8 Test Derivation from Timed Automata

Laura Brandán Briones[1] and Mathias Röhl[2]

[1] University of Twente
brandanl@cs.utwente.nl
[2] University of Rostock
mroehl@informatik.uni-rostock.de

8.1 Introduction

A real-time system is a discrete system whose state changes occur in real-numbered time [AH97]. For testing real-time systems, specification languages must be extended with constructs for expressing real-time constraints, the implementation relation must be generalized to consider the temporal dimension, and the data structures and algorithms used to generate tests must be revised to operate on a potentially infinite set of states.

There are various formalisms that use fictitious clocks for expressing timing constraints. These simplify reasoning about time by recording the timing of events with finite precision only and thereby approximate precise timing of activities. The set of nonnegative integers could be used as a time domain, with the restriction that the sequence of integer times must be non-decreasing. Behavior on a discrete time scale could be modeled with ordinary finite automata by adding a distinguished *tick* event to the set of its actions.

In dense time domains, which could be sub-domains of $\mathbb{Q}^{\geq 0}$ or $\mathbb{R}^{\geq 0}$, events may occur at different time points that lay arbitrarily close together. Detecting arbitrarily small variations would require infinite test cases. However, if two events may occur on different times but for an observer their ordering makes no difference for testing purposes these events may be considered to take place at the same point in time. Henzinger et al. showed that the digitization of clocks allows to distinguish all systems which are distinguishable in the dense time domain if the system can be modeled as a timed transition system [HMP92].

We start with a general introduction to timed automata and associated concepts. The main part presents three different techniques for the generation of real-time black-box conformance tests from timed automata with a dense time domain. The first approach allows for testing (a subclass of) nondeterministic timed automata, the second one concentrates on the exhaustive testing of deterministic timed automata, while the last approach facilitates the testing of deterministic timed automata with silent transitions. An automatic light switch is used as a running example for specifications and test suite derivation.

8.2 Timed Automata

Timed automata extend finite state automata with a finite set of clocks over a dense time domain [AD94]. All clocks increase monotonically at a uniform rate,

M. Broy et al. (Eds.): Model-Based Testing of Reactive Systems, LNCS 3472, pp. 201-231, 2005.

and measure the amount of time that has elapsed since they were started or reset. The choice of the next state of a timed automaton depends, in addition to the kind of an input symbol, on the occurrence time of the input symbol relative to the occurrence of previously read symbols. Each transition of the system may reset some of the clocks and have an associated enabling condition which is a constraint on the values of the clocks. A transition can be taken only if the current clock values satisfy its enabling condition. Timing constraints on clocks may be expressed by the following syntax.

Definition 8.1. For a set C of clock variables, the set $\Phi(C)$ of **clock constraints** φ, where $c \in C$ and $k \in \mathbb{Q}^{\geq 0}$, is defined inductively by

$$\varphi \stackrel{def}{=} c < k \mid c > k \mid c \leq k \mid c \geq k \mid \varphi_1 \wedge \varphi_2$$

Often, $c = k \stackrel{def}{=} c \leq k \wedge c \geq k$ and true $\stackrel{def}{=} 0 \leq c$ are used as abbreviations.

Definition 8.2. A **timed automaton** $\mathscr{A}$ is a tuple $\langle S, S_0, \Sigma, C, Inv, E \rangle$, where

- S is a finite set of locations
- $S_0 \subseteq S$ is a set of initial locations
- Σ is a finite alphabet that denotes the set of actions
- C is a finite set of clocks
- $Inv : S \to \Phi(C)$ associates a clock invariant to each location
- $E \subseteq S \times \Sigma \times \Phi(C) \times 2^C \times S$ gives the set of transitions [Alu99].

A transition $(s, a, \varphi, \lambda, s') \in E$ represents a change of location from $s \in S$ to $s' \in S$ on symbol $a \in \Sigma$. The clock constraint (guard) $\varphi \in \Phi$ specifies when the transition is enabled, and the set $\lambda \subseteq C$ gives the set of clocks to be reset when this transition is taken. Clock invariants constrain how long the automaton is allowed to stay in a certain location.

Example. We adopt the automatic Light Switch from Springintveld et al. as an example [SVD01]. The Light Switch can be specified by a timed automaton $\mathscr{A}$, with

- $S = \{s_0, s_1\}$
- $S_0 = \{s_0\}$
- $\Sigma = \{on, off\}$
- $C = \{c\}$
- $Inv(s_0) = \text{true}$, $Inv(s_1) = c \leq 5$
- $E = \{(s_0, on, \text{true}, \{c\}, s_1), (s_1, on, c < 5, \{c\}, s_1), (s_1, off, c = 5, \emptyset, s_0)\}$

Its behavior can be explained as follows. The state of the system in which the light is off is represented by s_0, and the state s_1 represents the situation where the light is on. The light can be turned on by pushing the *on* button. After five time units the switch turns itself *off*. Before that happens, the *on* button may be pushed again, which will leave the light on (cf. Figure 8.2).

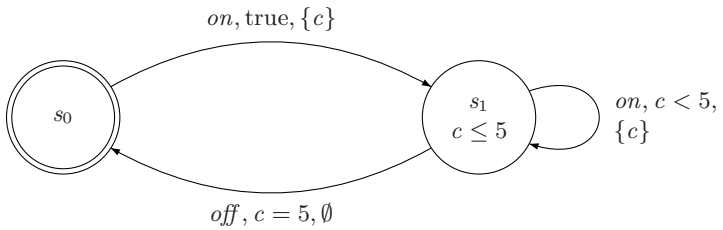


Fig. 8.1. A timed automaton specification of an automatic Light Switch

Remark 8.3. Timed automata were introduced by Alur and Dill [AD94] as a generalization of finite-state machines over infinite words [Tho90]. We only consider timed automata without acceptance conditions which are usually referred to as **timed safety automata** [HNSY92]. An introduction to acceptance is given in Section 19.2, whereas a discussion of acceptance conditions in the context of timed automata can be found elsewhere [HKWT95].

The behavior of a timed automaton $\mathscr{A}$ depends on both its current location and the actual values of all its clocks.

Definition 8.4. A **clock valuation** over a set of clocks C is a map ν that assigns to each clock $c \in C$ a value in $\mathbb{R}^{\geq 0}$. With $V(C)$ we denote the set of clock valuations over C. For $d \in \mathbb{R}^{\geq 0}$, $\nu + d$ denotes the clock interpretation which maps every clock c to the value $\nu(c) + d$. For $\lambda \subseteq C$, $\nu[\lambda := 0]$ denotes the clock interpretation for C which assigns 0 to each $c \in \lambda$, and agrees with ν over the rest of the clocks.

A labeled transition system $\mathscr{M}$ with uncountably many states can be used to define the possible behavior of a timed automata $\mathscr{A}$. A state of $\mathscr{M}$ has to be a pair $\langle s, \nu \rangle$ such that s is a location of $\mathscr{A}$ and ν is a clock valuation for C satisfying invariant $Inv_{\mathscr{A}}(s)$. Transitions of $\mathscr{M}$ represent either an elapse of time or a transition of $\mathscr{A}$.

Definition 8.5. The **semantics** of a timed automaton $\mathscr{A}$ is given by the LTS $\mathscr{M} = \langle Q, Q_0, L, \rightarrow \rangle$, where

- $Q = \{\langle s, \nu \rangle \in S_{\mathscr{A}} \times V(C_{\mathscr{A}}) \mid \nu \models Inv_{\mathscr{A}}(s)\}$
- $Q_0 \subseteq Q$ with $\langle s, \nu \rangle \in Q_0$ iff $s \in S_{0_{\mathscr{A}}}$ and $\nu(c) = 0$ for all clocks $c \in C_{\mathscr{A}}$
- $L = \Sigma_{\mathscr{A}} \cup \mathbb{R}^{\geq 0}$
- $\rightarrow \subseteq Q \times L \times Q$, which could be either
  - $(\langle s, \nu \rangle, d, \langle s, \nu + d \rangle)$ iff $d \in \mathbb{R}^{\geq 0}$ and for all $0 \leq d' \leq d$, $\nu + d' \models Inv_{\mathscr{A}}(s)$
  - $(\langle s, \nu \rangle, a, \langle s', \nu[\lambda := 0] \rangle)$ iff $(s, a, \varphi, \lambda, s') \in E_{\mathscr{A}}$ and $\nu \models \varphi$

Due to dense-time clocks, the transition system $\mathscr{M}$ for a timed automaton $\mathscr{A}$ has infinitely many states and operates on infinitely many symbols. Analysis of

safety requirements of real-time systems can be formulated as reachability problems for timed automata. Since the transition system $\mathscr{M}$ for a timed automaton $\mathscr{A}$ is infinite, reachability analysis constructs a quotient called the **region automaton** by partitioning the uncountable state space into finitely many regions [Alu99].

A timed automaton can be seen as accepting (or generating) timed words and thereby defining a timed language. Two timed automata are said to be equivalent if they accept the same timed language.

Definition 8.6. A **timed word** over an alphabet Σ is a finite sequence $(a_1, t_1) \ldots (a_n, t_n)$ of symbols $a_i \in \Sigma$ paired with nonnegative real numbers $t_i \in \mathbb{R}^{\geq 0}$ that are nondecreasing $(\forall\, i < n.t_i < t_{i+1})$. A **timed language** over Σ is a set of timed words over Σ.

Remark 8.7. Alur and Dill showed that a Büchi automaton (called region automaton) can be constructed that accepts exactly the set of untimed words that are consistent with the timed words accepted by a timed automaton [AD94]. The construction of the region automaton is PSPACE-complete.

Remark 8.8. Alur and Dill showed the language inclusion problem to be undecidable for nondeterministic timed automata but solvable in PSPACE for deterministic timed automata. The problem of deciding the emptiness of the language of a given timed automaton is PSPACE-complete for deterministic timed automata [AD94].

Deterministic timed automata form an important subclass of timed automata that are strictly less expressive than nondeterministic timed automata [AD94]. For timed automata to be deterministic multiple transitions starting at the same location with the same label are only allowed if their clock constraints are mutually exclusive. Thus, at most one of the transitions with the same action is enabled at a given time.

Definition 8.9. A timed automaton $\langle S, S_0, \Sigma, C, Inv, E\rangle$ is called **deterministic** iff

- $|S_0| = 1$, and
- for all $s \in S$, for all $a \in \Sigma$, for every pair of transitions of the form $\langle s, a, \varphi_1, \lambda_1, s_1\rangle \in E$ and $\langle s, a, \varphi_2, \lambda_2, s_2\rangle \in E$, $\varphi_1 \wedge \varphi_2$ is unsatisfiable.

Definition 8.10. Timed automata with **silent transitions** are gained by extending Definition 8.2 such that for a transition $(s, a, \varphi, \lambda, s') \in E$ an action a can be in $\Sigma \cup \tau$, where $\Sigma \cap \tau = \emptyset$. A transition $(s, a, \varphi, \lambda, s')$ is called a silent transition (often called ϵ-transition) when $a = \tau$. If, in addition $\lambda = \emptyset$ then we speak of a silent transition without reset.

Remark 8.11. Whereas silent transition do not increase the expressiveness of untimed automata they strictly increase the power of timed automata. Bérard et al. showed silent transitions with clock resets that lie on a directed cycle to be responsible for this increase in expressiveness [BPDG98].

8.3 Testing Event Recording Automata

Nielsen and Skou present a technique for the automatic generation of *real-time* black-box conformance tests for *non-deterministic* systems [NS03]. They start from a determinizable class of timed automata specifications called ERA, with a dense time interpretation. The tests are generated using a coarse grained equivalence class partition of the specification.

