

# Theory of Reactive Systems

Jan Peleska<sup>1</sup>

*Centre of Information Technology  
University of Bremen  
Germany  
July 8, 2010*

---

<sup>1</sup> Email: [jp@tzi.de](mailto:jp@tzi.de)

# 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:

$$\langle s_0, s_1, s_2, \dots \rangle$$

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.

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.

**Example 1.1** Suppose we observe temperature changes *temp* in a reactor at discrete points in time, and this results in a run

$$c =_{\text{def}} \langle (t_0, \text{temp}_0), (t_1, \text{temp}_1), (t_2, \text{temp}_2), \dots, (t_k, \text{temp}_k), \dots \rangle$$

where the state observations consist of tuples (timestamp  $t_i$ , temperature  $\text{temp}_i$  observed at time  $t_i$ ). Suppose further that we are interested in observing whether

a temperature threshold  $max$  is exceeded, and that the computation satisfies

$$\forall i \in \{0, \dots, k-1\} : temp_i \leq max$$

$$\forall i \in \{k, \dots, k+3\} : temp_i > max$$

$$\forall i \in \{k+4, \dots\} : temp_i \leq max$$

Introducing two events

- temp\_ok
- temp\_too\_high

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

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

□

### 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 1.2** Suppose we require in Example 1.1 that the temperature threshold in the reactor should never be exceeded for longer than  $\delta$  time units. This can be expressed by a formula referring to arbitrary computations

$$c =_{\text{def}} \langle (t_0, \text{temp}_0), (t_1, \text{temp}_1), (t_2, \text{temp}_2), \dots \rangle$$

in the following way:

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

On the event abstraction level, consider arbitrary computations

$$c_{event} =_{\text{def}} \langle (t_0, e_0), (t_1, e_1), \dots \rangle$$

Now the requirement can be expressed as

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

□

**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*.

**Exercise. 1.** Suppose there is a laboratory equipped with a laser and - for safety reasons - a door locking mechanism. If the laboratory is empty, a controller component locks the door and switches the laser on. If a person wants to enter the laboratory, a button has to be pushed, whereupon the controller switches the laser off and unlocks the door.

Right after switching-on, the laser is in the state *on* which automatically changes to *active* after a certain period of time. The same applies to the states *off* and *passive*.

Possible states of the door are *open* or *closed*. After the door has been opened, it closes automatically. A door counter counts how often the door has been opened or closed. We can assume, that at any time, at most one person can be in the laboratory.

Assuming  $t$ ,  $door$ ,  $dcnt$  and  $laser$  being variables which reflect current point in time, the door, door counter and laser states, computations have the form

$$c = \langle (t_0, door_0, dcnt_0, laser_0), (t_1, door_1, dcnt_1, laser_1), \dots \rangle$$

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

**1.1** Find logical formulas to express the following textual requirements:

- a) In the initial state, the door is closed, the door counter is 0 and the laser is in state *passive*.
- b) Being in the state *on*, the subsequent state of the laser has to be *active*. The same applies to the states *off* and *passive*.
- c) The laser state-change 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.
- e) The laser has to be in the state *passive*, if the room is not empty or the door is open.

**1.2** Introduce events in order to abstract concrete computations  $c$  to abstract computations  $c_E$ . Adapt the formulas a - e to the abstract computations using these events. □

## 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 [2, pp. 14] and [1].

**Definition 2.1** A *State Transition System* is a triple  $TS = (S, S_0, R)$ , 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*.

□

**Definition 2.2** A *Labelled Transition System* is a tuple  $LTS = (S, S_0, \Sigma, R)$ , where

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

□

State transition systems are the preferred mathematical model 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.

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

**Definition 2.3** A *Kripke Structure*  $K = (S, S_0, R, L)$  is a state transition system  $(S, S_0, R)$  augmented by a set  $AP$  of atomic propositions and a function

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

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  $\bar{R}$  suitable for Kripke structures by adding *self loops* to the terminal states in  $R$ :

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

### State Space of Valuation Functions.

Next, we specialise on specification formalisms where the state space can always be defined by a vector 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  $V = \{x_0, x_1, \dots\}$  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  $D = \bigcup_{x \in V} D_x$  the union over all domains of variables from  $V$ . A *valuation* is a partial mapping

$$s : V \not\rightarrow D$$

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

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

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  $s : V \not\rightarrow D$  is a consequence from the atomic propositions specifying exactly the valuation of each variable in the current state  $s$ , that is,

$$x_0 = s(x_0), x_1 = s(x_1), \dots \quad (*)$$

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 set of atomic propositions will therefore be

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

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

$$L(s) = \{x = d \mid x \in V \wedge s(x) = d\}$$

### First Order Representations.

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

Let  $s \in S$  a valuation and  $\phi$  a (first order) logical formula with free variables from  $V = \{x_0, x_1, \dots\}$ . We say that  $\phi$  *holds in state*  $s$  and write  $s \models \phi$ , if the formula evaluates to true when replacing every free variable  $x$  occurring in  $\phi$  by its valuation  $s(x)$ ; that is,  $\phi[s(x_0)/x_0, s(x_1)/x_1, \dots]$  is a tautology.

Based on the replacement concept, the initial state  $S_0$  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,

$$S_0 = \{s : V \not\rightarrow D \mid s \models I\}$$

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

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

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  $\psi$  a formula with free variables in  $V, V'$ . We say that  $\psi$  holds in  $(s, s')$  and write  $(s, s') \models \psi$  if

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

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 2.4** 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 both processes may have to release the CPU between two arbitrary statements.

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

<pre> 1   P0 { 2     do { s = 0; 3       while ( s == 0 ); 4       c0 = 1; // process data 5       c0 = 0; 6     } while (1); 7   } 8 </pre>	<pre> 1   P1 { 2     do { s = 1; 3       while ( s == 1 ); 4       c1 = 1; // process data 5       c1 = 0; 6     } while (1); 7   } 8 </pre>
--	--

To capture the complete state space, we add two program counters  $p_0, p_1$  in range  $\{1, 2, \dots, 7\}$  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  $p_0, p_1 \in \{1, 2, \dots, 7\}$ ,  $c_0, c_1, s \in \mathbb{B}$ . 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

$$\begin{aligned}
T \equiv & (p_0 = 1 \wedge p'_0 = 2 \wedge p'_1 = p_1 \wedge s' = s \wedge c'_0 = c_0 \wedge c'_1 = c_1) \vee \\
& (p_0 = 2 \wedge p'_0 = 3 \wedge p'_1 = p_1 \wedge s' = 0 \wedge c'_0 = c_0 \wedge c'_1 = c_1) \vee \\
& (p_0 = 3 \wedge s = 0 \wedge p'_0 = 3 \wedge p'_1 = p_1 \wedge s' = s \wedge c'_0 = c_0 \wedge c'_1 = c_1) \vee \\
& (p_0 = 3 \wedge s \neq 0 \wedge p'_0 = 4 \wedge p'_1 = p_1 \wedge s' = s \wedge c'_0 = c_0 \wedge c'_1 = c_1) \vee \\
& (p_0 = 4 \wedge p'_0 = 5 \wedge p'_1 = p_1 \wedge s' = s \wedge c'_0 = 1 \wedge c'_1 = c_1) \vee \\
& (p_0 = 5 \wedge p'_0 = 6 \wedge p'_1 = p_1 \wedge s' = s \wedge c'_0 = 0 \wedge c'_1 = c_1) \vee \\
& (p_0 = 6 \wedge p'_0 = 2 \wedge p'_1 = p_1 \wedge s' = s \wedge c'_0 = c_0 \wedge c'_1 = c_1) \vee \\
& (p_1 = 1 \wedge p'_1 = 2 \wedge p'_0 = p_0 \wedge s' = s \wedge c'_1 = c_1 \wedge c'_0 = c_0) \vee \\
& (p_1 = 2 \wedge p'_1 = 3 \wedge p'_0 = p_0 \wedge s' = 1 \wedge c'_1 = c_1 \wedge c'_0 = c_0) \vee \\
& (p_1 = 3 \wedge s = 1 \wedge p'_1 = 3 \wedge p'_0 = p_0 \wedge s' = s \wedge c'_1 = c_1 \wedge c'_0 = c_0) \vee \\
& (p_1 = 3 \wedge s \neq 1 \wedge p'_1 = 4 \wedge p'_0 = p_0 \wedge s' = s \wedge c'_1 = c_1 \wedge c'_0 = c_0) \vee \\
& (p_1 = 4 \wedge p'_1 = 5 \wedge p'_0 = p_0 \wedge s' = s \wedge c'_1 = 1 \wedge c'_0 = c_0) \vee \\
& (p_1 = 5 \wedge p'_1 = 6 \wedge p'_0 = p_0 \wedge s' = s \wedge c'_1 = 0 \wedge c'_0 = c_0) \vee \\
& (p_1 = 6 \wedge p'_1 = 2 \wedge p'_0 = p_0 \wedge s' = s \wedge c'_1 = c_1 \wedge c'_0 = c_0)
\end{aligned}$$

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:

- (i) **Construct the initial states:** This is done by finding all solutions  $s : V \not\rightarrow D$  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  $L(s) = \{x_0 = s(x_0), x_1 = s(x_1), \dots\}$ . 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.

- (ii) **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  $T[s(x_0)/x_0, s(x_1)/x_1, \dots]$ , that is  $T$ , with all prestate variables replaced by their actual values in  $s$ . Every solution  $s'$  gives rise to a new Kripke state with  $L(s') = \{x_0 = s'(x_0), x_1 = s'(x_1), \dots\}$ .

