Testing Real-Time Systems Using UPPAAL

Anders Hessel[1], Kim G. Larsen[2], Marius Mikucionis[2], Brian Nielsen[2], Paul Pettersson[1], and Arne Skou[2]

[1] Department of Information Technology, Uppsala University, P.O. Box 337, SE-751 05 Uppsala, Sweden
{hessel,paupet}@it.uu.se
[2] Department of Computer Science, Aalborg University, Fredrik Bajersvej 7E, DK-9220 Aalborg, Denmark
{kgl,marius,bnielsen,ask}@cs.aau.dk

Abstract. This chapter presents principles and techniques for model-based black-box conformance testing of real-time systems using the UPPAAL model-checking tool-suite. The basis for testing is given as a network of concurrent timed automata specified by the test engineer. Relativized input/output conformance serves as the notion of implementation correctness, essentially timed trace inclusion taking environment assumptions into account. Test cases can be generated offline and later executed, or they can be generated and executed online. For both approaches this chapter discusses how to specify test objectives, derive test sequences, apply these to the system under test, and assign a verdict.

1 Introduction

Many computer-based systems monitor and control a physical environment through sensors and actuators. The physical laws governing the environment induce a set of real-time constraints which the system must obey in order to achieve satisfactory or safe operation. Thus the computer system must not only produce correct result or reaction, but must do so at the correct time; neither too early nor too late. For a *real-time system* the timely reaction is just as important as the kind of reaction.

Testing real-time systems is even more challenging than testing untimed reactive systems, because the tester must now consider *when* to stimulate system, *when* to expect responses, and how to assign verdicts to the observed timed event sequence. Further, the test cases must be executed in real-time, i.e., the test execution system itself becomes a real-time system.

In this chapter we introduce a formal approach to model-based black-box conformance testing of real-time systems. We aim both at introducing timed testing to readers that are new in the area by giving many examples, and to more experienced readers by being formally precise and by touching on more advanced topics.

1.1 Approach and Chapter Outline

Real-time influences all aspects of test generation: The specification language must allow for the specification of real-time constraints. The conformance

R.M. Hierons et al. (Eds.): Formal Methods and Testing, LNCS 4949, pp. 77–117, 2008.

(implementation) relation must define what real-time behavior should be considered correct. It should be possible to specify what parts of the specified behavior should be tested. This can be done through test purposes, coverage criteria, or random exploration. Finally, the test generation algorithm must analyze the real-time specification, select and instantiate test cases, and output these in a timed test notation language. This computation must be done efficiently in order to handle large and complex specifications.

The timed automata formalism has become a popular and widespread formalism for specifying real-time systems. We adopt the particular UPPAAL style of timed automata. UPPAAL style timed automata have proven very expressive and convenient, but can still be analyzed efficiently. Section 2 introduces timed automata, their formal semantics in terms of timed labeled transition systems, and how to use timed automata to model and specify the behavior of real-time systems.

In the timed testing research community there is still no consensus on the exact conformance relation to use to evaluate the correctness of an implementation compared to its specification. Timed trace inclusion captures many of our intuitive expectations as well as having desired formal properties and is consistent with the widely accepted untimed input/output conformance-relation of Tretmans. We propose relativized timed input/output conformance relation between model and implementation under test (IUT) which coincides with timed trace inclusion taking assumptions about the environment behavior explicitly into account. In addition to allowing explicit and independent modelling of the environment, it also has some nice theoretical properties that allow testing effort to be reused when the environment or system requirements change. Relativized real-time input-output conformance is presented in Section 3.

Common approaches to test selection include test purposes or coverage criteria. When a model-checker is to be used to generate test sequences, the model is typically explicitly annotated with auxillary variables or automata that allow the test purpose or coverage criterion to be formulated as a reachability property that can be issued to the model-checker. In this chapter we present a more elegant approach where test purposes and coverage criteria can be formulated as *observer automata* that can be automatically superimposed on the model. This avoids explicit changes to the model, and allows the user to specify his own coverage criteria with relative ease. Observers and test generation using model-checking are presented in Section 4.

Given the model and observer automata, the problem becomes how to implement a test generator that efficiently can generate the required test suite. In Section 4.4 we propose an efficient algorithm that extends the basic reachability algorithm in UPPAAL with a compact bit-vector encoding of the specified coverage criteria.

The chapter illustrates two different approaches to timed testing which can be viewed as two extremes in a spectrum of possible approaches, offline and online testing, as depicted in Figure 1. In between these extremes are approaches that precompute a strategy or reduced specification (with particular test purpose in

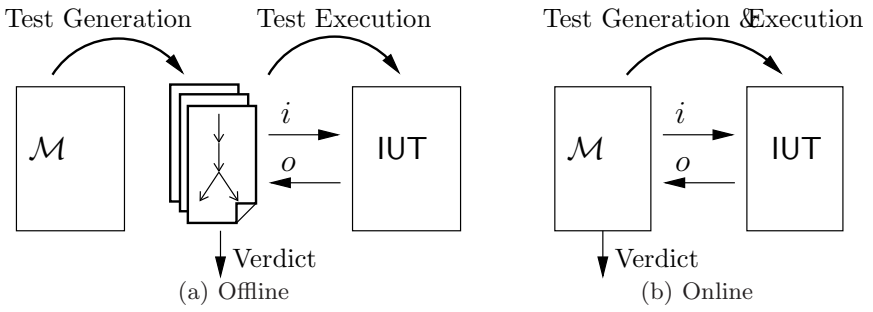


Fig. 1. Online vs. Offline Test Generation

mind) later to be executed online. Offline and online testing are compared below and discussed in detail in Sections 4 and 5.

1.2 Offline Test Generation

In *offline test generation* the test suite is pre-computed completely from the specification before it is executed on the implementation under test. Offline test generation inherits a general advantage of automated model-based testing such that when the requirements or model change, test cases can be automatically re-generated to reflect the change, rather than manually updating every test case and test script.

The advantages of offline test generation are that test cases are easier, cheaper, and faster to execute because all time constraints in the specification have been resolved at test generation time, and in addition, that the test suite can be generated with some a-priori guarantees, e.g., that the specification is structurally covered, or that a given set of test-objectives are met as fast or with as few resources as possible.

There are two main disadvantages of offline test generation. One is that the specification must be analyzed in its entirety, which often results in a state-explosion which limits the size of the specification that can be handled. Another problem is non-deterministic implementations and specifications. In this case, the output (and output timing) cannot be predicted, and the test case must be adaptive. Typically, the test case takes the form of a test-tree that branches for all possible outcomes. This may lead to very large test cases. In particular for real-time systems the test case may need to branch for all time instances where an output could arrive.

Offline test generators therefore often limit the expressiveness and amount of non-determinism of the specification language. This has been a particular problem for offline test generation from timed automata specifications, because the technique of determinizing the specification cannot be directly applied.

Given a restricted class of deterministic and output urgent timed automata we show in Section 4 how it is possible to use the unmodified UPPAAL

model-checker to synthesize test cases that are guaranteed to take the least possible time to execute. We also define a language for defining test purposes and coverage criteria, and present an efficient test generation algorithm.

1.3 Online Testing

Another testing approach is *online (on-the-fly) testing* that combines test generation and execution. Here the test generator interactively interprets the model, and stimulates and observes the IUT. Only a single test input is generated from the model at a time which is then immediately executed on the IUT. Then the produced output (if any) by the IUT as well as its time of occurrence are checked against the specification, a new input is produced and so forth until it is decided to end the test, or an error is detected. Typically, the inputs and delays are chosen randomly. An observed test run is a trace consisting of an alternating sequence of (input or output) actions and time delays.

There are several advantages of online testing. Testing may potentially continue for a long time (hours or even days), and therefore long, intricate test cases that stress the IUT may be executed. The state-space-explosion problem experienced by many offline test generation tools is reduced because only a limited part of the state-space needs to be stored at any point in time. Further, online test generators often allow more expressive specification languages, especially wrt. allowed non-determinism in real-time models: Since they are generated event-by-event they are automatically adaptive to the non-determinism of the specification and implementation. Online testing has proven an effective error detection technique [59, 62, 6].

A disadvantage is that the specification must be analyzed online and in real-time which require very efficient test generation algorithms to keep up with the implementation and specified real-time requirements. Also the test runs are typically long, and consequently the cause of a test failure may be difficult to diagnose. Although some guidance is possible, test generation is typically randomized which means that satisfaction of coverage criteria cannot be a priory guaranteed, but must instead be evaluated post mortem.

In Section 5 we present a sound and complete algorithm for online testing of real-time systems from timed automata specifications allowing full non-determinism. We describe an extension of Uppaal, named TRON, that implements this algorithm, and give an application example. We furthermore show how testing can be viewed as the two sub-problems of environment emulation and system monitoring, and we show how TRON can be configured to perform both combined or independently.

2 Specification of Real-Time Systems

This section formally presents our semantic framework, and introduces timed input/output transition systems (TIOTS), timed automata (TA), and our relativized timed input/output conformance relation.

2.1 Environment and System Modelling

An embedded system interacts closely with its environment which typically consists of the controlled physical equipment (the plant) accessible via sensors and actuators, other computer based systems or digital devices accessible via communication networks using dedicated protocols, and human users. A major development task is to ensure that an embedded system works correctly in its real operating environment. Due to lack of resources it is not feasible to validate the system for all possible (imaginary) environments. Also it is not necessary if the environments are known to a large extent. However, the requirements and the assumptions of the environment should be clear and explicit.

We denote the system being developed IUT, and its real operating environment RealENV. These communicate by exchanging *input* and *output* signals (seen from the perspective of IUT). Using a model-based development approach, the environment assumptions and system requirements are captured through abstract behavioral models denoted $\mathcal{E}$ and $\mathcal{S}$ respectively, communicating on abstract signals $i \in A_{in}$ and $o \in A_{out}$ corresponding (via a suitable abstraction) to the real *input* and *output*, see Figure 2.

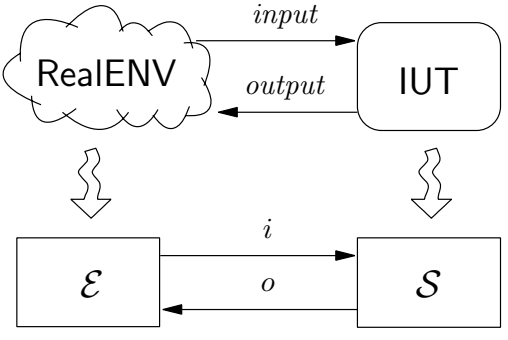


Fig. 2. Abstraction of a system

Modelling the environment explicitly and separately and taking this into account during test generation has several advantages: 1) the test generation tool can synthesize only relevant and realistic scenarios for the given type of environment, which in turn reduces the number of required tests and improves the quality of the test suite; 2) the engineer can guide the test generator to specific situations of interest; 3) a separate environment model avoids explicit changes to the system model if testing must be done under different assumptions or use patterns.

2.2 Timed I/O Transition Systems

To define our testing framework formally we need to introduce a semantic foundation for real-time systems. We use it to model systems and to define the formal semantics of timed automata. A timed input/output transition system (TIOTS)

is a labelled transition system where actions have been classified as inputs or outputs, and where dedicated delay labels model the progress of time. In our case we use the set of positive real-numbers to model time. Below we also extend commonly used notation for labeled transition systems to TIOTS.

Formal Definition of TIOTS. We assume a given set of actions A partitioned into two disjoint sets of output actions A_{out} and input actions A_{in}. In addition we assume that there is a distinguished unobservable action $\tau \notin A$. We denote by A_τ the set $A \cup \{\tau\}$.

A *timed I/O transition system* (TIOTS) $\mathcal{S}$ is a tuple $(S, s_o, A_{in}, A_{out}, \rightarrow)$, where

- S is a set of states, $s_0 \in S$,
- and $\rightarrow \subseteq\ S \times (A_\tau \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the usual constraints of *time determinism* (if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$ then $s' = s''$), *time additivity* (if $s \xrightarrow{d_1} s'$ and $s' \xrightarrow{d_2} s''$ then $s \xrightarrow{d_1+d_2} s''$), and *zero-delay* (for all states $s \xrightarrow{0} s$). $d, d_1, d_2 \in \mathbb{R}_{\geq 0}$, and $\mathbb{R}_{\geq 0}$ denotes non-negative real numbers.

Notation for TIOTS. Let $a, a_{1\ldots n} \in A$, $\alpha \in A_\tau \cup \mathbb{R}_{\geq 0}$, and $d, d_{1\ldots n} \in \mathbb{R}_{\geq 0}$. We write $s \xrightarrow{\alpha}$ iff $s \xrightarrow{\alpha} s'$ for some s'. We use $\Rightarrow$ to denote the τ-abstracted transition relation such that $s \stackrel{a}{\Rightarrow} s'$ iff $s \xrightarrow{\tau}{}^* \xrightarrow{a} \xrightarrow{\tau}{}^* s'$, and $s \stackrel{d}{\Rightarrow} s'$ iff $s \xrightarrow{\tau}{}^* \xrightarrow{d_1} \xrightarrow{\tau}{}^* \xrightarrow{d_2} \xrightarrow{\tau}{}^* \cdots \xrightarrow{\tau}{}^* \xrightarrow{d_n} \xrightarrow{\tau}{}^* s'$ where $d = d_1 + d_2 + \cdots d_n$. We extend $\Rightarrow$ to sequences in the usual manner.

$\mathcal{S}$ is *strongly input enabled* iff $s \xrightarrow{i}$ for all states s and for all input actions i. It is *weakly input enabled* iff $s \stackrel{i}{\Rightarrow}$ for all states s and for all input actions i. We assume that input actions (seen from the system point of view) are controlled by the environment and outputs are controlled by the system. An input enabled system cannot refuse input actions. However it may decide to ignore the input by executing a transition that results in the same state.

$\mathcal{S}$ is *non-blocking* iff for any state s and any $t \in \mathbb{R}_{\geq 0}$ there is a timed output trace $\sigma = d_1 o_1 \ldots o_n d_{n+1}$, $o_i \in A_{out}$, such that $s \stackrel{\sigma}{\Rightarrow}$ and $\sum_i d_i \geq t$. Thus $\mathcal{S}$ will not block time in any input enabled environment. This property ensures that a system will not force or rush its environment to deliver an input, and vice versa, the environment will never force outputs from the system. Time is common for both the system and its environment, and neither controls it.

