Timed Testing with TorX

Henrik Bohnenkamp* and Axel Belinfante

Formal Methods and Tools,
Department of Computer Science, University of Twente,
Postbus 217, NL-7500 AE Enschede, The Netherlands
{bohnenka, belinfan}@cs.utwente.nl

Abstract. TorX is a specification-based, on-the-fly testing tool that tests for ioco conformance of implementations w.r.t. a formal specification. This paper describes an extension of TorX to not only allow testing for functional correctness, but also for correctness w.r.t. timing properties expressed in the specification. An implementation then passes a timed test if it passes according to ioco, and if *occurrence times* of outputs or of quiescence signals are legal according to the specification. The specifications are described by means of non-deterministic safety timed automata. This paper describes the basic algorithms for ioco, the necessary modifications to standard safety timed automata to make them usable as an input formalism, a test-derivation algorithm from timed automata, and the concrete algorithms implemented in TorX for timed testing. Finally, practical concerns with respect to timed testing are discussed.

Keywords: Model-based on-the-fly Testing, Timed Automata, Real-Time Testing, TorX, Tools.

1 Introduction

Testing is one of the most natural, intuitive and effective methods to increase the reliability of software. Formal methods have been employed to analyse and systematise the testing idea in general, and to define notions of correctness of implementations with respect to specifications in particular. Moreover, practical approaches to testing have been derived from testing theories [4, 15, 8, 9]. The *ioco* testing theory [14] reasons about black-box conformance testing of software components. Specifications and implementations are modeled as labelled transition systems (LTS) with inputs and outputs. An important ingredient of the theory is the notion of *quiescence*, *i.e.,* the absence of output, which is considered to be observable. Quiescence provides additional information on the behaviour of the implementation under test (IUT) and therefore allows to distinguish better between correct and faulty behaviour. The *ioco* theory defines a notion of correctness, the ***ioco*** implementation relation, and defines how to derive sound

* This work is supported by the Dutch National Senter project TANGRAM.

J.S. Fitzgerald, I.J. Hayes, and A. Tarlecki (Eds.): FM 2005, LNCS 3582, pp. 173–188, 2005.
© Springer-Verlag Berlin Heidelberg 2005

test-cases from the specification. The set of all *ioco* test-cases (which is usually of infinite size) is exhaustive, *i.e.,* in theory it is possible to distinguish all faulty from all *ioco*-correct implementations by executing all test-cases. In practice, *ioco*-test-cases can be used to test software components and to find bugs. The testing tool TorX has been developed [4, 15] to derive *ioco* test-cases automatically from a specification, and to apply them to an IUT. TorX does *on-the-fly testing*, *i.e.,* test-case derivation and execution is done simultaneously. TorX has been used successfully in several industry-relevant case-studies [2, 3].

This paper is about an extension of TorX to allow testing of real-time properties: *real-time testing*. Real-time testing means that the decisions whether an IUT has passed or failed a test is not only based on which outputs are observed, given a certain sequence of inputs, but also on *when* the outputs occur, given a certain sequence of inputs *applied at predefined times*. Our approach is influenced by, although independent of, the *tioco* theory [6], an extension of *ioco* to real-time testing. Whereas the *tioco* theory provides a formal framework for timed testing, we describe in this paper an algorithmic approach to real-time testing, inspired by the existing implementation of TorX. We use as input models non-deterministic *safety timed automata*, and describe the algorithms developed to derive test-cases for timed testing.

Related Work. Real-time testing has recently come more and more into focus of research. In [11, 12] approaches are described in which timed automata are used as specification formalism, and algorithms are described to do on-the-fly timed testing based on these specifications. These approaches are most similar to the one we describe in this paper. However, the big difference is that we take in our approach *quiescence* into account.

TorX itself has in fact already been used for timed testing [2]. Even though the approach was an ad-hoc solution to test for some timing properties in a particular case study, the approach has shown a lot of the problems that come with practical real-time testing, and has provided solutions to many of them. This early case-study has accelerated the implementation work for our TorX extensions immensely.

Structure of the Paper. In Section 2, we introduce *ioco*, describe the central algorithms of TorX, and comment on *tioco*. In Section 3, we introduce the class of models we use to describe specifications, and describe the algorithms necessary to do testing. In Section 4, we describe an abstract algorithm to derive test-cases from timed automata, and describe how we have implemented this in TorX. In Section 5, we address practical issues regarding timed testing. We conclude with Section 6.

Notational Convention. We will frequently define structures by means of tuples. If we define a tuple $T = (e_1, e_2, \dots, e_n)$, we often will use a kind of *record* notation known from programming languages in order to address the components of the tuple, *i.e.,* we will write $T.e_i$ if we mean component e_i for T, for $i = 1, \dots, n$.

2 Preliminaries

2.1 The *ioco* Way of Testing

In this section we give a summary of the *ioco* theory (*ioco* is an abbreviation for "*IO-conformance*"). Details can be found in [14].

