Online On-the-Fly Testing of Real-time Systems

Marius Mikucionis
Kim G. Larsen
Brian Nielsen

BRICS Report Series RS-03-49

ISSN 0909-0878 December 2003

Online On-the-Fly Testing of Real-time Systems [*]

Marius Mikucionis Kim G. Larsen Brian Nielsen
{marius,kgl,bnielsen}@cs.auc.dk

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7B, 9220 Aalborg Øst, Denmark

Abstract

In this paper we present a framework, an algorithm and a new tool for online testing of real-time systems based on symbolic techniques used in UPPAAL model checker. We extend the UPPAAL timed automata network model to a test specification which is used to generate test primitives and to check the correctness of system responses including the timing aspects. We use timed trace inclusion as a conformance relation between system and specification to draw a test verdict. The test generation and execution algorithm is implemented as an extension to UPPAAL and experiments carried out to examine the correctness and performance of the tool. The experiment results are promising.

1 Introduction

The goal of testing is to gain confidence in a physical computer based system by means of executing it. More than one third of typical project resources is spent on testing embedded and real-time systems, but still it remains ad-hoc, based on heuristics, and error-prone. Moreover, it is estimated that 99% of processors produced today are targeted for embedded applications. Real-time and embedded systems require a special attention be given to timing where the moment of input and output event appearance is as important as the event itself. Therefore special attention must be paid to timing during testing. The goal of conformance testing is to check whether the behavior of the system under test (IUT) is correct (conforming) to that of its specification. We follow a model driven approach where a formal model (or specification) defines the required (real-time) observable behavior of the IUT, and from this we automatically derive and execute real-time test cases to determine whether the IUT is conforming.

Test cases can be generated *offline* where the complete test scenarios and verdicts are computed a-priori and before execution. The offline test generation is often based on a coverage criterion of the model like in [11, 5], on a test purpose as e.g. [12], or a fault-model as [10].

[*]Funded by Basic Research in Computer Science URL: `http://www.brics.dk/`.

A new approach to model based test generation is *online* testing that combines test generation and execution: only a single test primitive is generated from the model at a time which is then immediately executed on the IUT. Then the output produced by the IUT as well as its time of occurrence is checked against the specification, a new test primitive is produced and so forth until it is decided to end the test. An observed test run is a timed trace consisting of an alternating sequence of (input or output) actions and time delays. We use the term *on-the-fly* to describe a test generation and execution algorithm that computes a set of states and stimuli incrementally as needed and directed by the test execution.

There are several advantages of online on-the-fly testing. First, testing may potentially continue for a long time. A single test run may take hours or even weeks), and therefore very long, intricate, and stressful test cases may be executed. Second, the state-space-explosion problem experienced by many offline test generation tools is reduced because only a very limited part of the state-space need to be stored at any point in time. Third, offline test generators 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, see e.g. [6, 7].

On-the-fly testing from Promela [3] and LOTOS specifications for non-timed systems have been implemented in the TORX [2] tool, and practical application to real case studies show promising results [13, 2, 9]. However, TORX provides no support for real-time constraints.

In this paper we describe a technique for online testing of real-time systems, which consists of the following main contributions:

Real-time testing framework: We present the framework for automatic online testing of real-time systems from a densely timed automata model of the system under test and its environment.

Online Testing algorithms: We propose a set of algorithms for on-the-fly testing which ensure that the IUT is subjected to relevant tests and that verdicts are given according the specification. The algorithms use symbolic techniques derived from model-checking to efficiently represent and operate on infinite state-sets.

Tool implementation: We have implemented the algorithms by extending the UPPAAL tool. UPPAAL is a timed automata model checker developed jointly by a group of researches at Uppsala University and Aalborg University. We adopt the UPPAAL timed automata specification language and extend the verification engine for the on-the-fly testing. The most important feature of UPPAAL is the use of symbolic algorithms and data structures to represent and manipulate the real-valued clocks of timed automata.

Example: We demonstrate the applicability of our technique using a small case.

Experiments: We have conducted an experiment using our tool and technique to test an emulated IUT against non-trivial model of a train controller. We examine both the error detection capability and its performance. The results are promising.

2 Real-time Testing Framework

First we present the concept of timed I/O automaton as theoretical background used in model checking for reasoning about the real-time systems. We proceed with test setup and discuss how timed automata model is extended for test specification. In the last subsection we define what IUTs we consider to be conforming to specification and give an intuition for the test verdict.

2.1 Timed Automata

Let X be a set of non-negative real-valued variables called *clocks*, and $Act = \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ a set of input actions $\mathcal{I}$ and output-actions $\mathcal{O}$, (denoted $a?$ and $a!$), and the non-synchronizing action (denoted τ). Let $\mathcal{G}(X)$ denote the set of *guards* on clocks being conjunctions of simple 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* (timed automata) over (Act, X) is a tuple (L, ℓ_0, I, E), where L is a set of locations, $\ell_0 \in L$ is an initial location, $I : L \to \mathcal{G}(X)$ assigns invariants to locations, and E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times Act \times \mathcal{U}(X) \times L$. We write $\ell \xrightarrow{g,\alpha,u} \ell'$ iff $(\ell, g, \alpha, u, \ell') \in E$.