Lets expand our initial state  $\boxed{1,1,0,0,0}$ : Replacing the prestate variables in  $T$  with these values results in formula

$$\begin{aligned} T[1/p_0, 1/p_1, 0/s, 0/c_0, 0/c_1] &\equiv \\ (p'_0 = 2 \wedge p'_1 = 1 \wedge s' = 0 \wedge c'_0 = 0 \wedge c'_1 = 0) \vee \\ (p'_1 = 2 \wedge p'_0 = 1 \wedge s' = 0 \wedge c'_1 = 0 \wedge c'_0 = 0) \end{aligned}$$

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. Observe that we can also represent the Kripke structure as an infinite tree which is called the *computation tree*.  $\square$

### 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 2.4. Since a state  $s$  may occur in more than one place of the computation tree we use tree nodes  $N = S \times 2^{AP} \times \mathbb{N}$ :  $(s, P, n) \in N$  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  $TC = (N, \rho, \text{succ}, \text{pred})$  with

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

The algorithm is shown in Fig. 1. 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

---

```

function computationTree(in ( $S, S_0, R, L$ ) : KripkeStructure) : ( $N, \rho, \text{succ}, \text{pred}$ )
begin
   $n := 1$ ;  $M := \{(s, L(s), n) \mid s \in S_0\}$ ;  $N := \{\rho\} \cup M$ ;
   $\text{succ} := \{\rho \mapsto S_0\} \cup \{s \mapsto \emptyset \mid s \in S_0\}$ ;
   $\text{pred} := \{(s, L(s), n) \mapsto \rho \mid s \in S_0\} \cup \{\rho \mapsto \perp\}$ 
  while  $M \neq \emptyset$  do
     $M' := \emptyset$ ;
    foreach  $(s, L(s), n) \in M$  do
      foreach  $s' \in S$  do
        if  $(s, s') \in R$  then
           $N := N \cup \{(s', L(s'), n + 1)\}$ ;
           $\text{succ}(s, L(s), n) := \text{succ}(s, L(s), n) \cup \{(s', L(s'), n + 1)\}$ ;
           $\text{succ}(s', L(s'), n + 1) := \emptyset$ ;
           $\text{pred}(s', L(s'), n + 1) := (s, L(s), n)$ ;
          if  $(\forall k \in \{1, \dots, n\} : \text{pr}_1(\text{pred}^k(s', L(s'), n + 1)) \neq s')$  then
             $M' := M' \cup \{(s', L(s'), n + 1)\}$ 
          endif
        endif
      enddo
    enddo
     $M := M'$ 
     $n := n + 1$ ;
  enddo
   $\text{computationTree} := (N, \rho, \text{succ}, \text{pred})$ ;
end

```

---

Fig. 1. Algorithm for generating a finite portion of the computation tree associated with a Kripke Structure  $(S, S_0, R, L)$ .

model to prove or disprove assertions about the model with are of a certain restricted nature, to be discussed in the next section.

**Exercise. 2.** Consider the specification model of component  $C$  in Fig. 3.  $C$  inputs  $x \in \{0, 1, 2\}$  and outputs to  $y \in \{-1, 0, 1, 2, \dots\}$ . 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 10 to 11 can only be taken if  $x > y$  holds, which means, that the current valuation  $s : V \not\rightarrow D$  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 2.4, specify the initial state and the transition relation as logical formulas.  $\square$

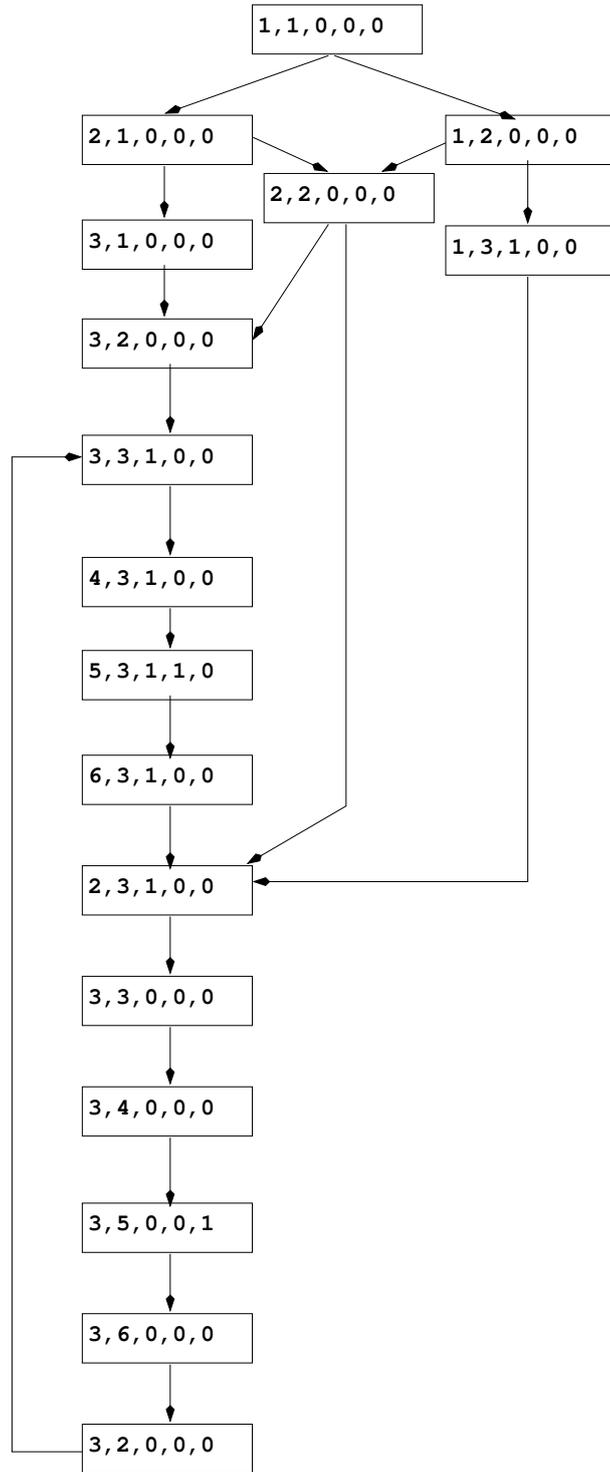
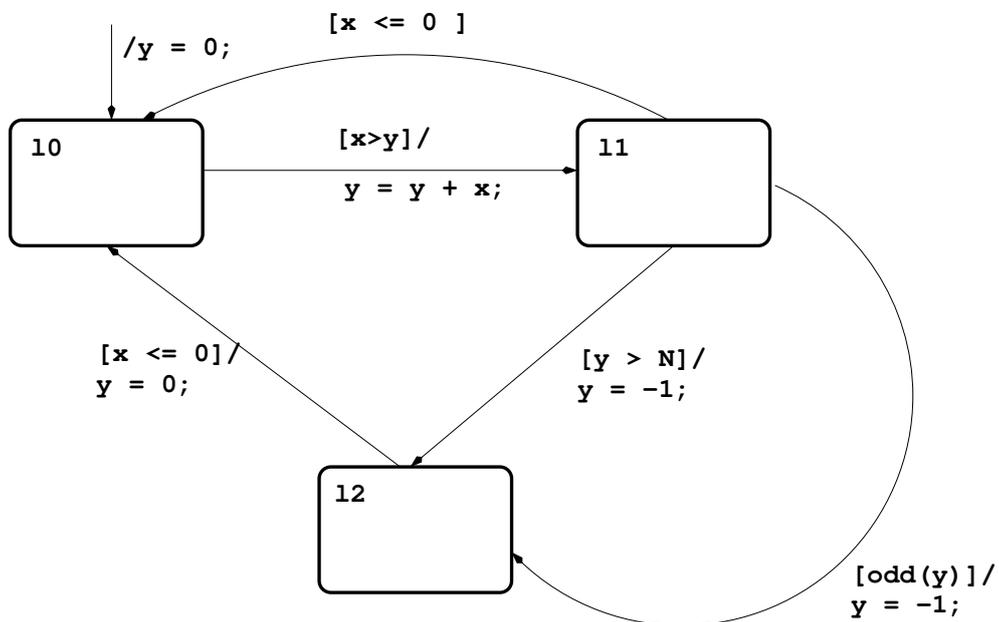
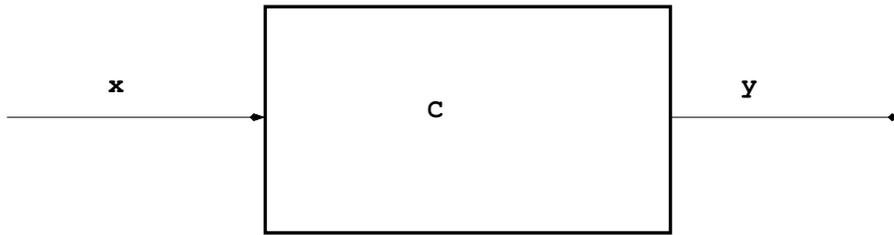


Fig. 2. Kripke structure for the processes  $P_0 \parallel P_1$  from Example 2.4.

**Exercise. 3.** Following the algorithm described in Fig. 1, draw the initial part of the computation tree representing 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  $N = 2$ . □

Fig. 3. Model of component  $C$ .

### 3 Property Specification With Temporal Logic

#### 3.1 The Computation Tree Logic $CTL^*$

##### 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:

- (i) Every atomic proposition  $p \in AP$  is a state formula.
- (ii) If  $f$  and  $g$  are state formulas then  $\neg f, f \wedge g, f \vee g$  are state formulas.
- (iii) If  $f$  is a *path formula* then  $\mathbf{E} f, \mathbf{A} f$  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  $\mathbf{X} f, \mathbf{F} f, \mathbf{G} f, f \mathbf{U} g, f \mathbf{R} g$  are path formulas.

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

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

### Semantics of CTL\* formulas.

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

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

to express that  $\phi$  holds in state  $s$  of  $M$ . We write

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

to express that  $\psi$  holds along path  $\pi$  through  $M$ . For CTL\* formulas  $\phi$  we say  $\phi$  *holds in the Kripke model  $M$*  and write

$$M \models \phi$$

---

$M, s \models p$	$\equiv p \in L(s)$
$M, s \models \neg\phi$	$\equiv M, s \not\models \phi$
$M, s \models \phi_1 \vee \phi_2$	$\equiv M, s \models \phi_1$ 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 $\pi$ from $s$ such that $M, \pi \models \psi$
$M, s \models \mathbf{A} \psi$	$\equiv$ on every path $\pi$ 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 $M, \pi^j \models \psi_2$

---

Fig. 4. Semantics of CTL\* formulas.

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

The inductive definition of  $\models$  is given in Fig. 4, where  $p$  denotes atomic propositions from  $AP$ ,  $\phi, \phi_i$  denote state formulas and  $\psi, \psi_j$  denote path formulas:

**Exercise. 4.** Using the syntax rules of CTL\* formulas and a syntax tree representation, prove or disprove that the following formulas conform to the CTL\*-syntax ( $a, b, c \in AP$ ):

- (i)  $\mathbf{AG}(\mathbf{XFa} \wedge \neg(\mathbf{bUGc}))$
- (ii)  $\mathbf{AXG}\neg a \wedge \mathbf{EFG}(a \vee \mathbf{A}(b\mathbf{U}a))$

□

**Exercise. 5.** Using the Kripke structure displayed in Fig. 2 prove or disprove the following CTL\*-assertions, using the semantic definition described in Fig. 4 in a step-by-step manner. For each of the formulas, give a textual interpretation of their meaning.