8.3.1 Model

Event Recording Automata (ERA) were proposed by Alur, Fix and Henzinger [AFH94] as a determinizable subclass of timed automata and have language inclusion as a decidable property (like all deterministic timed automata).

Like a timed automaton [AD94], an ERA has a set of clocks, which can be used in guards (clock constrains) and be reset when an action is taken. In ERA, however, each action a is uniquely associated with a clock c_a, called the **event clock** of a. Whenever an action a is executed the event clock c_a is automatically reset. No further clock assignments are permitted. The event clock c_a thus *records* the amount of time passed since the last occurrence of a. No silent τ-actions or location invariants are permitted. These restrictions ensure determinizability [AFH94].

Definition 8.12. An **Event Recording Automaton** (ERA) $\mathscr{A}$ is a tuple $\langle S, s_0, \Sigma, E \rangle$, where

- S is a non-empty (finite) set of locations
- $s_0 \in S$ is the initial location
- Σ is a finite set of actions
- $E \subseteq S \times \Sigma \times \Phi(C) \times S$ is the set of transitions
  where
  - $C = \{c_a \mid a \in \Sigma\}$ is the set of real-valued clocks
  - $\Phi(C)$ is the set of clock constraints (or guards), these guards are generated by the syntax $\varphi ::= \gamma \mid \varphi \wedge \varphi$, where γ is a constraint of the form $c_1 \sim k$ or $c_1 - c_2 \sim k$ with: $\sim \in \{\leq, <, =, >, \geq\}$, k a non-negative integer constant, and $c_1, c_2 \in C$.

All actions are *urgent*, meaning that synchronization between two automata takes place immediately when the parties have enabled a pair of complementary actions. The complementary actions are actions by which the automata synchronize, in our cases input and output actions, denoted as _?, _! respectively. The requirement of *urgent* actions is needed because with non-urgent observable actions the synchronization delay could be unbounded.

Example. Figure 8.2 shows an ERA which describe the behavior of the automatic Light Switch. The initial location is indicated by double circle. Formally, the ERA is given by $\langle S, s_0, \Sigma, E \rangle$, where

- $S = \{s_0, s_1\}$
- s_0 is the initial state

- $\Sigma = \{on?, off!\}$
- $E = \{(s_0, on?, \text{true}, s_1), (s_1, on?, c_{on} < 5, s_1), (s_1, off!, c_{on} = 5, s_0)\}$

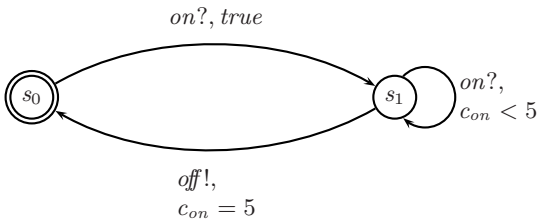


Fig. 8.2. ERA specification for an automatic Light Switch [SVD01]

The determinization procedure for ERAs is given by Alur, Fix and Henzinger [AFH94], and is conceptually a simple extension of the method used for the untimed case, only now the guards must be taken into account.

8.3.2 Symbolic Representation

Timed automata (a network of ERAs) with a dense time interpretation cannot be analyzed by finite state techniques, since the timed transition system associated with it has infinitely many states. Therefore, it must be analyzed symbolically [NS03]. Similar to the region automaton [Alu99] which partitions the state space into finitely many regions, here *zone* is used instead, in the following way.

The state of a network of timed automata is represented by a pair $\langle \overline{s}, \overline{\nu} \rangle$, where $\overline{s}$ is the vector of the automata's current location, and $\overline{\nu}$ is the vector of their current clock values. A *zone* z is a conjunction of clock constraints of the form $c_1 \sim k$ or $c_1 - c_2 \sim k$ with $\sim \in \{\leq, <, =, >, \geq\}$ or equivalently, the solution set to these constraints. A symbolic state $[\overline{s}, z]$ represents the (infinite) set of states $\{\langle \overline{s}, \overline{\nu} \rangle \mid \overline{\nu} \in z\}$.

Example. The graphical view of the symbolic state $[s_1, z]$ for the ERA of example 8.3.1, with $z = c_{on} < 5$ is shown in Figure 8.3.

Zones can be represented and manipulated efficiently by the *difference bound matrix* (DBM) data structure [Bel57]. The use of zones allows us to compute:

- The symbolic state that results after take a transition from a given source symbolic state
- The reachable state space. Forward reachability analysis starts in the initial state $(\overline{s}_0, \overline{0})$ and computes the symbolic states that can be reached by execute an action from an exists one, or by let time pass. When a new symbolic state is included in one previously visited, no further exploration of the new state needs to take place. Forward the reachability analysis terminates when no new state can be reached

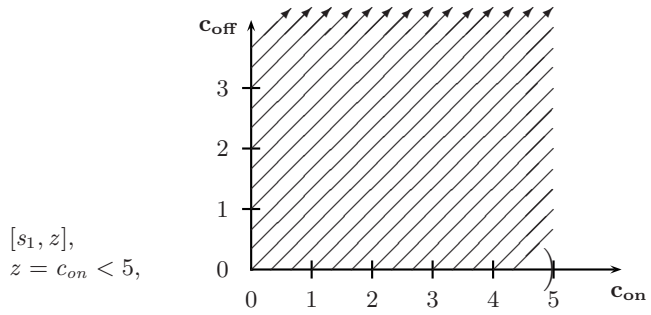


Fig. 8.3. A symbolic state and the solution set corresponding to the zone z

- Given a symbolic path to a symbolic state, a concrete timed *trace* leading to it (or a subset thereof) can be computed by propagating its constraints back along the symbolic path used to reach it, and by choosing specific time points along this *trace*

Remark 8.13. To ensure soundness of the produced tests, symbolic reachability analysis is needed to select only states for testing that are reachable, and to compute only timed *traces* that are actually part of the specification.

8.3.3 Testing

As opposed to exhaustive testing, a test *selection criterion* is used in this case (or *coverage criterion*), i.e. a rule that describe which behavior or which requirements should be tested. *Coverage* is a metric of completeness with respect to a test selection criterion.

For real-time systems it is proposed to partition the clock valuations into domains and ensure that each such domain is tested systematically.

Example. In our example of the automatic Light Switch, a partition domain for c_{on} could be as shown in Figure 8.4.

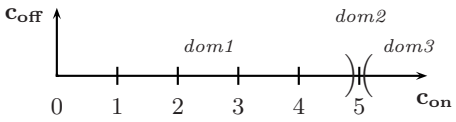


Fig. 8.4. ERA Domains Graph

The selection criterion used here is based on partition the state space of the specification into coarse equivalence classes, and require that the test suite for

each class yields a set of required observations of the implementation when it is expected to be a state in that class. Like in the Hennessy's works [HN83], the following abstract syntax is used:

(1) **after** σ **must** A,
(2) **can** σ,
(3) **after** σ **must** $\emptyset$

where $\sigma \in Act^*$ and $A \subset Act$. Informally, (1) is successful if at least one of the observations in A (called a *must* set) can be observed whenever the *trace* σ is served, (2) is successful if σ is a prefix of the observed system, and (3) is successful if this not case (i.e. σ is not a prefix). Using this notation, each class is decorated with the *simple deadlock* observations of the forms **after** ε **must** A (a *must* property), **after** a **must** $\emptyset$ (a *refusal* property), and **can** a (a *may* property) that should be satisfied in that class (this idea was taken from the testing preorder).

A test case consists of a timed *trace* which lead to a desired state in a coarse equivalence class followed by one of the *simple deadlock* observations.

Now, we present the state partitioning definition, which is used to construct the equivalence class graph. This graph is a transformation of the initial automata, which preserve all the information from it. And moreover, the equivalence class graph is what is effectively used in the test derivation process.

The State Partitioning works as follows. Let S' be a vector location in the determinized automaton, note that S' can be a set of locations of the original automaton. Therefore, this control location S' will have the clock valuations partitioned such that two clock valuations belong to the same equivalence class if and only if they enable precisely the same outgoing transitions from S', i.e. the locations are equivalent with respect to the enabled transitions.

An equivalence class is represented by a pair $[S', p]$, where S' is a set of location vectors, and p is the inequation which describe the clock constraints that must hold for that class, i.e. $[S', p]$ is the set of states $\{\langle S', \overline{\nu}\rangle \mid \overline{\nu} \in p\}$. Further, to obtain equivalence classes that are continuous convex polyhedra, and to enable the reuse of existing efficient symbolic techniques (as used in model checking), this constraint is rewritten in disjunctive normal form. Each disjunct form is treated as an equivalence class.

Definition 8.14. State Partitioning $\Psi(S')$
Let S' be a set of location vectors, $E(S')$ the set of transitions from a location in S'. If E is a set of transitions with $\Gamma(E)$ we denote the set of guards of the set E.

$$\Gamma(E) = \{\varphi \in \Phi(C) \mid \overline{s} \xrightarrow{\varphi, a} \overline{s'} \in E\}$$

Let P be a constraint over clock inequations γ composed using the logical connectives ($\wedge, \vee$, or $\neg$). DNF(P) denotes a function that rewrites constraint P to its equivalent disjunctive normal form, i.e. such that $\bigvee_i \bigwedge_j \gamma_{ij} = P$. Each conjunct in disjunctive form can be written as a guard $\varphi \in \Phi(C)$. The disjunctive normal form can be interpreted as a disjunction of guards such that $\bigvee_i \varphi_i = \bigvee_i \bigwedge_j \gamma_{ij}$. Let

$$\Psi(S') = \{P_{E'} \mid \ E' \in 2^{E(S')} \wedge P_{E'} = \bigwedge_{\varphi \in \Gamma(E')} \varphi \wedge \bigwedge_{\varphi \in \Gamma(E(S')-E')} \neg\varphi\}$$

Then, the set of guards φ_i whose disjunction equals the disjunctive normal form is denoted as GDNF, i.e,

$$\text{GDNF}(P_{E'}) = \{\varphi_i \in \Phi(C) \mid \ \bigvee_i \varphi_i = DNF(P_{E'})\}$$

and finally $\Psi_{dnf}(S')$ is:

$$\Psi_{dnf}(S') = \bigcup_{P_{E'} \in \Psi(S')} GDNF(P_{E'}).$$

To make this definition more understandable we show the next example. Using our example of the automatic Light Switch, we present the procedure for find the equivalences classes for $S' = \{s_1\}$.

Example. Let $S' = \{s_1\}$, then the transitions from S' are:

$$E(S') = \{(s_1, on?, c_{on} < 5, s_1), (s_1, off!, c_{on} = 5, s_0)\}$$

the guards of $E(S')$ are:

$$\Gamma(E(S')) = \{c_{on} < 5, c_{on} = 5\}$$

only for simplicity we will present $2^{\Gamma(E(S'))}$ instead of $2^{E(S')}$:

$$2^{\Gamma(E(S'))} = \{\emptyset, \{c_{on} < 5, c_{on} = 5\}, \{c_{on} < 5\}, \{c_{on} = 5\}\}$$

and:

$$\begin{aligned}\Psi(S') = \{&(c_{on} \geq 5) \wedge (c_{on} \neq 5), (c_{on} < 5) \wedge (c_{on} \neq 5),\\ &(c_{on} \geq 5) \wedge (c_{on} = 5), (c_{on} < 5) \wedge (c_{on} = 5)\}\end{aligned}$$

the disjunctive normal form of $\Psi(S')$ is :

$$\Psi_{dnf}(S') = \{c_{on} > 5, c_{on} < 5, c_{on} = 5, \emptyset\}$$

Then we have: $[s_1, c_{on} > 5], [s_1, c_{on} < 5]$ and $[s_1, c_{on} = 5]$ as states for our equivalence class graph.