The *ioco* Theory. A *labelled transition system* (LTS) is a tuple $(S, s_0, Act, \rightarrow)$, where S is a set of states, $s_0 \in S$ is the initial state, Act is a set of labels, and $\rightarrow \subseteq S \times Act \cup \{\tau\} \times S$ is the transition relation. Transitions $(s, a, s') \in \rightarrow$ are frequently written as $s \xrightarrow{a} s'$. τ is the *invisible* action. The set of all transition systems over label set Act is denoted as $\mathcal{L}(Act)$. Assume a set of input labels L_I, and a set of output labels L_U, $L_I \cap L_U = \emptyset$, $\tau \notin L_I \cup L_U$. Elements from L_I are often suffixed with a "?" and elements from L_U with an "!" to allow easier distinction. An LTS $L \in \mathcal{L}(L_I \cup L_U)$ is called an Input/Output transition system (IOTS) if L is *input-enabled*, *i.e.,* $\forall s \in S, \forall i? \in L_I : \exists s' \in L.S : s \xrightarrow{i?} s'$. Input-enabledness ensures that IOTS can never deadlock. However, it might be possible that from certain states no outputs can be produced without prior input. This behaviour is described by the notion of *quiescence*: let $L \in \mathcal{L}(L_I \cup L_U)$, and $s \in L.S$. Then s is *quiescent* (denoted $\delta(s)$), iff $\forall a \in L_U \cup \{\tau\} : \neg\exists s' \in L.S : s \xrightarrow{a} s'$. We introduce the *quiescence label*, $\delta \notin L_I \cup L_U \cup \{\tau\}$, and define the δ-*closure* $\Delta(L) = (L.S, L.s_0, L_I \cup L_U \cup \{\tau\} \cup \{\delta\}, \rightarrow')$, where $\rightarrow' = L.\!\rightarrow \cup \{(s, \delta, s) \mid s \in L.S \wedge \delta(s)\}$. It is this definition of quiescence which makes it necessary to postulate strongly convergent LTS, *i.e.,* which do not have infinite computations with only a finite trace. We introduce some more notation to deal with transition systems. Assume LTS L. For $a \in Act \cup \{\tau\}$, we write $s \xrightarrow{a}$, iff $\exists s' \in L.S : s \xrightarrow{a} s'$. We write $s \xrightarrow{a_1,\ldots,a_n} s'$ iff $\exists s_1, s_2, \ldots, s_{n-1} \in L.S : s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{n-1} \xrightarrow{a_n} s'$. We write $s \Longrightarrow s'$ iff $s \xrightarrow{\tau,\ldots,\tau} s'$, and $s \stackrel{a}{\Longrightarrow} s'$ iff $\exists s'', s''' \in L.S : s \Longrightarrow s'' \xrightarrow{a} s''' \Longrightarrow s'$. Let $L \in \mathcal{L}(L_I \cup L_U)$. For a state $s \in L.S$, the set of *suspension traces* from s, denoted by $Straces(s)$, are defined as $Straces(s) = \{\sigma \in (L_I \cup L_U \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Longrightarrow}\}$, where $\Longrightarrow$ is defined on top of $\Delta(L).\!\rightarrow$. We define $Straces(L) = Straces(L.s_0)$. For $L \in \mathcal{L}(L_I \cup L_U)$ and $s \in L.S$, we define $out(s) = \{o \in L_U \mid s \xrightarrow{o}\} \cup \{\delta \mid \delta(s)\}$, and, for $S' \subseteq L.S$, $out(S') = \bigcup_{s \in S'} out(s)$. Furthermore, for $s \in L.S$, $\underline{s\ \textbf{after}\ \sigma} = \{s' \in L.S \mid s \stackrel{\sigma}{\Longrightarrow} s'\}$, and for $S \subseteq L.S$, $\underline{S\ \textbf{after}\ \sigma} = \bigcup_{s \in S} \underline{s\ \textbf{after}\ \sigma}$. We define $\underline{L\ \textbf{after}\ \sigma} = \underline{L.s_0\ \textbf{after}\ \sigma}$.

Let $Spec, Impl \in \mathcal{L}(L_I \cup L_U)$ and let $Impl$ be an IOTS. Then we define

$$Impl\ \textbf{ioco}\ Spec \Leftrightarrow \forall \sigma \in Straces(Spec) : out(\underline{Impl\ \textbf{after}\ \sigma}) \subseteq out(\underline{Spec\ \textbf{after}\ \sigma}).$$

Testing for *ioco* Conformance: Test-Case Derivation. To test a real system, we need a specification of it. From the specification test-cases can be derived that are sound with respect to *ioco*, *i.e.,* their execution will never lead to a test failure if the implementation is *ioco*-correct. Test cases are deterministic, finite, non-cyclic LTS with two special states **pass** and **fail**, which are supposed to be *terminating*. Test-cases are defined in a process-algebraic notation, with the following syntax: $T \longrightarrow \textbf{pass} \mid \textbf{fail} \mid a;T \mid \sum_{i=1}^{n} a_i T_i$, for

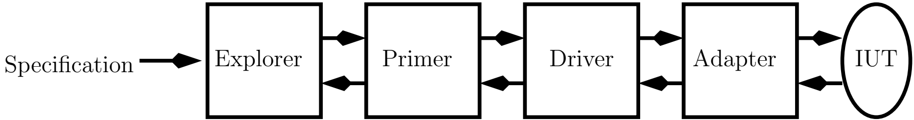

Fig. 1. The TorX tool architecture

$a, a_1, \ldots, a_n \in L_I \cup L_U \cup \{\delta\}$. Assuming an LTS $L \in \mathcal{L}(L_I \cup L_U)$ as a specification, test cases are defined recursively (with finite depth) according to the following rules. Starting with the set $S = \{L.s_0\}$,

1. $T :=$ **pass** is a test-case;
2. $T := a; T'$ is a test-case, where $a \in L_I$ and, assuming that $S' = \underline{S \textbf{ after } a}$ and $\underline{S' \neq \emptyset}$, T' is a test-case derived from set S';
3. For $\overline{out(S)} = (L_U \cup \{\delta\}) \setminus out(S)$,

$$T := \sum_{x \in \overline{out(S)}} x; \textbf{fail} \quad + \sum_{x \in out(S)} x; T_x$$

   is a test-case, where the T_x for $x \in out(S)$ are test-cases derived from the respective sets $S_x = \underline{S \textbf{ after } x}$.

2.2 On-the-Fly *ioco* Testing: TorX

In Figure 1 we see the tool structure of TorX. We can distinguish four tool components (not counting the IUT): EXPLORER, PRIMER, DRIVER and ADAPTER. The EXPLORER is the software component that takes a specification as input and provides access to an LTS representation of this specification. The PRIMER is the software component that is *ioco* specific. It implements part of the test-case derivation algorithm for the *ioco* theory. In particular, the PRIMER interacts directly with the EXPLORER, *i.e.,* the representation of the specification, in order to compute so-called *menus*. Menus are sets of transitions with input, output or δ labels, which according to the model are allowed to be applied to the IUT or allowed to be observed.