- (i)  $\mathbf{AG}\neg(c_0 \wedge c_1)$
- (ii)  $\mathbf{A}(\mathbf{F}c_0 \wedge \mathbf{G}(c_0 \Rightarrow \mathbf{F}(c_1 \wedge \mathbf{F}c_0)))$

Justify why the first assertion could be proved on the finite representation of the Kripke structure's computation tree as explained in algorithm 1 while this is not possible for the second assertion.  $\square$

### 3.2 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), (iv), (v) 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  $AP$ )

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

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 3.1** The CTL\* formula  $\mathbf{A}(\mathbf{F}\mathbf{G}f)$  (*On every path,  $f$  will finally hold in all states*) has no equivalent in CTL.  $\square$

**Theorem 3.2** *Every CTL formula can be expressed by means of the operators  $\neg, \vee, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}$ .*

**Proof.** Obviously  $\psi_1 \wedge \psi_2$  can be expressed as  $\neg(\neg\psi_1 \vee \neg\psi_2)$ . The theorem now follows from the fact that the following equivalences hold for all CTL path formulas

$\psi, \psi_1, \psi_2$ :

1.  $\mathbf{AX}\psi \equiv \neg\mathbf{EX}(\neg\psi)$
2.  $\mathbf{EF}\psi \equiv \mathbf{E}(\mathbf{trueU}\psi)$
3.  $\mathbf{AG}\psi \equiv \neg\mathbf{EF}(\neg\psi)$
4.  $\mathbf{AF}\psi \equiv \neg\mathbf{EG}(\neg\psi)$
5.  $\mathbf{A}(\psi_1\mathbf{U}\psi_2) \equiv \neg\mathbf{E}(\neg\psi_2\mathbf{U}(\neg\psi_1 \wedge \neg\psi_2)) \wedge \neg\mathbf{EG}\neg\psi_2$
6.  $\mathbf{A}(\psi_1\mathbf{R}\psi_2) \equiv \neg\mathbf{E}(\neg\psi_1\mathbf{U}\neg\psi_2)$
7.  $\mathbf{E}(\psi_1\mathbf{R}\psi_2) \equiv \neg\mathbf{A}(\neg\psi_1\mathbf{U}\neg\psi_2)$
8.  $\mathbf{E}\phi \equiv \mathbf{E}(\mathbf{falseU}\phi)$  if  $\phi$  does not contain  $\mathbf{E}, \mathbf{A}, \mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}$
9.  $\mathbf{A}\phi \equiv \neg\mathbf{E}(\mathbf{falseU}\neg\phi)$  if  $\phi$  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. 4, to be performed by the reader in Exercise 6.  $\square$

**Exercise. 6.** Prove the 9 semantic equivalences used in the proof of Theorem 3.2.

### 3.3 The Computation Tree Logics $\mathbf{ACTL}^*$ and $\mathbf{ACTL}$

If we restrict  $\mathbf{CTL}^*$  formulas to universal quantification only, the resulting computation tree logic is called  $\mathbf{ACTL}^*$ . More precisely,  $\mathbf{ACTL}^*$  only admits  $\mathbf{CTL}^*$  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  $\mathbf{CTL}$  formulas to universal quantification is called  $\mathbf{ACTL}$ .

**Example 3.3**  $\mathbf{AFAX}a$  is an  $\mathbf{ACTL}$  formula, but  $\mathbf{AGEFa}$  is not in  $\mathbf{ACTL}^*$ , since its  $\mathbf{E}$ -free representation  $\mathbf{AG}\neg\mathbf{AG}\neg a$  is not in positive normal form.  $\square$

In Section 5.4 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  $\mathbf{ACTL}^*$  or  $\mathbf{ACTL}$ , respectively.

## 4 CTL 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  $K = (S, S_0, R, L)$ . The specification is most frequently expressed by a temporal logic formula  $\phi$ ; an alternative specification formalism is *trace logic*.

In the general case we wish to identify all states  $s \in S$  where  $\phi$  holds, i. e.,  $s \models \phi$ . In most practical applications the objective is to prove that  $\phi$  holds in every initial state  $s \in S_0$  and in every state which is reachable from some initial state by  $n$ -fold application of the transition relation  $R$ ; this is written  $K \models \phi$ .

In this section we investigate property checking for Kripke structures against CTL formulas. 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  $x = \nu$  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.

### 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  $\phi_i$  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  $\phi_i$ . To this end, a labelling function  $L_\phi : S \rightarrow CTL$  is used.
- The algorithm stops when the last formula  $\phi_i$  having been processed coincides with the specification  $\phi$ .
- The result of the algorithm is the set  $S_\phi =_{\text{def}} \{s \in S \mid \phi \in L_\phi(s)\}$ .
- The Kripke model  $(S, S_0, R, L)$  satisfies  $\phi$  if its initial states are part of  $S_\phi$ , that is,

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

---

```

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

```

---

Fig. 5. Main algorithm for CTL property checking against Kripke structures.

### Syntax tree representation of CTL formulas.

From Section 3.2 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

- $\varepsilon$ : 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  $\phi$  is as follows:

- (i) If  $\phi \in AP$  then  $t(\phi) = T(\varepsilon, \phi, \varepsilon)$ .
- (ii) If  $\phi = \neg\phi_1$  then  $t(\phi) = T(\varepsilon, \neg, t(\phi_1))$ .
- (iii) If  $\phi = \phi_0 \vee \phi_1$  then  $t(\phi) = T(t(\phi_0), \vee, t(\phi_1))$ .
- (iv) If  $\phi = \mathbf{EX}\phi_1$  then  $t(\phi) = T(\varepsilon, \mathbf{EX}, t(\phi_1))$ .
- (v) If  $\phi = \mathbf{E}(\phi_0 \mathbf{U} \phi_1)$  then  $t(\phi) = T(t(\phi_0), \mathbf{EU}, t(\phi_1))$ <sup>2</sup>.
- (vi) 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  $\phi$ , its leaves (i. e. its atomic propositions) can be extracted by means of the function  $\text{leaves} : \text{Tree} \rightarrow 2^{AP}$  by means of the following recursive definition:

- (i)  $\text{leaves}(T(\varepsilon, \phi, \varepsilon)) = \{\phi\}$
- (ii)  $\text{leaves}(T(\varepsilon, \neg, t(\phi_1))) = \text{leaves}(t(\phi_1))$
- (iii)  $\text{leaves}(T(t(\phi_0), \vee, t(\phi_1))) = \text{leaves}(t(\phi_0)) \cup \text{leaves}(t(\phi_1))$
- (iv)  $\text{leaves}(T(\varepsilon, \mathbf{EX}, t(\phi_1))) = \text{leaves}(t(\phi_1))$
- (v)  $\text{leaves}(T(t(\phi_0), \mathbf{EU}, t(\phi_1))) = \text{leaves}(t(\phi_0)) \cup \text{leaves}(t(\phi_1))$
- (vi)  $\text{leaves}(T(\varepsilon, \mathbf{EG}, t(\phi_1))) = \text{leaves}(t(\phi_1))$

---

<sup>2</sup> 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  $T(t(\phi_0), \mathbf{EU}, t(\phi_1))$

---

```

procedure calcLabel(in  $(S, S_0, R, L)$  : KripkeStructure;
                    in  $\phi$  : CTL;
                    inout  $\text{label} : S \rightarrow 2^{\text{CTL}}$ )
begin
  if  $\phi \in AP$  then
    foreach  $s \in S$  do
      if  $\phi \in L(s)$  then
         $\text{label}(s) := \text{label}(s) \cup \{\phi\}$ ;
      endif
    enddo
  elseif  $t(\phi) = T(\varepsilon, \neg, t(\phi_1))$  then
    calcLabel $((S, S_0, R, L), \phi_1, \text{label})$ ;
    foreach  $s \in S$  do
      if  $\phi_1 \notin \text{label}(s)$  then
         $\text{label}(s) := \text{label}(s) \cup \{\phi\}$ ;
      endif
    enddo
  elseif  $t(\phi) = T(t(\phi_0), \vee, t(\phi_1))$  then
    calcLabel $((S, S_0, R, L), \phi_0, \text{label})$ ;
    calcLabel $((S, S_0, R, L), \phi_1, \text{label})$ ;
    foreach  $s \in S$  do
      if  $\phi_0 \in \text{label}(s) \vee \phi_1 \in \text{label}(s)$  then
         $\text{label}(s) := \text{label}(s) \cup \{\phi\}$ ;
      endif
    enddo
  elseif  $t(\phi) = T(\varepsilon, \mathbf{EX}, t(\phi_1))$  then
    calcLabel $((S, S_0, R, L), \phi_1, \text{label})$ ;
    foreach  $s \in S$  do
      if  $\exists s' \in S : R(s, s') \wedge \phi_1 \in \text{label}(s')$  then
         $\text{label}(s) := \text{label}(s) \cup \{\phi\}$ ;
      endif
    enddo
  elseif  $t(\phi) = T(t(\phi_0), \mathbf{EU}, t(\phi_1))$  then
    calcLabel $((S, S_0, R, L), \phi_0, \text{label})$ ; calcLabel $((S, S_0, R, L), \phi_1, \text{label})$ ;
    calcLabelEU $((S, S_0, R, L), \phi_0, \phi_1, \text{label})$ ;
  elseif  $t(\phi) = T(\varepsilon, \mathbf{EG}, t(\phi_1))$  then
    calcLabel $((S, S_0, R, L), \phi_1, \text{label})$ ;
    calcLabelEG $((S, S_0, R, L), \phi_1, \text{label})$ ;
  endif
end

```

---

Fig. 6. Label calculation – syntax-driven control algorithm.

---

```

procedure calcLabelEU(in ( $S, S_0, R, L$ ) : KripkeStructure;
                       in  $\phi_0 : \text{CTL}$ ; in  $\phi_1 : \text{CTL}$ ;
                       inout  $\text{label} : S \rightarrow 2^{\text{CTL}}$ )
begin
   $T := \langle s \in S \mid \phi_1 \in \text{label}(s) \rangle$ ;
  foreach  $s \in T$  do  $\text{label}(s) := \text{label}(s) \cup \{\mathbf{E}(\phi_0 \mathbf{U} \phi_1)\}$ ;
  while  $T \neq \langle \rangle$  do
     $s := \text{hd}(T)$ ;
     $T := \text{tail}(T)$ ;
    foreach  $u \in \{v \in S \mid R(v, s)\}$  do
      if  $\mathbf{E}(\phi_0 \mathbf{U} \phi_1) \notin \text{label}(u) \wedge \phi_0 \in \text{label}(u)$  then
         $\text{label}(u) := \text{label}(u) \cup \{\mathbf{E}(\phi_0 \mathbf{U} \phi_1)\}$ ;
         $T := T \frown \langle u \rangle$ ;
      endif
    enddo
  enddo
end

```

---

Fig. 7. Algorithm for labelling states with  $\mathbf{E}(\phi_0 \mathbf{U} \phi_1)$  formulas.