The state space of the ERA specification is a graph of equivalence classes. A node in this graph corresponds to an equivalence class. A transition between two nodes is labeled with an action, and represents the possibility of execute an action in a state in the source node, wait some amount of time, and thereby enter in a state in the target node. The graph is constructed by start from an existing node $[S', p]$ (initially the equivalence class of the initial location), and then for each enabled action a, compute the set of locations S'' that can be entered by execute the a action from the current equivalence class. Then the partitions p' of location S'' can be computed according to Definition 8.14. Every $[S'', p']$ is then an a successor of $[S', p]$. Only equivalence classes whose constraints have solutions need to be represented. The equivalence class graph is defined inductively in the Algorithm 11.

Each equivalence class $[S', p]$ is decorated with the action sets M, C, R from the testing preorder, as it is shows in definition 8.15.

Algorithm 11 Equivalence Class Graph

input: ERA determinized specification *Spec*
output: A equivalence Class Graph

$S'_0 = \{\overline{s_0}\}$
$E = \emptyset$ // *E the set of transition*
$N = \{[S'_0, p] \mid p \in \Psi_{dnf}(S'_0) \wedge p \neq \emptyset\}$ // *N is the set of nodes*
$N' = N$ // *N' is the set of new nodes*
while $N' \neq \emptyset$ **then**
  $N''' = \emptyset$
  foreach $[S', p] \in N'$
    foreach $a \in \Sigma$:
      $S'' = \{\overline{s'} \mid \exists\ \overline{s} \in S' : \overline{s} \xrightarrow{\varphi, a} \overline{s'}\}$
      if $S'' \neq \emptyset$ **then**
        $N'' = \{[S'', p'] \mid p' \in \Psi_{dnf}(S'') \wedge p' \neq \emptyset\}$
        $E = E \bigcup \{([S', p], a, [S'', p']) \mid [S'', p'] \in N'' \wedge (p \wedge \varphi) \neq \emptyset\}$
    $N''' = N''' \bigcup N''$
  $N' = N''' - (N''' \bigcap N)$
  $N = N \bigcup N'$

Definition 8.15. Decorated Equivalence Classes
Define $\text{Must}([S', p]) = \{A \mid \exists \langle S', \overline{\nu}\rangle : \langle S', \overline{\nu}\rangle \in [S', p] : \langle S', \overline{\nu}\rangle \models \textbf{after } \epsilon \textbf{ must } A\}$
$\text{Sort}([S', p]) = \{a \mid \exists \langle S', \overline{\nu}\rangle : \langle S', \overline{\nu}\rangle \in [S', p] : \langle S', \overline{\nu}\rangle \xrightarrow{a}\}$

- $M([S', p]) = \text{Must}([S', p])$
- $C([S', p]) = \text{Sort}([S', p])$
- $R([S', p]) = \Sigma - \text{Sort}([S', p])$

where ϵ denote the empty sequence.

If σ is a timed *trace* that lead to $[S', p]$ and $A \in M([S', p])$ then: **after** σ **must** A, is a test to be passed for that class. Similarly: **after** $\sigma \cdot a$ **must** $\emptyset$, is a test to be passed if $a \in R([S', p])$, and **can** $\sigma \cdot a$ if $a \in C([S', p])$. The number of generated tests can be reduced by remove tests that are logically passed by another test, i.e. the **must** sets can be reduced to $M([S', p]) = min_{\subseteq}\text{Must}([S', p])$ (where $min_{\subseteq}(M)$ gives the set of minimal elements of M under subset inclusion), and the actions observed during the execution of a **must** test can be removed from the **may** tests, i.e. $C([S', p]) = \text{Sort}([S', p]) - \bigcup_{A \in M([S', p])} A$.

Example. The equivalence classes graph for the automatic Light Switch are shown in Figure 8.5.

The equivalence class graph preserves all timed traces of the specification, and the required deadlock information for the Hennessy test [HN83] of the specification by the M, C and R action sets is stored in each node. The non-determinism found in the original specification is therefore maintained, but is represented differently, in a way that is more convenient for test generation: a test is composed

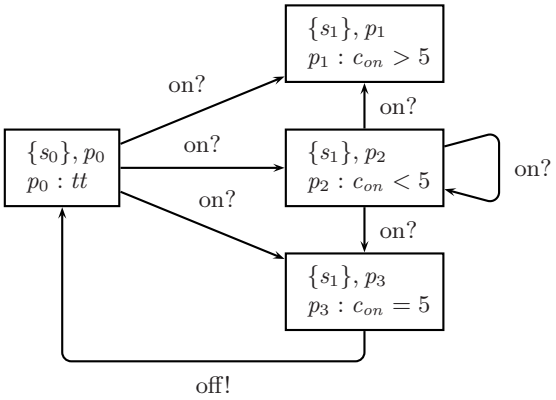


Fig. 8.5. ERA Equivalence Class Graph for the Light Switch

of a *trace* (a deadlock observation possible in the specification thereafter) and its associated verdict. This information can be simply found by following a path in the equivalence class graph.

Even the equivalence class graph have the necessary information for generate timed Hennessy tests, it also contains behavior and states not found in the specification, and use such behavior will result in irrelevant and unsound tests (in the same way as in model checking after use zones it is necessary to make a reachability analysis). To ensure soundness, only *traces* and deadlock properties actually contained in the specification should be used in a generated test. Therefore, the specification is interpreted symbolically, and the tests is generated from a representation of only the reachable states and behavior.

Algorithm 12 represents the test generation procedure. Step 1 constructs the equivalence class graph. The result of step 2 is the *symbolic reachability graph*. Nodes in this graph consist of symbolic states $[S', z/p]$ where S' is a set of location vectors, and z is a constraint characterizing a set of reachable clock valuations also in p, i.e. $z \subseteq p$. A transition represents that the target state is reachable by execute an action from the source state and then wait for some amount of time. The nodes in the reachability graph are decorated with the set M, C and R. Step 4 initializes an empty set *Tested* that contains the symbolic states from which test have to be generated so far. Steps 5 and further contain the test generation process.

This algorithm only generates tests for the *first* symbolic state that reaches a given partition, and uses the set *Tested* to ignore subsequent passes over the same partition. This ensures that all the *may*, *must*, and *refusal* properties are only generated once per partition, thus reduce the number of produced test cases.

This theory and algorithm have been implemented in a prototype tool called RTCAT. RTCAT inputs an ERA specification in AUTOGRAPH format, see [BRRdS96]. A specification may consist of several ERA operating in parallel and communicating via shared clocks and integer variables, but no silent actions (τ)

Algorithm 12 Overall Test Case Generation

input: ERA specification *Spec*
output: A complete cover set of timed Hennessy properties

```
1  Compute Spec_p = Equivalence Class Graph(Spec)
2  Compute Spec_r = Reachability Graph(Spec_p)
3  Label every [S', z/p] ∈ Spec_r with the sets M, C, R
4  Tested := ∅
5  foreach [S', z/p] ∈ Spec_r                    // traverse Spec_r
6      if ∄ z' : [S', z'/p] ∈ Tested then
7          Tested := Tested ∪ {[S', z/p]}  // enumerate tests
8              Choose ⟨s̄, v̄⟩ ∈ [S', z/p]
9              Compute a concrete timed trace σ from ⟨s̄0, 0̄⟩ to ⟨s̄, v̄⟩
10             Make Test Cases:
11                 if A ∈ M([S', p]) then after σ must A, is a relevant test
12                 if a ∈ C([S', p]) then can σ · a, is a relevant test
13                 if a ∈ R([S', p]) then after σ · a must ∅, is a relevant test
```

are allowed. The application of this technique to a realistic specification shows "promising results: the test suite is quite small, is constructed quickly, and with a reasonable memory usage" [NS03].

8.4 Testing Deterministic Timed Automaton

Springintveld, Vaandrager and D'Argenio [SVD01] showed that *exhaustive* testing of *trace equivalence* for *deterministic* timed automaton with *dense time* interpretation is theoretically possible, but quite infeasible in practice. A grid algorithm for bounded time-domain automaton is presented, which capture the real-time behaviors using finitely many points.

8.4.1 Model

The timed I/O automaton model is used here, which is a finite (untimed) automaton together with a timing annotation. This model is equivalent to the original timed automaton [AD94] with some restrictions in order to makes exhaustive test derivation feasible. A timed I/O automaton makes exhaustive test derivation feasible if it does not have silent τ-transitions, is deterministic, is input enabled and has isolated output as we will show later.

A *finite automaton* $\mathscr{A}'$[1] is a rooted labeled transition system with Q (the set of states) and E (the transition relation $\rightarrow$) finite. We will fix some useful notations and definitions. An *execution fragment* of the LTS $\mathscr{A}'$ is a finite or infinite alternating sequence $q_0\, a_1\, q_1\, a_2\, q_2 \ldots$ of states and actions of $\mathscr{A}'$ ($a_i \in L_{\mathscr{A}'}$ and $q_i \in L_{\mathscr{A}'}$), beginning with a state, and if it is finite also ending with a state,

[1] the reason why we use $\mathscr{A}'$ instead of $\mathscr{A}$ here, is only notational. Then $\mathscr{A}'$ denote a automaton and $\mathscr{A}$ will denote a timed automaton

such that for all $i > 0, q_{i-1} \xrightarrow{a_i} q_i$. An *execution* of $\mathscr{A}'$ is an execution fragment that begins with the initial state q_0 of $\mathscr{A}'$. A state q of $\mathscr{A}'$ is *reachable* if it is the last state of some finite execution of $\mathscr{A}'$. σ is a **distinguishing trace** of q and q' if it is either a *trace* of q but not of q', or the other way around (for the definition of *traces* see Appendix: Label Transition Systems). If $\delta \in E$ and $\delta = (q, a, q')$ we denote $\mathbf{src}(\delta) = q$, $\mathbf{act}(\delta) = a$ and $\mathbf{trg}(\delta) = q'$.

Definition 8.16. Let $\mathscr{B}$ be an LTS. A relation $R \subseteq Q_{\mathscr{B}} \times Q_{\mathscr{B}}$ is a *bisimulation* on $\mathscr{B}$ iff whenever $R(q_1, q_2)$, then

- $q_1 \xrightarrow{a} q_1'$ implies that there is a $q_2' \in Q_{\mathscr{B}}$ such that $q_2 \xrightarrow{a} q_2'$ and $R(q_1', q_2')$
- $q_2 \xrightarrow{a} q_2'$ implies that there is a $q_1' \in Q_{\mathscr{B}}$ such that $q_1 \xrightarrow{a} q_1'$ and $R(q_1', q_2')$

States q, q' of LTSs $\mathscr{B}$ and $\mathscr{B}'$, respectively, are *bisimilar* if there exists a bisimulation R on the disjoint union of $\mathscr{B}$ and $\mathscr{B}'$ (with arbitrary initial state) that relates q to q'. In such a case, we write : $q \simeq q'$. LTSs $\mathscr{B}$ and $\mathscr{B}'$ are *bisimilar*, notation $\mathscr{B} \simeq \mathscr{B}'$, if $q_0 \simeq q_0'$ for q_0 the initial state of $\mathscr{B}$ and q_0' the initial states of $\mathscr{B}'$.

It is well known that if $\mathscr{B}$ is deterministic, for all states q, q' of $\mathscr{B}$, $\mathscr{B} : q \simeq q'$ if and only if *traces* $(q) =$ *traces*(q'). As a consequence, two deterministic LTSs $\mathscr{B}$ and $\mathscr{B}'$ are bisimilar iff they have the same sets of traces.

