Chapter 9

Timed model-based testing

Authors: H. Bohnenkamp, A. Belinfante

9.1 Introduction

Testing is one of the most natural, intuitive and widely used methods to check the quality of software. One of the emerging and promising techniques for test automation is *model-based testing*. In model based testing, a *model* of the desired behavior of the *implementation under test* (IUT) is the starting point for test generation. In addition, this model serves as the oracle for test result analysis. Large amounts of test cases can, in principle, be algorithmically and automatically generated from the model.

Most model-based testing methods deal with black-box testing of functionality. This implies that the kind of properties being tested concern the functionality of the system. Functionality properties express whether the system correctly does what it should do in terms of correct responses to given stimuli, as opposed to, e.g., performance, usability, or reliability properties. In black-box testing, the specification is the starting point for testing. The specification prescribes what the IUT should do, and what it should not do, in terms of the behavior observable at its external interfaces. The IUT is seen as a black box without internal detail, as opposed to white-box testing, where the internal structure of the IUT, such as the program code, is the basis for testing. In this chapter we will consider black-box software testing of functionality properties.

Model-based testing should be based on formal methods: methods which allow the precise, mathematical definition of the meaning of models, and of notions of correctness of implementations with respect to specification models. One of the formal theories for model-based testing uses labeled transition systems (LTS) with inputs and outputs as models, and a formal implementation relation called *ioco* for defining conformance between an IUT and a specification [117, 118]; see also Chapter 11. An important ingredient of this theory is the notion of *quiescence*, i.e., the absence of output, which is considered to be observable. Quiescence provides additional information on the behavior of the IUT, and therefore allows to distinguish better between correct and faulty behavior. Moreover, the *ioco* theory defines how to derive sound 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 [10, 119] 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 [7, 9, 122]. Alternative approaches are, e.g., TGV [65], the AGEDIS TOOL SET [57], and SPEC EXPLORER [36, 84].

This chapter 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 certain times*. Our approach is influenced by, although independent of, the *tioco* theory [26], an extension of *ioco* to real-time testing. Whereas the *tioco* theory provides a formal framework for timed testing, we describe in this chapter an algorithmic approach to real-time testing, inspired by the existing implementation of TORX. We use as input models nondeterministic *safety timed automata*, and describe the algorithms developed to derive test cases for timed testing.

Related Work. Other approaches to timed testing, based on timed automata, exist, described in particular in [74, 85]. The big difference is that we take *quiescence* into account in our approach.

TORX itself has in fact already been used for timed testing [7]. 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 Chapter. In Section 9.2, we introduce *ioco*, and describe the central algorithms of TORX. In Section 9.3, we introduce the class of models we use to describe specifications for timed testing and the adaptations to make it usable with TORX. In Section 9.4, we describe an algorithm for timed on-the-fly testing. In Section 9.5, we address practical issues regarding timed testing. We conclude with Section 9.6.

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 $i = 1, \dots, n$.

9.2 Preliminaries

9.2.1 The *ioco* way of testing

In this section we give a summary of the *ioco* theory (*ioco* is an abbreviation for "*Input-Output-Conformance*"). Details can be found in [117, 118].

The *ioco* Theory A *labeled 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 behavior 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 *$\delta$-closure* $\Delta(L) = (L.S, L.s_0, L_I \cup L_U \cup \{\tau\} \cup \{\delta\}, \rightarrow')$, where $\rightarrow' = L.\!\rightarrow \cup \{(s, \delta, s) \mid s \in L.S \wedge \delta(s)\}$.

We introduce some more notation to deal with a transition system 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 $s \overset{a}{\Longrightarrow} s'$ iff $\exists s'', s''' \in L.S : s \Longrightarrow s'' \xrightarrow{a} s''' \Longrightarrow s'$. The extension to $s \xRightarrow{a_1 \cdots a_n} s'$ is defined similarly as above.