To model potential implementations it is useful to define the properties of *isolated outputs* and *determinism*. $\mathcal{S}$ is deterministic if for all delays or actions $\alpha \in A_\tau \cup \mathbb{R}_{\geq 0}$, and all states s, whenever $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ then $s' = s''$. That is, the successor state of an action is always uniquely known.

We say that $\mathcal{S}$ has *isolated outputs* if whenever $s \xrightarrow{o}$ for some output action o, then $s \not\xrightarrow{\tau}$ and $s \not\xrightarrow{d}$ for all $d > 0$ and whenever $s \xrightarrow{o'}$ then $o' = o$. A system with isolated outputs will only offer one output at a time, and will never retract an offered output by performing internal actions or delays.

Finally, a TIOTS exhibits *output urgency* iff whenever an output (or τ) is enabled, it will occur immediately, i.e., whenever $s \xrightarrow{\alpha}, \alpha \in A_{out} \cup \{\tau\}$ then

$s \stackrel{d}{\not\rightarrow}, d \in \mathbb{R}_{\geq 0}$. An output urgent system will deliver the output immediately when ready.

An observable *timed trace* $\sigma \in (A \cup \mathbb{R}_{\geq 0})^*$ is of the form $\sigma = d_1 a_1 d_2 \ldots a_k d_{k+1}$. We define the observable timed traces $\mathsf{TTr}(s)$ of a state s as:

$$\mathsf{TTr}(s) = \{\sigma \in (A \cup \mathbb{R}_{\geq 0})^* \mid s \stackrel{\sigma}{\Rightarrow}\} \tag{1}$$

For a state s (and subset $S' \subseteq S$) and a timed trace σ, s After σ is the set of states that can be reached after σ:

$$s \ \mathsf{After}\ \sigma = \{\, s' \mid s \stackrel{\sigma}{\Rightarrow} s' \,\}, \quad S' \ \mathsf{After}\ \sigma = \bigcup_{s \in S'} s \ \mathsf{After}\ \sigma \tag{2}$$

The set $\mathsf{Out}(s)$ of observable outputs or delays from states $s \in S' \subseteq S$ is defined as:

$$\mathsf{Out}(s) = \{\, a \in A_{out} \cup \mathbb{R}_{\geq 0} \mid s \stackrel{a}{\Rightarrow} \}, \quad \mathsf{Out}(S') = \bigcup_{s \in S'} \mathsf{Out}(s) \tag{3}$$

TIOTS Composition. Let $\mathcal{S} = (S, s_0, A_{in}, A_{out}, \rightarrow)$ and $\mathcal{E} = (E, e_o, A_{out}, A_{in}, \rightarrow)$ be TIOTSs. Here E is the set of environment states and the set of input (output) actions of $\mathcal{E}$ is identical to the output (input) actions of $\mathcal{S}$. The parallel composition of $\mathcal{S}$ and $\mathcal{E}$ forms a *closed system* $\mathcal{S} \parallel \mathcal{E}$ whose observable behavior is defined by the TIOTS $(S \times E, (s_0, e_0), A_{in}, A_{out}, \rightarrow\ )$ where $\rightarrow$ is defined as

$$\frac{s \stackrel{a}{\rightarrow} s' \quad e \stackrel{a}{\rightarrow} e'}{(s,e) \stackrel{a}{\rightarrow} (s',e')} \quad \frac{s \stackrel{\tau}{\rightarrow} s'}{(s,e) \stackrel{\tau}{\rightarrow} (s',e)} \quad \frac{e \stackrel{\tau}{\rightarrow} e'}{(s,e) \stackrel{\tau}{\rightarrow} (s,e')} \quad \frac{s \stackrel{d}{\rightarrow} s' \quad e \stackrel{d}{\rightarrow} e'}{(s,e) \stackrel{d}{\rightarrow} (s',e')} \tag{4}$$

2.3 Timed Automata

Timed automata [2] is an expressive and popular formalism for modelling real-time systems. Essentially a timed automaton is an extended finite state machine equipped with a set of real-valued clock-variables that track the progress of time and that can guard when transitions are allowed.

Formal Definition of Timed Automata. Let X be a set of $\mathbb{R}_{\geq 0}$-valued variables called *clocks*. Let $\mathcal{G}(X)$ denote the set of *guards* on clocks being conjunctions of constraints of the form $x \bowtie c$, and let $\mathcal{U}(X)$ denote the set of *updates* of clocks corresponding to sequences of statements of the form $x := c$, where $x \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$. A *timed automaton* over (A, X) is a tuple (L, ℓ_0, I, E), where

- L is a set of locations, $\ell_0 \in L$ is an initial location,
- $I : L \rightarrow \mathcal{G}(X)$ assigns invariants to locations, and
- E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times A_\tau \times \mathcal{U}(X) \times L$.

We write $\ell \xrightarrow{g,\alpha,u} \ell'$ iff $(\ell, g, \alpha, u, \ell') \in E$.

The semantics of a TA is defined in terms of a TIOTS over states of the form $s = (\ell, \bar{v})$, where ℓ is a location and $\bar{v} \in \mathbb{R}^X_{\geq 0}$ is a clock valuation satisfying the invariant of ℓ. Intuitively, a timed automaton can either progress by executing an edge or by remaining in a location and letting time pass:

$$\frac{\forall d' \leq d.\ I_\ell(d')}{(\ell, \bar{v}) \stackrel{d}{\rightarrow} (\ell, \bar{v} + d)} \quad \frac{\ell \xrightarrow{g,\alpha,u} \ell' \ \wedge g(\bar{v}) \ \wedge I_{\ell'}(\bar{v}'),\ \bar{v}' = u(\bar{v})}{(\ell, \bar{v}) \stackrel{\alpha}{\rightarrow} (\ell', \bar{v}')} \tag{5}$$

In delaying transitions, $(\ell, \bar{v}) \xrightarrow{d} (\ell, \bar{v} + d)$, the values of all clocks of the automaton are incremented by the amount of the delay d, denoted $\bar{v} + d$. The automaton may delay in a location ℓ as long as the invariant I_ℓ for that location remains true. Discrete transitions $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}')$ correspond to execution of edges $(\ell, g, \alpha, u, \ell')$ for which the guard g is satisfied by $\bar{v}$, and for which the invariant of the target location $I_{\ell'}$ is satisfied by the updated clock valuation $\bar{v}'$. The target state's clock valuation $\bar{v}'$ is obtained by applying clock updates u on $\bar{v}$.

Uppaal Timed Automata. Throughout this chapter we use Uppaal syntax to illustrate TA, and the figures are direct exports from Uppaal. Uppaal allows construction of large models by composing timed automata in parallel and lets these communicate using shared discrete and clock variables and synchronize (rendezvous-style) on complementary input and output actions, as well as broadcast actions.

Initial locations are marked using a double circle. Edges are by convention labeled by the triple: guard, action, and assignment in that order. The internal τ-action is indicated by an absent action-label. Committed locations are indicated by a location with an encircled "C". A committed location must be left immediately by the next transition taken in the system. An urgent location (encircled "U") must be left without letting time pass, but allows interleaving by other automata. Finally, bold-faced clock conditions placed under locations are location invariants. In addition to clocks, Uppaal also allows integer variables to be used in guards and assignments.

The latest version further supports a safe subset of C-code in assignments and guards, and C-data-structures.

Example 1. Fig. 3 shows a TA modelling the behavior of a simple light-controller. The user interacts with the controller by touching a touch sensitive pad. The light has three intensity levels: `OFF`, `DIMMED`, and `BRIGHT`. Depending on the timing

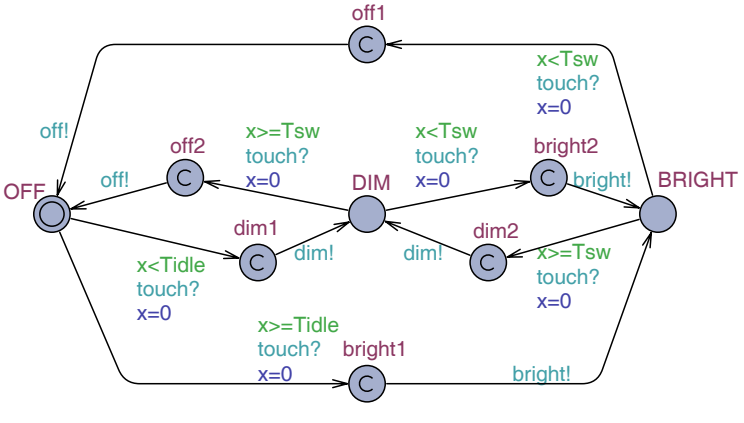
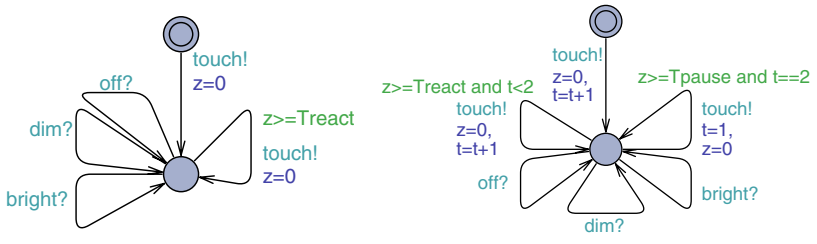


Fig. 3. Light Controller

between successive touches (recorded by the clock x), the controller toggles the light levels. For example, in dimmed state, if a second touch is made quickly (before the switching time $T_{sw} = 4$ time units) after the touch that caused the controller to enter dimmed state (from either off or bright state), the controller increases the level to bright. Conversely, if the second touch happens after the switching time, the controller switches the light off. If the light controller has been in off state for a long time (longer than or equal to $T_{idle} = 20$), it should reactivate upon a touch by going directly to bright level.

The simple light controller can perform the execution sequence $(\texttt{OFF}, x = 0) \xrightarrow{5} (\texttt{OFF}, x = 5) \xrightarrow{touch?} (\texttt{dim1}, x = 0) \xrightarrow{dim!} (\texttt{DIM}, x = 0) \xrightarrow{3.14} (\texttt{DIM}, x = 3.14) \xrightarrow{touch?} (\texttt{bright2}, x = 0) \xrightarrow{bright!} (\texttt{BRIGHT}, x = 0)$ resulting in the observable trace $\sigma = 5 \cdot touch? \cdot dim! \cdot 3.14 \cdot touch! \cdot bright!$. Note that $\{(\texttt{OFF}, x = 0)\}$ After $\sigma = \{(\texttt{BRIGHT}, x = 0)\}$, $\mathsf{Out}\big(\{(\texttt{OFF}, x = 0)\} \text{ After } \sigma\big) = \mathbb{R}_{\geq 0}$, but $\mathsf{Out}\big((\texttt{bright2}, x = 0)\big) = \{bright!\} \cup \{0\}$.



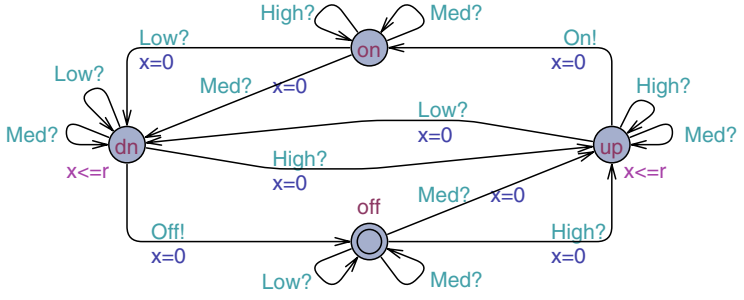
(a) (b)

Fig. 4. Two possible environment models for the simple light switch

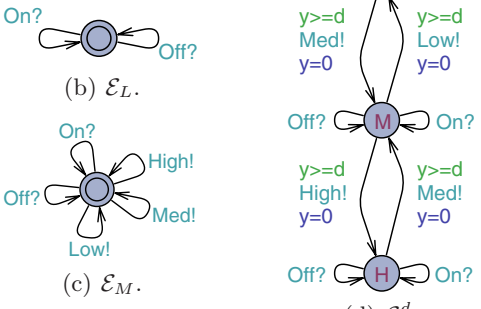
Figure 4 shows two possible environment models for the simple light controller. Figure 4(a) models a user capable of performing any sequence of touch actions. When the constant T_{react} is set to zero he is arbitrarily fast. A more realistic user is only capable of producing touches with a limited rate; this can be modeled setting T_{react} to a non-zero value. Figure 4(b) models a different user able to make two quick successive touches (counted by integer variable `t`), but which then is required to pause for some time (to avoid cramp), e.g., $T_{pause} = 5$.

The TA shown in Figure 3 and Figure 4 respectively can be composed in parallel on actions $A_{in} = \{\mathsf{touch}\}$ and $A_{out} = \{\mathsf{off}, \mathsf{dim}, \mathsf{bright}\}$ forming a closed network (to avoid cluttering the figures we may sometimes omit making them explicitly input enabled; for the unspecified inputs there is a non-drawn self looping edge that merely consumes the input without changing the location).

Example 2. Figure 5(a) shows a timed automaton specification $\mathcal{C}^r$ for a controller whose goal is to control and keep the room temperature in *Med* range by turning *On* and *Off* the room cooling device. The controller is required: 1) to turn *On* the cooling device within an allowed reaction time r when the room temperature reaches *High* range, and 2) to turn it *Off* within r when the temperature drops to *Low* range. Observe how location invariants are used to force the automaton to



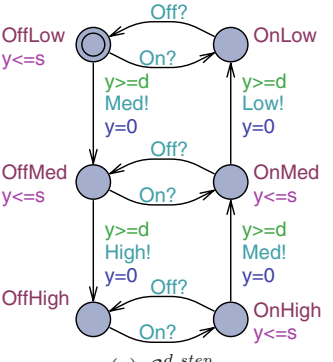
(a) $\mathcal{C}^r$: simple cooling controller with reaction time r.



(b) $\mathcal{E}_L$.

(c) $\mathcal{E}_M$.

(d) $\mathcal{E}_1^d$.



(e) $\mathcal{E}_2^{d,step}$.

Fig. 5. Timed automata of simple controller and various environments

leave the `dn` and `up` locations before the reaction time has elapsed, in consequence producing the output at some time before the required reaction time. When the room temperature is medium the cooling is allowed to be either on or off.