The PRIMER is triggered by the DRIVER. The DRIVER is the only active component and acts therefore as the motor of the TorX tool chain. It decides whether to apply a stimulus to the IUT, or whether to wait for an observation from the ADAPTER, and it channels information between PRIMER and ADAPTER.

The ADAPTER has several tasks: i) interface with the IUT; ii) translate abstract actions to concrete actions and and apply the latter to the IUT; iii) observe the IUT and translate observations to abstract actions; iv) detect absence of an output over a certain period of time and signal quiescence.

The recursive definition of test-cases as described in Section 2.1 allows to derive and execute test-cases simultaneously, *on-the-fly*. The core algorithm is the computation of *menus* from a set of states S. The output menu contains transitions labeled with the actions from the *out*-set $out(S)$. The input menu contains all inputs that are allowed to be applied to the IUT, according to the

specification. The reason to keep transitions, rather than actions, in menus is that it is necessary to know the destination states which can be reached after applying an input or observing an output. The computation of a menu requires for each state in S the bounded exploration of a part of the state-space. Recursive descent into the state-space is stopped if a transition with an input or output label is seen.

The algorithm for the computation of menus is given in Fig. 2. We assume an LTS $Spec \in \mathcal{L}(L_I \cup L_U)$. Input to the algorithm is a set S of states. Initially, $S = \{L.s_0\}$. After trace $\sigma \in (L_I \cup L_U \cup \{\delta\})^*$ has been observed, $S = \underline{L \textbf{ after } \sigma}$. Note that the transitions with δ labels are implicitly added to the *out* set when appropriate. Therefore, the EXPLORER does not have to deal with the δ-closure of the LTS it represents.

```
Algorithm Compute_Menu
1   input: Set of states S
2   output: Set of transitions in, out
3   in := ∅
4   out := ∅
5   already_explored := ∅
6   foreach s ∈ S
7     already_explored := already_explored ∪ {s}
8     S := S \ {s}
9     is_quiescent := true
10    foreach q -a-> q' ∈ Spec.→ ∩ ({s} × Act ∪ {τ} × L.S)
11      if a = τ
12        is_quiescent := false
13        if q' ∉ already_explored : S := S ∪ {q'}
14      else :
15        if a ∈ L_I : in := in ∪ {q -a-> q'}
16        else :
17          out := out ∪ {q -a-> q'}
18          is_quiescent := false
19    end
20    if is_quiescent : out := out ∪ {s -δ-> s}
21  end
22  return(in,out)
```

Fig. 2. Menu computation

Given the computed menus *in*, *out*, the DRIVER component decides how to proceed with the testing. The algorithm is given in Fig. 3. In principle, the DRIVER has to choose between the three different possibilities that have been given for the *ioco* test-case algorithm in Section 2.1: i) termination, ii) applying an input in set *in*, or iii) waiting for an output.

With the variables *wait* and *stop* we denote a probabilistic choice: whenever one of them is references they are either **false** or **true**. The driver control loop therefore terminates with probability one. The choice between ii) and iii) is also done probabilistically: if the ADAPTER has no observation to offer to the DRIVER, the variable *wait*is consulted. To describe the algorithm of the DRIVER, we enhance the definition of $\underline{\cdot \textbf{ after } \cdot}$ to menus. If M is a menu, then we define $\underline{M \textbf{ after } a} = \{q' \mid (q \xrightarrow{a} q') \in M\}$.

```
Algorithm Driver_Control_Loop
1   input: —
2   output: Verdict pass or fail
3   (in, out) = Compute_Menu({s0})
4   while ¬stop :
5     if ADAPTER.has_output() ∨ wait:
6       if out after ADAPTER.output() = ∅: terminate(fail)
7       (in, out) = Compute_Menu(out after ADAPTER.output())
8     else:
9       choose i? ∈ {a | q -a-> q' ∈ in}
10      if ADAPTER.apply_input(i?) :
11        (in, out) = Compute_Menu(in after i?)
12  end
13  terminate(pass)
```

Fig. 3. Driver Control Loop

Quiescence in Practice. From the specification point-of-view, quiescence is a reachability property. In the real world, a non-quiescent implementation will produce an output after some finite amount time. If an implementation never produces an output, it is quiescent. Therefore, from an implementation point-

of-view, quiescence can be seen as a timing property, and one that can not be detected in finite time. In theory, this makes quiescence detection impossible. However, in practice it is possible to work with approximations to quiescence. A system that is supposed to work at a fast pace, like in the order of milli-seconds, can certainly be considered as being quiescent, if after two days of waiting no output has appeared. Even two hours, if not two minutes of waiting might be a sufficient to conclude that the system is quiescent. It seems to be plausible to approximate quiescence by waiting for a properly chosen time interval after the occurrence of the latest event. This is the approach chosen for TorX. The responsibility to detect quiescence and to send a synthetic action, the *quiescence signal*, lies with the ADAPTER.

2.3 *tioco* Testing Theory

Even though development of the *tioco* theory and our own work described here has been mostly independent from each other, some important decisions made for *tioco* have been adapted for our own approach.

In *tioco*, the formalism used to model specification and implementation of timed systems are so-called *timed transition systems* with input and output labels (TIOTS). Timed transition systems are LTS with an explicit notion of time and delay. An implementation relation ***tioco*** is defined, and also a test-case derivation algorithm. *tioco* is meant as an extension of *ioco* to timed testing. Therefore, the theory has to deal with quiescence. As explained in Section 2.2, quiescence is in real life a property related to time, and the methods to approximate the occurrence of quiescence is reused from the approach chosen for TorX. However, since TIOTS have an explicit notion of time, the quiescence approximation approach has to be taken explicitly into account in the definition of test-cases. It is in principle straightforward to define quiescence on the level of TIOTS in terms of reachability, but the derived test-cases must define explicitly when a δ is allowed to be *observed*. In order to define this unambiguously, an assumption is made which must be met by an implementation in order to ensure the soundness of *tioco* testing.