For a state $s \in \Delta(L).S$, the set of *suspension traces* from s, denoted by $Straces(s)$, are defined as $Straces(s) = \{\sigma \in (L_I \cup L_U \cup \{\delta\})^* \mid s \overset{\sigma}{\Longrightarrow}\}$, where $\Longrightarrow$ is defined on top of $\Delta(L).\!\rightarrow$. We define $Straces(L) = Straces(\Delta(L).s_0)$. For $s \in \Delta(L).S$, we define $out(s) = \{o \in L_U \mid s \xrightarrow{o}\} \cup \{\delta \mid \delta(s)\}$, and, for $S' \subseteq \Delta(L).S$, $out(S') = \bigcup_{s \in S'} out(s)$. Furthermore, for $s \in \Delta(L).S$ and $\sigma \in (L_I \cup L_U \cup \{\delta\})^*$, $\underline{s \textbf{ after } \sigma} = \{s' \in \Delta(L).S \mid s \overset{\sigma}{\Longrightarrow} s'\}$, and for $S \subseteq \Delta(L).S$, $\underline{S \textbf{ after } \sigma} = \bigcup_{s \in S} \underline{s \textbf{ after } \sigma}$. We define $\underline{L \textbf{ after } \sigma} = \underline{\Delta(L).s_0 \textbf{ after } \sigma}$.

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

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

The last line basically says that an implementation is only correct with respect to *ioco* if and only if all the outputs it produces, or quiescent phases, are predicted by, and thus correct according to, the specification.

Testing for *ioco* Conformance: Test Case Derivation The most important property of the *ioco* theory is that it is possible to derive test cases from specifications *automatically*. If an IUT fulfills certain assumptions (these assumptions are commonly known as *testing hypothesis*) then the *ioco*-test cases are *sound*: failing an *ioco*-test case implies that the IUT is not *ioco*-conformant with the specification. Test cases are described as 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: $T ::= \textbf{pass} \mid \textbf{fail} \mid a;T \mid \sum_{i=1}^{n} a_i T_i$, for $a, a_1, \ldots, a_n \in L_I \cup L_U \cup \{\delta\}$. Assuming an LTS $\mathit{Spec} \in \mathcal{L}(L_I \cup L_U)$ as a specification, test cases are defined recursively (with finite depth) according to the following rules. Starting with the set $S = \{s \mid \Delta(\mathit{Spec}).s_0 \Longrightarrow s\}$,

(1) $T := \textbf{pass}$ 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 $S' \neq \emptyset$, 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 $S_x = \underline{S \textbf{ after } x}$.

With not too much phantasy it is possible to imagine an algorithm which is constructing test cases according to the three rules given above.

9.2.2 On-the-fly *ioco* testing: TORX

In Figure 9.1 we see the tool structure of TORX. We can distinguish four tool components (not counting the IUT): EXPLORER, PRIMER, DRIVER and ADAPTER. The

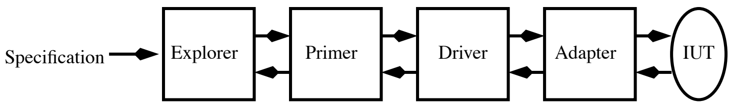

Figure 9.1: The TORX tool architecture.

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 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

```
Algorithm Compute_Menu
1   input: Set of states $S$
2   output: Sets of transitions $in$, $out$
3   $in := \emptyset$
4   $out := \emptyset$
5   $already\_explored := \emptyset$
6   foreach $s \in S$
7       $already\_explored := already\_explored \cup \{s\}$
8       $S := S \setminus \{s\}$
9       $is\_quiescent :=$ true
10      foreach $s \xrightarrow{a} q' \in Spec.\rightarrow$
11          if $a = \tau$
12              $is\_quiescent :=$ false
13              if $q' \notin already\_explored$ : $S := S \cup \{q'\}$
14          else :
15              if $a \in L_I$ : $in := in \cup \{s \xrightarrow{a} q'\}$
16              else :
17                  $out := out \cup \{s \xrightarrow{a} q'\}$
18                  $is\_quiescent :=$ false
19      end
20      if $is\_quiescent$ : $out := out \cup \{s \xrightarrow{\delta} s\}$
21  end
22  return($in$,$out$)
```

Figure 9.2: Menu computation.