---

```

procedure calcLabelEG(in ( $S, S_0, R, L$ ) : KripkeStructure;
                       in  $\phi_1 : \text{CTL}$ ;
                       inout  $\text{label} : S \rightarrow 2^{\text{CTL}}$ )
begin
   $S' := \{s \in S \mid \phi_1 \in \text{label}(s)\}$ ;
   $\text{SCC} := \{C \mid C \text{ is a nontrivial SCC of } S'\}$ 
   $T := \langle s \mid \exists C \in \text{SCC} : s \in C \rangle$ ;
  foreach  $s \in T$  do  $\text{label}(s) := \text{label}(s) \cup \{\mathbf{EG}\phi_1\}$ ;
  while  $T \neq \langle \rangle$  do
     $s := \text{hd}(T)$ ;
     $T := \text{tail}(T)$ ;
    foreach  $u \in \{v \in S' \mid R(v, s)\}$  do
      if  $\mathbf{EG}\phi_1 \notin \text{label}(u)$  then
         $\text{label}(u) := \text{label}(u) \cup \{\mathbf{EG}\phi_1\}$ ;
         $T := T \frown \langle u \rangle$ ;
      endif
    enddo
  enddo
end

```

---

Fig. 8. Algorithm for labelling states with  $\mathbf{EG}\phi_1$  formulas.

## 5 Data Abstraction

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

### 5.1 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

$$\begin{aligned} TS/\sim &=_{\text{def}} (S/\sim, S_0/\sim, R/\sim) \\ S_0/\sim &=_{\text{def}} \{[s_0] \mid s_0 \in S_0\} \\ R/\sim &=_{\text{def}} \{([s], [s']) \mid \exists u \in [s], u' \in [s'] : R(u, u')\} \end{aligned} \tag{1}$$

### 5.2 Auxiliary Variables and Associated Equivalence Classes

Let us consider now again only state spaces  $S$  whose elements are variable valuations  $s : V \not\rightarrow D$ ,  $V = \{x_1, x_2, \dots\}$ . Let  $AUX = \{a_1, a_2, \dots\}$  a set of fresh variables such that  $V \cap AUX = \emptyset$ . Let  $e_i(x_1^i, x_2^i, \dots)$  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

$$\begin{aligned} s_e &: V \cup AUX \not\rightarrow D \\ \text{dom } s_e &= \text{dom } s \cup \{a_i \in AUX \mid x_1^i, x_2^i, \dots \in \text{dom } s\} \\ s_e|_V &= s \text{ that is, } \forall x \in V \cap \text{dom } s_e : s_e(x) = s(x) \\ \forall a_i \in AUX \cap \text{dom } s_e &: s_e(a_i) = e_i(s(x_1^i), s(x_2^i), \dots) \end{aligned}$$

Observe that the expressions  $e_i(x_1^i, x_2^i, \dots)$  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  $TS_e = (S_e, S_{0e}, R_e)$  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*

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

Observe that, given a valuation  $(s : V \not\rightarrow D) \in S$ , its equivalence class  $[s]$  may also be regarded as a valuation function on the variables from  $AUX$  by setting

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

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

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

**Lemma 5.1** *Suppose that the initial state  $S_0$  is characterised by first-order predicate  $\mathcal{I}$  with free variables in  $V = \{x_1, x_2, \dots\}$ , and that the transition relation  $R \subseteq S \times S$  is characterised by predicate  $\mathcal{R}$  with free variables in  $V$  and  $V' =_{\text{def}} \{x'_1, x'_2, \dots\}$ . Then the respective predicates for  $TS_e/\sim$  are given by*

$$\mathcal{I}/\sim(a_1, a_2, \dots) =_{\text{def}} \exists \xi_1, \xi_2, \dots : (\forall i : a_i = e_i(\xi_1, \xi_2, \dots)) \wedge \mathcal{I}[\xi_1/x_1, \xi_2/x_2, \dots] \quad (2)$$

$$\begin{aligned} \mathcal{R}/\sim(a_1, a_2, \dots, a'_1, a'_2, \dots) &=_{\text{def}} \exists \xi_1, \xi_2, \dots, \xi'_1, \xi'_2, \dots : \\ &\forall i : (a_i = e_i(\xi_1, \xi_2, \dots) \wedge a'_i = e_i(\xi'_1, \xi'_2, \dots)) \wedge \\ &\mathcal{R}[\xi_1/x_1, \xi_2/x_2, \dots, \xi'_1/x'_1, \xi'_2/x'_2, \dots] \end{aligned} \quad (3)$$

**Proof.** From (1) and the fact that  $\mathcal{I}$  characterises  $S_0$  we conclude that

$$S_{0e}/\sim = \{[s_0] : AUX \not\rightarrow D \mid s_0 : V \cup AUX \not\rightarrow D \wedge \mathcal{I}[s_0(x_1)/x_1, s_0(x_2)/x_2, \dots]\}$$

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

$$\begin{aligned} \bar{S} &=_{\text{def}} \{s_a : AUX \not\rightarrow D \mid \mathcal{I}/\sim[s_a(a_1)/a_1, s_a(a_2)/a_2, \dots]\} = \\ &\{s_a : AUX \not\rightarrow D \mid \exists \xi_1, \xi_2, \dots : (\forall i : s_a(a_i) = e_i(\xi_1, \xi_2, \dots)) \wedge \mathcal{I}[\xi_1/x_1, \xi_2/x_2, \dots]\} \end{aligned}$$

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

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

Now we show  $\bar{S} \subseteq S_{0e}/\sim$ : Let  $s_a \in \bar{S}$ , then there exist  $\xi_1, \xi_2, \dots$  such that  $(\forall i : s_a(a_i) = e_i(\xi_1, \xi_2, \dots)) \wedge \mathcal{I}[\xi_1/x_1, \xi_2/x_2, \dots]$ . Now define a valuation  $s_0 : V \not\rightarrow D$  by  $s_0(x_i) =_{\text{def}} \xi_i, i = 1, 2, \dots$ . This  $s_0$  is contained in  $S_0$  and therefore  $[s_0] \in S_{0e}/\sim$ , since  $\mathcal{I}[\xi_1/x_1, \xi_2/x_2, \dots]$  and therefore  $\mathcal{I}[s_0(x_1)/x_1, s_0(x_2)/x_2, \dots]$

holds. Since  $s_a(a_i) = e_i(\xi_1, \xi_2, \dots) = e_i(s_0(x_1), s_0(x_2), \dots)$ , the construction of  $s_0$  implies  $s_a = [s_0]$ , so  $s_a \in S_{0e}/\sim$ , and this shows  $\bar{S} \subseteq S_{0e}/\sim$  and proves (2).

For proving (3), recall from (1) that the transition relation of the factorised transition system  $TS_{e/\sim}$  is defined by

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

We define

$$\bar{R} =_{\text{def}} \{(s_a, s'_a) \mid \mathcal{R}/\sim[s_a(a_1)/a_1, s_a(a_2)/a_2, \dots, s'_a(a_1)/a'_1, s'_a(a_2)/a'_2, \dots]\}$$

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

To show that  $R/\sim \subseteq \bar{R}$ , suppose that  $([s], [s']) \in R/\sim$ . By definition of  $[\cdot]$ ,  $R/\sim$  and  $\mathcal{R}$  there exists  $u, u' : V \not\rightarrow D$  such that

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

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

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

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

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

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

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

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

Now define

$$s : V \not\rightarrow D; s(x_i) \mapsto \xi_i, \quad s' : V \not\rightarrow D; s'(x_i) \mapsto \xi'_i, i = 1, 2, \dots$$

Then  $[s] = s_a$  and  $[s'] = s'_a$  and  $\mathcal{R}[s(x_1)/x_1, s(x_2)/x_2, \dots, s'(x_1)/x'_1, s'(x_2)/x'_2, \dots]$  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.  $\square$

### 5.3 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  $e_i(x_1^i, x_2^i, \dots)$  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

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

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

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

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

$$K_e/\sim = (S_e/\sim, S_{oe}/\sim, R_e/\sim, L_e/\sim)$$

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  $\varphi$  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  $\varphi$  proven to be valid in  $K_e/\sim$  is also valid in  $K_e$ .

**Example 5.2** Consider the Kripke Structure depicted in Fig. 9, 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 halt-state in front of the traffic light, it makes sense to introduce a Boolean auxiliary variable

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

Factorisation against the equivalence relation introduced by  $\mathbf{stops}$  leads to the abstracted Kripke structure shown in Fig. 10.

Now suppose we wish to prove that  $\mathbf{EF}(\mathbf{tl} = \mathbf{green})$  holds for the Kripke structure of the original model in Fig. 9. The assertion can be readily expressed on abstract level as  $\mathbf{EF}(\neg \mathbf{stops})$  which obviously holds on abstract level, since every path in Fig. 10 visits  $(\mathbf{m1}, \neg \mathbf{stops})$ . Similarly, the condition  $\mathbf{AF}(\mathbf{tl} = \mathbf{red} \vee \mathbf{tl} = \mathbf{yellow})$  can be expressed in an abstract way as  $\mathbf{AF} \mathbf{stops}$ . 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  $\mathbf{EG}(\mathbf{stops})$

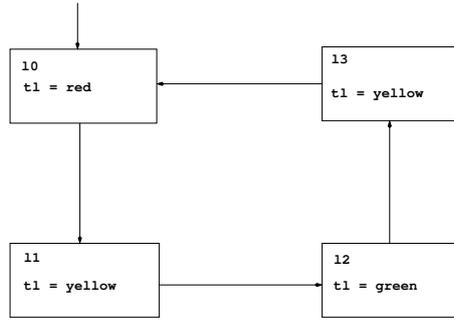
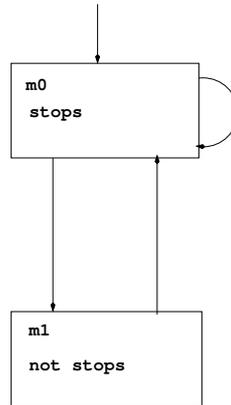


Fig. 9. Kripke structure of traffic light controller from Example 5.2.

Fig. 10. Abstracted Kripke structure induced by auxiliary variable `stops` in Example 5.2.

which obviously does not hold in the concrete model. Conversely, the concrete model satisfies  $\mathbf{AF}(\neg\text{stops})$  which is false on abstract level.  $\square$

**Exercise. 7.** Consider the slightly modified specification model from Exercise 2, now shown in Fig. 11. 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 can be proved using the explicit CTL model checking algorithms on the abstracted Kripke structure:

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

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.