Let C be a set of clocks with $c \in C$, then define $\mathbf{dom}(c) \stackrel{def}{=} J \cup \{\infty\}$, were J is a bounded interval over $\mathbb{R}$ with infimum and supremum in $\mathbb{Z}$ and $\mathbf{intv}(c) \stackrel{def}{=} dom(c) - \{\infty\}$. The **terms** over C (denoted as $T(C)$) are expressions generated by the grammar $e := c \mid k \mid e+k$, with $c \in C$ and $k \in \mathbb{Z}^{\infty}$, i.e. $\mathbb{Z} \cup \{\infty\}$. Let $F(C)$ be the boolean combinations of inequalities of the form $e \leq e'$ or $e < e'$ with $e, e' \in T(C)$. A (*simultaneous*) **assignment** over C is a function μ from C to $T(C)$, the set of all these functions is denoted as $M(C)$. If φ is a constraint over C and μ an assignment, then $\varphi[\mu]$ denotes the constraint obtained from φ by replacing each variable $c \in C$ by $\mu(c)$. Finally a **clock valuation** over C is a map ν that assigns to each clock $c \in C$ a value in its domain (this set of valuations is denoted as $V(C)$). We say that ν *satisfies* φ, notation $\nu \vDash \varphi$, if φ evaluates to true under valuation ν.

In the next definition is presented the timing annotation for a finite automaton, which is a set of clocks, a set of invariants for each state, a set of guards, which allowed the transition to be made of not, Ass the assignments for each transition, and ν_0 the initial clock valuation.

Definition 8.17. A **timing annotation** for a given finite automaton $\mathscr{A}' = \langle Q, q_0, E \rangle$ is a tuple $\mathscr{T} = \langle C, Inv, \Phi, Ass, \nu_0 \rangle$, where

- C is a finite set of clocks
- $Inv : Q \to F(C)$ associates an invariant to each state
- $\Phi : E \to F(C)$ associates a guard to each transition

- $Ass : E \to M(C)$ associates an assignment to each transition s.t. for each $\delta \in E$:

$$Inv(\mathbf{src}(\delta)) \wedge \Phi(\delta) \Rightarrow \bigwedge_{c \in C} (Ass(\delta)(c) \in dom(c)) \wedge Inv(\mathbf{trg}(\delta))[Ass(\delta)]$$

- $\nu_0 \in V(C)$ is the initial *clock valuation*. It should hold that $\nu_0 \vDash Inv(q_0)$ and, for all $c, \nu_0(c) \in \mathbb{Z}^\infty$.

Above all, we present the timed I/O automata, which, as we already say, is a finite automaton together with a timed annotation and some restrictions. These restrictions are fundamentals to prove future theorems for the discretization of the state space.

Definition 8.18. A timed I/O automaton (TIOA) is a triple $\mathscr{A} = \langle \mathscr{A}', \mathscr{T}, \mathscr{P} \rangle$, where $\mathscr{A}'$ is a finite automaton with $L_{\mathscr{A}'} \cap \mathbb{R}^{>0} = \emptyset$ (to do not confuse labels of actions with labels of time), $\mathscr{T}$ is a timing annotation for $\mathscr{A}'$ and $\mathscr{P} = (\mathcal{I}, \mathcal{O})$ is a partitioning of $L_{\mathscr{A}'}$ in input actions and output actions. The following properties must hold, for all $\delta, \delta' \in E_{\mathscr{A}'}$ and $q \in Q_{\mathscr{A}'}$:

- (Determinism) if $\mathbf{src}(\delta) = \mathbf{src}(\delta')$, $\mathbf{act}(\delta) = \mathbf{act}(\delta')$ and $\Phi(\delta) \wedge \Phi(\delta')$ is satisfiable then $\delta = \delta'$
- (Isolated outputs) if $\mathbf{src}(\delta) = \mathbf{src}(\delta')$, $\mathbf{act}(\delta) \in \mathcal{O}$ and $\Phi(\delta) \wedge \Phi(\delta')$ is satisfiable then $\delta = \delta'$
- (Input enabling) every input is always enabled within the interior of the invariant of each location and only within it
- (Progressiveness) for every state of its operational semantics ($\mathscr{OS}(\mathscr{A})$, defined as follows) there exists an infinite execution fragment that starts in this state, contains no input actions, and in which the sum of the delays diverges.

In order to not confuse, and following the previous implicit convention, in a TIOA $\mathscr{A}$ we will use S as the set of locations and Σ as the set of actions. In contrast to the associated operational semantics $\mathscr{OS}(\mathscr{A})$, where Q is the set of states and L is the set of actions.

Example. Figure 8.6 depicts the timed I/O automaton which represent the Light Switch.

The operational semantics of $\mathscr{A}$ (denoted as $\mathscr{OS}(\mathscr{A})$) is defined as the LTS $\langle Q, L, q_0, \rightarrowtail \rangle$, with Q, L and q_0 similarly as in previous Definition 8.5, and $\rightarrowtail$ being the smallest relation that satisfies the following two rules, for all $(s, \nu), (s', \nu') \in Q$, $a \in \Sigma, \delta \in E$ and $d \in \mathbb{R}^{>0}$:

- $$\frac{\delta : s \xrightarrow{a} s',\ \nu \models \Phi(\delta),\ \nu' = \nu \circ Ass(\delta)}{(s,\nu) \overset{a}{\rightarrowtail} (s',\nu')}$$

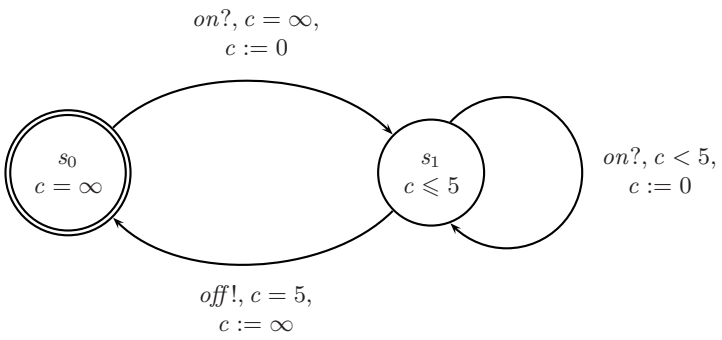


Fig. 8.6. TIOA specification for a Light Switch [SVD01]

- $$\frac{\forall\, 0 \leq d' \leq d \;:\; \nu \oplus d' \models Inv(s)}{(s,\nu) \overset{d}{\rightarrowtail} (s,\nu \oplus d)}$$

where the actions in $\mathbb{R}^{>0}$ are referred to as *time delays* and

$$(\nu \oplus d)(c) \overset{def}{=} \begin{cases} \nu(c) + d & \text{if } (\nu(c) + d) \in intv(c) \\ \infty & \text{otherwise} \end{cases}$$

The following lemma, which is a direct corollary of the definitions, gives four basic properties of the operational semantics of a timed I/O automaton.

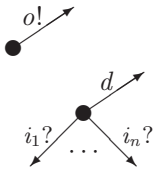
Lemma 8.19. *Let $\mathscr{A}$ be a TIOA, then*

- *$\mathscr{OS}(\mathscr{A})$ is deterministic*
- *$\mathscr{OS}(\mathscr{A})$ possesses Wang's time additivity property:*

$$q \overset{d+d'}{\rightarrowtail} q' \text{ iff } \exists\, q'' : q \overset{d}{\rightarrowtail} q'' \wedge q'' \overset{d'}{\rightarrowtail} q'$$

- *Each state of $\mathscr{OS}(\mathscr{A})$ has either*
  - *a single outgoing transition labeled with an output action, or*
  - *both outgoing delay transitions and outgoing input transitions (one for each input action), but no outgoing output transitions*

  States of the second type are called ***stable***



- *For each state $q \in Q_{\mathscr{OS}(\mathscr{A})}$, there exists a unique finite sequence of output actions σ and a unique stable state q' such that $q \overset{\sigma}{\rightarrowtail} q'$.*

8.4.2 Discretization

The construction of a finite subautomaton used, for the discretization of the state space, is based on the fundamental concept of a *region* due to Alur and

Dill [AD94]. The key idea behind the definition of a region is that, even though the number of states of the LTS $\mathscr{OS}(\mathscr{A})$ is infinite, not all of these states are distinguishable via constraints. If two states corresponding to the same location agree on the internal parts of all the clock values, and also in the order of the fractional parts of all the clocks, then these two states cannot be distinguished.

Definition 8.20. The equivalence relation $\cong$ over the set $V(C)$ of clocks valuations is given by: $\nu \cong \nu'$ if and only if $\forall\ c, c' \in C$:

- $\nu(c) = \infty$ iff $\nu'(c) = \infty$
- if $\nu(c) \neq \infty$ then $\lfloor \nu(c) \rfloor = \lfloor \nu'(c) \rfloor$ and ($fract(\nu(c)) = 0$ iff $fract(\nu'(c)) = 0$)
- if $\nu(c) \neq \infty \neq \nu(c')$ then $fract(\nu(c)) \leq fract(\nu(c'))$ iff $fract(\nu'(c)) \leq fract(\nu'(c'))$

where $\forall\, k \in \mathbb{R}$ (in this case a valuation of a clock), $\lfloor k \rfloor$ denotes the largest number in $\mathbb{Z}$ that is not greater than k, and $\lceil k \rceil$ denotes the smallest number in $\mathbb{Z}$ that is not smaller than k and $fract(k)$ is the fractional part of k (so $fract(k) = k - \lfloor k \rfloor$).

A *region* is an equivalence class of valuations induced by $\cong$.

Example. Figure 8.7 shows the 11 regions of the c_{on} clock from the Light Switch.

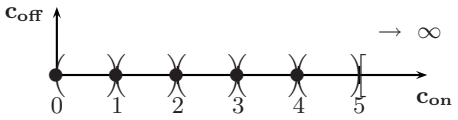


Fig. 8.7. Regions of the c_{on} clock from the Light Switch example

Lemma 8.21. *For all clock constraints φ:*

$$\textit{if}\ \ \nu \cong \nu'\ \ \textit{then}\ \ \nu \vDash \varphi\ \textit{iff}\ \nu' \vDash \varphi$$

The equivalence relation $\cong$ on the clock valuations of a TIOA can be extended to an equivalence relation on states, by defining

$$(s, \nu) \cong (s', \nu') \stackrel{def}{=} (s = s' \wedge \nu \cong \nu')$$

A *region* of a TIOA is an equivalence class of states induced by $\cong$.

Because testing is based on distinguishing sequences (cf. Chapter 4), it is necessary to have an automaton that can distinguish each sequences that is used. Correspondingly, the Grid Automaton will be presented after present all its necessary ingredients.

Let $\mathbb{G}^n$ be the set of integer multiples of 2^{-n}, for some sufficiently large natural number n. If t is a real number, we use the notation[2] $\lfloor t \rfloor_n$ for the largest

[2] do not confuse with the notation $\lfloor\ \rfloor$ without subindice

number in $\mathbb{G}^n$ that is not greater than t, and $\lceil t \rceil_n$ for the smallest number in $\mathbb{G}^n$ that is not smaller than t. We write $[t]_n$ for the fraction $(\lfloor t \rfloor_n + \lceil t \rceil_n)/2$, note that $[t]_n \in \mathbb{G}^{n+1}$. For a TIOA $\mathscr{A}$ and its $\mathscr{OS}(\mathscr{A})$ associated, write Q^n for the set of states $(s, \nu) \in Q$ such that, for each clock $c, \nu(c) \in \mathbb{G}^n \cup \{\infty\}$.