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 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 9.2.1 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

```
Algorithm Driver_Control_Loop
1    input: —
2    output: Verdict pass or fail
3    $(in, out) = \textit{\textbf{Compute\_Menu}}(\{s_0\})$
4    while $\neg stop$ :
5       if ADAPTER.$has\_output() \vee wait$:
6          $o =$ ADAPTER.$output()$
7          $M = \underline{out \textbf{ after } o}$
8          if $M = \emptyset$: $terminate(\textbf{fail})$
9          $(in, out) = \textit{\textbf{Compute\_Menu}}(M)$
10      else:
11         choose $i? \in \{a \mid q \xrightarrow{a} q' \in in\}$
12         if ADAPTER.$apply\_input(i?)$ :
13            $(in, out) = \textit{\textbf{Compute\_Menu}}(\underline{in \textbf{ after } i?})$
14   end
15   $terminate(\textbf{pass})$
```

Figure 9.3: Driver Control Loop.

state space. Recursive descent into the state space is stopped when a state is seen that has no outgoing τ transitions, or that was visited before.

The algorithm for the computation of menus is given in Figure 9.2. We assume LTS *Spec* $\in \mathcal{L}(L_I \cup L_U)$. Input to the algorithm is a set S of states. Initially, $S = \{Spec.s_0\}$. After trace $\sigma \in (L_I \cup L_U \cup \{\delta\})^*$ has been observed, $S = \underline{Spec \textbf{ after } \sigma}$. Note that the transitions with δ labels are implicitly added to the out set when appropriate (line 20). Therefore, the EXPLORER does not have to compute the δ-closure of the LTS it represents.

Given the computed menus *in*, out, the DRIVER component decides how to proceed with the testing. The algorithm is given in Figure 9.3. In principle, the DRIVER has to choose between the three different possibilities that have been given for the *ioco* test case algorithm in Section 9.2.1: 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 they are referenced, a dice is thrown and depending on the outcome either **false** or **true** is returned. The driver control loop therefore terminates with probability one, because eventually *stop* will return **true**. 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 $\underline{M \textbf{ after } a} = \{q' \mid (q \xrightarrow{a} q') \in M\}$.

Quiescence in Practice From the specification point-of-view, quiescence is a structural property of the LTS. 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 milliseconds, 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 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.

9.3 Timed testing with timed automata

In this section we describe timed automata, the formalism which we use to formulate specifications for timed testing.

9.3.1 Timed automata

A timed automaton is similar to an LTS with some extra ingredients: apart from states, actions and transitions, there are *clocks*, *clock constraints*, and *clock resets*. Timed automata states are actually called locations, and transitions *edges* or *switches*.

Clocks are entities to measure time. They take nonnegative values from a time domain $\mathbb{T}$ (usually the nonnegative real numbers) and advance linearly as time progresses with the same rate. Let $\mathcal{C}$ be the set of clocks. Clock constraints are boolean expressions of a restricted form: 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 $\mathcal{B}(\mathcal{C})$. Clock constraints evaluate to either true or false. Since they depend on clock valuations, which change over time, also the evaluation of clock constraints changes generally over time. Clock constraints are used in two places in a timed automata: as *guards* and as *invariants*. Every transition has a guard, which describes the conditions (depending on the clock valuations) under which the transition is enabled, i.e., can be executed. Locations, on the other hand, are associated with an invariant. An invariant describes the conditions under which it is allowed to be in its corresponding location. Invariants describe an urgency condition: a location must be left before an invariant evaluates to false.

Clock resets are subsets of $\mathcal{C}$ and are associated to transitions. If a transition is executed, all clocks in the corresponding clock set are set to 0. As before, the action set is divided into inputs and outputs.

In Figure 9.4 we see an example for a timed automaton. This timed automaton has 7 locations, named S_0 to S_6. There is only one clock, c. The switches are named $e_0, \ldots, e_{12}$ and are labeled with actions, ending on ? or !, distinguishing inputs from outputs. Transitions without action labels are considered to be internal, i.e., labeled