Definition 1 (*tioco* Quiescence Prerequisite). *For an implementation Impl there is an $M \in \mathbb{R}$ such that*

- *Impl produces an output within M time units, counted from the last input or output, or,*
- *if it does not, then Impl is quiescent.*

It is in general the responsibility of the system designer to ensure that this assumption holds and to provide a reasonable value for M. In general, there will be systems which can never fulfil this property. Quiescence for TIOTS is (informally) defined as follows.

Definition 2 (*tioco* Quiescence). *A state in a TIOTS is quiescent iff there is no state reachable by τ-steps or by delaying, where a transition with an output label is enabled.*

Note that this definition of quiescence is more general than the one for *ioco*. A state that can make a τ-step can in *tioco* still be considered as quiescent, whereas in *ioco* not.

Related to the handling of quiescence is another property in the *tioco* theory which we adopt: the *no-forced-input* property. A system must not be forced to get inputs at a certain time in order to proceed. This basically states that if a state is quiescent, *i.e.,* if it can only proceed by accepting inputs, it must be ensured that there is no urgency requirement on the application of an input. If a state in an TIOTS specification waits for inputs, it must be allowed to wait for these inputs forever.

3 Absolute-Time Timed Automata

The input formalism chosen for our timed-testing extensions of TorX are non-deterministic safety timed automata [10]. In this section we will introduce the necessary background needed to describe our testing approach.

3.1 From Timed Automata to Zones

Our approach makes use of zone-based semantics of timed automata known from the literature. A comprehensive treatment on semantics and algorithms for timed and zone automata is given in [5]. In the following we will give a nano-tutorial on this subject.

A time domain $\mathbb{T}$ is a totally ordered, well-founded additive monoid with neutral element 0 that is also the minimum in the ordering, and with $d + d' \leq d$ iff $d' = 0$, for all $d \in \mathbb{T}$. In the following we assume a fixed time domain $\mathbb{T}$. Let $\mathcal{C}$ be a set of clock variables. An *atomic clock-constraint* is an inequality of the form $b_l \prec x - y \prec b_u$ or $b_l \prec x \prec b_u$, for $x, y \in \mathcal{C}$, $\prec \in \{<, \leq\}$, and $b_l, b_u \in \mathbb{T}$ with $b_l \leq b_u$. Clock constraints are conjunctions of atomic clock constraints. The set of all clock constraints over clock set $\mathcal{C}$ is denoted by $\mathcal{B}(\mathcal{C})$. Atomic clock constraints of the form $b_l \prec x - y \prec b_u$ are also called *clock-difference constraint.*

Definition 3 (Timed Automaton). *A timed automaton T is a tuple $(N, \mathcal{C}, Act, l_0, E, I)$, where*

- *N is a finite set of locations,*
- *$\mathcal{C}$ is a set of clock variables,*
- *Act is a set of labels,*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times Act \cup \{\tau\} \times 2^{\mathcal{C}} \times N$ is the set of edges*
- *$I : N \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations.*

We define $\mathcal{A}(Act)$ to be the set of timed automata over the label set Act.

The edges $(l, g, a, r, l') \in E$ are abbreviated $l \xrightarrow{g,a,r} l'$, where $g \in \mathcal{B}(\mathcal{C})$ is called *guard* of the edge, and $r \subseteq \mathcal{C}$ *clock reset.* Guards and invariants are clock constraints. Note that in the literature the set of clocks is usually not explicitly

mentioned in the definition of a timed automaton, but in the following it is necessary to remember on which set of clocks the semantics of a timed automaton is defined.

A *clock valuation* is a function $u : \mathcal{C} \to \mathbb{T}$. For $d \in \mathbb{T}$, we define $(u + d)(c) = u(c) + d$. If a valuation u satisfies a clock constraint $C \in \mathcal{B}(\mathcal{C})$, *i.e.,* if the relational expression obtained by replacing all occurrences of clock names c by $u(c)$ evaluates to true, we write $u \in C$. If $r \subseteq \mathcal{C}$, then $u[r \mapsto 0](c) = 0$, if $c \in r$, and $u[r \mapsto 0](c) = u(c)$, otherwise.

The semantics of a timed automaton is a transition system where states are pairs (l, u) of locations and clock valuations. Initial state is $(l_0, \{c \mapsto 0 \mid c \in \mathcal{C}\})$. Transitions are defined as follows.

$$\frac{u \in I(l) \quad (u+d) \in I(l)}{(l,u) \xrightarrow{d} (l,u+d)} (d \in \mathbb{T}) \qquad \frac{l \xrightarrow{g,a,r} l' \quad u \in g \quad u' = u[r \mapsto 0] \quad u' \in I(l')}{(l,u) \xrightarrow{a} (l',u')} \tag{1}$$

If $\mathbb{T}$ is a continuous set, like $\mathbb{R}^+$, the transition system defined by the two rules have a continuous state-space. It is well known, however, that under certain conditions it is possible to abstract from the continuous transitions defined above, and derive a discrete representation of the timed automaton, the *region automaton* [1]. More efficient in time and space however is the construction of a *zone automaton* [5], which is an abstraction of the region automaton. A *clock zone* is the maximal set of clock valuations that satisfy a given clock constraint. In order to define the semantics of a timed automaton in terms of a zone automaton, a number of operations on clock zones are defined (in decreasing order of precedence). The time-passing operator $\Uparrow z$, which is defined as $\Uparrow z = \{u + d \mid u \in z, d \in \mathbb{T}\}$; Conjunction of clock zones $z \wedge z'$, defined as $z \wedge z' = z \cap z'$. Clock reset $z[r \mapsto 0]$ for $r \subseteq \mathcal{C}$, defined as $z[r \mapsto 0] = \{u[r \mapsto 0] \mid u \in z\}$.