$\square$

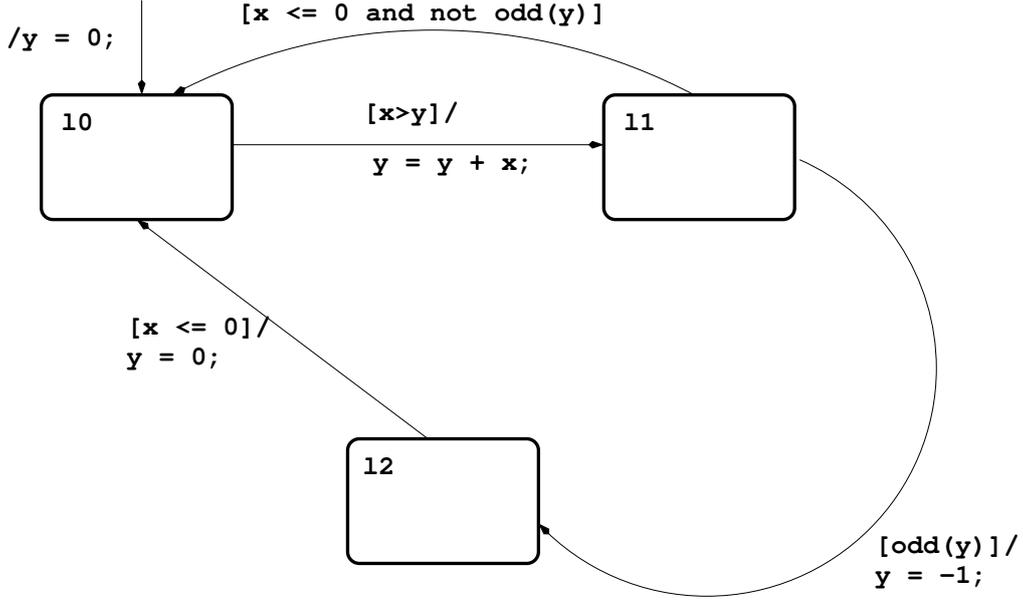


Fig. 11. Model for Exercise 7.

#### 5.4 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*:

**Definition 5.3** [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  $(s, s') \in H$ :

- (i)  $L(s) \cap AP' = L'(s')$
- (ii)  $\forall s_1 \in S : R(s, s_1) \Rightarrow \exists s'_1 \in S' : R'(s', s'_1) \wedge H(s_1, s'_1)$

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

$$\forall s_0 \in S_0 : \exists s'_0 \in S'_0 : H(s_0, s'_0)$$

□

Before exploiting the simulation concept in Theorem 5.7 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  $K_e/\sim$ :

**Theorem 5.4** Given  $\sim$ , equivalence classes  $[s]$ ,  $AP_e$ ,  $L_e$ ,  $K_e$ ,  $K_e/\sim$  as introduced in Section 5.3 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  $K_e \preceq K_e/\sim$  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 5.3, 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

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

Therefore condition (i) of Definition 5.3 holds.

Now let  $s_1 \in S_e$  such that  $R(s, s_1)$ . By construction of  $R/\sim$  in Section 5.1 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 5.3 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,  $K_e \preceq K_e/\sim$ , and this completes the proof.  $\square$

**Definition 5.5** Let  $K \preceq K'$  with simulation relation  $H \subset S \times S'$  and  $H(s, s')$ . Suppose  $\pi$  is a path in  $K$  starting at  $s$  and  $\pi'$  a path starting at  $s'$  in  $K'$ . We say that  $\pi$  and  $\pi'$  correspond to each other if

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

$\square$

**Lemma 5.6** Let  $K \preceq K'$  with simulation relation  $H \subset S \times S'$  and  $H(s, s')$ . Then for every path  $\pi$  in  $K$  starting at  $s$  there is a corresponding path  $\pi'$  in  $K'$  starting at  $s'$ .

**Proof.** Since  $\pi$  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 5.3 successively on  $\pi(0), \pi(1), \pi(2), \dots$  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  $\pi'$  is a path in  $K'$ , and it corresponds to  $\pi$  by construction.  $\square$

**Theorem 5.7** Assume  $K \preceq K'$ . Then for every ACTL\* formula  $\phi$  with atomic propositions in  $AP'$

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

**Proof.** Let  $\phi$  an ACTL\* formula as defined in Section 3.3. 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

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

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

(1) If  $\phi$  is an atomic proposition, then  $(K, s) \models \phi$  if and only if  $\phi \in L(s)$ . Since  $(K', s') \models \phi$  by assumption,  $\phi$  must be contained in  $AP'$ . Because  $K'$  simulates  $K$  and  $L(s) \cap AP' = L'(s')$  holds (condition (i) of Definition 5.3). 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  $\phi$  is an ACTL\* formula  $\phi_1$  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  $\phi_i$  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  $\phi_i$  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  $\phi$  a state formula, such that  $(K, s) \models \phi$  whenever  $(K', s') \models \phi$ . Let  $\pi$  a path with  $\pi(0) = s$ , and  $\pi'$  its corresponding path in  $K'$ , starting at  $s' = \pi'(0)$  (this path exists according to Lemma 5.6). 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  $\pi'$  corresponding to  $\pi$ .

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

(7) Let  $\phi = \psi_1 \vee \psi_2$ , such that  $\psi_i$  are path formulas where  $K, \pi \models \psi_i$  whenever  $K', \pi' \models \psi_i$  for  $i = 1, 2$  on a path  $\pi'$  corresponding to  $\pi$ . 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  $\pi'$  corresponding to  $\pi$ .

(8) Let  $\phi = \psi_1 \wedge \psi_2$ , such that  $\psi_i$  are path formulas where  $K, \pi \models \psi_i$  whenever  $K', \pi' \models \psi_i$  for  $i = 1, 2$  on a path  $\pi'$  corresponding to  $\pi$ . 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  $\pi'$  corresponding to  $\pi$ .

(9) Let  $\phi = \mathbf{X}\psi$  and  $\psi$  a path formula such that  $K, \pi \models \psi$  holds whenever  $K', \pi' \models \psi$  holds on a path  $\pi'$  corresponding to  $\pi$ . Now  $K', \pi' \models \mathbf{X}\psi$  is equivalent to  $K', \pi'^1 \models \psi$ . Since  $\pi'^1$  corresponds to  $\pi^1$  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.  $\square$

**Exercise. 8.a** Give the following explanations regarding the proof of Theorem 5.7:

- (i) Give a detailed formal explanation why the theorem follows from (\*).
- (ii) Give a formal syntax specification for ACTL\* similar to EBNF notation introduced for CTL\* formulas in Section 3.1.
- (iii) Explain how ACTL\* is inductively defined according to Definition A.1:
  - (a) What might be a suitable universe  $U$ ?
  - (b) What is the base set  $B$ ?
  - (c) Which are the constructors  $r \in K$ ?
- (iv) Explain how the proof of Theorem 5.7 applies the principle of structural induction.

□

**Theorem 5.8** 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  $L : S \rightarrow 2^{AP}$ . 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  $K \preceq K'$ .

**Proof.** See Exercise 8.

□

**Exercise. 8.** Prove Theorem 5.8, using the facts on first order representations given in Section 2.

□

### 5.5 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 5.9** [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  $B(s, s')$ :

- (i)  $L(s) = L'(s')$
- (ii)  $\forall s_1 \in S : R(s, s_1) \Rightarrow \exists s'_1 \in S' : R'(s', s'_1) \wedge B(s_1, s'_1)$
- (iii)  $\forall s'_1 \in S' : R'(s', s'_1) \Rightarrow \exists s_1 \in S : R(s, s_1) \wedge B(s_1, s'_1)$

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<sup>3</sup>:

**Theorem 5.10** *If  $K \equiv K'$  and  $\phi \in \text{CTL}^*$ , then*

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

□

### 5.6 Predicate Abstraction

With the knowledge of Section 5.3 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  $a_i = e_i(x_1^i, x_2^i, \dots)$  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.

**Definition 5.11** Let  $K = (S, S_0, R, L)$  a Kripke structure with variables from  $V = \{x_1, \dots, x_n\}$  and  $\phi$  a predicate with free variables over  $V$ . Let  $AUX = \{a_1, \dots, a_k\}$  a set of auxiliary variables defining an abstraction relation via expressions  $a_i = e_i(x_1^i, x_2^i, \dots), i = 1, \dots, k$ . Then the *lifting* of  $\phi$  with respect to this abstraction is denoted by  $[\phi]$  and defined as

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

□

**Theorem 5.12** *Let  $K = (S, S_0, R, L)$  a Kripke structure with variables from  $V = \{x_1, \dots, x_n\}$  and  $\phi$  a predicate with free variables over  $V$ . Let  $AUX = \{a_1, \dots, a_k\}$  a set of auxiliary variables defining an abstraction relation via expressions  $a_i = e_i(x_1^i, x_2^i, \dots), i = 1, \dots, k$ . Let  $K' = (S', S'_0, R', L')$  denote the abstracted Kripke structure obtained by factorisation with  $\sim$  as described in Section 5.3. 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*

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

<sup>3</sup> For a proof, see [2, pp. 171].

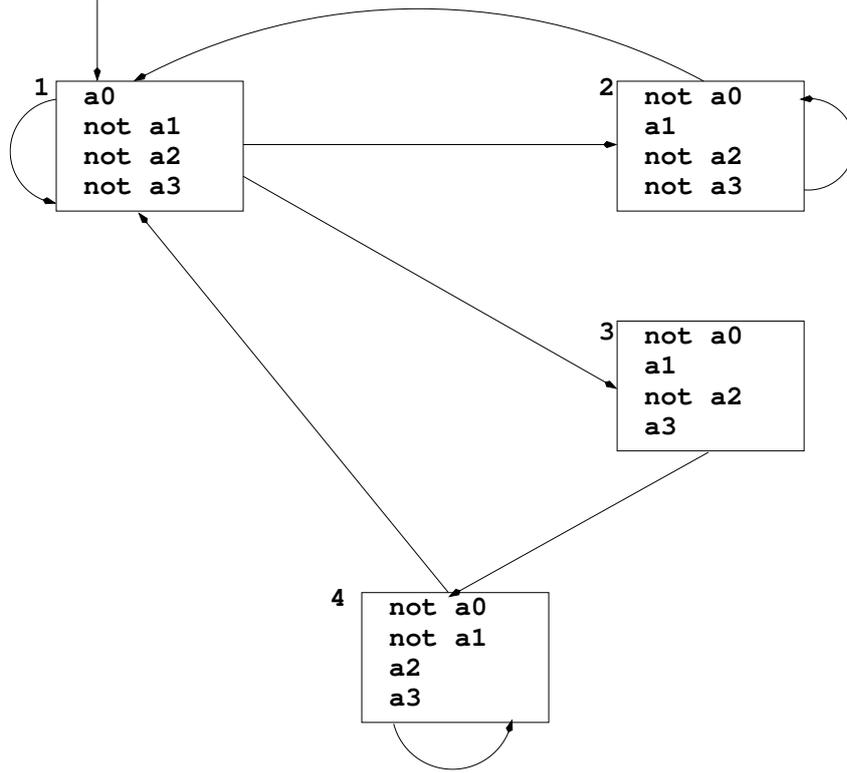


Fig. 12. Kripke structure for abstracted model from Example 5.13.

