (S, S_0, R, L, AP)$ with state space $S \subseteq V \to D$ and a set of well-typed expressions $e_1, \ldots, e_n$. Assuming that $AP = \{v = d \mid v \in V, d \in D_v\}$ and therefore $\forall s \in S : L(s) = \{v = s(v) \mid v \in V\}$, we can systematically construct a new KS $K' = (S', S_0', R', L', AP')$ as follows.

1. Define $V' = \{z_1, \ldots, z_n\}$, where $z_i \notin V$ and n is the number of selected expressions e_i.
2. Define $S' = \{s' : V' \to D' \mid \forall z_i \in V' : s(z_i) \in D_{e_i}\}$ with $D = \bigcup_{i=1}^{n} D_{e_i}$.
3. Define relation $H \subseteq S \times S'$ by

$$H = \{(s, s') \in S \times S' \mid \forall z_i \in V' : s'(z_i) = s(e_i)\}$$

4. Define the initial state space by

$$S'_0 = \{s' \in S' \mid \exists s \in S_0 : H(s, s')\}$$

5. Define the transition relation R' by

$$R' = \{(s'_1, s'_2) \in S' \times S' \mid \exists (s_1, s_2) : R(s_1, s_2) \wedge H(s_1, s'_1) \wedge H(s_2, s'_2)\}$$

6. Define the atomic propositions AP' by

$$AP' = \{z_i = c_i \mid z_i \in V', c_i \in D_{e_i}\}$$

7. Define the labelling function L' in the natural way by

$$\forall s' \in S' : L'(s') = \{p \in AP' \mid s'(p)\}$$

Theorem 15 *Given* K *and well-typed expressions* $e_1, \ldots, e_n$*, the KS* K' *constructed as specified above simulates* K *with simulation relation* H*.*

Proof. By construction, H and K' fulfil properties (i) and (ii) of Definition 13. It remains to show that the function

$$g : L(S) \to L'(S);\ g(L(s)) = L'(s') \text{ if and only if } H(s, s')$$

is well-defined. To this end, suppose that $L(s) = L(r)$ for some $s, r \in S$. According to our assumptions about K, this means that $\forall v \in V : s(v) = r(v)$. As a consequence, $s(e_i) = r(e_i)$ for $i = 1, \ldots, n$. Now the construction of H implies that $H(s, s')$ and $H(r, s')$ with $s'(z_i) = s(e_i) = r(e_i)$ for $i = 1, \ldots, n$. This means that s and r are abstracted to the same state s', and therefore g is well-defined. □

Theorem 16 *Given* K, K' *as introduced above, suppose that* K *has initial condition* $\mathcal{I}$*, and that its transition relation* R *can be represented in propositional form by* $\mathcal{R}$*. Then the associated propositions of* K' *are*

given by

$$\begin{aligned}
\mathcal{I}' &\equiv \exists \xi_1 \in D_{\nu_1}, \ldots, \xi_m \in D_{\nu_m} : \mathcal{I}[\xi_i/\nu_i \mid i = 1, \ldots, m] \wedge \\
&\qquad \bigwedge_{j=1}^{n} z_j = e_j[\xi_i/\nu_i \mid i = 1, \ldots, m] \\
\mathcal{R}' &\equiv \exists \xi_1, \xi_1' \in D_{\nu_1}, \ldots, \xi_m, \xi_m' \in D_{\nu_m} : \mathcal{R}[\xi_i/\nu_i, \xi_i'/\nu_i' \mid i = 1, \ldots, m] \wedge \\
&\qquad \bigwedge_{j=1}^{n} z_j = e_j[\xi_i/\nu_i \mid i = 1, \ldots, m] \wedge \\
&\qquad \bigwedge_{j=1}^{n} z_j' = e_j[\xi_i'/\nu_i \mid i = 1, \ldots, m]
\end{aligned}$$