We denote the set of all clock zones on clocks in $\mathcal{C}$ as $\mathcal{Z}(\mathcal{C})$, or just $\mathcal{Z}$, if $\mathcal{C}$ is clear from the context. We define $\mathit{Succ}(z, i, g, r, i') = (\Uparrow z \wedge i \wedge g)[r \mapsto 0] \wedge i')$ for clock zone $z \in \mathcal{Z}$, clock resets $r \subseteq \mathcal{C}$ and clock constraints i, g, i'.

The state space of a zone automaton underlying a timed automaton $T = (N, \mathcal{C}, \mathit{Act}, l_0, E, I)$ is a sub-set of $N \times \mathcal{Z}$, and, following [5], its elements are called *zones* (without "*clock*-"). The zone automaton $\mathit{ZA}(T)$ of T is a labelled transitions system $(S, s_0, \mathit{Act}, \twoheadrightarrow) \in \mathcal{L}(\mathit{Act})$, where $S \subseteq N \times \mathcal{Z}$, $s_0 = (l_0, \{c = 0 \mid c \in \mathcal{C}\})$, and $\twoheadrightarrow$ is defined by the following rule:

$$\frac{l \xrightarrow{g,a,r} l' \quad z' = \mathit{Succ}(z, I(l), g, r, I(l')) \neq \emptyset}{(l, z) \overset{a}{\twoheadrightarrow} (l', z')}. \tag{2}$$

Zone automata derived by this rule are discrete, but in general still infinite. For a certain class of timed automata it is however possible to construct a finite quotient of zones by so-called normalisation (*cf.* [5]). The use of normalisation will however not be necessary for our purposes.

3.2 Absolute Time in Zone Automata

For our testing approach we have decided to measure time absolutely, *i.e.,* testing-relevant events like the application of an input to the IUT or the observation of an output from the IUT is time stamped in absolute time, measured

from "system start". When "system start" is, is an arbitrary choice. Using absolute time does not have any particular advantage or disadvantage. Our approach would work equally well with relative time, *i.e.,* with measurements of the time that passes between two observable events. The choice for absolute time was the fact that some simple computations on time stamps were not necessary.

We have therefore to introduce a notion of absolute time in timed automata. We do this by introducing a special clock, denoted by ABS, a clock which is never referenced in the considered timed automaton, and which therefore is never reset. So, if $T = (N, \mathcal{C}, Act, l_0, E, I)$ is a timed automaton, we define the absolute-time version $\text{ABS}(T)$ of T as $T = (N, \mathcal{C} \cup \{\text{ABS}\}, Act, l_0, E, I)$. Note that clock zones are defined relative to a clock set. All clocks in the clock set are considered in order to compute successor clock zones. Adding ABS to the clock set adds therefore one more dimension to the clock zones. We define the absolute-time zone automaton of a timed automaton T with clock set $T.\mathcal{C}$ as the zone automaton $ZA(\text{ABS}(T))$. Given a zone $q = (l, z)$ of $\text{ABS}(T)$, the valuations of the absolute time clock ABS in clock zone z describes the time interval in which it is allowed to sojourn in zone q. In the following, we will denote the projection of a clock zone $z \in \mathcal{Z}(\mathcal{C} \cup \{\text{ABS}\})$ on the absolute times scale as $z^{\downarrow}$, *i.e.,* $z^{\downarrow} = \{u(\text{ABS}) \mid u \in z\}$.

3.3 Inputs and Outputs in Timed Automata

We distinguish again a set of input labels, L_I, and a set of output labels, L_U, and special symbol δ to denote quiescence. We consider now the set of timed automata $\mathcal{A}(L_I \cup L_U)$.

The semantics of timed automata, as defined with (1) defines a timed transition system (TTS), and assuming the label sets L_I and L_U for the timed automaton, this TTS is a TIOTS. Therefore, we can apply in principle the quiescence definition of [6] to define quiescence for Timed Automata with inputs and outputs. The definition is however not useful to detect quiescence *algorithmically.* Fortunately, it is possible to express the conditions for quiescence on the level of the timed automaton itself, by modifying and adding switches. Similar to the *ioco* case, we will call such a modified version of a timed automaton T the δ-closure of T. The definition of the δ-closure below takes the *tioco* definition of quiescence as well as the *tioco* Quiescence Prerequisite (*cf.* Definitions 1 and 2) into account. We therefore assume the existence of a real number M which is mentioned in Definition 1, and denote the δ-closure of T as $\Delta_M(T)$.

Definition 4 (δ-closure of a timed automaton). *Let* $T = (N, \mathcal{C}, Act, l_0, E, I)$ *be a timed automaton with* $Act = L_I \cup L_U$*, and let* $M \in \mathbb{R}, M > 0$*. Then the* δ*-closure* $\Delta_M(T)$ *of* T *is a timed automaton* $(N', \mathcal{C}', Act', l'_0, E', I')$*, where* $N' = N$*,* $\mathcal{C}' = \mathcal{C} \cup \{\text{QC}\}$*,* $Act' = Act \cup \{\delta\}$*,* $l'_0 = l_0$*,* $I' = I$*, and* $E' = E_1 \cup E_2 \cup E_3 \cup E_4$ *with* $E_1 = \{(e.l, e.g \wedge (\text{QC} < M), e.a, e.r \cup \{\text{QC}\}, e.l') \mid e \in E \wedge e.a \in L_U\}$*,* $E_2 = \{(e.l, e.g, e.a, e.r \cup \{\text{QC}\}, e.l') \mid e \in E \wedge e.a \in L_I\}$*,* $E_3 = \{e \mid e \in E \wedge e.a = \tau\}$ *and* $E_4 = \{(l, \text{QC} > M, \delta, \emptyset, l) \mid l \in N\}$*.*