The following lemma shown that given any state (q) in $\mathbb{G}^n$ for all $a \in \Sigma$ and $d \in \mathbb{G}^n$, labels of a transition in the semantic $(\rightarrowtail)$, the target state (q') of that transition is also in $\mathbb{G}^n$.

Lemma 8.22. *Let $q \in Q^n$, then*

- *If $q \overset{a}{\rightarrowtail} q'$ with $a \in \Sigma$ then $q' \in Q^n$*
- *If $q \overset{d}{\rightarrowtail} q'$ with $d \in \mathbb{G}^n$ then $q' \in Q^n$.*

Moreover, for a *distinguishing trace* of length m for two states in Q^n, a *trace* can be derived in which all delay actions are in the grid set $\mathbb{G}^{n+m}$.

Theorem 8.23. *Let $\mathscr{A}, \mathscr{B}$ be TIOAs and theirs associated semantics $\mathscr{OS}(\mathscr{A})$, $\mathscr{OS}(\mathscr{B})$, let $(r, r') \cong (s, s')$ for states $r \in Q_{\mathscr{A}}, r' \in Q_{\mathscr{B}}, s \in Q^n_{\mathscr{A}}$ and $s' \in Q^n_{\mathscr{B}}$, and let $\sigma = a_1 a_2 \dots a_m$ be a* distinguishing trace *for r and r'. Then there exists a* distinguishing trace $\tau = b_1 b_2 \dots b_m$ *for s and s' such that, for all $j \in [1, \dots, m]$, if a_j is an input or output action then $b_j = a_j$, and if a_j is a delay action then $b_j \in \mathbb{G}^{n+j}$ with $\lfloor a_j \rfloor \leq b_j \leq \lceil a_j \rceil$.*

This theorem allows to *transform* each *distinguishing trace* into one in which all delay actions are in a grid set, and shown that there is a dependency between the length of the *trace* and the granularity of the grid: the longer the *trace* the finer the grid. This is due to the fact that the distinguish power of a *distinguishing trace* for two states r and r' entirely depends on the regions traversed when applying σ to r and r', respectively. Moreover, we can conclude that the grid size depends on the number of states, not just on the number of clocks.

In order to obtain a grid size that is fine enough to distinguish all pairs of different states, the following theorem establishes an upper bound on the length of minimal *distinguishing traces*.

Theorem 8.24. *Suppose $\mathscr{A}$ and $\mathscr{B}$ are TIOAs with the same input actions, and r and s are states of $\mathscr{OS}(\mathscr{A})$ and $\mathscr{OS}(\mathscr{B})$, respectively : $r \not\simeq s$ (with $\simeq$ denoting bisimilarity 8.16). Then, there exists a* distinguishing trace *for r and s of length at most the number of regions of $Q_{\mathscr{A}} \times Q_{\mathscr{B}}$.*

Finally, we are in position of define the Grid Automaton. For each TIOA $\mathscr{A}$ and natural number n, the grid automaton $\mathscr{G}(\mathscr{A}, n)$ is defined as the subautomaton of $\mathscr{OS}(\mathscr{A})$ in which each clock value is in the set $\mathbb{G}^n \cup \{\infty\}$, and the only delay action is 2^{-n}. Note that since in the initial state of $\mathscr{OS}(\mathscr{A})$ all clocks take values in $\mathbb{Z}^\infty$, it is always included as a state of $\mathscr{G}(\mathscr{A}, n)$. Moreover, since $\mathscr{G}(\mathscr{A}, n)$ has a finite number of states and actions, $\mathscr{G}(\mathscr{A}, n)$ is a finite automaton.

Definition 8.25. Let $\mathscr{A} = \langle S, \Sigma, s_0, E \rangle$ be a TIOA, its $\mathscr{OS}(\mathscr{A}) = \langle Q, L, q_0, \rightarrowtail \rangle$ and $n \in N$. The grid automaton $\mathscr{G}(\mathscr{A}, n)$ is the LTS $\mathscr{A}'' = \langle Q', L', q'_0, \rightarrowtail' \rangle$ given by

- $Q' = Q^n$
- $L' = \Sigma \cup \{2^{-n}\}$
- $q'_0 = q_0$
- for all $q, q' \in Q'$ and $a \in L'$, $q \overset{a}{\rightarrowtail}' q'$ iff $q \overset{a}{\rightarrowtail} q'$.

The grid automaton is the restriction of $\mathscr{OS}(\mathscr{A})$ to the time steps in 2^{-n}, therefore $\mathscr{G}(\mathscr{A}, n)$ is finite.

Example. In Figure 8.8 the grid automaton of our example of the Light Switch for $n = 2$ is presented. Here we denote the initial state as $<< >>$, for distinguish it from the double circle denoting the initial state in a TIOA.

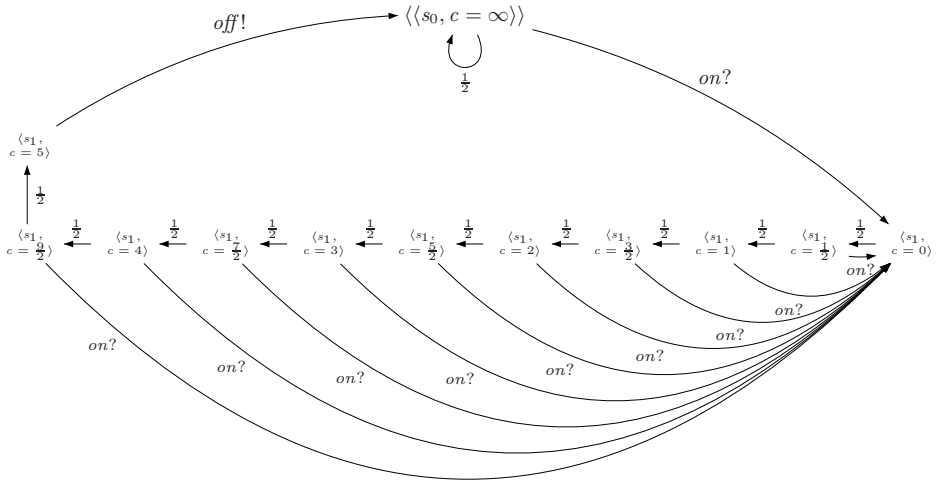


Fig. 8.8. The grid automaton $\mathscr{G}(\mathscr{A}, n)$ with $\mathscr{A}$ as the Light Switch automaton and $n = 2$

Corollary 8.26. *Let $\mathscr{A}$ and $\mathscr{B}$ be TIOA with the same input actions, and let n be at least the number of regions of $S_{\mathscr{A}} \times S_{\mathscr{B}}$, then*

$$\mathscr{A} \simeq \mathscr{B} \quad \textit{iff} \quad \mathscr{G}(\mathscr{A}, n) \simeq \mathscr{G}(\mathscr{B}, n).$$

Using the grid automaton with the appropriate degree of granularity the problem of decide bisimulation equivalence of TIOA is reduced to the problem of decide bisimulation equivalence of their finite subautomata.

8.4.3 Testing

A **test sequence** for a TIOA $\mathscr{A}$ is a finite sequence of delays and input actions of $\mathscr{A}$ (we denoted the set of this sequences as *Exp*). A test sequence σ can be

applied to $\mathscr{A}$ starting from any state s of its $\mathscr{OS}(\mathscr{A})$. The application of σ to $\mathscr{A}$ in s uniquely determines a finite, maximal execution fragment in $\mathscr{OS}(\mathscr{A})$.

How to perform a test sequence is shown in the following definition. The outcome of performing a test sequence on $\mathscr{A}$ is described in terms of an auxiliary labeled transition system T.

Definition 8.27. The test sequence is the LTS $T = \langle (Exp \times Q), \Sigma, (\epsilon, s_0), \twoheadrightarrow \rangle$ with $(Exp \times Q)$ as its set of states, where Exp is the test sequence to be executed, Σ is a set of actions, (ϵ, s_0) is (arbitrarily chosen) initial state, and a transition relation $\twoheadrightarrow$ that is inductively defined as the least relation satisfying the following four rules, for all $q, q' \in Q, \sigma \in Exp, i \in \mathcal{I}, o \in \mathcal{O}$ and $d, d' \in \mathbb{R}^{>0}$:

- $\dfrac{q \overset{o!}{\longmapsto} q'}{(\sigma,q) \overset{o!}{\twoheadrightarrow} (\sigma,q')}$
- $\dfrac{q \overset{i?}{\longmapsto} q'}{(i?\sigma,q) \overset{i?}{\twoheadrightarrow} (\sigma,q')}$
- $\dfrac{q \overset{d}{\longmapsto} q'}{(d\sigma,q) \overset{d}{\twoheadrightarrow} (\sigma,q')}$
- $\dfrac{q \overset{d'}{\longmapsto} q',\ \sup\{t \in \mathbb{R}^{>0} \mid q \overset{t}{\longmapsto}\} = d' < d}{(d\sigma,q) \overset{d'}{\twoheadrightarrow} ((d-d')\sigma,q')}$

The first rule says that output actions are always performed autonomously, i.e. independently of the input of the intended test sequence. Instead, input actions are only performed if they are explicitly specified in the test sequence. This is stated by the second rule. Similarly, the third rule says that a delay can occur only when it is both specified by the test sequence and allowed by $\mathscr{A}$. In some cases, a delay specified in the test sequence cannot occur since it is interrupted by an autonomous output action of $\mathscr{A}$. In such a case, the part of the delay up to the output action is executed, while the rest is postponed until $\mathscr{A}$ stops doing output actions autonomously. This last case is expressed by the fourth rule.

Theorem 8.28. *Let $\mathscr{A}$ a TIOA and T its test sequence, then*

- *each state of T has at most one outgoing transition, and*
- *T does not have an infinite execution fragment.*

Theorem 8.28 allows us to define $exec(\sigma, q)$ as the execution fragment of $\mathscr{OS}(\mathscr{A})$ obtained by projecting the states in the unique maximal execution fragment of T that starts in (σ, q) on their second component. We define $outcome(\sigma, q)$, the *outcome of the sequence σ in state q*, as the *trace* of the execution fragment that is induced by performing the test sequence:

$$outcome(\sigma, q) \overset{def}{=} trace(exec(\sigma, q))$$

Deriving and Applying a Test Suite It is assumed that the behavior of the IUT (Implementation Under Test) is accurately modeled by a TIOA *Impl*. Then the IUT conforms to the specification *Spec* if *Impl* is bisimilar to *Spec*.

The method of building test suites is similar to Chow's classical algorithm for Mealy machines [Cho78] (cf. Chapter 4). A test suite consists of a finite set of test sequences which should be applied to the implementation. Each sequence consists of the concatenation of two sequences. The initial part of a test sequence is taken from a *transition cover* P for a grid subautomaton of *Spec*, i.e. a set of test sequences that together exercise every transition of the subautomaton.

Definition 8.29. Let $\mathscr{A}$ be a TIOA, $n \in \mathbb{N}$, $\mathscr{A}'' = \mathscr{G}(\mathscr{A}, n)$. A *transition cover* for $\mathscr{A}''$ is a finite collection $P \subseteq Exp_n$ of test sequences, such that $\epsilon \in P$ and, for all transitions $q \overset{a}{\rightarrowtail}' q'$ of $\mathscr{A}''$ with q reachable (within $\mathscr{A}''$) and stable (Definition 8.19), P contains test sequences σ and $\sigma \cdot a$ such that $q_0 \overset{\sigma}{\rightarrowtail}' q$.