Proof. Initial condition. By condition 4 of the construction recipe for K' specified above, the initial states of K' are given by

$$S_0' = \{s' \in S' \mid \exists s \in S_0 : H(s, s')\}$$

Using the propositional representation of S_0, this can be re-written as

$$\begin{aligned}
S_0' &= \{s' \in S' \mid \exists s \in S : s \models \mathcal{I} \wedge H(s, s')\} \\
&= \{s' \in S' \mid (\exists s \in S : \mathcal{I}[s(\nu)/\nu \mid \nu \in V]) \wedge H(s, s')\} \\
&= \{s' \in S' \mid \exists s \in S, \xi_1 \in D_{\nu_1}, \ldots \xi_m \in D_{\nu_m} : \\
&\qquad \mathcal{I}[\xi_i/\nu_i \mid i = 1, \ldots, m] \wedge \bigwedge_{i=1}^{m} s(\nu_i) = \xi_i \wedge H(s, s')\} \\
&= \{s' \in S' \mid \exists s \in S, \xi_1 \in D_{\nu_1}, \ldots \xi_m \in D_{\nu_m} : \\
&\qquad \mathcal{I}[\xi_i/\nu_i \mid i = 1, \ldots, m] \wedge \bigwedge_{i=1}^{m} s(\nu_i) = \xi_i \wedge \bigwedge_{j=1}^{n} s'(z_j) = s(e_j)\} \\
&= \{s' \in S' \mid \exists \xi_1 \in D_{\nu_1}, \ldots \xi_m \in D_{\nu_m} : \\
&\qquad \mathcal{I}[\xi_i/\nu_i \mid i = 1, \ldots, m] \wedge \bigwedge_{j=1}^{n} s'(z_j) = e_j[\xi_i/\nu_i \mid i = 1, \ldots, m]\} \\
&= \{s' \in S' \mid \big(\exists \xi_1 \in D_{\nu_1}, \ldots \xi_m \in D_{\nu_m} : \mathcal{I}[\xi_i/\nu_i \mid i = 1, \ldots, m] \wedge \\
&\qquad \bigwedge_{j=1}^{n} z_j = e_j[\xi_i/\nu_i \mid i = 1, \ldots, m]\big)[s'(z_k)/z_k \mid k = 1, \ldots, n]\} \\
&= \{s' \in S' \mid \mathcal{I}'[s'(z_j)/z_j \mid j = 1, \ldots, n]\}
\end{aligned}$$

This proves that $\mathcal{I}'$ is a propositional representation of the initial condition of K'.

Transition Relation. This is shown in analogy to the proof above (see Exercise 20). □

Exercise 26. Prove the correctness of predicate $\mathcal{R}'$ in Theorem 16.

□

Example 27. We illustrate the construction principle for simulations and the application of Theorem 16, using again the traffic light controller example introduced above. First we describe the initial state and transition relation of the concrete traffic light system in propositional form (see notation in the example above):

$$\begin{aligned}
\mathcal{I} &\equiv c = 0 \wedge tl = \text{red} \\
\mathcal{R} &\equiv (c = 0 \wedge tl = \text{red} \wedge c' = 1 \wedge tl' = \text{yellow}) \vee \\
&\quad (c = 1 \wedge tl = \text{yellow} \wedge c' = 2 \wedge tl' = \text{green}) \vee \\
&\quad (c = 2 \wedge tl = \text{green} \wedge c' = 3 \wedge tl' = \text{yellow}) \vee \\
&\quad (c = 3 \wedge tl = \text{yellow} \wedge c' = 0 \wedge tl' = \text{red})
\end{aligned}$$

To construct the simulation according to the recipe above, we define one Boolean expression

$$e_1 = (tl = \text{red} \vee tl = \text{yellow})$$

Step 1. Define $V' = \{\text{stops}\}$.

Step 2. Define $S' = \{m_i : V' \to \mathbb{B} \mid i = 0, 1 \wedge m_i(\text{stops}) = 1 - i\}$. (Recall that we identify Boolean values `false`, `true` with 0,1.)

Step 3. Define the simulation relation by

$$H = \{(s, m) \in S \times S' \mid m(\text{stops}) = (s(tl) = \text{red} \vee s(tl) = \text{yellow})\}$$

Step 4. Calculate the initial state as first order expression by application of Theorem 16.

$$\begin{aligned}
\mathcal{I}' &\equiv \exists \xi_0 \in \{0,1,2,3\}, \xi_1 \in \{\text{red}, \text{yellow}, \text{green}\} : \\
&\qquad \mathcal{I}[\xi_0/c, \xi_1/tl] \wedge \text{stops} = e_1[\xi_0/c, \xi_1/tl] \\
&\equiv \exists \xi_0 \in \{0,1,2,3\}, \xi_1 \in \{\text{red}, \text{yellow}, \text{green}\} : \\
&\qquad \xi_0 = 0 \wedge \xi_1 = \text{red} \wedge \text{stops} = (\xi_1 = \text{red} \vee \xi_1 = \text{yellow}) \\
&\equiv (\text{stops} = \texttt{true})
\end{aligned}$$