**Proof.** Applying Definition 5.11 on  $\mathcal{I}$  and  $\mathcal{R}$  yields

$$\begin{aligned}
 [\mathcal{I}] &\equiv \exists \xi_1, \dots, \xi_n : (\forall i = 1, \dots, k : a_i = e_i(\xi_1, \dots, \xi_n)) \wedge \mathcal{I}[\xi_1/x_1, \dots, \xi_n/x_n] \\
 [\mathcal{R}] &\equiv \exists \xi_1, \dots, \xi_n : \exists \xi'_1, \dots, \xi'_n : (\forall i = 1, \dots, k : a_i = e_i(\xi_1, \dots, \xi_n)) \wedge \\
 &\quad (\forall i = 1, \dots, k : a'_i = e_i(\xi'_1, \dots, \xi'_n)) \wedge \\
 &\quad \mathcal{R}[\xi_1/x_1, \dots, \xi_n/x_n, \xi'_1/x'_1, \dots, \xi'_n/x'_n]
 \end{aligned}$$

According to Lemma 5.1 these formulas represent initial condition  $\mathcal{I}/\sim$  and transition relation  $\mathcal{R}/\sim$  of  $K'$ .  $\square$

**Example 5.13** Consider again the model displayed in Fig. 11 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

$$\begin{aligned}
a_0 &= 10 \\
a_1 &= 11 \\
a_2 &= 12 \\
a_3 &= \text{odd}(y)
\end{aligned} \tag{4}$$

Note, that  $a_0, \dots, a_3$  form not the simplest abstraction possible to show the required property - indeed, abstraction by  $a_0$  and  $a_3$  would suffice. The effect of the coarser abstraction would be, that proving several other formulas like  $\mathbf{AG}(\neg 12 \vee \text{odd}(y))$  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. 11 we derive

$$\mathcal{I}(10, 11, 12, x, y) \equiv 10 \wedge \neg 11 \wedge \neg 12 \wedge y = 0$$

**Step. 2.** Specify formula for the transition relation of the concrete system: Evaluating Fig. 11 again, we derive

$$\begin{aligned}
\mathcal{R}(10, 11, 12, x, y, 10', 11', 12', x', y') &\equiv \\
&((10 \wedge x \leq y \wedge y' = y \wedge 10') \vee \\
&(10 \wedge x > y \wedge y' = y + x \wedge 11') \vee \\
&(11 \wedge x \leq 0 \wedge \neg \text{odd}(y) \wedge y' = y \wedge 10') \vee \\
&(11 \wedge \text{odd}(y) \wedge y' = -1 \wedge 12') \vee \\
&(11 \wedge x > 0 \wedge \neg \text{odd}(y) \wedge y' = y \wedge 11') \vee \\
&(12 \wedge x \leq 0 \wedge y' = 0 \wedge 10') \vee \\
&(12 \wedge x > 0 \wedge y' = y \wedge 12')) \wedge \\
&((10 \wedge \neg 11 \wedge \neg 12) \vee (\neg 10 \wedge 11 \wedge \neg 12) \vee (\neg 10 \wedge \neg 11 \wedge 12)) \wedge \\
&((10' \wedge \neg 11' \wedge \neg 12') \vee (\neg 10' \wedge 11' \wedge \neg 12') \vee (\neg 10' \wedge \neg 11' \wedge 12'))
\end{aligned}$$

**Step. 3.** Compute the abstracted initial condition  $\mathcal{I}/\sim = [\mathcal{I}]$ : Applying Definition 5.11 on  $[\mathcal{I}]$  for the given abstraction (4) results in

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

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

tion 5.11 on  $[\mathcal{R}]$  for the given abstraction (4) results in

$$\begin{aligned}
[\mathcal{R}](a_0, a_1, a_2, a_3, a'_0, a'_1, a'_2, a'_3) \equiv \\
& \exists \xi_0, \xi_1, \xi_2, \xi_3, \xi_4, \xi'_0, \xi'_1, \xi'_2, \xi'_3, \xi'_4 : \\
& a_0 = \xi_0 \wedge a_1 = \xi_1 \wedge a_2 = \xi_2 \wedge a_3 = \text{odd}(\xi_4) \wedge \\
& a'_0 = \xi'_0 \wedge a'_1 = \xi'_1 \wedge a'_2 = \xi'_2 \wedge a'_3 = \text{odd}(\xi'_4) \wedge \\
& ((\xi_0 \wedge \xi_3 \leq \xi_4 \wedge \xi'_4 = \xi_4 \wedge \xi'_0) \vee \\
& (\xi_0 \wedge \xi_3 > \xi_4 \wedge \xi'_4 = \xi_4 + \xi_3 \wedge \xi'_1) \vee \\
& (\xi_1 \wedge \xi_3 \leq 0 \wedge \neg \text{odd}(\xi_4) \wedge \xi'_4 = \xi_4 \wedge \xi'_0) \vee \\
& (\xi_1 \wedge \text{odd}(\xi_4) \wedge \xi'_4 = -1 \wedge \xi'_2) \vee \\
& (\xi_1 \wedge \xi_3 > 0 \wedge \neg \text{odd}(\xi_4) \wedge \xi'_4 = \xi_4 \wedge \xi'_1) \vee \\
& (\xi_2 \wedge \xi_3 \leq 0 \wedge \xi'_4 = 0 \wedge \xi'_0) \vee \\
& (\xi_2 \wedge \xi_3 > 0 \wedge \xi'_4 = \xi_4 \wedge \xi'_2)) \wedge \\
& ((\xi_0 \wedge \neg \xi_1 \wedge \neg \xi_2) \vee (\neg \xi_0 \wedge \xi_1 \wedge \neg \xi_2) \vee (\neg \xi_0 \wedge \neg \xi_1 \wedge \xi_2)) \wedge \\
& ((\xi'_0 \wedge \neg \xi'_1 \wedge \neg \xi'_2) \vee (\neg \xi'_0 \wedge \xi'_1 \wedge \neg \xi'_2) \vee (\neg \xi'_0 \wedge \neg \xi'_1 \wedge \xi'_2)) \equiv \\
& ((a_0 \wedge a'_3 = a_3 \wedge a'_0) \vee (a_0 \wedge a'_3 \wedge a'_1) \vee (a_0 \wedge \neg a'_3 \wedge a'_1) \vee \\
& (a_1 \wedge \neg a_3 \wedge a'_3 = a_3 \wedge a'_0) \vee (a_1 \wedge \neg a_3 \wedge a'_3 = a_3 \wedge a'_1) \vee (a_1 \wedge a_3 \wedge a'_3 \wedge a'_2) \vee \\
& (a_2 \wedge \neg a'_3 \wedge a'_0) \vee (a_2 \wedge a'_3 = a_3 \wedge a'_2)) \wedge \\
& ((a_0 \wedge \neg a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge a_1 \wedge \neg a_2) \vee (\neg a_0 \wedge \neg a_1 \wedge a_2)) \wedge \\
& ((a'_0 \wedge \neg a'_1 \wedge \neg a'_2) \vee (\neg a'_0 \wedge a'_1 \wedge \neg a'_2) \vee (\neg a'_0 \wedge \neg a'_1 \wedge a'_2))
\end{aligned}$$

The resulting abstracted Kripke structure is displayed in Fig. 12, and it is trivial to see from the graphic representation that  $\mathbf{AG}(\neg 10 \vee \neg \text{odd}(y))$  holds, because this formula is equivalent to  $\mathbf{AG}(\neg a_0 \vee \neg a_3)$  and the Kripke structure in Fig. 12 simulates the concrete system from Fig. 11 by construction.  $\square$