This specification is non-deterministic in two ways. First, there are several next states to a *Med* temperature, e.g.,

$$\{(\texttt{off}, x = 0)\} \text{ After } 5 \cdot \mathit{Med}? = \{(\texttt{off}, x = 5), (\texttt{up}, x = 0)\}. \tag{6}$$

Second, the controller switches state *within* the reaction time r, but it is unknown when. Thus from e.g., state $(\texttt{up}, x = 0)$ the controller may execute any of the observable traces, $d \cdot \mathit{On}!, 0 \leq d \leq r$. Note that

$$\mathsf{Out}\big((\texttt{up}, x = 0)\big) = \{\mathit{On}!\} \cup \{d \mid 0 \leq d \leq r\}. \tag{7}$$

The intention of this specification (given our conformance relation) is to allow implementation freedom to the manufacturer wrt. exact functionality, speed, timing tolerances, etc..

The Uppaal Tool. In the Uppaal tool it is possible to edit, simulate and check properties of Uppaal timed automata in a graphical environment. The

property specification language supports safety, liveness, deadlock, and response properties.

In this chapter we use the UPPAAL tool for offline test generation by expressing the test case generation problem as a safety property that can be solved by reachability analysis. Safety properties are used to expresses requirements of the form "the model can never reach an undesired state". The dual properties like "the system can reach a desired state", are usually referred to as reachability properties.

When checking a safety property, the UPPAAL tool performs symbolic reachability analysis of the network of timed automata to search for reachable states where the property is satisfied (or not satisfied). If a state that satisfies the property is found, UPPAAL generate a diagnostic traces witnessing a submitted safety property. Currently UPPAAL supports three options for diagnostic trace generation: *some trace* leading to the goal state, the *shortest trace* with the minimum number of transitions, and *fastest trace* with the shortest accumulated time delay. The underlying algorithm used for finding time-optimal traces is a variation of the A*-algorithm [5,39]. Hence, to improve performance it is possible to supply a heuristic function estimating the remaining cost from any state to the goal state.

To perform reachability analysis of (densely) timed automata UPPAAL uses a (finite) symbolic representation of the state space and symbolic computation steps.

A *symbolic state* is of the form (ℓ, D), where ℓ is a control location of a timed automaton and D is a convex subset of $\mathbb{R}^{|X|}_{\geq 0}$, i.e. it represents the (potentially infinite) set of concrete states $\{(\ell', \bar{v}) \mid \ell' = \ell \wedge \bar{v} \in D\}$. The initial symbolic state is (ℓ_0, D_0), where $D_0 = \{\ \bar{v} \mid (\ell_0, \bar{v}_0) \xrightarrow{d} (\ell_0, \bar{v})\ \}$ and $\bar{v}_0$ is the clock valuation assigning all clocks to zero.

A *symbolic computation step* $(\ell, D) \xrightarrow{\alpha} (\ell', D')$ consists of performing an action followed by some delay, and can be performed iff $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}')$, and $D' = \left\{\ \bar{v}'' \mid (\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}') \ \wedge\ (\ell', \bar{v}') \xrightarrow{d} (\ell', \bar{v}'') \ \wedge\ \bar{v} \in D\ \right\}$.

It is possible to represent a convex subset D as a so-called *difference bounded matrix* [21] that can be efficiently manipulated by constraint-solving techniques [53], implemented as model-checking tools such as UPPAAL and Kronos [20].

3 Relativized Timed Conformance

In this section we define our notion of conformance between TIOTSs. Our notion derives from the input/output conformance relation (ioco) of Tretmans and de Vries [58, 63] by taking time and environment constraints into account. Under assumptions of weak input enabledness our relativized timed conformance relation (denoted rtioco_e) coincides with relativized timed trace inclusion. Like ioco, this relation ensures that the implementation has only the behavior allowed by the specification. In particular, 1) it is not allowed to produce an output at a time when one is not allowed by the specification, 2) it is not allowed to omit producing an output when one is required by the specification.

The ioco relation operates with the concept of *quiescence* allowing (eternal) absence of outputs to be observed by means of a finite time out, and be equalized with a special observable action, resulting in a more discriminating relation. It is debatable whether the same abstraction is reasonable in real-time systems. Briones et al. have proposed relations that allows this [12], see also the discussion in Section 6.1. Our relation takes the view that only finite progress of time can be observed in a real-time system. Thus, rtioco_e offers the notion of time-bounded (finite) quiescence, that—in contrast to ioco's conceptual eternal quiescence—can be observed in a real-time system.

Formal Definition of rtioco_e. Let $\mathcal{S} = (S, s_0, A_{in}, A_{out}, \rightarrow)$ be an weak-input enabled and non-blocking TIOTS. An *environment* for $\mathcal{S}$ is itself a weak-input enabled and non-blocking TIOTS $\mathcal{E} = (E, e_o, A_{out}, A_{in}, \rightarrow)$ with reversed inputs and outputs.

Given an environment $e \in E$ the e-relativized timed input/output conformance relation rtioco_e between system states $s, t \in S$ is defined as:

$$s \ \mathsf{rtioco}_e \ t \quad \text{iff} \quad \forall \sigma \in \mathsf{TTr}(e).\, \mathsf{Out}\big((s,e) \ \mathsf{After} \ \sigma\big) \subseteq \mathsf{Out}\big((t,e) \ \mathsf{After} \ \sigma\big)$$

Whenever $s \ \mathsf{rtioco}_e \ t$ we will say that s is a correct implementation (or refinement) of the specification t under the environmental constraints expressed by e. Under the assumption of weak input-enabledness of both $\mathcal{S}$ and $\mathcal{E}$ we may characterize relativized conformance in terms of trace-inclusion as follows:

Lemma 1. *Let $\mathcal{S}$ and $\mathcal{E}$ be input-enabled with states $s, t \in S$ and $e \in E$ resp., then*

$$s \ \mathsf{rtioco}_e \ t \quad \text{iff} \quad \mathsf{TTr}(s) \cap \mathsf{TTr}(e) \subseteq \mathsf{TTr}(t) \cap \mathsf{TTr}(e)$$

Thus if $s \ \mathsf{rtioco}_e \ t$ does not hold then there exists a trace σ of e such that $s \stackrel{\sigma}{\Rightarrow}$ but $t \stackrel{\sigma}{\not\Rightarrow}$. Given the notion of relativized conformance it is natural to consider the preorder on environments based on their discriminating power, i.e. for environments e and f:

$$e \sqsubseteq f \qquad \text{iff} \qquad \mathsf{rtioco}_f \subseteq \mathsf{rtioco}_e \tag{8}$$

(to be read f is more discriminating than e). It follows from the definition of rtioco that $e \sqsubseteq f$ iff $\mathsf{TTr}(e) \subseteq \mathsf{TTr}(f)$. In particular there is a most (least) discriminating (weakly) input enabled and non-blocking environment U (O) given by $\mathsf{TTr}(U) = (A \cup \mathbb{R}_{\geq 0})^*$ $\big(\mathsf{TTr}(O) = (A_{out} \cup \mathbb{R}_{\geq 0})^*\big)$. The corresponding conformance relation rtioco_U (rtioco_O) specializes to simple timed trace inclusion (timed output trace inclusion) between system states.

Moreover, because we treat environment constraints explicitly and separately, rtioco_e has some nice theoretical and practical attractive properties that allows the tester to re-use testing effort if either the environment assumption is strengthened, or if the system specification is weakened. Assume that $i \ \mathsf{rtioco}_e \ s$, then without re-testing

$$\textit{if } s \sqsubseteq s' \textit{ then } i \ \mathsf{rtioco}_e \ s' \tag{9}$$

$$\textit{if } e' \sqsubseteq \ e \textit{ then } i \ \mathsf{rtioco}_{e'} \ s \tag{10}$$

In the following we exemplify how our conformance relation discriminates systems, and illustrate the potential power of environment assumptions and how this can help to increase the relevance of the generated tests for a given environment.

Example 3. Consider the simple cooling controller $\mathcal{C}^r$ of Figure 5(a), where r is a parameter r with its reaction time, and the environment in Figure 5(c).

Take $\mathcal{C}^{10}$ to be the specification and assume that the implementation behaves like $\mathcal{C}^{12}$. Clearly, $\mathcal{C}^8$ $\mathsf{rtioco}_{\mathcal{E}_M}$ $\mathcal{C}^6$ because $\sigma = 0 \cdot \mathit{Med}! \cdot 7 \cdot \mathit{On}! \in \mathsf{TTr}(\mathcal{C}^8)$, but $\sigma \notin \mathsf{TTr}(\mathcal{C}^6)$, or alternatively, $\mathsf{Out}\big(\mathcal{C}^8 \text{ After } 0 \cdot \mathit{Med}! \cdot 7\big) = \{\mathit{On}!\} \cup \mathbb{R}_{\geq 0} \not\subseteq \mathsf{Out}\big(\mathcal{C}^6 \text{ After } 0 \cdot \mathit{Med}! \cdot 7\big) = \mathbb{R}_{\geq 0}$ (recall that C^r may remain in location *off* on input *Med* and not produce any output). The implementation can thus perform an output at a time not allowed by the specification.

Next, suppose $\mathcal{C}^r$ is implemented by a timed automaton $\mathcal{C}'^r$ equal to $\mathcal{C}^r$, except the transition $up \xrightarrow{Low} dn$ is missing, and replaced by a self loop $up \xrightarrow{Low} up$.

They are distinguishable by the timed trace $0{\cdot}\mathit{Med?}{\cdot}0{\cdot}\mathit{High?}{\cdot}0{\cdot}\ \mathit{Low?}{\cdot}0{\cdot}\ \mathit{On!}$ in the implementation that is not in the specification (switches the compressor *Off* instead).

Example 4. Figure 5(c) shows the universal (most general) and completely unconstrained environment $\mathcal{E}_M$ where room temperature may change unconstrained and may change (discretely) with any rate. This may not be realistic in the given physical environment, and there may be less need to test the controller in such an environment, as temperature normally evolves slowly and continuously, e.g., it cannot change drastically from *Low* to *High* and back unless through *Med*. Similarly, most embedded and real-time systems also interact with physical environments and other digital systems that— depending on circumstances—can be assumed to be correct and correctly communicate using well defined interfaces and protocols.

Figures 5(b) to 5(e) show four possible environment assumptions for $\mathcal{C}^r$. Figure 5(c) and Figure 5(b) shows respectively the most discriminating and least discriminating environments. Figure 5(d) shows the environment model $\mathcal{E}_1^d$ where the temperature changes through *Med* range and with a speed bounded by d. Figure 5(e) shows an even more constrained environment $\mathcal{E}_2^{d,s}$ that assumes that the cooling device works, e.g., temperature changes with an upper and lower speed bounded by d and s.

Notice that $\mathcal{E}_2$ and $\mathcal{E}_1$ have less discriminating power and thus may not reveal faults found under more discriminating environments. However, if the erroneous behavior is impossible in the actual operating environment the error may be irrelevant. Consider again the implementation $\mathcal{C}'^r$ from above. This error can be detected under $\mathcal{E}_0$ and $\mathcal{E}_1^k$ if $k = 3d$ and $r > k$, via the timed trace that respects $d{\cdot}\mathit{Med?}{\cdot}d{\cdot}\mathit{High?}{\cdot}d{\cdot}\mathit{Med?}{\cdot}d{\cdot}\mathit{Low?}{\cdot}\varepsilon{\cdot}\ \mathit{On!}$, $\varepsilon \leq r$. The specification would produce *Off*. The error cannot be detected under $\mathcal{E}_1$ if it is too slow $3d > r$, and never under $\mathcal{E}_2$ for no value of d.

In the extreme the environment behavior can be so restricted that it only reflects a single test scenario that should be tested. In our view, the environment assumptions should be specified explicitly and separately.

4 Offline Test Generation

In this section, we describe an offline test generation approach for real-time systems specified as timed automata. In order to specify that a certain level of thoroughness is achieved in the testing we shall require that a generated test suite satisfies a given coverage criterion. For untimed systems coverage criteria have been studied by researchers for many years, and a number specific coverage criteria have been proposed in the literature, including [45,52,14,15,17,26,41,51, 49, 23, 54]. In comparison, research in real-time coverage criteria is still a more immature area where not many general results are available. Therefore, most of the coverage criteria and test generation techniques described in this section were originally proposed for testing of untimed systems. However, they can often be adopted for the domain of real-time system. For example, the well-known all-definitions use-pair coverage criterion [26,41] (described in Sections 4.2 and 4.3), can be applied to definitions and uses of timers, as well as data variables.

We will see in Section 4.2 how test case generation can be performed by reformulating the problem as a model-checking problem that can be solved by a model-checking tool like Uppaal. This will require that the original system is annotated with variables that are needed to formulate the test case generation problem as a model-checking problem. For intricate coverage criteria, it can be cumbersome to find and manually do the right model annotations. The auxiliary variables also add extra complexity to the timed automata model. In Section 4.3 we present a formal language to specify coverage criteria and we review an algorithm which handles the extra information directly in the algorithm. In this way the process becomes more user friendly, and the coverage information can be dealt with more efficiently using a bit-vector representation.

In order to make offline test case generation applicable to timed automata specifications, we shall assume that the underlying TIOTS is *deterministic*, *weakly input enabled*, *output urgent*, with *isolated outputs* as defined in Section 2.2. This means that the $\mathcal{S}$ is assumed to react deterministically to any input provided, and will always be able to accept input from the test case. At any state, the $\mathcal{S}$ is also assumed to always have at most one output action that will occur immediately.

Further, as discussed in Section 2, we shall assume that the test specification is given as a closed network of TA that can be partitioned into one subnetwork $\mathcal{S}$ specifying the required behavior of the IUT, and one subnetwork $\mathcal{E}$ modelling the behavior of its intended environment RealENV, as depicted in Fig. 2.

4.1 Test-Case Generation by Model-Checking

When generating test cases by model-checking, the idea is to formulate the problem as a reachability problem that can be solved with an existing model-checking tool. As mentioned, we will use the Uppaal tool introduced in Section 2.3 to perform reachability analysis of timed automata. More precisely, we shall use a boolean combination of comparisons between integer constants and variables in the model to characterise a desired state to be reached.

In Section 4.2, we will describe in more details how UPPAAL's ability to produce traces witnessing a posed reachability property can be used to produce test cases for a given test purpose or coverage criteria. First, we describe how diagnostic traces can be interpreted as test cases.