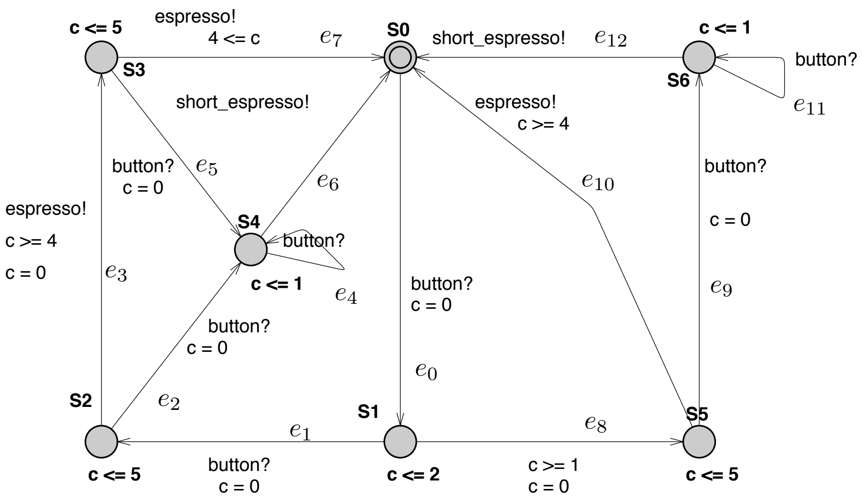

Figure 9.4: Timed Automaton.

with τ (e_8 in the example). The timed automaton describes the behavior of a coffee machine of which behavior depends on time. Starting location is S_0. Pressing a button (button?) brings us with e_0 to location S_1. Clock c is reset. The invariant of S_1 makes sure that within 2 seconds the location must be left again. This happens either by pressing the button again, which brings us with e_1 to location S_2. Alternatively, the internal transition e_8 can be executed: the guard enables it after 1 second of idling in S_1. Reaching location S_2 means that we have asked for two espressos. Consequently, going from S_2 to S_0 (edges e_3 and e_7) gives us two outputs espresso!. If we are in location S_5, we have pressed the button only once, and we can go with only one output espresso! back to location S_0. In locations S_2, S_3 and S_5 the invariants are always $c \leq 5$, and the transitions labeled with espresso! have a guard $c \geq 4$. This means that, since all transitions except those leading into S_0 reset clock c, one shot of espresso is produced within 4 and 5 seconds[1]. In locations S_2, S_3 and S_5 it is also always possible to receive another button press. This press cuts the coffee production short, i.e., via intermediate locations S_4 or S_6, we reach location S_0. The output is then, consequently, a short espresso (short_espresso!), which is obtained within 1 second.

A formal definition of a timed automaton follows.

Definition 1 (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 (partitioned into L_I and L_U as before), $l_0 \in N$ is the initial location, $E \subseteq N \times (Act \cup \{\tau\}) \times \mathcal{B}(\mathcal{C}) \times 2^{\mathcal{C}} \times N$ is the set of edges (or switches),*

[1]Actually a good espresso needs a bit longer than that.

and $I : N \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations. We define $\mathcal{A}(Act)$ to be the set of timed automata over the label set Act.

If $e = (l, a, g, r, l') \in E$, we also write $l \xrightarrow{a,g,r} l'$, where a is the action label, g is the guard, and r the clock reset.

9.3.2 Quiescence

Using a timeout to approximate quiescence has immediate impact on an approach to timed testing. Whereas in the un-timed case quiescence detection via time-out can not be described in the theory itself, in timed testing it should and actually must be described: a timeout is a timing property which influences therefore a timed test run. Incorporating the quiescence timeout into the timed testing technique is, as it turns out, a problem with a straightforward solution, which we will describe below. However, there is one assumption that must be made on the behavior of implementations.

Definition 2 *For an implementation Impl there is an $M \in \mathbb{T}$ such that*

- *Impl produces an output within M time units, counted from the last input or output, or,*
- *if it does not, then Impl will never ever produce an output again (without prior input).*

Only if this assumption on a real implementation holds, our test approach will work. This assumption is thus part of the above mentioned testing hypothesis.

A timed automaton to be used as a specification must be modified in order to express when quiescence is allowed to be accepted. To do that, it is necessary to know what M to assume. Then,

(1) an extra clock QC is added to the timed automaton;

(2) a self-loop labeled with special action δ is added to each location. Its guard is QC $\geq M$;

(3) clock QC is added to the clock reset of every transition labeled with an input or output;

(4) the guard of every output transition is extended with QC $< M$.

If $\mathcal{A}$ is a timed automaton, we denote this modified timed automaton as $\Delta_M(\mathcal{A})$.