**Exercise. 9.** 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   }

```

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

$$\alpha : \mathbb{N}_0 \rightarrow \mathbb{N}_0; \quad x \mapsto \lceil \log_2 x \rceil$$

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:

- (i) Specify initial condition  $\mathcal{I}$  and transition formula  $\mathcal{R}$  of the concrete program fragment above.
- (ii) Now use the abstraction  $a_1 = \alpha(x), a_2 = \alpha(y)$ . and calculate the abstracted formulas  $[\mathcal{I}]$  and  $[\mathcal{R}]$ .
- (iii) 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.

□

### 5.7 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  $\phi'$  *approximates*  $\phi$  if  $\phi \Rightarrow \phi'$ .

**Definition 5.14** Let  $\phi$  a predicate in negation normal form with free variables in  $V = \{x_1, x_2, \dots\}$ . Given an abstraction  $a_i = e_i(x_1, x_2, \dots), i = 1, 2, \dots$ , the *approximation of  $\phi$*  is denoted by  $\mathcal{A}(\phi)$ .  $\mathcal{A}(\phi)$  has free variables in  $\{a_1, a_2, \dots\}$  and is defined inductively by the following rules:

- (i) If  $\phi$  is an atomic proposition<sup>4</sup>, then  $\mathcal{A}(\phi) =_{\text{def}} [\phi]$ .
- (ii) If  $\neg\phi$  is a negated atomic proposition, then  $\mathcal{A}(\neg\phi) =_{\text{def}} [\neg\phi]$ .
- (iii)  $\mathcal{A}(\phi_1 \wedge \phi_2) =_{\text{def}} \mathcal{A}(\phi_1) \wedge \mathcal{A}(\phi_2)$
- (iv)  $\mathcal{A}(\phi_1 \vee \phi_2) =_{\text{def}} \mathcal{A}(\phi_1) \vee \mathcal{A}(\phi_2)$
- (v)  $\mathcal{A}(\exists x : \phi) =_{\text{def}} \exists a : \mathcal{A}(\phi)$
- (vi)  $\mathcal{A}(\forall x : \phi) =_{\text{def}} \forall a : \mathcal{A}(\phi)$

□

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

**Theorem 5.15** *Let  $\phi$  a predicate in negation normal form with free variables in  $V = \{x_1, x_2, \dots\}$ . Given an abstraction  $a_i = e_i(x_1, x_2, \dots), i = 1, 2, \dots$ , the lifted version of  $\phi$  implies its approximated version, i. e.,*

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

**Proof.** The proof is performed by structural induction over the formula  $\phi$ .

**Step 1.** If  $\phi$  is atomic or the negation of an atom,  $\mathcal{A}(\phi) = [\phi]$ , so there is nothing to prove.

**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

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

**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  $i = 1, 2, \dots$  and that  $\phi = \phi(x, x_1, x_2, \dots)$ . Then

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

**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.  $\square$

**Theorem 5.16** *Given a Kripke structure  $K = (S, S_0, R, L)$  with variables in  $V = \{x_1, x_2, \dots\}$ , initial condition  $\mathcal{I}$  and transition formula  $\mathcal{R}$ . Given an abstraction  $a_i = e_i(x_1, x_2, \dots), i = 1, 2, \dots$ . Let  $K' = (S', S'_0, R', L')$  denote the Kripke structure with variables  $\{a_1, a_2, \dots\}$ , initial condition  $\mathcal{A}(\mathcal{I})$  and transition relation  $\mathcal{A}(\mathcal{R})$ . Then*

$$K \preceq K'$$

**Proof.** Let  $K''$  denote the abstracted Kripke structure with variables  $\{a_1, a_2, \dots\}$ , initial condition  $[\mathcal{I}]$  and transition formula  $[\mathcal{R}]$ . From Theorem 5.12 and Theorem 5.4 we know that  $K''$  simulates  $K$ . From Theorem 5.15 we know that  $[\mathcal{I}] \Rightarrow \mathcal{A}(\mathcal{I})$  and  $[\mathcal{R}] \Rightarrow \mathcal{A}(\mathcal{R})$ . Now Theorem 5.8 implies that  $K'$  simulates  $K''$ . Since  $\preceq$  is transitive, the theorem follows.  $\square$

**Exercise. 10.** Given a Kripke structure  $K = (S, S_0, R, L)$  we use the following notation:

- $K_s =_{\text{def}} (S, \{s\}, R, L)$  for  $s \in S$
- $s_0 \preceq s_1 \equiv_{\text{def}}$  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**

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

□

## 6 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 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.

### 6.1 Abstract Syntax of Timed State Machines

*Timed State Machines*  $s$  consist of *locations*  $\ell \in \text{Loc}(s)$  (also called *control states*) and *transitions*

$$\tau = (\ell, p, g, \alpha, \ell') \in \Sigma(s) \subseteq L(s) \times P \times G \times A \times L(s)$$

connecting source and target locations  $\ell$  and  $\ell'$ , 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  $g \in \text{Bexpr}(V)$  denotes the guard condition of  $\tau$  which is a Boolean expression over symbols from  $V$ . For timer

symbols  $t \in T$  occurring in  $g$  we only allow Boolean conditions  $\text{elapsed}(t, c)$  with constants  $c$ . Intuitively speaking,  $\text{elapsed}(t, c)$  evaluates to **true** if at least  $c$  time units have passed since  $t$ 's most recent reset.

Transition component  $\alpha \in A = \mathbb{P}(V \times \text{Expr}(V))$  denotes a set of value assignments to variables in  $V$ , according to expressions from  $\text{Expr}(V)$ . For a pair  $a = (v, e) \in A$ ,  $\text{var}(a) =_{\text{def}} v$  and  $\text{expr}(a) =_{\text{def}} e$  denote the projections on variable and expression, respectively. For timer symbols  $t \in T$  only resets  $(t, \text{reset})$  are allowed. A transition without accompanying assignments is associated with an empty set  $\alpha = \emptyset$ . Function

$$\omega : L_s \rightarrow \mathbb{P}(\Sigma(s)); \ell \mapsto \{(\bar{\ell}, p, g, \alpha, \ell') \in \Sigma(s) \mid \bar{\ell} = \ell\}$$

maps locations to their outgoing transitions. Each state machine  $s$  has a specific *start location*  $\text{start}(s)$ . Exactly one transition must leave  $\text{start}(s)$ , and the guard of this transition has to be **true**.

The *parallel composition* of timed state machines  $s_1, \dots, s_n$  operating over the same set  $V$  of variables is denoted by

$$\parallel_{i=1, \dots, n} s_i$$

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.

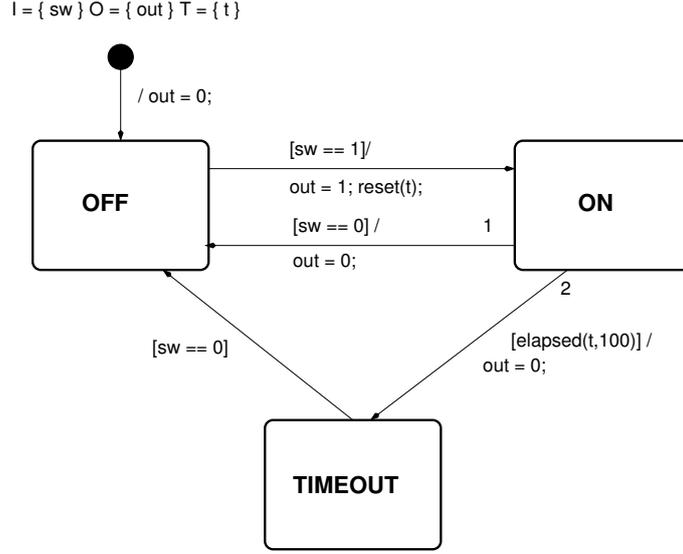
**Example 6.1** Fig. 13 shows an example of a simple switching mechanism involving a timer  $t$ : The start location is marked by the black bullet. Initially, the device controlled by this mechanism is switched off by setting the control output  $\text{out}$  to 0. If the switch  $\text{sw}$  is set to 1 then the device is switched on by means of output  $\text{out} = 1$ . A timer is set, so that the device is automatically switched off after 100 time units. In that case, the input switch  $\text{sw}$  has to be reset first, before the device can be switched on again. Otherwise, if the switch  $\text{sw}$  is reset to 0 before the timer elapses, the device is switched off at once and switched on again as soon as  $\text{sw} = 1$ .  $\square$

## 6.2 Semantics of Timed State Machines

The semantics of timed state machines is based on *timed state transition systems*  $TSTS = (S, S_0, R)$ : The state space  $S$  consists of valuation functions  $s : L \cup V \rightarrow D$  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  $\ell$ ,  $s(\ell) \in \mathbb{B}$ ,  $s(\ell) = \text{true}$  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

$$\hat{t} = z - z_0$$

Fig. 13. Timed state machine  $s$  for switch with timeout.

or, equivalently,

$$\frac{d\hat{t}}{dz} = 1$$

which will occur in the invariants introduced below, which are part of the transition relation.

**Example 6.2** For the example from Fig. 13, this results in the following initial state:

$$S_0 = \{ s \in S \mid s \models I \}$$

$$I \equiv \text{start}(s) \wedge \hat{t} = 0 \wedge \text{INV}$$

$$\text{INV} \equiv (\text{start}(s) \vee \text{OFF} \vee \text{ON} \vee \text{TIMEOUT}) \wedge$$

$$\neg(\text{start}(s) \wedge \text{OFF}) \wedge \neg(\text{start}(s) \wedge \text{ON}) \wedge \neg(\text{start}(s) \wedge \text{TIMEOUT}) \wedge$$

$$\neg(\text{OFF} \wedge \text{ON}) \wedge \neg(\text{OFF} \wedge \text{TIMEOUT}) \wedge \neg(\text{ON} \wedge \text{TIMEOUT}) \wedge \frac{d\hat{t}}{dz} = 1$$

□

Transitions are classified as

- *discrete transitions*
- *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  $\alpha = \{a_1, \dots, a_k\}$  is defined as

$$\begin{aligned} \epsilon(\alpha) \equiv_{\text{def}} & (\forall a \in \alpha \wedge \text{var}(a) \in V - T : \text{var}(a)' = \text{expr}(a)) \wedge \\ & (\forall a \in \alpha \wedge \text{var}(a) \in T : \text{var}(a)' = \hat{t}) \end{aligned}$$

A state machine transition  $\tau = (\ell_0, p, g, \alpha, \ell_1)$  may be *triggered* (or, synonymously, it may *fire*) if

$$\text{trigger}_s(\ell_0, p, g, \alpha, \ell_1) \equiv_{\text{def}} \ell_0 \wedge g \wedge (\forall (\ell_0, \bar{p}, \bar{g}, \bar{\alpha}, \bar{\ell}_1) \in \omega_s(\ell_0) : \bar{p} \geq p \vee \neg \bar{g})$$