From Diagnostic Traces to Test Cases. Let A be a TA composition of an IUT model $\mathcal{S}$ and a model $\mathcal{E}$ of its intended environment RealENV. A diagnostic trace produced by UPPAAL for a given reachability question on A demonstrates the sequence of moves to be made by each of the system components and the required clock constraints needed to reach the target location. A (concrete) diagnostic trace will have the form:

$$(s_0, e_0) \xrightarrow{\gamma_0} (s_1, e_1) \xrightarrow{\gamma_1} (s_2, e_2) \xrightarrow{\gamma_2} \cdots (s_n, e_n)$$

where s_i, e_i are states of the $\mathcal{S}$ and $\mathcal{E}$, respectively, and γ_i are either time-delays or synchronization (or internal) actions. The latter may be further partitioned into purely $\mathcal{S}$ or $\mathcal{E}$ transitions (hence invisible for the other part) or synchronizing transitions between the IUT and the RealENV (hence observable for both parties).

A *test sequence* is an alternating sequence of concrete delay actions and observable actions. From the diagnostic trace above a *test sequence*, $\lambda \in A_{in} \cup A_{out} \cup \mathbb{R}_{\geq 0}$, may be obtained simply by projecting the trace to the $\mathcal{E}$-component, while removing invisible transitions, and summing adjacent delay actions. Finally, a *test case* to be executed on the real IUT implementation may be obtained from λ by the addition of *verdicts*.

First note that with the assumptions made on the underlying TIOTS made above, the conformance relation specializes to timed trace inclusion, as discussed in Section 3. Thus, after any input sequence, the implementation is allowed to produce an output only if the specification is also able to produce that output. Similarly, the implementation may delay (thereby staying silent) only if the specification also may delay. The test sequences produced by our techniques are derived from diagnostic traces, and are thus guaranteed to be included in the specification.

To clarify the construction we may model the test case itself as a TA A_λ for the test sequence λ. Locations in A_λ are labelled using two distinguished labels, `PASS` and `FAIL`. The execution of a test case is now formalized as the composition of the test case automaton A_λ and IUT A_I.

$$\text{IUT passes } \lambda \textit{ iff } A_\lambda \parallel A_I \not\rightarrow^* \texttt{FAIL}$$

A_λ is constructed such that a *complete execution* terminates in a `FAIL` state if the IUT cannot perform λ and such that it terminates in a `PASS` state if the IUT can execute all actions of λ. The construction is illustrated in Figure 6.

4.2 Coverage-Based Test Case Generation

We shall see how test cases satisfying a given *coverage criterion* can be generated by model-checking. A common approach to the generation of test cases is to first

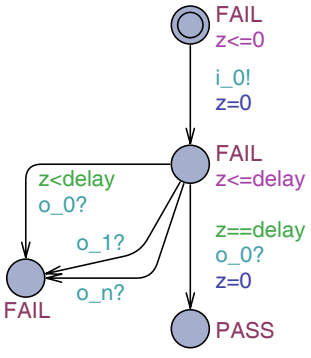


Fig. 6. Test case automaton for the sequence $i_0! \cdot delay \cdot o_0?$

manually formulate a set of informal test purposes and then to formalize these such that the model can be used to generate one or more test cases for each test purpose.

Test Purposes. A test purpose is a specific test objective (or property) that the tester would like to observe on the IUT. We will formulate the test purpose as a property that can be checked by reachability analysis of the combined $\mathcal{E}$ and $\mathcal{S}$ model. Different techniques can be used for this purpose. Sometimes the test purpose can be directly transformed into a simple model-checking property expressed as a boolean combination of automata locations. In other cases it may require decoration of the model with auxiliary flag variables. Another technique is to replace the environment model with a more restricted one that matches the behavior of the test purpose only.

Example 5. We exemplify these two approaches using the following two test purposes expressing test objectives of the simple light controller in Example 1.

TP1: Check that the light can become bright.
TP2: Check that the light switches off after three successive touches.

The test purpose **TP1** can be formulated as a simple reachability property requiring that eventually the `lightContoller` can enter location `BRIGHT`. Generating the *shortest* diagnostic trace results in the test sequence:

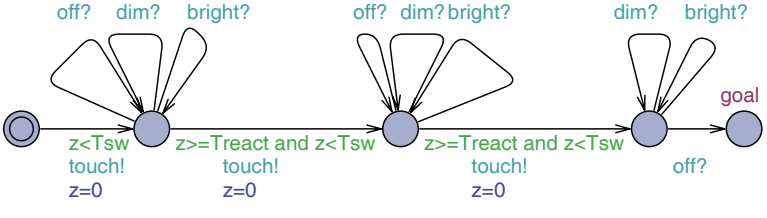


Fig. 7. Test Environment for TP2

$$20 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{bright}?$$

However, the *fastest sequence* satisfying the test purpose is

$$0 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{dim}? \cdot 0 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{bright}?$$

The test purpose **TP2** can be formulated by a reachability property requiring that a location in a specific environment automaton can be reached. In Figure 7 an environment automaton `tpEnv` for **TP2** is shown. The automaton restricts the possible user input so that there is at least `Treact` time units in between two consecutive touches. The fastest test sequence satisfying the test purpose is:

$$0 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{dim}? \cdot \mathit{Treact} \cdot \mathit{touch}! \cdot 0 \cdot \mathit{bright}? \cdot \mathit{Treact} \cdot \mathit{touch}! \cdot 0 \cdot \mathit{off}?$$

Coverage Criteria. Often the tester is interested in creating a test suite that ensures that the specification or implementation is covered in a certain way. This ensures that a certain level of thoroughness has been achieved in the test generation process. Here we explain how test sequences with guaranteed coverage of the IUT model can be computed by model-checking, effectively giving automated tool support.

A large suite of coverage criteria have been proposed in the literature, such as statement, transition, and definition-use coverage, each with its own merits and application domain. We explain how to apply some of these to TA models (more coverage criteria will be introduced in Section 4.3).

Edge Coverage: A test sequence satisfies the *edge-coverage criterion* [45] if, when executed on the model, it traverses every edge of the selected TA-components. Edge coverage can be formulated as a reachability property in the following way: add an auxiliary variable e_i of type boolean (initially false) for each edge to be covered (typically realized as a bit array in UPPAAL), and add to the assignments of each edge i an assignment $e_i :=$ **true**; a test suite can be generated by formulating a property requiring that a state can be reached in which all e_i variables are true, i.e., $(e_0{==}\textbf{true} \wedge e_1{==}\textbf{true} \wedge \ldots \wedge e_n{==}\textbf{true})$. The auxiliary variables are needed to enable formulation of the coverage criterion as a reachability property using the UPPAAL property specification language which is a restricted subset of *timed computation tree logic* (TCTL) [3].

Location Coverage: A test sequence satisfies the *location-coverage criterion* [45] if, when executed on the model, it visits every location of the selected TA-components. To generate test sequences with location coverage, we introduce an auxiliary variable b_i of type boolean (initially false for all locations except the initial) for each location ℓ_i to be covered. For every edge with destination ℓ_i: $\ell' \xrightarrow{g,a,u} \ell_i$ add to the assignments u $b_i{:=}\textbf{true}$; the reachability property will then require all b_i variables to be true.

Definition-Use Pair Coverage: The definition-use pair criterion [17] is a data-flow coverage technique where the idea is to cover paths in which a variable is *defined*, i.e. appears in the left-hand side of an assignment, and later is *used*, i.e. appears in a guard or the right-hand side of an assignment.

We use (v, e_d, e_u) to denote a *definition-use pair* (DU-pair) for variable v if e_d is an edge where v is defined and e_u is an edge where v is used. A DU-pair (v, e_d, e_u) is valid if e_u is reachable from e_d and v is not redefined in the path from e_d to e_u. A test sequence covers (v, e_d, e_u) iff (at least) once in the sequence, there is a valid DU-pair (v, e_d, e_u). A test sequence satisfies the (all-uses) DU-pair coverage criterion of v if it covers all valid DU-pairs of v.

To generate test sequences with definition-use pair coverage, we assume that the edges for a model are enumerated, so that e_i is the number of edge i. We introduce an auxiliary data-variable v_d (initially **false**) with value domain $\{\mathbf{false}\} \cup \{1 \ldots |E|\}$ to keep track of the edge at which variable v was last defined, and a two-dimensional boolean array du of size $|E| \times |E|$ (initially **false**) to store the covered pairs. For each edge e_i at which v is defined we add $v_d := e_i$, and for each edge e_j at which v is used we add the conditional assignment $\mathit{if}\,(v_d \neq \mathbf{false})\mathit{then}\ du[v_d, e_j] := \mathbf{true}$. Note that if v is both used and defined on the same edge, the array assignment must be made before the assignment of v_d.

The reachability property will then require all $du[i, j]$ representing valid DU-pairs to be true for the (all-uses) DU-pair criterion. Note that a test sequence satisfying the DU-pair criterion for several variables can be generated using the same encoding, but extended with one auxiliary variable and array for each covered variable.

Example 6. The light switch in Figure 3 requires a bit-array of 12 elements (one per edge). When the environment can touch arbitrarily fast the generated fastest edge covering test sequence has the accumulated execution time 28. The solution (there might be more traces with the same fastest execution time) generated by UPPAAL is:

$$0 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{dim}? \cdot 0 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{bright}?\cdot$$
$$0 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{off}? \cdot 20 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{bright}?\cdot$$
$$4 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{dim}? \cdot 4 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{off}?$$

4.3 Test Case Generation Using Observers

As described in the previous section, it is in principle possible to generate test cases by annotating UPPAAL timed automata with auxiliary variables, and solve the problem by reachability analysis. However, for more intricate coverage criteria it can be cumbersome and very time-consuming to find the proper model annotations. Another problem with using model-checking algorithms and tools to generate test cases is that they are not really tailored for the problem, which may lead to problems with performance.

In this section, we shall present another approach to offline test case generation for real-time systems modeled as timed automata. Instead of using model annotations and reachability properties to specify coverage criteria, we shall present a language of *observers* as a generic and formal specification language for coverage criteria. We shall further see how to adapt a model-checking algorithm to

internally handle information about coverage, so that test-case generation can be performed in a more efficient way.

The observers presented here are based on the notion of observers described by Blom et.al., in [7]. In their setting, observers are used to express coverage criteria of test cases generated from system specification described as extended finite state machines (EFSMs). In this section, we shall review their work and adapt the results to our setting, i.e., for timed automata specifications of real-time systems. We first describe how observers are used to specify coverage criteria.

The Observer Language. As we have seen, a coverage criterion typically consists of a (rather large) set of items that should be "covered" or examined by the test suite. The set of items to be covered is derived from a more general criterion, requiring that some property ψ should be fulfilled, where ψ is a logical property characterizing the items to be covered. For example, ψ could be satisfied for all locations or edges of a model, to characterize the location of edge coverage criteria mentioned in the previous section. In the following, we will use the term *coverage item* for an item satisfying ψ, and assume that a coverage criterion is to cover as many coverage items ψ as possible of a model.

Using standard techniques from model-checking and run-time verification it is possible to represent a coverage item by an observer that monitors how a timed automaton executes. Whenever a coverage item characterized by the observer is fulfilled, the observer will "accept" the trace. We shall assume that an observer can observe the actions in a trace of an automaton, and also other details about the timed automata performing the action, such as the source and target locations, and the values of its state variables. This will make it possible to characterise a wide range of coverage criteria as observers.

Formally, an *observer* of a timed automaton $\mathcal{S} = (L, \ell_0, I, E)$ is a tuple (Q, q_0, Q_f, B) where

- Q is a finite set of *observer locations*
- q_0 is the *initial observer location.*
- $Q_f \subseteq Q$ is a set of *accepting observer locations.*
- B is a set of edges, each of form $q \xrightarrow{b} q'$ where $q, q' \in Q$ and b is a predicate that depend on the $\mathcal{S}$ transition $(\ell, \overline{v}) \xrightarrow{\alpha} (\ell', \overline{v}')$. The evaluation of b can depend on an input/output action α, and/or the syntactic edge $\ell \xrightarrow{g,\alpha,u} \ell'$ the $\mathcal{S}$ transition is derived from.[1]

In many cases, the initial location q_0 has an edge to itself with the predicate **true**. We use the symbol $\bullet$ to represent q_0 together with such a self-loop. Similarly, we assume that each $q_f \in Q_f$ has an edge to itself with the predicate **true**. We use the symbol $\circledcirc$ to represent accepting locations. Intuitively, the loop in q_0 is often used to allow the observer to "non-deterministically" start monitoring at any point in a timed trace. The loop in each q_f is used to allow an observer to stay in an accepting location.

[1] For UPPAAL timed automata extended by variables, b can also depend on the variables.

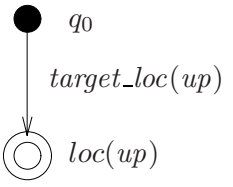


Fig. 8. An observer for location coverage of location *up*

Example 7. As a very simple example, consider the observer shown in Figure 8 characterizing the coverage item *"visit location up of the automaton"*. It has an initial location q_0 and an accepting location $loc(up)$. The predicate $target\_loc(up)$ is satisfied when location *up* is reached in the monitored timed automata. Hence, the observer could e.g., be used to express that location *up* should be covered in automaton $\mathcal{C}^r$ of Figure 5.

Intuitively, observers have the following semantics: At any specific instant an observer operates in one or *serveral* of its locations, say $Q_i \subseteq Q$. At each transition, the observer traverses all outgoing edges from each location $q \in Q_i$, whose predicates are satisfied (enabled) due to the monitored transition of $\mathcal{S}$. Note that more than one (or none) of the outgoing edges can be enabled. Thus the possible successors of a single location q can be zero or more locations. This means that, if there is a path to an accepting location q_f, that can be reached by choosing the "right" enabled edge after each transition of $\mathcal{S}$, the observer will find that path, like a non-deterministic automaton would do. In that sense, an observer will monitor and find all possible coverage items. Later in this section, we will define formally how observers monitor coverage criteria.

Since, a coverage criterion typically stipulates that a set of coverage items should be covered, the notion of observers is extended with a parameterization mechanism so that they can specify a *set of* coverage items. Parameterized observers are observers, in which locations and edges may have parameters that range over given domains. Each possible instantiation of a parameter gives a certain observer location or edge. For each specified coverage item, the observer has an *accepting* (possibly parameterized) location which (for convenience) is given the name of the corresponding coverage item. When the accepting location is reached, the trace has covered the corresponding coverage item.

Example 8. The coverage criterion *"visit all locations of* $\mathcal{C}^r$*"* can be represented by a parameterized observer with one initial state, and one parameterized accepting location, named $loc(\mathtt{L})$, where L is a parameter that ranges over locations in automaton $\mathcal{C}^r$. For each value ℓ of L, the location $loc(\ell)$ is entered when the automaton enters location ℓ. A parameterized observer for location coverage is shown in Figure 9(a).