9.3.3 From timed automata to zone LTS

TorX assumes that a specification is modeled in terms of labeled transition systems. In order to use TorX for timed testing, it is thus necessary to derive an LTS representation from a timed automaton. Such an LTS will be called a *zone LTS*. The technical details are not of interest here and can be found in [15]. Important to know, however, are the following facts. We assume a timed automaton $\Delta_M(\mathcal{A}) = (N, \mathcal{C}, Act \cup \{\delta\}, l_0, E, I)$.

(1) Time is measured in absolute time counted from system start (i.e., from the time when the initial state of the LTS was initially entered).

(2) States of the underlying LTS are of the form (l, z), where $l \in N$, and z is a so-called clock zone. The whole tuple is called a zone; z describes information about time. In particular, it defines an interval $z^{\downarrow} = [t_1, t_2] \subseteq \mathbb{T}$ which describes that only for absolute time $t \in [t_1, t_2]$ the state (l, z) might be entered.

(3) If we have a transition $(l, z) \xrightarrow{a} (l', z')$, we also write $(l, z) \xrightarrow{a@[t_1,t_2]} (l', z')$, if $z'^{\downarrow} = [t_1, t_2]$.

(4) For every transition $(l, z) \xrightarrow{a} (l', z')$ there is a corresponding edge $e = l \xrightarrow{a,g,r} l' \in E$.

(5) Given state (l, z), the successor clock zone of z for edge $e = l \xrightarrow{a,g,r} l'$ is denoted $z' = \mathit{Succ}(z, e)$. The successor state of (l, z) for e is then consequently (l', z'). It can happen that $\mathit{Succ}(z, e)^{\downarrow} = \emptyset$, which indicates that the switch e can not be executed, i.e., (l', z') is then not a successor state of (l, z).

(6) Zones can be instantiated. If $(l, z) \xrightarrow{a@[t_1,t_2]} (l', z')$ (with corresponding edge $e \in E$), and $t \in [t_1, t_2]$, then we can derive a new successor state (l', z'') of (l, z) such that $z''^{\downarrow} = [t, t]$. We denote this instantiated successor clock zone as $z'' = \mathit{Succ}(z, e, t)$.

The first transition of the zone LTS of Figure 9.4 corresponds to switch e_0 and has the form $(S_0, z^0) \xrightarrow{\texttt{button?}@[0,\infty]} (S_1, z^1)$ for intial clock zone z^0 and $z^1 = \mathit{Succ}(z^0, e_0)$. Since e_0 has no guard, the button can be pressed any time, i.e., between absolute time 0 and ∞. If `button`? is pressed at time t, then we derive $z^1_t = \mathit{Succ}(z^0, e_0, t)$. Then we can derive from edge e_1 the transition $(S_1, z^1_t) \xrightarrow{\texttt{button?}@[t+0,t+2]} (S_2, z^2)$ with $z^2 = \mathit{Succ}(z^1_t, e_1)$. From e_8, we can derive a transition $(S_1, z^1_t) \xrightarrow{\tau@[t+1,t+2]} (S_5, z^5)$, and with e_{10} also $(S_5, z^5) \xrightarrow{\texttt{espresso!}@[t+5,t+7]} (S_0, z^6)$, where $z^5 = \mathit{Succ}(z^1_t, e_8)$ and $z^6 = \mathit{Succ}(z^5, e_{10})$. This derivation shows that if the button is pressed once at, say, time 10, then without further interference we can expect an espresso between time 15 and 17.

9.4 Timed automata testing with TORX

In the following we describe the algorithms implemented in TORX for timed testing. We assume a timed automaton $\mathit{Spec} \in \mathcal{A}(L_I \cup L_U)$ and consider its δ-closure $\Delta_M(\mathit{Spec})$ for an appropriately chosen value M. Similarly to the un-timed case, TORX computes input- and output-menus, and chooses between applying an admissible input and waiting for an output.

```
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 $(l, z) \in S$
7        $already\_explored := already\_explored \cup \{(l, z)\}$
8        $S := S \setminus \{(l, z)\}$
9        foreach $e \in \{e' \in E \mid e'.l = l\}$
10           if $z' = Succ(z, e) \wedge z'^{\downarrow} \neq \emptyset$ :
11               if $e.a = \tau$: $S := S \cup \{(e.l', z')\}$
12               else :
13                   if $e.a \in L_I$ : $in := in \cup \{(l, z) \xrightarrow{a} (e.l', z')\}$
14                   else : $out := out \cup \{(l, z) \xrightarrow{a} (e.l', z')\}$
15       end
16   end
17   return($in$,$out$)
```

Table 9.1: Computation of menus from timed automata.