The trailing part of a test sequence is taken from a set Z, which is a *characterization set* for a grid subautomaton of *Impl*, meaning that for every pair of non-bisimilar grid states, Z contains a sequence that distinguishes between them.

Definition 8.30. Let p a state of $\mathscr{A}$, q a state of $\mathscr{B}$, and let σ be a test sequence for $\mathscr{A}$ and $\mathscr{B}$. σ distinguishes p from q if $\text{outcome}_{\mathscr{A}}(\sigma, p) \neq \text{outcome}_{\mathscr{B}}(\sigma, q)$. If Z is a set of test sequences for $\mathscr{A}$ and $\mathscr{B}$, written $p \approx_Z q$ means that no test sequence in Z distinguishes p from q.

The ability of always being able to bring the machine back to its initial state is used. In the timed case, it is not reasonable to consider the reset as an instantaneous operation: typically, some time will elapse between the moment when it is requested the machine to go to its initial state, and the moment at which the reset operation has been completed. But, it is not difficult to prove that the maximal time that can elapse between the occurrence of a reset action and the time at which the initial state is reached is always less than the number of regions of $\mathscr{A}$.

Then, the test suite is defined for a given TIOA as follows.

Definition 8.31. Let $\mathscr{A}$ be a TIOA and $n \in \mathbb{N}$. Let P be a transition cover for $\mathscr{G}(\mathscr{A}, n)$ and Z a characterization set for the TIOA model of the IUT. The test suite for $\mathscr{A}$ generated from P and Z with grid size n is defined by

$$\text{test-suite}(\mathscr{A}, n, P, Z) \overset{def}{=} P \cdot Z \cdot \{\text{reset max}\}$$

i.e. the concatenation of the transition cover, the characterization set and the reset time.

Definition 8.32. A state of a TIOA is *quiescent* if each execution fragment starting in that state that contains an output action also contains an input action.

Algorithm 13 is the testing algorithm that applies each test case from the test suite to an implementation (the prove of correctness is showed in [SVD01]). This algorithm is restricted to TIOAs with a *quiescent* initial state, where the machine waits for stimulus from its environment before producing any output.

Algorithm 13 Test Generation

input: A TIOA *Spec*, the specification automaton, with reset action *reset*, reset time max, and a *quiescent* initial state.
An Implementation Under Test (IUT), a device that accepts inputs from I_{Spec} and produces outputs in $\mathcal{O}_{\text{Spec}}$.
A natural number n.
A natural number m.
output: A verdict PASS or FAIL

```
1  Let  X = I_Spec ∪ {2^-n}
2      Determine a (minimal) finite transition cover P for G(Spec, n)
3      For all test sequences σ ∈ test-suite(Spec, n, P, X^(m-1)) do
4          Apply test sequence σ to the IUT
5          Return FAIL and halt if outcome of the IUT differs from
               outcome_Spec(σ, s^0_Spec)
6      Return PASS and halt
```

This algorithm results in a huge number of sequences. Therefore, it cannot be claimed to be itself of practical value. Rather, the major contribution here is the TIOA model and the demonstration that an algorithm to derive a (complete) test suite does exist. Moreover, there are ways to reduce the number of tests, and make the time delays within the tests manageable [SVD01].

8.5 Testing Networks of UPPAAL Timed Automata

Cardell-Oliver [CO00] presents a test generation method for networks of *deterministic timed automata* on a *dense time* base. Timed automata are extended with persistent data variables and are allowed to have silent transitions. Test generation is based on test views that partition events into visible (relevant) and hidden events according to a certain test purpose. By only testing for visible events the size of the resulting test suite can be reduced. The work presented is a generalization of previous work by Cardell-Oliver and Glover [COG98] that was applicable only for specifications with a discrete clock model.

8.5.1 Model

For model specification, UPPAAL timed automata [LPY97] are adopted. UPPAAL timed automata (UTA) extend Alur and Dill's model of timed automata with (integer) data variables. With UTA, networks of deterministic timed automata can be specified. This allows for closed world specifications of systems, i.e.

the behavior of an system's environment can be specified explicitly. Synchronization between components takes place by complementary actions of automata, i.e. by simultaneous occurrence of an output event $a!$ and an input event $a?$, with $a \in \Sigma$, respectively. Each automaton $\mathscr{A}_i$ can use a set of integer variables Var_i that is a subset of a set of global integer variables Var. Guards on transitions are extended to apply for both clocks and data variables.

Definition 8.33. An **UPPAAL timed automata** $\mathscr{A}$ is a tuple $\langle S, s_0, \Sigma, C, Inv, E \rangle$, where

- S is a finite set of locations
- s_0 is the initial location
- $\Sigma = \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ is a finite set of actions, partitioned into input actions, output actions, and the silent action
- C is a finite set of (real-valued) clocks
- $Inv : S \to \Phi(C)$ assigns clock invariants to locations
- $E \subseteq S \times \Sigma \times \Phi(C, Var_{\mathscr{A}_i}) \times 2^R \times S$ is the set of transitions.

Transitions $(s, a, \varphi, r, s') \in E$ are denoted by $s \xrightarrow{a,\varphi,r} s'$, where a is the action to be performed, φ the guard of the transition, and r a set of assignments for clocks and data variables. Clock variables can be reset to an integer constant $l \in \mathbb{Z} \cup \{-1\}$. A reset to -1 denotes a turn-off of the according clock variable. Data variables can be reset to integer expressions of the form $v := k * v + k'$, where $v \in Var_{\mathscr{A}_i}$ and $k, k' \in \mathbb{Z}$. R is used to denote the set of all possible reset operations.

Remark 8.34. The definition of UTA mainly follows the one presented by Bengtsson et al. [BLL+95]. The definition given here omits urgent synchronization but includes silent transitions as well as location invariants.

For testing purposes, clock constraints in guards and invariants are required to be closed ($<$ and $>$ are not allowed) and domains for clocks and data variables are required to be finite.

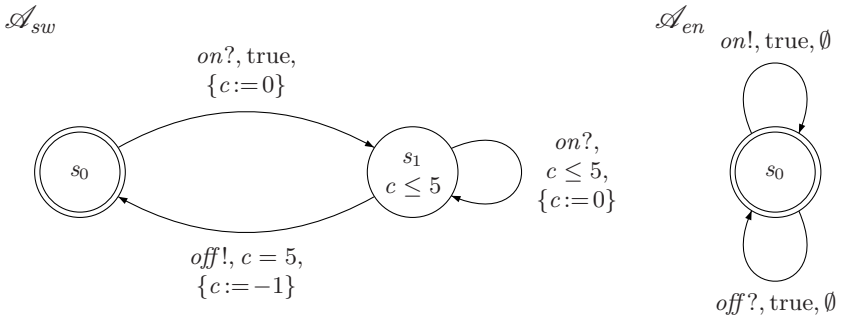


Fig. 8.9. UTA specification of the Light Switch $\mathscr{A}_{sw}$ and its environment $\mathscr{A}_{en}$

Example. The Light Switch can be defined by an UTA $\mathscr{A}_{sw} = \langle S, s_0, \Sigma, C, Inv, E \rangle$, where

- $S = \{s_0, s_1\}$
- $\Sigma = \{on, off\}$, with $\mathcal{I} = \{on\}$ and $\mathcal{O} = \{off\}$
- $C = \{c\}$
- $Inv(s_0) = \text{true}$, $Inv(s_1) = c \leq 5$
- $E = \{(s_0, on?, \text{true}, \{c := 0\}, s_1), (s_1, on?, c \leq 5, \{c := 0\}, s_1),$
  $(s_1, on?, c = 5, \{c := \text{-}1\}, s_0)\}$.

Specification of the environment can be done analogously (cf. Figure 8.9).

The definition of the semantics of UPPAAL timed automata is based on timed transition systems with an uncountable set of states.

Definition 8.35. A **timed transition system** (**TTS**) over a set of actions Σ and a time domain $\mathbb{R}^{\geq 0}$ is a tuple $\mathscr{M} = \langle Q, L, \longrightarrow, q_0 \rangle$ of a set of states Q, an initial state $q_0 \in Q$, and a set of labels $L \subseteq \Sigma \cup \mathbb{R}^{\geq 0}$, a transition relation $\longrightarrow \subseteq Q \times L \times Q$ that has to satisfy the following properties ($\forall\, q, q', q'' \in Q \wedge \forall\, d, d_1, d_2 \in \mathbb{R}^{\geq 0}$):

- *time determinism:* if $q \xrightarrow{d} q' \wedge q \xrightarrow{d} q''$ then $q' = q''$
- *time additivity:* $q \xrightarrow{d_1+d_2} q''$ iff $q \xrightarrow{d_1} q' \xrightarrow{d_2} q''$
- *0-delay:* $q \xrightarrow{0} q'$ iff $q = q'$.

Since specifications of real-time systems in UPPAAL are generally networks of automata, a LTS $\mathscr{M}$ has to be constructed for parallel compositions of UTA. The set $P = \{p_1, \ldots, p_n\}$ is used to contain the names of all components that are part of the specification, with p_i being the name of the component specified by the automaton $\mathscr{A}_i$. The set of channels usable for synchronization is given by $Ch = (\bigcup_i \mathcal{I}_{\mathscr{A}_i}) \cap (\bigcup_i \mathcal{O}_{\mathscr{A}_i})$.

States of $\mathscr{M}$ are pairs $(\overline{s}, \nu)$, where $\overline{s}$ is a vector holding the current control locations for each component (automaton) and ν maps each clock to a value in the time domain as well as each data variable to an integer value.

Transition labels of $\mathscr{M}$ are either delays $d \in \mathbb{R}^{\geq 0}$ or event triples (p_i, a, r) with p_i being the name of the automaton executing an action a, that could either be a silent action or an output action (which implies the occurrence of an complementary input actions of another automaton). An action a leads to the execution of a set of resets r that contains resets for clocks, variables, and locations. Location resets explicitly denote a change of location of a component which results in an update of the according element in $\overline{s}$. The set of all possible reset statements is given by $\mathcal{R} \subseteq 2^{\bigcup_i R_{\mathscr{A}_i} \cup R_i^s}$, with $R_{\mathscr{A}_i}$ being the usual resets of $\mathscr{A}_i$ and R_i^s being the set of resets for locations of $\mathscr{A}_i$.