Without loss of generality we will, in the following description of observers, use a single timed automaton corresponding to the TIOTS $\mathcal{S}$ in Section 2. Internal actions of the $\mathcal{E}$ will not affect the observer and the extension to a network of timed automata is straight forward.

How Observers Monitor Coverage Criteria. In test case generation an observer observes the transitions of the timed automaton monitored. Reached accepting locations correspond to covered coverage items. We formally define the execution of an observer in terms of a composition between a timed automaton and an observer, which has the form of a *superposition* of the observer onto the timed automaton. Each state of this superposition consists of a state of the timed automaton, together with a set of currently occupied observer locations.

If a predicate b on an observer edge is satisfied by a timed automaton transition $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}')$ we write $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}') \models b$. Formally, the superposition of an observer (Q, q_0, Q_f, B) onto a timed automaton $\mathcal{S}$ is defined as follows:

- *States* are of the form $\langle (\ell, \bar{v}) | \mathcal{Q} \rangle$, where $(\ell, \bar{v})$ is a state of the timed automaton, and $\mathcal{Q}$ is a set of locations of the observer.
- The *initial state* is the tuple $\langle (\ell_0, \bar{v}_0) | \{q_0\} \rangle$, where $(\ell_0, \bar{v}_0)$ is the initial state of the timed automaton, and q_0 is the initial location of the observer.
- A *computation step* is defined by the following two rules
  - $\langle (\ell, \bar{v}) | \mathcal{Q} \rangle \overset{\alpha}{\rightsquigarrow} \langle (\ell', \bar{v}') | \mathcal{Q}' \rangle$ if $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}')$ and $\mathcal{Q}' = \left\{ q' \mid q \xrightarrow{b} q' \text{ and } q \in \mathcal{Q} \text{ and } (\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}') \models b \right\}$
  - $\langle (\ell, \bar{v}) | \mathcal{Q} \rangle \overset{d}{\rightsquigarrow} \langle (\ell, \bar{v}') | \mathcal{Q} \rangle$ if $(\ell, \bar{v}) \xrightarrow{d} (\ell, \bar{v}')$
- A state $\langle (\ell, \bar{v}) | \mathcal{Q} \rangle$ of the superposition *covers* the coverage item represented by the location $q_f \in Q_f$ if $q_f \in \mathcal{Q}$.

Note that the way the set $\mathcal{Q}$ is updated essentially results in an (on-the-fly) subset construction of the parameterized observer. Initially, $\mathcal{Q}$ contains only the initial observer location q_0. In the subsequent computation steps, $\mathcal{Q}$ contains the set of all occupied observer locations, representing already covered and partially covered coverage items. In each discrete action step, the set of occupied observer locations $\mathcal{Q}'$ is obtained by generating all possible successors to the locations in $\mathcal{Q}$, i.e. all q' such that there exists a $q \in \mathcal{Q}$ and an edge $q \xrightarrow{b} q' \in B$ with b satisfied by the computation step of the timed automaton. The observer set $\mathcal{Q}$ is not affected by delay transitions, indicating that the the notion of observers presented in this chapter can not observe time delays.

Both the initial and all accepting observer locations (most commonly) have implicit self-loops with predicate *true*. This means that in the superposition of the observer onto a timed automaton, the initial observer location q_0 is always occupied and all reached accepting observer locations (representing covered coverage items) are guaranteed to remain in $\mathcal{Q}$. As mentioned before, The fact that q_0 is always occupied can be intuitively understood as allowing for the observer to non-deterministically start monitoring a timed automaton (or an IUT) at *any* computation step of a run (or at any point during test execution).

Example 9. Figure 9 shows observers specifying a number of coverage criteria described in the literature [17].

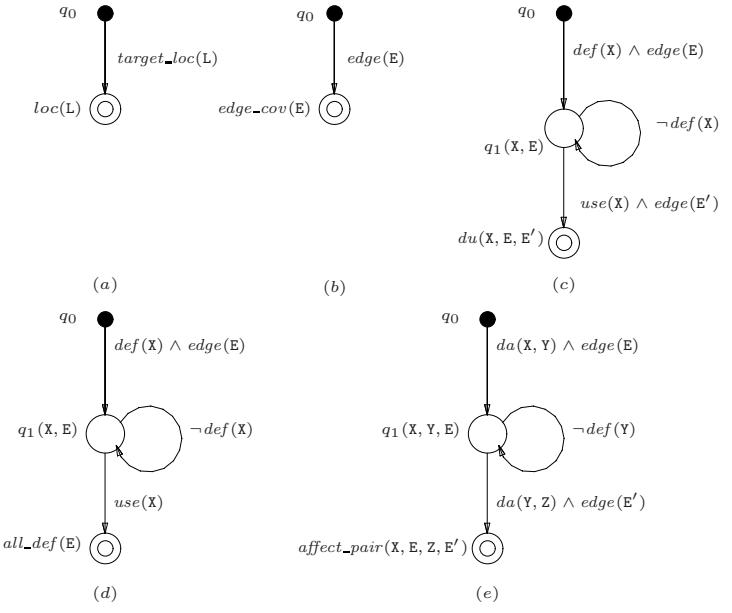


Fig. 9. Five examples of coverage criteria expressed as observers

The *all-locations* [45] coverage criteria is specified by the observer shown in Figure 9(a), where the parameter L is any location in a timed automaton (if restricted to one automaton). If the observer is superposed onto a TIOTS consisting of the timed automaton $\mathcal{C}^r$ in Figure 5, we have that $\mathtt{L} = \{on, dn, off, up\}$ and the edge of the parameterized observer represents one edge for each location in the automaton $\mathcal{C}^r$ i.e. an edge guarded by $target\_loc(on)$ with target location $loc(on)$ etc. Here $target\_loc(\mathtt{L})$ is a predicate which evaluates to true if the observer monitors an edge of the timed automaton $\mathcal{C}^r$ with the target location L. The set of possible coverage items is thus $\{loc(on), loc(dn), loc(off), loc(up)\}$.

The *all-edges* [45] coverage observer in Figure 9(b) is similar to the all-location coverage observer. Here $edge(\mathtt{E})$ is a predicate which evaluates to true if the observer monitors edge E of the timed automaton $\mathcal{C}^r$. The edges of the timed automaton $\mathcal{C}^r$ in Figure 5 are $\mathtt{E}=\{e_0, \ldots, e_{15}\}$[2], and thus the set of possible coverage items when the observer is superposed onto the timed automaton is $\{edge\_cov(e_i) \mid e_i \in \mathtt{E}\}$.

The *all-definition use-pairs* (all-uses [17], reach coverage [26, 41]) coverage observer is shown in Figure 9(c). It uses the two predicates $def(\mathtt{X})$ and $use(\mathtt{X})$ that are true if X is defined and used on the monitored edge, respectively (as defined in Section 4.2). The observer has an accepting location $du(\mathtt{X}, \mathtt{E}, \mathtt{E}')$, where X is a variable name, E is an edge on which X is defined, and $\mathtt{E}'$ an edge on which X is used. Variable X may not be redefined in the trace between E and $\mathtt{E}'$. If the

[2] We assume that the edges can be referred to by indexes 0 to 15.

observer monitors the execution sequence $(\texttt{OFF}, x = 0) \xrightarrow{5} (\texttt{OFF}, x = 5) \xrightarrow{touch?} (\texttt{dim1}, x = 0) \xrightarrow{dim!} (\texttt{DIM}, x = 0) \xrightarrow{3.14} (\texttt{DIM}, x = 3.14) \xrightarrow{touch?} (\texttt{bright2}, x = 0) \xrightarrow{bright!} (\texttt{BRIGHT}, x = 0)$ of the timed automaton in Figure 3 the only covered coverage item is $du(x, \texttt{OFF} \xrightarrow{touch?} \texttt{dim1}, \texttt{DIM} \xrightarrow{touch?} \texttt{bright2})$.

The *all-definitions* [51] coverage observer of Figure 9(*d*) is similar to the all-definition use-pairs coverage except that only the defining edges are required to be covered. When the observer is superposed with the timed automaton in Figure 3 the set of accepting locations is { $all\_def(\texttt{OFF} \xrightarrow{touch?} \texttt{bright1})$, $all\_def(\texttt{BRIGHT} \xrightarrow{touch?} \texttt{dim2})$, $all\_def(\texttt{DIM} \xrightarrow{touch?} \texttt{bright2})$, $all\_def(\texttt{OFF} \xrightarrow{touch?} \texttt{dim1})$, $all\_def(\texttt{DIM} \xrightarrow{touch?} \texttt{off2})$, $all\_def(\texttt{BRIGHT} \xrightarrow{touch?} \texttt{off1})$ }. The *all affect-pairs* (Ntafos' required k-Tuples [49]) coverage observer is shown in Figure 9(*e*). It uses the predicate $da(x, y)$ that is true if the observer monitors a transition in which the value of variable x affects the value of variable y. The observer accepts whenever a variable x affects a variable z via another variable y. In this case we require that x directly affects y which, without redefinition, directly affects z.

A Symbolic Semantics of Observers. The way observers monitor coverage criteria, as defined above for timed automata, will result in an infinite state space due to the dense representation of time. Therefore, before presenting the test case generation algorithm, we shall introduce a finite-state *symbolic* semantics based on the symbolic semantics of timed automata described in Section 2.3.

Formally, the symbolic semantics of observers superposed onto a timed automaton is defined as follows:

- *Symbolic states* are of the form $\langle (\ell, D) | \mathcal{Q} \rangle$, where (ℓ, D) is a symbolic state of the timed automaton, and $\mathcal{Q}$ is a set of observer locations.
- A *initial symbolic state* is a tuple $\langle (\ell_0, D_0) | \{q_0\} \rangle$, where (ℓ_0, D_0) is the initial symbolic state of the timed automaton, and q_0 is the initial observer location.
- A computation step is a triple $\langle (\ell, D) | \mathcal{Q} \rangle \overset{\alpha}{\Rrightarrow} \langle (\ell', D') | \mathcal{Q}' \rangle$ for ℓ' and α such that $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}')$,
  $D' = \left\{ \bar{v}'' \mid (\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}') \ \wedge \ (\ell', \bar{v}') \xrightarrow{d} (\ell', \bar{v}'') \ \wedge \ \bar{v} \in D \right\}$, and
  $\mathcal{Q}' = \left\{ q' \mid q \xrightarrow{b} q' \ \wedge \ q \in \mathcal{Q} \ \wedge \ (\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}') \models b \right\}$.

Note that the evaluation of b does not depend on the clock values of the observed timed automata. Thus, if $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}')$ is a valid transition satisfying b, then any valid transition $(\ell, \bar{v}'') \xrightarrow{\alpha} (\ell', \bar{v}''')$ in $(l, D) \xrightarrow{\alpha} (l', D')$ will also satisfy b.

4.4 Test Case Generation with Observers

In test case generation with observers, we use the superposition of an observer onto a timed automaton, and view the test case generation problem as a state-space exploration problem. To cover a single coverage item q_f is the problem of finding a trace

$$tr = \langle (\ell_0, \bar{v}_0) | \{q_0\} \rangle \overset{d}{\rightsquigarrow} \overset{\alpha}{\rightsquigarrow} \ldots \overset{d'}{\rightsquigarrow} \overset{\alpha'}{\rightsquigarrow} \overset{d''}{\rightsquigarrow} \langle (\ell, \bar{v}) | \mathcal{Q} \rangle \text{ such that } q_f \in \mathcal{Q} \tag{11}$$

It can be shown, that the problem can also be stated based on the symbolic semantics as

$$tr = \langle (\ell_0, D_0) | \{q_0\} \rangle \overset{\alpha}{\Rightarrow} \ldots \overset{\alpha'}{\Rightarrow} \langle (\ell, D) | \mathcal{Q} \rangle \text{ such that } q_f \in \mathcal{Q} \tag{12}$$

We will use $\omega(tr) = \alpha \ldots \alpha'$ to denote the *word* of the trace tr, or just ω whenever tr is clear from the context. In general, a single trace tr may cover several accepting locations of the observer. We say that the trace ω covers n accepting observer states if there are n accepting states in $\mathcal{Q}$, and we use $|Q_f \cap \mathcal{Q}|$ to denote the number of accepting states in $\mathcal{Q}$.

Algorithm 1. Test generation for maximum coverage.

```
1  PASS:= ∅; MAX := 0; ω_max := ω_0;
2  WAIT:= {⟨⟨(ℓ_0, D_0)|{q_0}⟩, ω_0⟩};
3  while WAIT≠ ∅ do
4      select ⟨⟨(ℓ, D)|Q⟩, ω⟩ from WAIT;
5      if |Q_f ∩ Q| > MAX then
6          ω_max := ω; MAX := |Q_f ∩ Q|;
7      if for all ⟨(ℓ, D')|Q'⟩ in PASS: Q ⊈ Q' or D ⊈ D' then
8          add ⟨(ℓ, D)|Q⟩ to PASS;
9          for all ⟨(ℓ'', D'')|Q''⟩ such that ⟨(ℓ, D)|Q⟩ ⇛^α ⟨(ℓ'', D'')|Q''⟩ do
10             add ⟨⟨(ℓ'', D'')|Q''⟩, ωα⟩ to WAIT;
11 return ω_max and MAX;
```

We are now ready to describe the test case generation algorithm [7]. We shall restrict the presentation to an algorithm generating a single trace. The same technique can be used to produce sets of traces to cover many coverage items. Alternatively, the timed system model $\mathcal{S}$ can be annotated with edges that reset the system to its initial state. A generated trace can then be interpreted as a set of test cases separated by the reset edges [27].

An abstract algorithm to compute test case is shown in Algorithm 1. The algorithm computes the maximum number of coverage items that can be visited (MAX), and returns a trace with maximum coverage (ω_{max}). The two main data structures WAIT and PASS are used to keep track of the states waiting to be explored, and the states already explored, respectively.