holds. This means that for  $\tau$  to fire,  $s$  must reside in location  $\ell_0$ ,  $\tau$ 's guard condition has to evaluate to **true** and no higher-priority transition emanating from  $\ell_0$  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  $\alpha$  is defined by the set of left-hand side variables and timers that are changed by this action:

$$W(\alpha) =_{\text{def}} \{\text{var}(a) \mid a \in \alpha\}$$

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  $\parallel_{i=1, \dots, n} s_i$  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  $\text{trigger}_D$  is defined as follows:

$$\text{trigger}_D \equiv_{\text{def}} \exists i \in \{1, \dots, n\}, \tau \in \Sigma(s_i) : \text{trigger}_{s_i}(\tau)$$

The *invariant*  $\text{Inv}$  states that

- every state machine may be in at most one location at time,
- every variable only takes values in its specified domain,
- the current time behaves like an ideal clock.

$$\text{Inv} \equiv_{\text{def}}$$

$$(\forall i \in \{1, \dots, n\}, \ell_0, \ell_1 \in \text{Loc}(s_i) : \ell_0 \wedge \ell_1 \Rightarrow \ell_0 = \ell_1) \wedge$$

$$(\forall v \in V : v \in D_v) \wedge$$

$$\frac{d\hat{t}}{dz} = 1$$

Components  $\Phi_D$  and  $\Phi_T$  denote the discrete and delay transition aspects of the complete transition relation  $\Phi$ , respectively: if  $\text{trigger}_D$  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

$$\begin{aligned}\Phi_D \equiv_{\text{def}} & (\hat{t}' = \hat{t}) \wedge (\forall v \in I : v' = v) \wedge \\ & (\forall i \in \{1, \dots, n\}, \tau \in \Sigma(s_i) : \text{trigger}(\tau) \Rightarrow \epsilon(\tau)) \wedge \\ & (\forall v \in V - I : \text{written}(v) \vee v' = v)\end{aligned}$$

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

$$\text{written}(v) \equiv_{\text{def}} (\exists i \in \{1, \dots, n\}, \tau \in \Sigma(s_i) : \text{trigger}(\tau) \wedge v \in W(\tau))$$

The delay component  $\Phi_T$  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.

$$\begin{aligned}\Phi_T \equiv_{\text{def}} & (\hat{t}' > \hat{t}) \wedge \\ & (\forall i \in \{1, \dots, n\}, \ell \in \text{Loc}(s_i) : \ell' \Leftrightarrow \ell) \wedge \\ & (\forall v \in V - I : v' = v) \wedge \\ & (\forall i \in \{1, \dots, n\}, (\ell_0, p, g, \alpha, \ell_1) \in \Sigma(s_i) : \\ & \quad (\exists \bar{g} \in \text{Bexpr}, t \in T, c \in \mathbb{N} : g \equiv \bar{g} \wedge \text{elapsed}(t, c)) \Rightarrow \\ & \quad (\hat{t}' \leq c + t \vee \hat{t} \geq c + t))\end{aligned}$$

**Example 6.3** For the example from Fig. 13, this results in the following transition relation:

$$\begin{aligned}R \equiv & \text{INV} \wedge \text{INV}' \wedge ((\text{start}(s) \wedge \text{sw}' = \text{sw} \wedge t' = t \wedge \hat{t}' = \hat{t} \wedge \text{out}' = 0 \wedge \text{OFF}') \vee \\ & (\text{OFF} \wedge \text{sw} = 0 \wedge \hat{t}' > \hat{t} \wedge \text{out}' = \text{out} \wedge t' = t \wedge \text{OFF}') \vee \\ & (\text{OFF} \wedge \text{sw} = 1 \wedge \text{sw}' = \text{sw} \wedge \hat{t}' = \hat{t} \wedge \text{out}' = 1 \wedge t' = \hat{t} \wedge \text{ON}') \vee \\ & (\text{ON} \wedge \text{sw} = 1 \wedge \hat{t}' > \hat{t} \wedge (\hat{t} - t) < 100 \wedge (\hat{t}' - t) \leq 100 \wedge \text{out}' = \text{out} \wedge t' = t \wedge \text{ON}') \vee \\ & (\text{ON} \wedge \text{sw} = 1 \wedge (\hat{t} - t) \geq 100 \wedge \hat{t}' = \hat{t} \wedge \text{sw}' = \text{sw} \wedge \text{out}' = 0 \wedge t' = t \wedge \text{TIMEOUT}') \vee \\ & (\text{ON} \wedge \text{sw} = 0 \wedge \hat{t}' = \hat{t} \wedge \text{sw}' = \text{sw} \wedge \text{out}' = 0 \wedge t' = t \wedge \text{OFF}') \vee \\ & (\text{TIMEOUT} \wedge \text{sw} = 1 \wedge \hat{t}' > \hat{t} \wedge t' = t \wedge \text{out}' = 0 \wedge \text{TIMEOUT}') \vee \\ & (\text{TIMEOUT} \wedge \text{sw} = 0 \wedge \hat{t}' = \hat{t} \wedge \text{sw}' = \text{sw} \wedge \text{out}' = 0 \wedge t' = t \wedge \text{OFF}'))\end{aligned}$$

□

**Exercise. 11.** Apply the concept of predicate abstraction introduced in Section 5.6 in order to prove that the sample model displayed in Fig. 13 satisfies the properties

- $\mathbf{AG}(\neg \text{ON} \vee (\hat{t} - t \leq 100))$
- $\mathbf{A}(\mathbf{G}(\text{sw} = 1) \Rightarrow \mathbf{F}(\text{ON} \wedge (\hat{t} - t) > 50))$

To this end, proceed as follows:

- (i) Perform different abstractions for each of the two properties.
- (ii) For each property,
  - (a) Define the relevant auxiliary variables  $a_i$  and the associated abstraction expressions  $e_i(\dots \text{concrete variables} \dots)$ .
  - (b) Lift the initial state predicate  $I$  defined above to its abstracted predicate  $[I]$ , as defined in Definition 5.11.
  - (c) Lift the transition relation predicate  $R$  defined above to its abstracted version  $[R]$ .
  - (d) From  $[I]$  and  $[R]$ , formally derive the Kripke structure of the abstracted system.
  - (e) Evaluating the Kripke structure, give an informal argument why the property is satisfied.

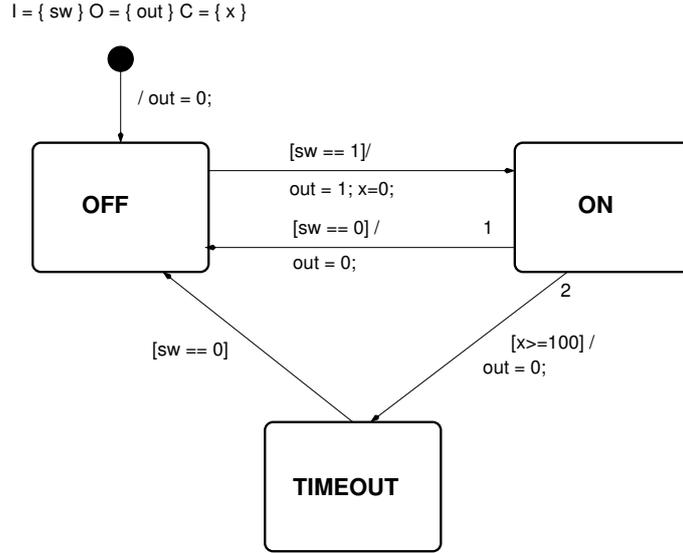
□

### 6.3 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 parallelism 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 assignments: 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.

Fig. 14. Timed state machine  $s$  with clock instead of timer variable.

#### 6.4 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 5. Given a timed state machine  $s$  with timers  $t_i \in T$  and current time  $\hat{t}$  the auxiliary variables

$$x_i(\hat{t}, t_i) =_{\text{def}} (\hat{t} - t_i), \quad t_i \in T$$

are called clock variables; let  $C$  denote the set of all these  $x_i$ . 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 5.2. Then, if  $K$  denotes the Kripke structure associated with  $s$ , it is easy to see that  $K/\sim$  is bisimilar to  $K$ .

**Example 6.4** The timed state machine shown in Fig. 13 and described in Example 6.3 can be modelled with clocks instead of timer variables as shown in Fig. 14: instead of timer variable  $t \in T$  we introduce a clock  $x$ . The  $\text{reset}(t)$  command is transformed into a reset of the clock to zero. The  $\text{elapsed}(t,c)$  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 6.3:

$$S_0/\sim = \{s \in S/\sim \mid s \models I/\sim\}$$

$$I/\sim \equiv \text{start}(s) \wedge \hat{t} = 0 \wedge \text{INV}/\sim$$

$$\text{INV}/\sim \equiv (\text{start}(s) \vee \text{OFF} \vee \text{ON} \vee \text{TIMEOUT}) \wedge$$

$$\neg(\text{start}(s) \wedge \text{OFF}) \wedge \neg(\text{start}(s) \wedge \text{ON}) \wedge \neg(\text{start}(s) \wedge \text{TIMEOUT}) \wedge$$

$$\neg(\text{OFF} \wedge \text{ON}) \wedge \neg(\text{OFF} \wedge \text{TIMEOUT}) \wedge \neg(\text{ON} \wedge \text{TIMEOUT}) \wedge$$

$$\frac{d\hat{t}}{dz} = 1 \wedge \frac{dx}{dz} = 1$$

$$R/\sim \equiv \text{INV}/\sim \wedge \text{INV}'/\sim \wedge ((\text{start}(s) \wedge \text{sw}' = \text{sw} \wedge x' = x \wedge \hat{t}' = \hat{t} \wedge \text{out}' = 0 \wedge \text{OFF}') \vee$$

$$(\text{OFF} \wedge \text{sw} = 0 \wedge \hat{t}' > \hat{t} \wedge \text{out}' = \text{out} \wedge x' = x + \hat{t}' - \hat{t} \wedge \text{OFF}') \vee$$

$$(\text{OFF} \wedge \text{sw} = 1 \wedge \text{sw}' = \text{sw} \wedge \hat{t}' = \hat{t} \wedge \text{out}' = 1 \wedge x' = x \wedge \text{ON}') \vee$$

$$(\text{ON} \wedge \text{sw} = 1 \wedge \hat{t}' > \hat{t} \wedge x' = x + \hat{t}' - \hat{t} \wedge x < 100 \wedge x' \leq 100 \wedge \text{out}' = \text{out} \wedge x' = x \wedge \text{ON}') \vee$$

$$(\text{ON} \wedge \text{sw} = 1 \wedge x \geq 100 \wedge \hat{t}' = \hat{t} \wedge \text{sw}' = \text{sw} \wedge \text{out}' = 0 \wedge x' = x \wedge \text{TIMEOUT}') \vee$$

$$(\text{ON} \wedge \text{sw} = 0 \wedge \hat{t}' = \hat{t} \wedge \text{sw}' = \text{sw} \wedge \text{out}' = 0 \wedge x' = x \wedge \text{OFF}') \vee$$

$$(\text{TIMEOUT} \wedge \text{sw} = 1 \wedge \hat{t}' > \hat{t} \wedge x' = x + \hat{t}' - \hat{t} \wedge \text{out}' = 0 \wedge \text{TIMEOUT}') \vee$$

$$(\text{TIMEOUT} \wedge \text{sw} = 0 \wedge \hat{t}' = \hat{t} \wedge \text{sw}' = \text{sw} \wedge \text{out}' = 0 \wedge t' = t \wedge \text{OFF}')$$

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.  $\square$

## References

- [1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- [2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
- [3] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Wiley, Teubner, Stuttgart, 2 edition, 1987.

## A Structural Induction

In this section the principle of *structural induction* is introduced. The material is taken from [3, pp. 8].

**Definition A.1** [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

- (i)  $B \subseteq A$
- (ii) If  $a_1, \dots, a_n \in A$  and  $((a_1, \dots, a_n), a) \in r$  for some constructor  $r \in K$ , then  $a \in A$ .

**Theorem A.2 (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*

- (i) Induction basis.  $P(x)$  holds for all  $x \in B$ .
- (ii) Induction step. *If  $P(a_i), i = 1, \dots, n$  holds for  $a_1, \dots, a_n \in A$  (induction hypothesis) and  $((a_1, \dots, a_n), a) \in r$  for some constructor  $r \in K$ , then  $P(a)$  also holds.*

*Then  $P(a)$  holds for all  $a \in A$ .* □