The idea behind this definition is the following. The assumption is that the IUT fulfils the *tioco* Quiescence Prerequisite. Therefore, it will only produce

outputs within M time units since the last input or output has been seen. This means that in the specification every location in which it is allowed to stay after M time units have passed, a δ should be accepted. Moreover, if more then M time units have passed, no output in the timed automaton needs to be enabled anymore. Therefore, we add the clock $\textsc{qc} \notin \mathcal{C}$. It measures the time since the last observable behaviour of the IUT has happened. Consequently, every switch which has an input or output label resets clock $\textsc{qc}$. The δ label is added to the action set, and every location in N gets a self-loop switch with δ label, which is only enabled if $\textsc{qc} > M$. The set E' thus comprises the disjoint sets $E_1, \ldots, E_4$. E_1 contains all edges of E with output label, where the guards are extended with the constraint $\textsc{qc} < M$. $\textsc{qc}$ is reset if the switch is taken. E_2 contains all edges of E with an input label, but with the clock reset extended by $\textsc{qc}$ again. E_3 contains all (unmodified) switches of E with a τ label. E_4 contains only self-loops with δ label. These switches denote the occurrence of a quiescence signal. The guard for all of these switches is $\textsc{qc} > M$. Note that the clock $\textsc{qc}$ is not reset, since otherwise switches with output labels could become enabled again.

Similar to the *tioco* theory, we postulate the *no-forced-input* property (see Section 2.3, *cf.* [6]). In the context of timed automata, we thus require that, whenever it is possible to accept quiescence in a location, it must always be possible to stay in that location forever.

4 Timed Automata Testing with TorX

The timed testing approach we have implemented in TorX is based on the absolute-time zone automata, derived from δ-closed timed automata.

4.1 Test-Cases

In order to define the test-cases we are executing with TorX, we adapt the definition of $\underline{\cdot \textbf{ after } \cdot}$ (*cf.* Section 2.1) to work on zones.

Definition 5 ($\underline{\cdot \textbf{ after}_\textbf{t} \cdot}$). *Let $T = (N, \mathcal{C}, L_I \cup L_U, l_0, E, I)$ be a timed automaton, and let $ZA(\textsc{Abs}(\Delta(T))) = (S, s_0, L_I \cup L_U \cup \{\delta\}, \longrightarrow\!\!\!\!\!\!\rightarrow)$ the absolute-time zone automaton derived from its δ-closure. Let $S' \subseteq S$. Then, for $a \in L_I \cup L_U \cup \{\delta\}$ and $t \in \mathbb{T}$,*

$$\underline{S' \textbf{ after}_\textbf{t}\ a@t} = \{(l', z'') \mid \exists (l, z) \in S' : (l, z) \xrightarrow{a}\!\!\!\!\rightarrow (l', z') \\ \textit{and } z'' = (z' \wedge \textsc{Abs} = t) \neq \emptyset\} \tag{3}$$

The set $\underline{S' \textbf{ after}_\textbf{t}\ a@t}$ contains all those zones which can be reached by executing action a at time t from clock zones in S'. Moreover, the successor zones reflect the fact that $\textsc{Abs} = t$ at the time of entering.

Based on this definition, we can give a semi-formal definition of the timed test-cases that are being executed with TorX. As for *ioco* (*cf.* Section 2.1), we

Table 1. Computation of menus from timed automata

Algorithm ***Compute_Menu_TA***	
1	**input**: Set of zones S
2	**output**: Set of zone automata transitions in, out
3	$in := \emptyset$
4	$out := \emptyset$
5	$already_explored := \emptyset$
6	**foreach** $s = (l, z) \in S$
7	$already_explored := already_explored \cup \{s\}$
8	$S := S \setminus \{s\}$
9	**foreach** $e \in \{e' \in E \mid e.l = l\}$
10	**if** $z' = Succ(z, I(e.l), e.g, e.r, I(e.l')) \neq \emptyset$:
11	**if** $e.a = \tau$: $S := S \cup \{(e.l', z')\}$
12	**else** :
13	**if** $e.a \in L_I$: $in := in \cup \{s \xrightarrow{a} (e.l', z')\}$
14	**else** : $out := out \cup \{s \xrightarrow{a} (e.l', z')\}$
15	**end**
16	**end**
17	**return**(in, out)

express the test-cases in a process-algebra-like notation[1]. We distinguish again three steps.

1. $T :=$ **pass** is a test-case;
2. Application of input:

$$T := i@t; T' \quad + \sum_{\substack{t' < t \wedge o \in L_U \\ \underline{S \ \mathbf{after_t} \ o@t'} = \emptyset}} o@t'; \mathbf{fail} \quad + \sum_{\substack{t' < t \wedge o \in L_U \\ \underline{S \ \mathbf{after_t} \ o@t'} \neq \emptyset}} o@t'; T_{o@t'}$$

   for $i \in L_I$ and for $t \in \mathbb{T}$ chosen such that $S' = \underline{S \ \mathbf{after_t} \ i@t} \neq \emptyset$, and for T' being a test-case derived from S', and the $T_{o@t'}$ test-cases derived from $\underline{S \ \mathbf{after_t} \ o@t'}$. Note that it is necessary to take outputs into account which do arrive at the time $t' < t$.
3. Waiting for outputs or signalling of quiescence:

$$T := \sum_{\substack{o \in L_U \cup \{\delta\} \\ \underline{S \ \mathbf{after_t} \ o@t} = \emptyset}} o@t; \mathbf{fail} \quad + \sum_{\substack{o \in L_U \cup \{\delta\} \\ \underline{S \ \mathbf{after_t} \ o@t} \neq \emptyset}} o@t; T_{o@t}.$$

   Here, all outputs including δ are considered. The outputs $o@t$ which yield an empty successor set $\underline{S \ \mathbf{after} \ o@t}$ result in a test failure. All other outputs lead to a test-case $T_{o@t}$, where $T_{o@t}$ is derived from $\underline{S \ \mathbf{after} \ o@t} \neq \emptyset$.

[1] semi-formal: we do abuse the Σ sign to denote non-deterministic choice over a potentially continuous set of possibilities, which is not well-defined.