Initially, the set of already explored states is empty and the only state waiting to be explored is the *extended state* $\langle\langle(\ell_0, D_0)|\{q_0\}\rangle, \omega_0\rangle$, where ω_0 is the empty trace. The algorithm then repeatedly examines extended states from WAIT. If a state $\langle(\ell, D)|\mathcal{Q}\rangle$ found in WAIT is included in a state $\langle(\ell, D')|\mathcal{Q}'\rangle$ in PASS, then obviously $\langle(\ell, D)|\mathcal{Q}\rangle$ does not need to be further examined. If not, all successor states that are reachable from $\langle(\ell, D)|\mathcal{Q}\rangle$ in one computation step are put on WAIT, with their traces extended with the action of the computation step from

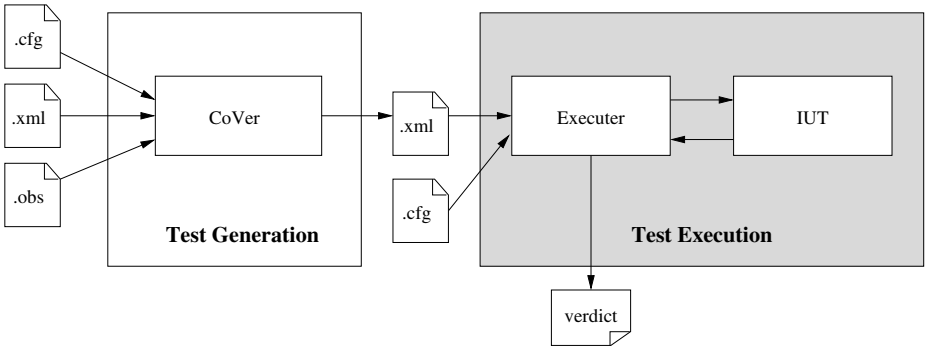


Fig. 10. UPPAAL CO✓ER setup

which they are generated. The state $\langle(\ell, D)|\mathcal{Q}\rangle$ is saved in PASS. The algorithm terminates when WAIT is empty.

The variables ω_{max} and MAX are initially set to the empty trace and 0, respectively. They are updated whenever an extended state is found in WAIT which covers a higher number of coverage items than the current value of MAX. Throughout the execution of the algorithm, the value of MAX is the maximum number of coverage items that have been covered by a single trace, and ω_{max} is one such trace. When the algorithm terminates, the two values MAX and ω_{max} are returned.

It has been shown in e.g. [40] how to extract a concrete diagnostic trace from traces generated by symbolic model-checkers for timed automata. The same technique can be directly applied to extract concrete traces with Algorithm 1. Thus, we can compute traces like Equation 11 from traces like Equation 12 generated by the algorithm. The results on soundness and completeness of symbolic model-checking for timed automata also applies to Algorithm 1 since the number of possible elements in the sets $\mathcal{Q}$ is guaranteed to be finite.

4.5 Tool Implementation

The concept of observers and the test case generation algorithm presented in this section have been implemented in a version of the UPPAAL tool, called UPPAAL CO✓ER [3] [28,29]. The current implementation uses bit-vector analysis techniques to represent and manipulate coverage, and supports an extended version of the observer language described in this section [7]. For a given coverage criterion (a set of) test cases can be generated from system specifications described as a network of UPPAAL timed automata [27].

A typical setup in which UPPAAL CO✓ER is used to test an IUT is shown in Figure 10. The setup is divided in two parts, a test *generation* part for generating

[3] More information about UPPAAL CO✓ER is available at the web site `http://user.-it.uu.se/~hessel/CoVer/`.

and transforming test cases into XML-format, and a test *execution* part that executes the tests on the IUT in a controlled environment.

The input to UPPAAL CO✓ER is a model, an observer, and a configuration file. The model is an UPPAAL timed automata network (.xml) with a system part and an environment part. The observer (.obs) expresses the coverage criterion that steers the exploration during test case generation. The configuration file (.cfg) describes the signals in the timed automata network that should be considered as external, i.e. the interactions between the system part and the environment part. The configuration file also specifies the variables that should be passed as parameters in the input/ output signals.

The UPPAAL CO✓ER tool produces a test suite consisting of a set of test cases (.xml) that are timed traces where each input and output signal has a list of parameters with values (according to the configuration file). An *Executer* interprets the test cases, executes them, and returns a verdict for each test case.

UPPAAL CO✓ER has been used in a large case study in collaboration with Ericsson, in which model-based testing was applied to test a WAP gateway [29]. In the case study, the session and transaction layers of the WAP protocol were modeled in detail as UPPAAL timed automata, and observers were used to specify the coverage criteria the test suites should satisfy. The UPPAAL CO✓ER tool was applied to generate test suites that were automatically translated into executable test scripts that revealed several discrepancies between the model and the actual implementation.

The observer techniques presented in this section have also been implemented in a tool operating on a subset of the functional language Erlang [8]. The tool has been applied in a case study in collaboration with the Swedish tele communication company Mobile Arts AB.

5 Online Testing

The previous section described offline test generation from timed automata specifications given test purposes or coverage criteria specified as observer automata or reachability properties, but was limited to deterministic specifications. However, for many real-time systems the ordering or timing of events cannot be known a priory, and hence its behavior can not be appropriately captured by a deterministic model.

Moreover, as elaborated in Section 6.3, timed automata cannot be determinized, and hence using determinization as intermediate step as is done by many untimed test generators is infeasible for timed automata, and other approaches are necessary. Here we present online testing which is a promising approach. We present a real-time online testing algorithm, its soundness, completeness and implementation.

5.1 Non-determinism and Time

In general non-determinism in specification is used as a means of *abstraction*. It may be that the exact circumstances in the implementation that lead to different

event orderings or timings are not known or would require a model with too many details. It may also be that the implementation internally exhibits non-determinism which cannot be observed or controlled by the tester, e.g., the exact arrival order and timings of external interrupts. A further typical use of non-determinism is to model *optional* behavior that is permitted, but not required by all implementations.

Non-determinism plays a particular role in real-time systems because it is used to express timing uncertainty. A typical real-time requirement is that the IUT must deliver an output within a given time bound, but as long as the deadline is satisfied, the IUT conforms. In TIOTS, this is specified as a non-deterministic choice between letting time pass and producing an output. In timed automata this is described syntactically by using an invariant on a location with the outgoing edge producing the output (see e.g., location l_2 of Figure 5(a) where the compressor is required to switch on (and off) within r time units.

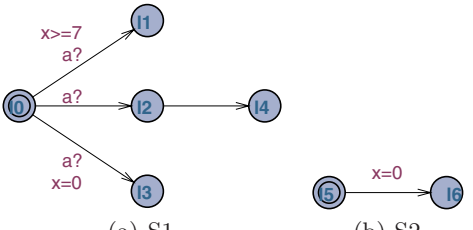
Further, outputs from the IUT may be delayed by an unpredictable amount of time in the communication software between the test host and IUT. Some timing tolerance on most output actions is often required.

A non-deterministic model may reach/occupy several possible states after having executed an action, and as a consequence it may have different possible next behaviors. This possible set of states represents the uncertainty the tester has about the exact state of a (conforming) IUT, and the tester must be prepared to accept any legal next behavior according to the state set.

Example 10. As examples, consider the simple compressor controller of Figure 5(a). Upon receiving a medium temperature reading the controller may either stay off or switch on the compressor, see Equation 6. Further consider the timed automata in Figure 11. The following equations list the states that can be reached after an observable action and a delay. Note that in the second case even a single transition can result in more (infinite with dense time) states. In this example it is not known when the clock x is reset on the internal transition.

$$\{\langle l_0, x = 3\rangle\} \text{ After } a = \{\langle l_2, x = 3\rangle, \langle l_4, x = 3\rangle, \langle l_3, x = 0\rangle\}$$

$$\{\langle l_5, x = 0\rangle\} \text{ After } 4 = \{\langle l_5, x = 4\rangle, \langle l_6, 0 \leq x \leq 4\rangle\}$$



(a) S1 (b) S2

Fig. 11. Two non-deterministic timed automata

Such non-deterministic timed specifications are algorithmically and computationally more complex to analyze than their untimed counter parts because they require symbolic manipulation of sets of infinite sets of states.

5.2 A Real-Time Online Testing Algorithm

The test specification input to Algorithm 2 consists of two *weakly input enabled* and *non-blocking* TIOTSs $\mathcal{S} \parallel \mathcal{E}$ respectively modelling the IUT and its environment. It maintains the current reachable state set $\mathcal{Z} \subseteq S \times E$ that the test specification can possibly occupy after the timed trace σ observed so far. Knowing this state-set allows it to choose appropriate inputs and to validate IUT outputs. Moreover, if the computed state set becomes empty ($\mathcal{S} \parallel \mathcal{E}$ After $\sigma = \emptyset$), the IUT has exhibited a timed trace not in the test specification, and the IUTcannot be rtioco conforming, see Section 3. The possible set of states is computed incrementally event by event.

Algorithm 2. Test generation and execution: $TestGenExe(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T)$.

```
1  Z := {(s0, e0)};                    // initialize the state set with initial state
2  while Z ≠ ∅ ∧ ♯iterations ≤ T do
3      switch between action, delay and restart randomly do
4          case action:                                          // offer an input
5              if EnvOutput(Z) ≠ ∅ then
6                  randomly choose i ∈ EnvOutput(Z);
7                  send i to IUT,;
8                  Z := Z After i;
9          case delay:                                       // wait for an output
10             randomly choose d ∈ Delays(Z);
11             sleep for d time units or wake up on output o at d' ≤ d;
12             if o occurs then
13                 Z := Z After d';
14                 if o ∉ ImpOutput(Z) then return fail ;
15                 else Z := Z After o
16             else Z := Z After d;                  // no output within d delay
17         case restart: Z := {(s0, e0)}; reset IUT;          // reset and restart
18 if Z = ∅ then return fail else return pass;
```

The tester can perform three basic actions: either send an input (enabled environment output) to the IUT, wait for an output for some time, or reset the IUT and restart. If the tester observes an output or a time delay it checks whether this is legal according to the state set. The state set is updated whenever an input is given, or an output or a delay is observed.

Illegal occurrence or absence of an output is detected if the state set becomes empty which is the result if the observed trace is not in the specification. The

functions used in Algorithm 2 are defined as: $\mathsf{EnvOutput}(\mathcal{Z}) = \{a \in A_{in} \mid \exists (s,e) \in \mathcal{Z}.e \xrightarrow{a}\}$, $\mathsf{ImpOutput}(\mathcal{Z}) = \{a \in A_{out} \mid \exists (s,e) \in \mathcal{Z}.s \xrightarrow{a}\}$, and $\mathsf{Delays}(\mathcal{Z}) = \{d \mid \exists (s,e) \in \mathcal{Z}.e \stackrel{d}{\Rightarrow}\}$ [4]. Note that EnvOutput is empty if the environment has no outputs to offer. Similarly, the Delays function cannot pick at random from the entire domain of real-numbers if the environment must produce an input to the IUT model before a certain moment in time.

5.3 Soundness and Completeness

Algorithm 2 constitutes a randomized algorithm for providing stimuli to (in terms of input and delays) and observing resulting reactions from (in terms of output) a given IUT. Under a testing hypothesis about the behavior of the IUT and given that the TIOTSs $\mathcal{S}$ and $\mathcal{E}$ satisfy the below given assumptions, the randomization used in Algorithm 2 may be chosen such that the algorithm is both complete and sound in the sense that it (eventually with probability one) gives the verdict "fail" in all cases of non-conformance and the verdict "pass" in cases of conformance.

The hypothesis is based on the results on digitization techniques in [57][5] which allow the dense-time trace inclusion problem between two sets of timed traces to be reduced to discrete time. In particular it suffices to choose unit delays in Algorithm 2 (assuming that the models and the IUT share the same magnitude of a time unit).

Moreover, if the behavior of the IUT is a non-blocking, input enabled, deterministic TIOTS with isolated outputs the reaction to any given timed input trace $\sigma = d_1 i_1 \ldots d_k i_k d_{i+1}$ is completely deterministic. More precisely, given the stimuli σ there is a unique $\rho \in \mathsf{TTr}(\mathsf{IUT})$ such that $\rho \uparrow A_{in} = \sigma$, where $\rho \uparrow A_{in}$ is the natural projection of the timed trace ρ to the set of input actions. If the IUT is allowed to be non-deterministic it cannot be guarenteed that all its behavior have been revealed.

Theorem 1. *Assume that the behavior of* IUT *may be modeled*[6] *as a weakly input enabled, non-blocking, deterministic TIOTS with isolated outputs,* $\mathsf{TTr}(\mathsf{IUT})$ *and* $\mathsf{TTr}(\mathcal{E})$ *are* closed under digitization *and that* $\mathsf{TTr}(\mathcal{S})$ *is* closed under inverse digitization*. Then Algorithm 2 with only unit delays is sound and complete in the following senses:*

[4] According to the definition of rtioco_e given in Section 3, all environment traces and delays must be considered, not only the delays that can occur in the parallel composition of $\mathcal{S}$ and $\mathcal{E}$; in a parallel composition a delay is only permitted if both components agree. Therefore $\mathsf{Delays}(\mathcal{Z})$ extracts the possible delays from the environment component e of the system state (s,e) to ensure that the algorithm will try to wait beyond the specified deadlines before supplying a new input.

[5] We refer the reader to [57] for the precise definition of digitization and inverse digitization.

[6] The assumption that the IUT can be modeled by a formal object in a given class is commonly referred to as the *test hypothesis*. Only its existence is assumed, not a known instance.

1. *Whenever* $TestGenExe(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T) = fail$ *then* $\mathsf{IUT}\ \mathsf{rtioco}_{\mathcal{E}}\ \mathcal{S}$.
2. *Whenever* $\mathsf{IUT}\ \mathsf{rtioco}_{\mathcal{E}}\ \mathcal{S}$ *then* $\mathsf{Prob}\big(TestGenExe(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T) = fail\big) \xrightarrow{T\to\infty} 1$

   where T *is the maximum number of iterations of the while-loop before exiting.*

Proof. The proof can be found in [38].

From [57, 34] it follows that the closure properties required in Theorem 1 are satisfied if the behavior of the IUT and the $\mathcal{E}$ are TIOTSs induced by closed timed automata (i.e. where all guards and invariants are non-strict) and $\mathcal{S}$ is a TIOTS induced by an open timed automaton (i.e. with guards and invariants being strict). In practice these requirements are not restrictive, e.g. for strict guards one can always scale the clock constants to obtain arbitrary high precision.

5.4 Tool Implementation

The online testing algorithm Algorithm 2 is implemented in a tool named Uppaal-TRON [38]: Uppaal extended for Testing Real-time systems ONline. It implements the setup shown in Figure 12.