Definition 8.36. The semantics of a network of UTA $\mathscr{A}_1, \ldots, \mathscr{A}_n$ is given by the TTS $\mathscr{M} = \langle Q, L, \rightarrow, q_0 \rangle$, where

- $Q = \{\langle \overline{s}, \nu \rangle \mid \overline{s}[i] \in S_{\mathscr{A}_i}, \nu \models Inv_{\mathscr{A}_i}(C_{\mathscr{A}_i})\}$, $\forall\, 1 \leq i \leq n$
- $q_0 = \langle \overline{s}_0, \nu_0 \rangle$ with $\overline{s}_0[i] = s_{0\mathscr{A}_i}$ and $\nu_0[i] = 0$, $\forall\, 1 \leq i \leq n$

- $L = \mathbb{R}^{\geq 0} \cup (P, Ch \cup \{\tau\}, \mathcal{R})$
- $\rightarrow \subseteq Q \times L \times Q$, that could be either
  - $\langle \overline{s}, \nu \rangle \xrightarrow{d} \langle \overline{s}, \nu \oplus d \rangle$ iff $\forall\, i$: $\nu \oplus d \models Inv_{\mathscr{A}_i}(\overline{s}[i])$
  - $\langle \overline{s}, \nu \rangle \xrightarrow{p_i, \tau, r} \langle \overline{s}[s'_{\mathscr{A}_i}/s_{\mathscr{A}_i}], r_i(\nu) \rangle$ iff $(s_i, \varphi, \tau, r_i, s'_i) \in E_{\mathscr{A}_i}$ and $\nu \models \varphi$, with $r = r_i \cup \{s_{\mathscr{A}_i} := s'_{\mathscr{A}_i}\}$
  - $\langle \overline{s}, \nu \rangle \xrightarrow{p_i, a, r} \langle \overline{s}[s'_{\mathscr{A}_i}/s_{\mathscr{A}_i}, s'_{\mathscr{A}_j}/s_{\mathscr{A}_j}], (r_i \cup r_j)(\nu) \rangle$ iff $(s_i, \varphi_i, a!, r_i, s'_i) \in E_{\mathscr{A}_i}$, $(s_j, \varphi_j, a?, r_j, s'_j) \in E_{\mathscr{A}_j}$, $\nu \models \varphi_i$, and $\nu \models \varphi_j$, with $r = r_i \cup r_j \cup \{s_{\mathscr{A}_i} := s'_{\mathscr{A}_i}, s_{\mathscr{A}_j} := s'_{\mathscr{A}_j}\}$

For a variable assignment ν and a delay d, $\nu \oplus d$ denotes the variable assignment after d. $\oplus$ models time-insensitiveness of all data variables and that all enabled clocks progress at the same rate:

$$\forall\, v \in Var : (\nu \oplus d)(v) = \nu(v), \text{and}$$
$$\forall\, c \in \bigcup_i C_{\mathscr{A}_i} : (\nu \oplus d)(c) = \begin{cases} \nu(c) + d & \text{if } \nu(c) \geq 0 \\ \nu(c) & \text{if } \nu(c) = -1 \end{cases}$$

Silent transitions result in the change of location of one component. According transitions in $\mathscr{M}$ express this change by replacing the ith element of the location vector $\overline{s}$ by a new location $s'_{\mathscr{A}_i}$ and applying the resets r to ν. Synchronizations between two components involve two location transitions, one for the sender $\mathscr{A}_i$ and one for the receiver $\mathscr{A}_j$. Consequently the ith and the jth element of $\overline{s}$ have to be replaced with $s'_{\mathscr{A}_i}$ and $s'_{\mathscr{A}_j}$ respectively, and the union of transition resets $r_i \cup r_j$ has to be applied to ν.

Remark 8.37. The definition given here follows Bengtsson et al. [BLL+95] in defining states as pairs of a location vector $\overline{s}$ and variable valuations ν.

An alternative to the use of a location vector would be to include for every component p_i a special variable loc_i, which holds the current location of the according process, into the set Var. States could then be defined as $S \subseteq (Var \rightarrow \mathbb{Z}) \cup (C \rightarrow \mathbb{R}^{\geq 0})$ [CO00].

Example. The possible behavior of the Light Switch specified by $\mathscr{A}_{sw}$ in the environment $\mathscr{A}_{en}$ is given by a TTS $\mathscr{M}_s = \langle Q, L, \longrightarrow, q_0 \rangle$, where

- $Q = \left\langle \binom{s_{\mathscr{A}_{sw}}}{s_{\mathscr{A}_{en}}}, c \rightarrow [0, 8] \right\rangle$, with $s_{\mathscr{A}_{sw}} \in S_{\mathscr{A}_{sw}}$ and $s_{\mathscr{A}_{en}} \in S_{\mathscr{A}_{en}}$
- $q_0 = \left\langle \binom{s_0}{s_0}, c = 0.0 \right\rangle$
- $L = [0, 8] \cup (\{sw, en\}, \{on, off\}, \{\{c := 0, s_{\mathscr{A}_{sw}} := s_1\}, \{c := \text{-}1, s_{\mathscr{A}_{sw}} := s_0\}\})$ (Resets for locations of the environment are omitted since $\mathscr{A}_{en}$ has only one location.)

For testing we constrain the time domain to $[0, 8]$. Note that due to the dense time domain, $\mathscr{M}_s$ has infinitely many states and infinitely many transitions (cf. Figure 8.10).

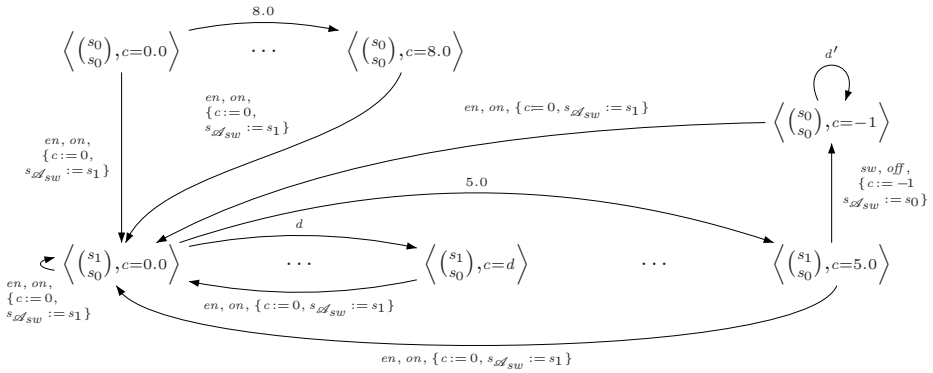


Fig. 8.10. Timed transition system $\mathscr{M}_s$ for $\mathscr{A}_{sw} \| \mathscr{A}_{en}$

8.5.2 Digitization

Timed transition systems are not directly amenable to testing. Besides their infiniteness, TTS *traces* include some *traces* that cannot be observed, e.g. delays that are not followed by visible events. Furthermore, observable TTS *traces* do not contain sufficient information to distinguish between input and output events.

A **testable timed transition systems** is a TTS but also a (deterministic) FSM. A TTTS $Spec = \langle Q, L, \longrightarrow, q_0 \rangle$ uses a subset $Q \subset Q_{\mathscr{M}}$ of states of the original TTS. Labels of the TTTS are timed event 4-tuples (d, io, a, r) with discrete delay $d \in \mathbb{N}$, $io \in \{inp, out\}$, and a and r as in $\mathscr{M}$. It is derived from a TTS $\mathscr{M}$ executing the following steps:

(1) *Digitize clocks*: Each timed *trace* with times in $\mathbb{R}^{\geq 0}$ is mapped onto a set of *traces* with times in $\mathbb{Z}$. For each reachable state q and for each delay $d \in \mathbb{R}^{\geq 0}$ within a lower and upper bound $LB \leq d \leq UB$ after which an event a can occur include for every $i \in \{LB, LB+1, \ldots, UB\}$ a transition from $\langle \overline{s}, \nu \rangle$ to $\langle \overline{s}, \nu \oplus i \rangle$ into the TTTS.

(2) *Distinguish between inputs and outputs of the SUT*: The set of network components can be partitioned into automata specifying the system under test $\mathscr{S}$ and automata describing the environment $\mathscr{E}$ of the SUT, with $\mathscr{S} \cap \mathscr{E} = \emptyset$. Each transition (p_i, a, r) of a TTS becomes in the TTTS $(0, inp, a, r)$ if $\mathscr{A}_i \in \mathscr{E}$, or $(0, out, a, r)$ if $\mathscr{A}_i \in \mathscr{S}$ respectively.

(3) *Distinguish between visible and invisible actions:* Visible events of a TTTS are defined by a test view $\mathcal{V} = (P' \subseteq P, Var' \subseteq Var, C' \subseteq C, Ch' \subseteq Ch)$. In the TTTS all $a \in Ch \backslash Ch'$ are replaced by τ. The reset set is reduced to only contain resets for elements of $\overline{s}$ with $p \in P'$, for variables $v \in Var'$, and for clocks $c \in C'$. All states with equal values for visible variables are considered to belong to the same visible equivalence class ($q =_{\mathcal{V}} q' \stackrel{def}{=} \forall\, p_i \in P' \forall\, v \in (Var' \cup C') : \nu(v) = \nu'(v) \wedge s[i] = s'[i]$, with $q = (\overline{s}, \nu)$ and $q' = (\overline{s}', \nu')$).

(4) *Normalize TTTS:* Not observable events could not be tested. Therefor, silent events are elided and delays of these omitted events are added to their following visible events. Each transition sequence of the form $q_0 \xrightarrow{d_1,inp,\tau,\{\}} q_1 \xrightarrow{d_2,out,a,r} q_2$ is replaced by $q_0 \xrightarrow{d_1+d_2,out,a,r} q_2$.
Subsequently, the TTTS has to be re-transformed into a deterministic transition system, since omitting events may have introduced non-determinism. Note that, normalization is not allowed to remove cycles of silent actions. At least one of the actions on such a cycle has to be made visible, i.e. the test view $\mathcal{V}$ has to be changed, to get a proper TTTS.

(5) *Minimize TTTS:* remove all states that are redundant, i.e. all but one that are in the same visible equivalence class and have the same set of *traces*. There might be states that are in the same visible equivalence class but do not have the same set of visible *traces*. Such states have to be kept.

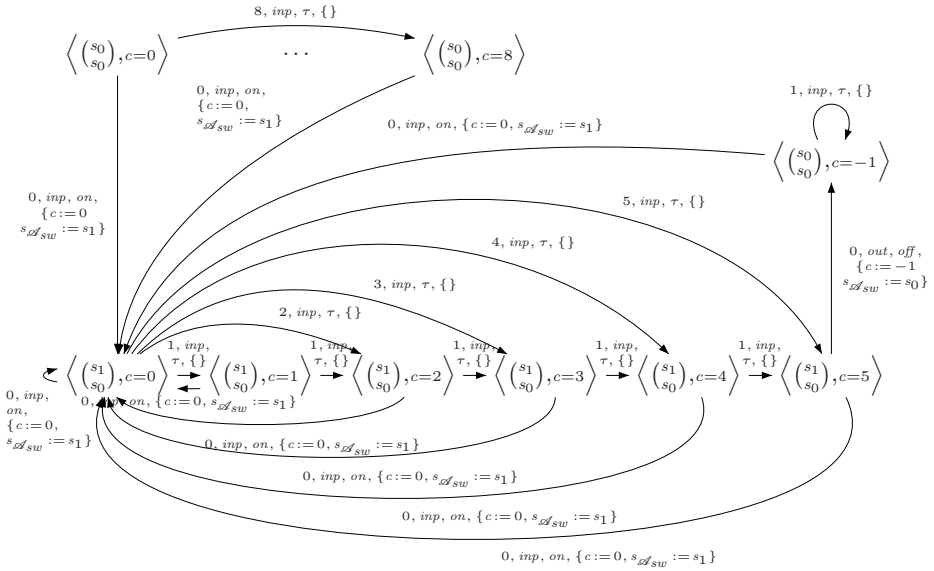


Fig. 8.11. A TTTS gained from TTS $\mathscr{M}_s$ after digitization and label transformation

Example. After digitization the TTS $\mathscr{M}_s$ is reduced to a TTTS with 15 states (cf. Figure 8.11).