Table 2. DRIVER control loop for timed systems

```
Algorithm Driver_Control_Loop_TA
1    input: —
2    output: Verdict pass or fail
3    (in, out) = Compute_Menu_TA({(l_0, {x = 0 | x ∈ C})})
4    while ¬ stop:
5        if ADAPTER.has_output() ∨ wait:
6            o@t := ADAPTER.output()
7            if out after_t o@t = ∅: terminate(fail)
8            (in, out) := Compute_Menu_TA(out after_t o@t, t)
9        else:
10           choose i@t ∈ {a@t' | (l, z) -a->> (l, z') ∈ in ∧ t' ∈ z'↓}
11           if ADAPTER.apply_input(i@t):
12               (in, out) = Compute_Menu_TA(in after_t i@t, t)
13   end
14   terminate(pass)
```

4.2 Menu Computation

In Table 1 the algorithm for menu computation ***Compute_Menu_TA*** is given. We assume a timed automaton $Spec \in \mathcal{A}(L_I \cup L_U)$ and consider its δ-closure $\Delta_M(Spec)$ for an appropriately chosen value M. The input of the algorithm is a set of zones S derived from $(ZA(\textsc{Abs}(\Delta_M(Spec))))$ (line 1). The output comprises two sets, the *in* menu and the *out* menu. (lines 3, 4, 17). The set *already_explored* is used to keep track of zones already explored (line 5). We have an outer loop over all states (*i.e.,* zones (l, z)) in the set S (lines 6–16). The contents of S varies during the computation. All states considered inside the loop are added to *already_explored* and removed from S (lines 7, 8). The inner loop (line 9 – 15) considers every switch e with source location l. First, the successor clock zone z' of z according to switch e is computed (line 10). If z' is not empty, transitions of the zone automaton are added to the sets *in* or *out*, depending on the labels of switch e (lines 11–14). Note that transitions with label δ are added to the *out* menu. In case of a τ label, the resulting zone is added to set S (line 11). In essence, the menu computation is a bounded state-space exploration of the zone automaton with sorting of the generated transitions according to their labels.

4.3 Driver Control Loop

We enhance the definition of $\underline{\cdot \ \mathbf{after_t} \ \cdot}$ to menus.

Definition 6 ($\underline{\cdot \ \mathbf{after_t} \ \cdot}$). *Let $T = (N, \mathcal{C}, Act, l_0, E, I)$ be a timed automaton, and let $ZA(\textsc{Abs}(T)) = (S, s_0, Act, \twoheadrightarrow)$ be its absolute-time zone automaton. Let $M \subseteq \twoheadrightarrow$. Then, for $a \in Act$ and $t \in \mathbb{T}$,*

$$\underline{M \ \mathbf{after_t} \ a@t} = \{(l', z'') \mid (l, z) \xrightarrow{a}\!\!\twoheadrightarrow (l', z') \in M \text{ and } z'' = (z' \wedge \textsc{Abs} = t) \neq \emptyset\} \tag{4}$$

If the set M is a menu computed by ***Compute_Menu_TA***, each transition $(l, z) \xrightarrow{a} (l', z')$ contains the interval of all times at which a us allowed to happen: the interval $z'^{\downarrow}$. The set $\underline{M\ \mathbf{after_t}\ a@t}$ then computes a set of successor zones from M which can be reached by executing a at exactly time t. For a zone $(l, z) \in \underline{M\ \mathbf{after_t}\ a@t}$, $z^{\downarrow} = [t, t]$ holds.

In Table 2, we see the algorithm for the DRIVER control loop of TORX, enhanced to deal with time. Menus are computed with ***Compute_Menu_TA***, and the successor states are computed with $\underline{\cdot\ \mathbf{after_t}\ \cdot}$. When an input is applied, not only an input $i? \in in$ is chosen, but also a time instance $t \in z'^{\downarrow}$ (line 10), at which time to apply the input.

The variables *wait* and *stop* have the same meaning as in the *ioco* algorithm (*cf.* Section 2.2).

4.4 *ioco*, *tioco*, and TorX

The algorithms for menu computation and test execution are very similar to the ones implemented for untimed TORX. However, there are some slight differences, which we will comment here.

The most important difference is that the δ-closure of the timed automaton can not be computed anymore by the PRIMER. Rather, the δ-closure is done beforehand, and the primer does not need to distinguish anymore between a δ label and arbitrary outputs. The reason for this is that quiescence is a timing property that has to be dealt with on zone-automaton level. These computations are however in the responsibility of the EXPLORER. As a consequence, contrary to our initial hopes, the algorithms that existed for untimed TORX can not be reused. However, the changes are simple and the principle remains the same.

Another big difference is that we allow for the more general definition of quiescence from the *tioco* theory. Attempts to use the more restricted *ioco* definition turned out to be not successful, since unsound test-cases could be produced.

5 Timed Testing in Practice

5.1 Notes on the Testing Hypothesis

The *Testing Hypothesis* is an important ingredient in the testing theory of Tretmans [14]. The hypothesis is that the IUT can be modelled by means of the model class which forms the basis of the testing theory. In case of *ioco* the assumption is that the IUT can be modelled as an input-enabled IOTS. Under this assumption, the results on soundness and completeness of *ioco*-testing do apply to the practical testing approach. In this paper, we have not defined a formalism that we consider as model for an implementation, so we can not really speak of a *testing hypothesis*. Still, it is important to give some hints on what properties a real IUT should have in order to make timed testing feasible. We mention four points.

First, we require input enabledness, as for the untimed case. That means, whenever it is decided to apply an input to the IUT, it is accepted, regardless

of whether this input really does cause a non-trivial state-change of the IUT or not.

Second, it is plausible to postulate that all time measurements are done relative to the same clock that the IUT refers to. In practice this means that the TorX adapter should run on the same host as the IUT and reference the same hardware clock. If measurements would be done by different clocks, measurement errors caused by clock skew and drifts might spoil the measurement, and thus the test run.

Third, as has been pointed out in Section 2.3, the *system designer* has to ensure that the implementation behaves such that quiescence can be detected according to Section 2.3, Def. 1.