We assume that the IUT is a black-box whose state is not directly observable, i.e., only physical *input* and *output* actions are observable. The *adapter* is an IUT specific hardware/software component that connects the IUT to TRON and is responsible for translating abstract input *"in"* test events into physical stimuli and physical IUT output observations into abstract model outputs *"out"*. All events are time-stamped at testing tool side, meaning that the adapter model should be included as part of implementation specification. TRON *engine* loads the test specification which is a network of timed automata partitioned into *models* of the *environment* and the IUT. The goal of TRON is to emulate and replace the environment of the IUT: stimulate the IUT with input that is deemed relevant by the environment part of the model, based on the timed sequence of input and output actions performed so far.

Because TRON executes on platforms whose execution cannot be entirely predicted and controlled (e.g. due to operating system scheduling and tool analysis performance issues), Algorithm 2 is implemented in such a way that TRON checks the validity of output with timing and also the actual timing of input

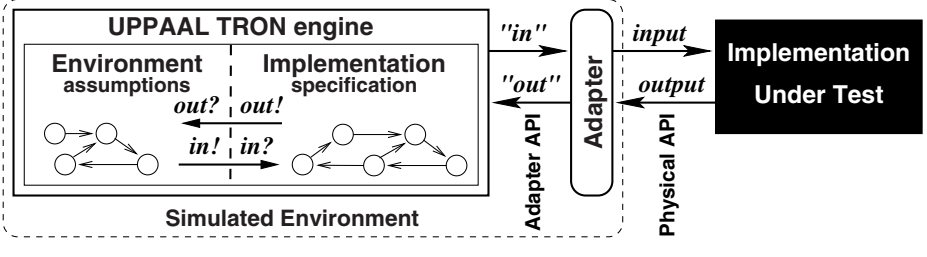


Fig. 12. TRON test setup

execution. TRON provides an application programming interface to enable programming of adapters, and provides the means for loading this as a dynamically linked library.

Internally, TRON uses matured efficient timed automata symbolic reachability algorithm from UPPAAL [4] to compute the symbolic state set which means that the model semantics is preserved and analysis is efficient for online testing. Thus, to compute the operator After the online testing algorithm manipulates sets of symbolic states (ℓ, D), see Section 2.3, and is constructed such that it terminates even if the model contains τ action loops. Further information about the implementation of the required symbolic operations can be found in [38].

To evaluate online testing we have created a number of small academic specifications and implementation (and mutants thereof). The results regarding both performance and error detection capability are promising. More details can be found in [38]. We have also evaluated online testing on an industrial case [44], an electronic refrigeration controller provided by the Danish manufacturer Danfoss A/S. Besides temperature based compressor regulation it has numerous features for handling alarms and defrosting cycles, etc.

We found that real-time online testing is an effective means of detecting discrepancies between the model and the implementation in practice. It also appears feasible performance-wise for such realistic models.

However, large and very non-deterministic models can run into a state explosion making it problematic to update the state-set in real-time which may limit the granularity of time constraints that can be checked in real-time. In a typical test run in the Danfoss case, the state-set varied typically between a few symbolic states and a few hundred symbolic states. Exploring these is unproblematic for the modern model-checking engine employed by TRON. Updating even medium sized state-sets with around a 100 states requires only a few milli-seconds of CPU-time on a modern PC. The largest encountered state-sets (around 3000 states) were very infrequent, and required around 300 milli-seconds.

Real-time online testing thus appears feasible for a large range of embedded systems.

5.5 Testing = Environment Emulation + Implementation Monitoring

On closer inspection it turns out that online testing consists of two logically different functions, namely *environment emulation* and IUT *monitoring*:

Environment Emulation: An environment emulator (completely or partly) replaces the real environment of the IUT, and stimulates the IUT with new inputs based on the history of previous inputs and observed outputs. An environment emulator thus executes online in real-time and actively stimulates the IUT, but does not assign verdicts to the observed trace.

IUT Monitoring: A monitor passively observes the timed input/output sequence produced between the IUTand its real-environment, and determines whether this behavior is (relativized input/output) conforming to the

specification. Hence, the monitor functions as a test oracle. Monitoring is also sometimes called *passive testing*.

The monitor can be executed in three different ways. It may run *real-time online* in which case non-conformance is reported immediately. This requires that the monitor has sufficient computational resources to analyze the model at the pace dictated by the IUT. Alternatively the monitor may be executed online, but at its own pace (virtual time). Events that are unprocessed are buffered until the monitor becomes ready. Non-conformance will be reported while the IUTis running, but typically some time after it has occurred. Finally, the monitor can be executed offline (post-mortem) on a collected (finite) trace.

Until now we have presented our framework, test-generation and execution algorithm, and TRON as a tool that performs environment emulation and online real-time monitoring as an integrated program.

However, in some situations it is beneficial to separate the two functions in different parts/tools. For example, the two functions can be performed by dedicated tools specialized for the particular function or executed on dedicated platforms (e.g., a hard real-time operating system/computer for environment emulation and a fast (soft-real-time) number-crunching computer for monitoring). Another example is performance. It may not be possible to evaluate a large detailed model of the IUT online in real-time (models of the IUT tends to be larger and much more detailed than the environment model). With a separate monitoring function this can be done afterwards or on a separate dedicated computer.

The explicit separation of the test specification into an environment part and an IUT part allows TRON to be configured easily to perform both pure emulation and monitoring as described in the following.

We denote the behavioral model of the IUT with input actions A_{in} and output actions A_{out} by $\mathcal{S}(A_{in}, A_{out})$. Similarly, we denote the environment by $\mathcal{E}(A_{out}, A_{in})$. Also let $U(A_{in}, A_{out})$ and $O(A_{in}, A_{out})$ denote respectively the most (universal) and least (passive) discriminating and least discriminating timed automata, see Section 3. The universal timed automaton is capable of performing any trace. The passive timed automata silently consume input actions.

To use TRON for pure environment emulator use the intended environment model $\mathcal{E}(A_{out}, A_{in})$ and replace the IUT-model $\mathcal{S}(A_{in}, A_{out})$ by $U(A_{in}, A_{out})$. In consequence TRON will produce timed traces only in $\mathcal{E}(A_{out}, A_{in})$. Non-conformance will never be reported because $U(A_{in}, A_{out})$ allows any timed trace. This configuration is depicted in Figure 13(a).

Similarly, pure monitoring can be achieved using a slightly modified IUT-model $\mathcal{S}' = \mathcal{S}(\emptyset, A_{in} \cup A_{out})$ where all input actions are changed to output actions, see Figure 13(b). This model contains the same traces (ignoring i/o labeling) as the original. The environment model must be completely passive and not contain any inputs (as seen from the IUT point of view), $O' = O(A_{in} \cup A_{out}, \emptyset)$. Thus, with no essential modification to TRON or Algorithm 2 the monitoring can be executed in simulated time or offline. If the monitor is uncertain about

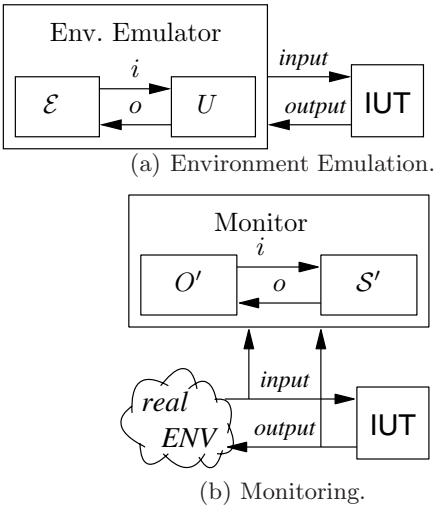


Fig. 13. Model based emulation and monitoring

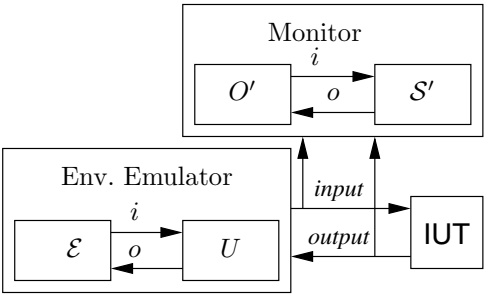


Fig. 14. Model-based Testing via Combined Environment Emulation and Monitoring

the state of the IUT when started, Algorithm 2 can be started with a different (over-approximated) state-set instead of the initial state.

Finally, we observe that online testing can be obtained by running two instances of TRON, one performing monitoring and the other environment emulation, see Figure 14. The two instances may possibly run on different computers.

6 Discussion and Future Work

Model-based test generation for real-time specifications has been investigated by others (see e.g., [50, 43, 11, 30, 22, 13, 56, 47, 36, 27, 42]), but remain relatively immature. In this section we discuss our approach to timed testing and compare to important related work. Also we mention topics for future work in the area.

6.1 Conformance Relations

The choice of conformance relation is important for both theoretical and practical reasons, yet there is still no wide spread consensus in the community about its definition.

Our relativized timed input/output conformance relation is a timed and environment-relativized extension of a solid and widespread implementation relation used in model based conformance testing of untimed systems, namely the input/output conformance relation by Tretmans [58]. Informally, input/output conformance requires for all specification traces that the implementation never produces an output not allowed by the specification, and that it never refuses to produce an output (forever stays quiescent) when the specification requires one. As also noted in [36, 42] a timed input/output conformance relation can be obtained (assuming input enabledness) as timed trace inclusion between the implementation and its specification.

A fundamental question is how quantitative properties like real-time can be observed of the physical IUT. E.g., can event occurrences be observed at time points or only within error bounds, and should such fundamental physical uncertainties be an explicit part of the theory? Similarly, does a concept like quiescence make sense in a real-time system, or are only time bounded (finite time) observations possible? New alternative timed implementation relations have been formulated by Briones and Brinksma in [12].

Another question is related to the goal of real-time testing. Timed trace inclusion does not allow the implementation to be faster than the specification. In some cases this may be unsafe. However, in many other cases it seems natural that the implementation should be allowed to be as fast as possible. Therefore faster-than type relations have been proposed [19, 46]. Thus there seem to be a unclear cut boundary between real-time correctness testing and performance testing.

6.2 Specification of Tests

The test cases to be executed on the IUT can be selected by different means. Typical approaches are test purposes, model-coverage criteria, fault-models (see e.g., [30, 22]) , or randomly.

Test purposes are specific observation objectives formulated by the test engineer, see e.g. [23, 54]. Another popular approach is to cover the model in the hope that a covering test suite is also thorough. Further, model coverage is an important measure for estimating the confidence the developers can have in the executed tests.

In typical approaches, the selection of test cases follows some particular coverage criterion, such as coverage of control states, edges, etc. For finite-state machines several approaches focus on particular coverage criteria, e.g., Bouquet and Legeard [9] synthesize test cases corresponding to combinations of choices of control flow and boundary values of state variables. Nielsen and Skou [48] generate test cases from timed automata that cover different time-domains represented as reachable symbolic states.

Since different coverage criteria are suitable in different situations, and satisfy different constraints on fault detection capability, cost, information about where potential faults may be located, etc., it is highly desirable that a test generation tool is able to generate test suites in a flexible manner, for a wide variety of different coverage criteria. In other words, a test generation tool should accept a simple specification of a coverage criterion, given in a language that can easily specify a large set of coverage criteria, and be able to generate test suites accordingly. Hong et al [32, 31] describe how flow-based coverage criteria can be expressed in temporal logic. Friedman et al [24] specifies coverage by giving a set of projections of the state space (e.g., on individual state variables, components of control flow) that should be covered, possibly under some restrictions.

The observer approach described in this chapter generalizes these approaches and provides such a flexible language. Test purposes can in some sense be regarded as coverage observers, but are not used to specify more generic coverage criteria and do not make use of parameterization, as we do.

Where offline test generation uses symbolic and constraint solving algorithms to satisfy a coverage criterion, online test generators typically uses cheap randomized choice techniques, and can thus not guarantee satisfaction of the coverage criteria, or only provide a probabilistic guarantee provided (unrealistic) long execution time. The achieved coverage of an online testing session can easily be evaluated post-mortem by comparing the collected timed trace with the model. This can for instance be done by executing the timed trace on the model suitably extended with auxiliary coverage or meta-variables, as described in Section 4.2. Another approach is to dynamically collect coverage information during the test run and use this to guide (reduce the random choices) toward uncovered parts of the model.

Except for the obvious extensions of untimed coverage criteria, there exists very little research [48, 16] that deals explicitly with real-time coverage criteria, i.e., criteria that tries to cover the time domain and timer/clock values of a timed specification. Future work includes formulating such real-time coverage criteria and extending the observer approach to allow easy specification of these.

6.3 Test Generation Algorithms

Many model-checker based test generators that generate tests from a coverage criterion invoke the model checker for each coverage item resulting in a single test case per coverage item, see e.g., [25]. This not only results in many test cases and a large test generation overhead, but also a large test execution overhead because many sub-sequences will be identical. It may be more efficient to cover several items by the same test, and generate a test suite that covers the model as much as possible, as our algorithm in Section 4.4. However, this requires that the model checker is extended with dedicated search and pruning algorithms and efficient bit vector encodings of the coverage criteria. We also expect such efficient encodings to play an important role in monitoring and guiding the online test generator toward a coverage goal.

Moreover, whereas most other work on optimizing test suites, e.g. [1, 60, 33], focus on minimizing the length of the test suite, our technique may also reduce

the actual execution time, because it considers that some events take longer to produce or and take real-time constraints into account. It may even produce the time optimal test sequences.

Most offline algorithms explicitly determinize the specification [18, 35, 47] as an intermediate step. However, for expressive formalisms like TA this approach is problematic because in general they cannot be determinized.

It is well-known that from the theory of timed automata that non-deterministic timed automata (unlike finite automata) cannot be determinized to a language equivalent deterministic timed automata [2]. It is also not in general possible to remove internal transitions from a timed automata (and when they can, it may be very costly) [61]. Much work on timed test generation from TA therefore restrict the amount and type of allowed non-determinism: [56, 22, 27] completely disallow non-determinism, [36, 47] restrict the use of clocks, guards or clock resets. This gives a less expressive and less flexible specification language. In contrast, online testing is automatically adaptive and only implicitly determinizes the specification, and only partially up to the concrete trace observed so far.

Our approach to online testing is inspired by the (untimed) algorithm proposed by Tretmans et. al. in [63, 6]. They have implemented online testing from Promela [63] and LOTOS specifications in the TORX [62] tool, and practical application to real case studies show promising results [59, 62, 6]. However, TORX provides no support for real-time. Similarly to Krichen and Tripakis [55, 42] we use symbolic reachability computation algorithms to track the current state-set for timed automata with unrestricted non-determinism. We extend the UPPAAL model-checker resulting in an integrated and mature testing and verification tool.