Let us now assume a test view $\mathcal{V} = (P', Var', C', Ch')$, where $P' = \{p_{en}\}$, $Var' = Var = \emptyset$, $C' = C = \{c\}$, and $Ch' = Ch = \{on, off\}$. Since $P' \subset P$ does not contain the name of the switch component p_{sw}, valuations and resets of the locations of the Switch become invisible. By using this view, and applying normalization the set of states can be reduced to contain only 3 states. We get the TTTS $Spec = (Q, L, \longrightarrow, q_0)$, where

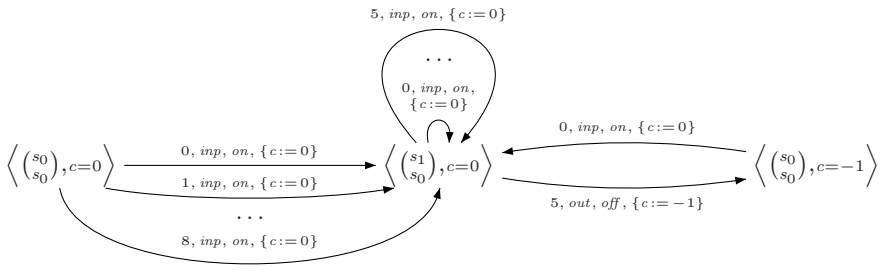


Fig. 8.12. The TTTS *Spec* after the application of a test view, normalization, and minimization

- $Q = \{q_0, q_1, q_2\} = \left\{ \left\langle \binom{s_0}{s_0}, c=0 \right\rangle, \left\langle \binom{s_1}{s_0}, c=0 \right\rangle, \left\langle \binom{s_0}{s_0}, c=\text{-}1 \right\rangle \right\}$
- $\longrightarrow = \{t_1, \ldots, t_{16}\} = \left\{ q_0 \xrightarrow{0,inp,on,\{c:=0\}} q_1, \ldots, q_0 \xrightarrow{8,inp,on,\{c:=0\}} q_1, \right.$
  $q_1 \xrightarrow{0,inp,on,\{c:=0\}} q_1, \ldots, q_1 \xrightarrow{5,inp,on,\{c:=0\}} q_1,$
  $\left. q_1 \xrightarrow{5,out,off,\{c:=\text{-}1\}} q_2,\ q_2 \xrightarrow{0,inp,on,\{c:=0\}} q_1 \right\}$

(cf. Figure 8.12)

8.5.3 Testing

The conformance relation for testable timed transition systems is trace equivalence. Formally, $\text{Conf}(Spec) \stackrel{def}{=} \{S \mid \text{traces}(Spec) = \text{traces}(S)\}$. A test suite for a TTTS *Spec* consists of one test case for every transition in *Spec*. A test case essentially consists of three parts. The first part reaches the source state of a transition. Secondly, the transition is executed. The third part has to verify that the execution of the transition has resulted in the target state specified by *Spec*, i.e. it is a state verification sequence.

The usage of test views dramatically simplifies the search for these *separating sequences*. With classical FSM testing techniques (without data variables and test views) each state needs to be distinguished form any other in the automaton (cf. Chapter 4). Since the normalization of the TTTS ensures that *Spec* is minimal and does only contain visible events we know exactly in which state we are after the execution of a certain *trace* (except for states that are in the same visible equivalence class). Hence, the third part of a transition test needs only to distinguish the target state of the transition to be tested from other states in their visible equivalence class. There may not exist a unique separating sequence for each such state (cf. Chapter 3), since *traces* of one state may be included in *traces* of other states. To distinguish these states, the *separating sequences* are paired with oracles that states whether the final event of the *trace* shall be observed.

Please note, that even if *Impl* is deterministic, from the tester's perspective it does not behave deterministically, because events produced by the implementation may occur at different points in time. Since the tester has no capability to control when output events of the SUT will eventually occur any possible *trace* has to be considered for both reaching a state and distinguishing a state. One of all possible *reach traces*, or *separating traces* respectively, had to be chosen on the fly during execution of the test, depending on the actual occurrence of an output event. If there is a *trace* that does not depend on the choices of the SUT we only need to consider this one for testing.

The conformance test algorithm (cf. Algorithm 14) takes a TTTS *Spec*, constructed using a View $\mathcal{V}$, as input and produces a finite set of *traces* each accompanied with an oracle (yes/no) for observing its final event.

Algorithm 14 TTTS Conformance Test Algorithm

input: TTTS $Spec = \langle Q, L, \longrightarrow, q_0 \rangle$, Test View $\mathcal{V}$
output: Test(*Spec*)

1 $Test(Spec) = \emptyset$
2 **for** every $q \in Q$ **do**
3 *// find all acyclic traces, i.e. that visit no state more than once, ending at q*
4 $reach(q) = \{\sigma \mid q_0 \xrightarrow{\sigma} q \wedge acyclic(\sigma)\}$
5 **for** every $q \in Q$ that is a transition's destination state **do**
6 **for** each $q' =_{\mathcal{V}} q$ **do**
7 *// distinguish q from all states in the same visible equivalence class*
8 **if** $q = q'$ **then** $\sigma = \langle\rangle$
9 **else** *// non trivial distinction of states*
10 **for** every $\sigma = l_1 \dots l_n$ with $l_1 \dots l_{n-1} \in \text{traces}(q) \cap \text{traces}(q')$ **do**
11 **if** $\sigma \notin \text{traces}(q) \cap \text{traces}(q')$ **then** *// σ distinguishes between q and q'*
12 *// pair σ with oracle whether the final event should be observed*
13 **if** $\sigma \in \text{traces}(q)$ **then** $\text{diff}(q, q') += \sigma * \text{yes}$
14 **else if** $\sigma \in \text{traces}(q')$ **then** $\text{diff}(q, q') += \sigma * \text{no}$
15 *// Compose a test for every transition*
16 **for** every $t = (q_1, l, q_2) \in \longrightarrow$ **do**
17 **for** each $q_i =_{\mathcal{V}} q_2$
18 $\text{Testfor}(t) \mathrel{+}= \sigma_1 \cdot l \cdot \sigma_i * R_i$, with $\sigma_1 \in reach(q_1)$ and $\sigma_i * R_i \in diff(q_2, q_i)$
19 $\text{Test}(Spec) \mathrel{+}= \text{Testfor}(t)$

Previous work did allow implementations to have extra states [COG98]. Now it is claimed that "the assumption of a bounded, small number of extra states is not appropriate for real-time systems" [CO00], because minor changes of a timed automata specification can result in a very large change in the size of its TTTS.

Definition 8.38. Real-Time Faults for TTTS: $Impl \in \text{NonConf}(Spec)$ if and only if

- *Impl* has no more states then *Spec* and
- *Impl* has a single transition fault or *Impl* can be transformed to *Spec* by a sequence of single transition faults.

It can be shown that for a TTTS specification *Spec*, the test suite *Test*(*Spec*) that is generated by the TTTS Test generation Algorithm detects any Impl $\in$ Nonconf(*Spec*) [CO00]. If the implementation satisfies the test hypotheses then all tests for *Spec* will be passed by the implementation if and only if the implementation is trace equivalent to *Spec*.

Example. $Spec = \langle Q_{Spec}, L, \longrightarrow, q_0 \rangle$, $\mathcal{V} = (\{s_{en}\}, \emptyset, \{c\}, \{on, off\})$

(1) *Reach all states*
- reach(q_0) = $\{\langle\rangle\}$
- reach(q_1) = $\{\langle 0, inp, on, \{c{:=}0\}\rangle, \ldots, \langle 8, inp, on, \{c{:=}0\}\rangle\}$
- reach(q_2) = $\{\langle 0, inp, on, \{c := 0\} \cdot 5, out, off, \{c := 1\}\rangle, \ldots, \langle 3, inp, on, \{c{:=}0\} \cdot 5, out, off, \{c{:=}1\}\rangle\}$

(2) Distinguish states in the same visible equivalence class: Since q_0 is not a destination state for some transition we do not need to distinguish between q_0 and q_1 although both are in the same visible equivalence class. q_2 has no other state in its visible equivalence class. Therefor, all *distinguishing traces* are trivial, i.e. $\{\langle\rangle\}$

(3) Pair traces with oracles.
- diff(q_1, q_1) = $\{\langle\rangle * yes\}$
- diff(q_2, q_2) = $\{\langle\rangle * yes\}$

(4) Compose tests for every transition.
- testfor(t_1) = $\langle 0, inp, on, \{c{:=}0\}\rangle * yes$
- ...
- testfor(t_{10}) = $\langle 0, inp, on, \{c{:=}0\} \cdot 1, inp, on, \{c{:=}0\}\rangle * yes$
- ...
- testfor(t_{15}) = $\langle 0, inp, on, \{c{:=}0\} \cdot 5, out, off, \{c{:=}1\}\rangle * yes$
- testfor(t_{16}) = $\langle 0, inp, on, \{c := 0\} \cdot 5, out, off, \{c := 1\} \cdot 0, inp, on, \{c := 0\}\rangle * yes$

Since the tester has control over the event *on* we can choose one trace of all possible *reach*() traces for each state, although the states may be reached by different traces. If *on* were under control of the SUT we had to include all possible *reach*() traces for the according states. Furthermore, if we allowed *off* events to occur between an lower and upper time bound we had to include all possible traces including an *off* event into the according *reach* sets.

Please note, that transitions with yes oracles may be included in longer transitions, e.g. testfor(t_{16}) subsumes testfor(t_1) and testfor(t_{15}).

8.6 Summary

All three approaches use timed automata with a dense time model for testing real-time systems. All need to partition the uncountable state space of the semantics of (networks of) timed automata into a finite number of states considered equivalent.

Nielsen and Skou use coarse-grained domains [NS03]. A fully automatic method for the generation of real-time test sequences from a subclass of timed automata called event-recording automata is proposed. The technique is based on the symbolic analysis of timed automata inspired by the UPPAAL model-checker. Test sequences are selected by covering a coarse equivalence class partitioning of the state space. They argue that the approach provides a heuristic that guarantees that a well-defined set of interesting scenarios in the specification has been automatically, completely, and systematically explored.

Springintveld, Vaandrager and D'Argenio proved that exhaustive testing with respect to bisimulation[3] of deterministic timed automata with a dense time interpretation is theoretically possible [SVD01]. Testing of timed systems is described as a variant of the bounded time-domain automaton (TA). The TA describing the specification is transformed into a region automaton, which in turn is transformed into another finite state automaton, referred to as a Grid Automaton. Test sequences are then generated from the Grid Automaton. The idea behind the construction of the Grid Automaton is to represent each clock region with a finite set of clock valuations, referred to as the representatives of the clock region. However, although being exact, their grid method is impractical because it generates "an astronomically large number of test sequences" [SVD01].

Cardell-Oliver presents a testing method for networks of deterministic timed automata extended with integer data variables [CO00]. Checking of trace equivalence is done only for parts of a system that are visibly observable. In addition to the usual time-discretization test views are used to discriminate between states depending on a test-purpose. Test views partition variables and events into visible and hidden ones. Equivalence on visible clocks and variables induces an equivalence relation on states. States that are evidently different, i.e. that are in different visible equivalence classes, need not be distinguished from each other. This significantly reduces the length of test suites.

	specs	time	det.	τ	network	impl. rel.	based on	exhaustive
[NS03]	ERA	$\mathbb{R}^{>0}$			$\surd$	trace inclusion	testing preorder	
[SVD01]	TIOA	$\mathbb{R}^{>0}$	$\surd$			bisimulation	W method	$\surd$
[CO00]	UTA	$\mathbb{R}^{>0}$	$\surd$	$\surd$	$\surd$	bisimulation	W method	

Table 8.1. Comparison

In practice, time resources used for test case generation and execution should be as small as possible and test coverage as high as possible. This general need on

[3] In the case of determinism, bisimulation and trace equivalence coincide [vG01]

effectiveness becomes even more evident in real-time testing. Exhaustive testing becomes infeasible for any system of considerable size. Some approaches for testing real-time systems (cf. Chapter 13) gain practicability by dropping formal rigorousness. However, safety-critical systems require for justified confidence into their behavior. Make timed automata based testing applicable to systems of realistic size, remains to be done.