Fourth, up to now we left open which time domain $\mathbb{T}$ to choose for our approach. The standard time domain used for timed automata are real numbers, however, in practice only floating-point numbers, rather than real numbers can be used. Early experiments have however shown that floats and doubles quite quickly cause numerical problems. Comparisons of time stamps turn out to be to inexact due to rounding and truncation errors. In the TorX implementation we use thus fixed-precision numbers, *i.e.,* 64 bit integers, counting micro-seconds. This happens to be the time representation used for the UNIX operating system family.

5.2 Limitations of Timed Testing

Even though the timed testing approach described in this paper seems to be easy enough, timed testing is not easy at all. Time is a complicated natural phenomenon. It can't be stopped. It can not be created artificially in a lab environment. Time runs forward, it runs everywhere, and, leaving Einstein aside, everywhere at the same pace. For timed testing this means that there is no time to waste. The testing apparatus, TorX, in this case, must not influence the outcome of the testing approach. However, the execution of TorX does consume time, and the question is when the execution time of TorX does influence the testing.

- Assume that input $i?$ is allowed to be applied at time $0 \leq t \leq b$. Assume that the testing tool needs $b/2$ to prepare to apply the input. Then the input can never be applied between time 0 and $b/2$. If there is an error hiding in this time interval, it will not be detected.
- Assume that the tester is too slow to apply $i?$ before b. Then this input can not be applied, and some behaviour of the IUT might never be exercised.

This basically means that the speed of the testing tool and the speed of communication between tester and IUT determine the maximal speed of the IUT that can be reliably tested.

Springintveld et al. [13] define an algorithm to derive test-cases for testing timed automata. They prove that their approach to test timed automata is possible and even complete, but in practice infeasible, due to the enormous number of test-cases to be run. This is likely also the case for our approach and

thus limits the extend to which timed testing can be useful. Automatic selection of meaningful test-cases might be an important ingredient in future extensions of our approach. For the time being, our goal is to find out how far we can get with timed testing *as is* in practice. This will be subject of our further research.

6 Conclusions and Further Work

In this paper we have presented Timed TorX, a tool for on-the-fly real-time testing. We use non-deterministic safety timed automata as input formalism to describe system specifications, and we demonstrate how to use standard algorithms for zone-computations in order to make our approach work. It turns out that the existing TorX algorithms, especially in the PRIMER and DRIVER can in principle be reused in order to deal with time. The major difference is that the δ-closure of the specification is now an explicit step, and cab not be done implicitly in the PRIMER anymore.

Our approach is strongly related to the *tioco* testing theory [6]. Esp. the notion of quiescence we have defined in Section 3.3 is strongly motivated by the *tioco* definition. We have much confidence that the δ-closure we have defined for timed automata ensures that the test-cases and the on-the-fly testing algorithm as presented in this paper are indeed an instantiation of the *tioco* theory. However, a formal proof of this assertion has still to be provided.

Timed testing relies on precise measurement of time stamps, but measurement errors can never be avoided. Timed automata live in an ideal world. It is perfectly normal to specify that a particular output should occur exactly two seconds after a certain input. But what if the output comes after 2.001 seconds? Should this considered to be a failure or not? One approach would be to allow for slack, *i.e.,* don't allow for discrete values but for intervals in the specification of occurrence times. However, this defers the problem only to the boundary of the intervals. An approach that is currently considered is to go away from hard pass/fail verdicts, but to define continuous metrics which allow to express quantitatively how far an implementation deviates from the specification. Work on this is based on [7].

Acknowledgements. We thank Conrado Daws, Ed Brinksma and Laura Brandán Briones for discussions on timed automata, *tioco* theory and timed testing in general. Furthermore we thank Jan Tretmans for helpful comments on quiescence. Tim Willemse pointed out a mistake in an earlier approach to implement quiescence.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comp. Science*, 126(2):183–235, 1994.
2. A. Belinfante. Timed testing with TorX: The Oosterschelde storm surge barrier. In M. Gijsen, editor, *Handout 8e Nederlandse Testdag*, Rotterdam, 2002. CMG.

3. A. Belinfante, J. Feenstra, L. Heerink, and R. G. de Vries. Specification based formal testing: The easylink case study. In *2nd Workshop Emb. Systems (PROGRESS '01)*, pages 73–82, 2001.
4. A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12th *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer, 1999.
5. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets:*, volume 3098 of *LNCS*, pages 87–124. Springer–Verlag, 2004.
6. Laura Brandán Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software (FATES '04)*, volume 3395 of *LNCS*, pages 64–78, 2005.
7. L. de Alfaro, M. Faella, and M. Stoelinga. Linear and branching metrics for quantatative transition systems. In *Proc. ICALP'04*, volume 3142 of *LNCS*, pages 97–109. Springer–Verlag, 2004.
8. J-C. Fernandez, C. Jard, T. Jeron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 348–359. Springer-Verlag, 1996.
9. J-C. Fernandez, C. Jard, T. Jeron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1–2):123–146, 1997. Special Issue on COST 247, Verification and Validation Methods for Formal Descriptions.
10. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Journal Inf. and Comp.*, 111(2):193–244, 1994.
11. M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In S. Graf and L. Mounier, editors, *Proc. 11th Int. SPIN Workshop (SPIN 2004)*, volume 2989 of *LNCS*, pages 109–126. Springer-Verlag, 2004.
12. M. Mikucionis, B. Nielsen, and K. G. Larsen. Real-time system testing on-the-fly. In K. Sere and M. Waldén, editors, *15th Nordic Workshop on Programming Theory*, number 34, pages 36–38. Abo Akademi, Department of Computer Science, Finland, 2003.
13. J.G. Springintveld, F.W. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1–2):225–257, 2001.
14. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
15. J. Tretmans and H. Brinksma. Torx: Automated model based testing. In A. Hartman and K. Dussa-Ziegler, editors, *Proc. 1st European Conf. on Model-Driven Software Engineering*, Nürnberg, 2003.