It seems likely that a combination of the strengths of offline and online testing will require the notion of games. In a two-player game one player is trying to reach a winning state by performing controllable game-moves while being affected by uncontrollable moves by the opponent. Translated into testing this corresponds to the situation where the tester is trying reach a state where the test purpose (or coverage criterion) is satisfied by giving controllable inputs to the IUT (the opponent) that responds by making uncontrollable and unpredictable output actions. The goal of the test generator is to compute a winning strategy that will partly be computed statically and partly be interpreted and computed dynamically. Although promising work is in progress on such timed games [10] the required concepts are not sufficiently well developed yet.

The UPPAAL framework is perfectly suited for exploring timed properties of the model, but there is little effort done toward combining it with more complicated test data generation. The recent release of UPPAAL supports C-like data declarations which would enable to combine and implement ideas from [37].

6.4 Real-Time Test Execution and Diagnostics

The execution of real-time test cases is also a challenge, both for online and offline generated tests, because the test execution system is a real-time system with deadlines and potential narrow tolerances. There are two main problems. One is that the host platform may cause unpredictable real-time performance

of the tester because of scheduling latency, competing processes, i/o activity and disturbances from competing processes. The other is that there is communication media between the tester and the IUT that must be factored into the test generation or execution. It introduces latency and uncertainty on the order and timing of observations. These problems are not only technical engineering problems, but also seem to require clarification at a semantic and theoretical level.

When non-conformance has been detected the next step is to diagnose why the run failed. It may be an error in the specification, the adaptor layer, or the implementation. If the error is in the implementation the exact cause has to be found and corrected, and regression testing must be performed.

For online testers these issues are especially problematic, because test sequences are typically very long and randomly generated, and hence are difficult to diagnose and reproduce for regression testing. The current TRON implementation assumes that the fault appears in the last testing step and gives a hint about what output was expected and when, and prints information about the last known non-empty state-set. While very helpful, it does not necessarily indicate the cause of the fault, which may have been caused by an internal fault executed by the IUT much earlier. Also TRON allows a recorded timed trace to be replayed against the implementation. However, doing so for long traces with narrow timing tolerances is technically very challenging.

In the future we plan to combine coverage facts with information about passed and failed test runs, in the hope that difference in coverage (of the model or code) could help locate the cause of the error, an approach inspired by the concept of delta-debugging [64].

7 Conclusions

In this chapter we reviewed progress on formal model-based testing of real-time systems. We presented a testing framework consisting of a formal, timed specification language, timed automata, and a formal real-time correctness relation, relativized input/output conformance. We conclude that this framework is solid, technically sound and works well in practice. Based on this common framework we demonstrated two extreme approaches to timed test generation. In offline (optimal) satisfaction of test purpose or coverage criterion is the aim, while online testing ensures thoroughness through volume and brute-force.

These approaches are implemented by (substantially) extending the efficient algorithms and data structures from the Uppaal model-checker. We find such a mature tool and efficient machinery important for the practical use of the test generation techniques.

Overall, we conclude that significant progress has been made in the area of timed testing, but also that many exciting and important challenges remain. These range from technical engineering problems to principal semantic (and perhaps philosophical) ones.

Acknowledgements

We would like to thank the anonymous reviewers for their detailed and constructive comments that greatly helped improving this presentation.

References

1. Aho, A.V., Dahbura, A.T., Lee, D., Uyar, M.Ü.: An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. IEEE Transactions on Communications 39(11), 1604–1615 (1991)
2. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126(2), 183–235 (1994)
3. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for Real-Time Systems. In: Proc. of Logic in Computer Science, Jun 1990, pp. 414–425. IEEE Computer Society Press, Los Alamitos (1990)
4. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Uppaal implementation secrets. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 3–22. Springer, Heidelberg (2002)
5. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J.: Efficient Guiding Towards Cost-Optimality in UPPAAL. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 174–188. Springer, Heidelberg (2001)
6. Belinfante, A., Feenstra, J., de Vries, R.G., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) 12th Int. Workshop on Testing of Communicating Systems, pp. 179–196. Kluwer Academic Publishers, Dordrecht (1999)
7. Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and generating test cases using observer automata. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 125–139. Springer, Heidelberg (2005)
8. Blom, J., Jonsson, B.: Automated test generation for industrial erlang applications. In: Proc. 2003 ACM SIGPLAN workshop on Erlang, Uppsala, Sweden, pp. 8–14 (August 2003)
9. Bouquet, F., Legeard, B.: Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 778–795. Springer, Heidelberg (2003)
10. Bouyer, P., Cassez, F., Fleury, E., Larsen, K.G.: Optimal Strategies in Priced Timed Game Autoamata. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, Springer, Heidelberg (2004)
11. Braberman, V., Felder, M., Marré, M.: Testing Timing Behaviors of Real Time Software. In: Quality Week 1997, San Francisco, USA, pp. 143–155 (April-May 1997)
12. Briones, L.B., Brinksma, E.: A Test Generation Framework for Quiescent Real-Time Systems. In: Grabowski, J., Nielsen, B. (eds.) International workshop on Formal Approaches to Testing of Software. Co-located with IEEE Conference on Automates Software Engineering 2004, Linz, Austria, pp. 64–78 (September 2004)
13. Cardell-Oliver, R.: Conformance Testing of Real-Time Systems with Timed Automata. Formal Aspects of Computing 12(5), 350–371 (2000)

14. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal 9(5), 193–200 (1994)
15. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering 4(3), 178–187 (1978)
16. Clarke, D., Lee, I.: Testing Real-Time Constraints in a Process Algebraic Setting. In: 17th International Conference on Software Engineering (1995)
17. Clarke, L.A., Podgurski, A., Richardsson, D.J., Zeil, S.J.: A formal evaluation of data flow path selection criteria. IEEE Trans. on Software Engineering 15(11), 1318–1332 (1989)
18. Cleaveland, R., Hennessy, M.: Testing Equivalence as a Bisimulation Equivalence. Formal Aspects of Computing 5, 1–20 (1993)
19. Cleaveland, R., Zwarico, A.E.: A Theory of Testing for Real-Time. In: Sixth Annual IEEE Symposium on Logic in Computer Science, pp. 110–119 (1991)
20. Daws, C., Olivero, A., Yovine, S.: Verifying ET-LOTOS programs with Kronos. In: Hogrefe, D., Leue, S. (eds.) Proc. of 7*th* Int. Conf. on Formal Description Techniques, North-Holland, Amsterdam (1994)
21. Dill, D.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
22. En-Nouaary, A., Dssouli, R., Khendek, F., Elqortobi, A.: Timed Test Cases Generation Based on State Characterization Technique. In: 19th IEEE Real-Time Systems Symposium (RTSS 1998), December 2–4 1998, pp. 220–229 (1998)
23. Fernandez, J.-C., Jard, C., Jéron, T., Viho, C.: An experiment in automatic generation of test suites for protocols with verification technology. Science of Computer Programming 29 (1997)
24. Friedman, G., Hartman, A., Nagin, K., Shiran, T.: Projected state machine coverage for software testing. In: Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 134–143 (2002)
25. Heimdahl, M.P.E., Rayadurgam, S., Visser, W., Devaraj, G., Gao, J.: Autogenerating Test Sequences Usiong Model Checkers: A Case Study. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, Springer, Heidelberg (2004)
26. Herman, P.M.: A data flow analysis approach to program testing. Australian Computer J. 8(3) (November 1976)
27. Hessel, A., Larsen, K.G., Nielsen, B., Pettersson, P., Skou, A.: Time-Optimal Real-Time Test Case Generation using UPPAAL. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 136–151. Springer, Heidelberg (2004)
28. Hessel, A., Pettersson, P.: A test generation algorithm for real-time systems. In: Ehrich, H.-D., Schewe, K.-D. (eds.) Proc. of 4th Int. Conf. on Quality Software, September 2004, pp. 268–273. IEEE Computer Society Press, Los Alamitos (2004)
29. Hessel, A., Pettersson, P.: Model-Based Testing of a WAP Gateway: an Industrial Study. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, Springer, Heidelberg (2007)
30. Higashino, T., Nakata, A., Taniguchi, K., Cavalli, A.R.: Generating Test Cases for a Timed I/O Automaton Model. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS), Budapest, Hungary, September 1–3, 1999. IFIP Conference Proceedings, vol. 147, pp. 197–214. Kluwer, Dordrecht (1999)
31. Hong, H.S., Cha, S.D., Lee, I., Sokolsky, O., Ural, H.: Data flow testing as model checking. In: ICSE 2003: 25th Int. Conf. on Software Enginering, May 2003, pp. 232–242 (2003)

32. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 327–341. Springer, Heidelberg (2002)
33. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: A Temporal Logic Based Theory of Test Coverage and Generation. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 327–341. Springer, Heidelberg (2002)
34. Ouaknine, J., Worrell, J.: Revisiting digitization, robustness, and decidability for timed automata. In: 18th IEEE Symposium on Logic in Computer Science (LICS 2003), Ottawa, Canada, June 2003, pp. 198–207. IEEE Computer Society Press, Los Alamitos (2003)
35. Jéron, T., Morel, P.: Test generation derived from model-checking. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 108–122. Springer, Heidelberg (1999)
36. Khoumsi, A., Jéron, T., Marchand, H.: Test cases generation for nondeterministic real-time systems. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, Springer, Heidelberg (2004)
37. Koopman, P.W.M., Alimarine, A., Tretmans, J., Plasmeijer, M.J.: Gast: Generic automated software testing. In: Peña, R., Arts, T. (eds.) IFL 2002. LNCS, vol. 2670, pp. 84–100. Springer, Heidelberg (2003)
38. Larsen, K., Mikucionis, M., Nielsen, B.: Online Testing of Real-time Systems using Uppaal. In: Grabowski, J., Nielsen, B. (eds.) International workshop on Formal Approaches to Testing of Software. Co-located with IEEE Conference on Automates Software Engineering 2004, Linz, Austria (September 2004)
39. Larsen, K.G., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., Romijn, J.: As cheap as possible: Efficient cost-optimal reachability for priced timed automat. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 493–505. Springer, Heidelberg (2001)
40. Larsen, K.G., Pettersson, P., Yi, W.: Diagnostic Model-Checking for Real-Time Systems. In: Proc. of Workshop on Verification and Control of Hybrid Systems III, October 1995. LNCS, vol. 1066, pp. 575–586. Springer, Heidelberg (1995)
41. Laski, J.W., Korel, B.: A data flow oriented program testing strategy. IEEE Transactions on Software Engineering SE-9(3), 347–354 (1983)
42. Krichen, M., Tripakis, S.: Black-box Conformance Testing for Real-Time Systems. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, Springer, Heidelberg (2004)
43. Mandrioli, D., Morasca, S., Morzenti, A.: Generating Test Cases for Real-Time Systems from Logic Specifications. ACM Transactions on Computer Systems 13(4), 365–398 (1995)
44. Mikucionis, M., Larsen, K.G., Nielsen, B., Skou, A.: Testing rea-time embedded software using uppaal-tron —an industrial case study. In: Embedded Software (EMSOFT), New Jersey, USA (September 2005)
45. Myers, G.: The Art of Software Testing. Wiley-Interscience, Chichester (1979)
46. Núñez, M., Rodríguez, I.: Conformance Testing Relations for Timed Systems. In: Grieskamp, W., Weise, C. (eds.) International workshop on Formal Approaches to Testing of Software, Co-located with Computer Aided Verification, Edinburgh, Scotland, UK (July 2005)
47. Nielsen, B., Skou, A.: Automated Test Generation from Timed Automata. In: Tools and Algorithms for the Construction and Analysis of Systems, April 2001, pp. 343–357 (2001)
48. Nielsenand, B., Skou, A.: Automated test generation from timed automata. International Journal on Software Tools for Technology Transfer 5, 59–77 (2003)

49. Ntafos, S.: A comparison of some structural testing strategies. IEEE Transaction on Software Engineering 14, 868–874 (1988)
50. Peleska, J., Amthor, P., Dick, S., Meyer, O., Siegel, M., Zahlten, C.: Testing Reactive Real-Time Systems. In: Material for the School – 5th International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 1998), Lyngby, Denmark (1998)
51. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering 11(4), 367–375 (1985)
52. RCTA, Washington D.C., USA. RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certifications (December 1992)
53. Rokicki, T.G., Myers, C.J.: Automatic verification of timed circuits. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 468–480. Springer, Heidelberg (1994)
54. Rusu, V., du Bousquet, L., Jéron, T.: An approach to symbolic test generation. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 338–357. Springer, Heidelberg (2000)
55. Tripakis, S.: Fault Diagnosis for Timed Automata. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, Springer, Heidelberg (2002)
56. Springintveld, J., Vaandrager, F., D'Argenio, P.R.: Testing Timed Automata. Theoretical Computer Science 254(1–2), 225–257 (2001)
57. Henzinger, T.A., Manna, Z., Pnueli, A.: What good are digital clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
58. Tretmans, J.: Testing concurrent systems: A formal approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
59. Tretmans, J., Belinfante, A.: Automatic testing with formal methods. In: EuroSTAR 1999: 7th European Int. Conference on Software Testing, Analysis & Review. Barcelona, Spain. EuroStar Conferences, Galway, Ireland, November 8–12 (1999)
60. Ümit Uyar, M., Fecko, M.A., Sethi, A.S., Amar, P.D.: Testing Protocols Modeled as FSMs with Timing Parameters. Computer Networks: The International Journal of Computer and Telecommunication Networking 31(18), 1967–1998 (1999)
61. Diekert, V., Gastin, P., Petit, A.: Removing epsilon-Transitions in Timed Automata. In: Reischuk, R., Morvan, M. (eds.) STACS 1997. LNCS, vol. 1200, pp. 583–594. Springer, Heidelberg (1997)
62. de Vries, R., Tretmans, J., Belinfante, A., Feenstra, J., Feijs, L., Mauw, S., Goga, N., Heerink, L., de Heer, A.: Côte de resyste in PROGRESS. In: STW Technology Foundation, editor, PROGRESS 2000 – Workshop on Embedded Systems, October 2000, pp. 141–148. The Netherlands, Utrecht (2000)
63. de Vries, R.G., Tretmans, J.: On-the-fly conformance testing using SPIN. Software Tools for Technology Transfer 2(4), 382–393 (2000)
64. Zeller, A., Hildebrandt, R.: Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering 28(2), 183–200 (2002)