Now calculate the transition relation as first order expression by application of Theorem 16.

$$
\begin{aligned}
\mathcal{R}' \quad \equiv \quad & \exists \xi_0, \xi_0' \in \{0,1,2,3\}, \xi_1, \xi_1' \in \{\text{red}, \text{yellow}, \text{green}\} : \\
& \quad \mathcal{R}[\xi_0/c, \xi_1/\text{tl}, \xi_0'/c', \xi_1'/\text{tl}'] \wedge \\
& \quad \text{stops} = e_1[\xi_0/c, \xi_1/\text{tl}] \wedge \text{stops}' = e_1[\xi_0'/c', \xi_1'/\text{tl}'] \\
\equiv \quad & \exists \xi_0, \xi_0' \in \{0,1,2,3\}, \xi_1, \xi_1' \in \{\text{red}, \text{yellow}, \text{green}\} : \\
& \quad \big((\xi_0 = 0 \wedge \xi_1 = \text{red} \wedge \xi_0' = 1 \wedge \xi_1' = \text{yellow}) \vee \\
& \quad (\xi_0 = 1 \wedge \xi_1 = \text{yellow} \wedge \xi_0' = 2 \wedge \xi_1' = \text{green}) \vee \\
& \quad (\xi_0 = 2 \wedge \xi_1 = \text{green} \wedge \xi_0' = 3 \wedge \xi_1' = \text{yellow}) \vee \\
& \quad (\xi_0 = 3 \wedge \xi_1 = \text{yellow} \wedge \xi_0' = 0 \wedge \xi_1' = \text{red})\big) \wedge \\
& \quad \big(\text{stops} = (\xi_1 = \text{red} \vee \xi_1 = \text{yellow}) \wedge \\
& \quad\ \ \text{stops}' = (\xi_1' = \text{red} \vee \xi_1' = \text{yellow})\big) \\
\equiv \quad & \exists \xi_0, \xi_0' \in \{0,1,2,3\}, \xi_1, \xi_1' \in \{\text{red}, \text{yellow}, \text{green}\} : \\
& \quad \big((\xi_0 = 0 \wedge \xi_1 = \text{red} \wedge \xi_0' = 1 \wedge \xi_1' = \text{yellow} \wedge \\
& \quad\ \ \text{stops} = (\xi_1 = \text{red} \vee \xi_1 = \text{yellow}) \wedge \\
& \quad\ \ \text{stops}' = (\xi_1' = \text{red} \vee \xi_1' = \text{yellow}))\big) \vee \\
& \quad (\xi_0 = 1 \wedge \xi_1 = \text{yellow} \wedge \xi_0' = 2 \wedge \xi_1' = \text{green} \wedge \\
& \quad\ \ \text{stops} = (\xi_1 = \text{red} \vee \xi_1 = \text{yellow}) \wedge \\
& \quad\ \ \text{stops}' = (\xi_1' = \text{red} \vee \xi_1' = \text{yellow}))\big) \vee \\
& \quad (\xi_0 = 2 \wedge \xi_1 = \text{green} \wedge \xi_0' = 3 \wedge \xi_1' = \text{yellow} \wedge \\
& \quad\ \ \text{stops} = (\xi_1 = \text{red} \vee \xi_1 = \text{yellow}) \wedge \\
& \quad\ \ \text{stops}' = (\xi_1' = \text{red} \vee \xi_1' = \text{yellow}))\big) \vee \\
& \quad (\xi_0 = 3 \wedge \xi_1 = \text{yellow} \wedge \xi_0' = 0 \wedge \xi_1' = \text{red} \wedge \\
& \quad\ \ \text{stops} = (\xi_1 = \text{red} \vee \xi_1 = \text{yellow}) \wedge \\
& \quad\ \ \text{stops}' = (\xi_1' = \text{red} \vee \xi_1' = \text{yellow}))\big) \\
\equiv \quad & (\text{stops} \wedge \text{stops}') \vee (\text{stops} \wedge \neg\text{stops}') \vee (\neg\text{stops} \wedge \text{stops}')
\end{aligned}
$$

Obviously this initial condition and transition relation of the simulation corresponds to the transition graph shown in Figure C.4. □