9.4.1 Menu computation

Based on the zone-LTS described above, TORX computes menus. The algorithm is similar to the one in Figure 9.2, but is specialized to account for the admitted time intervals of a zone. This algorithm ***Compute_Menu_TA*** is given in Table 9.1. The input of the algorithm is a set of zones S (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 $z'^{\downarrow}$ is not empty, transitions of the zone LTS 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, i.e., the δ action is not treated any different from an output. 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 LTS with sorting of the generated transitions according to their labels.

```
**Algorithm** ***Driver_Control_Loop_TA***
1    **input**: —
2    **output**: Verdict **pass** or **fail**
3    $(in, out) = \textbf{\textit{Compute\_Menu\_TA}}(\{(l_0, \{x = 0 \mid x \in \mathcal{C}\})\})$
4    **while** $\neg$ *stop*:
5        **if** ADAPTER.*has_output*() $\vee$ *wait*:
6            $o@t :=$ ADAPTER.*output*()
7            **if** $\underline{out\ \mathbf{after_t}\ o@t} = \emptyset$: *terminate*(**fail**)
8            $(in, out) := \textbf{\textit{Compute\_Menu\_TA}}(\underline{out\ \mathbf{after_t}\ o@t}, t)$
9        **else**:
10           choose $i@t \in \{a@t' \mid (l, z) \xrightarrow{a} (l, z') \in in \wedge t' \in z'^{\downarrow}\}$
11           **if** ADAPTER.*apply_input*($i@t$):
12               $(in, out) = \textbf{\textit{Compute\_Menu\_TA}}(\underline{in\ \mathbf{after_t}\ i@t}, t)$
13   **end**
14   *terminate*(**pass**)
```

Table 9.2: DRIVER control loop for timed systems.

9.4.2 Driver control loop

We define the following operator $\underline{\cdot\ \mathbf{after_t}\ \cdot}$, which maps menus on sets of zones.

Definition 3 ($\underline{\cdot\ \mathbf{after_t}\ \cdot}$) *Let $\Delta_M(\mathcal{A}) = (N, \mathcal{C}, Act \cup \{\delta\}, l_0, E, I)$ be a timed automaton, and let M be a menu. Then, for $a \in Act$ and $t \in \mathbb{T}$,*

$$\begin{aligned}\underline{M\ \mathbf{after_t}\ a@t} \quad = \quad & \{(l', z'') \mid (l, z) \xrightarrow{a} (l', z') \in M \\ & \textit{and } z'' = Succ(z, e, t) \textit{ with } z''^{\downarrow} \neq \emptyset\},\end{aligned} \tag{9.1}$$

where e is the respective switch corresponding to the $(l, z) \xrightarrow{a} (l', z')$ transition.

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 is 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.

In Table 9.2, 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 $i?$ 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 9.2.2).

9.5 Timed testing in practice

9.5.1 Notes on the testing hypothesis

The *testing hypothesis* is an important ingredient in the testing theory of Tretmans [117]. The hypothesis is that the IUT can be modeled 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 modeled 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 chapter, 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 un-timed 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 9.3.2, it is assumed that the implementation behaves such that quiescence can be detected according to Section 9.3.2, Definition 2. This an assumption, part of the test hypothesis, which the system designer may have to ensure.

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 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.

9.5.2 Limitations of timed testing

Even though the timed testing approach described in this chapter seems to be easy enough, timed testing is not easy at all. Time is a complicated natural phenomenon. It can not 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 behavior 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. [112] 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.

9.6 Conclusions

In this chapter we have presented Timed TORX, a tool for on-the-fly real-time testing. We use nondeterministic 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 TORX implementation is still in a prototype stage. Yet, small systems of the size of the coffee machine in Figure 9.4 can be tested.

Our approach is strongly related to the *tioco* testing theory [26]. In fact, it has been shown (although not published yet) that the testing technique described here is in fact a sound and exhaustive instance of the *tioco* theory.