The semantics of a timed automata is defined in terms of a timed transition system over states of the form $p = (\ell, \sigma)$, where ℓ is a location and $\sigma \in \mathbb{R}_{\geq 0}^{X}$ is a clock valuation satisfying the invariant of ℓ. Intuitively, there are two kinds of transitions: delay transitions and discrete transitions. In delay transitions, $(\ell, \sigma) \xrightarrow{d} (\ell, \sigma + d)$, the values of all clocks of the automaton are incremented with the amount of the delay, d. Discrete transitions $(\ell, \sigma) \xrightarrow{\alpha} (\ell', \sigma')$ correspond to execution of edges $(\ell, g, \alpha, u, \ell')$ for which the guard g is satisfied by σ. The clock valuation σ' of the target state is obtained by modifying σ according to updates u. We write $p \xrightarrow{\gamma}$ as a short for $\exists p'.\ p \xrightarrow{\gamma} p', \gamma \in Act \cup \mathbb{R}_{\geq 0}$. A timed trace is a sequence of alternating time delays and actions in Act.

A *network of timed automata* $\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ over (Act, X) is defined as the parallel composition of n timed automata over (Act, X). Semantically, a network again describes a timed transition system obtained from those of the components by requiring synchrony on delay transitions and requiring discrete transitions to synchronize on complementary actions (i.e. $a?$ is complementary to $a!$).

2.2 Test Specifications

The test framework consists of the IUT and its environment that is to be simulated by the tester. In our case the tester is a test computer equipped with a *test specification*, an online testing engine, and an *adapter*, see Figure 1.

An IUT usually operates in a particular environment – a collection of conditions and assumptions that the IUT is used under. Not every environment is realistic or reasonable and therefore only tests relevant to this environment should be executed. Moreover we may want to test how the system would behave under very specific conditions, e.g. try some test

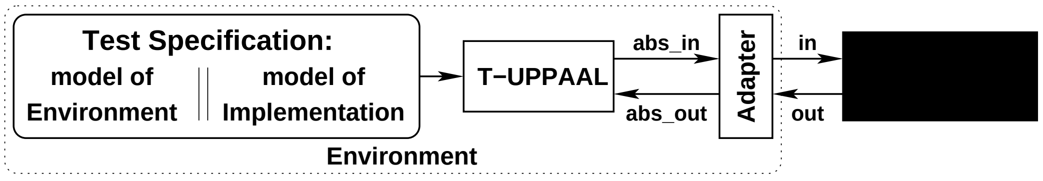
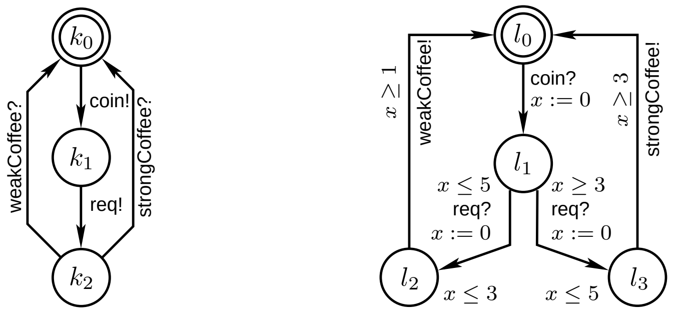


Figure 1: Framework of online testing using on-the-fly verification techniques.

purpose which leads to a failure. Also, by controlling the behavior and the amount of non-determinism in the environment model the user can guide or tune the test generation to run more efficiently. Therefore the test specification is allowed to be a parallel composition of a model of the implementation and a model of environment. A small example of a coffee machine model with its environment (one scientist) model is presented in Figure 2.



(a) Scientist. (b) Coffee machine.

Figure 2: Timed system model for famous coffee machine.

The adapter component translates the abstract input and output representation (*abs_in*, *abs_out*) into real actions (*in*, *out*) applied to and received from the IUT. We assume that the IUT and its model are input enabled to be able to accept any input offered at any moment in time. The tester is allowed to offer any input allowed by the environment from the test specification.

The test specification is given as a closed network of timed automata that can be partitioned into one subnetwork modeling the behavior of the IUT, and one modeling the behavior of its environment (ENV). The environment model can be replaced with a completely unconstrained one that allows all possible interaction sequences. We assume that the tester can take the place of the environment and control the IUT via a distinguished set observable input and output actions. We allow the full UPPAAL timed automata language, including non-deterministic (both w.r.t. actions and timing) specifications.

2.3 Conformance Relation.

A conformance relation defines what IUT behaviors are considered correct compared to its specification. In the non-timed setting the formal conformance relation *ioco* [3] has proved suitable in practice and is used in many model based testing tools. Informally, the ioco-relation requires that the implementation never produces an output that is not allowed by the specification, and that the implementation is quiescent (eternally unable to produce outputs) only when the specification may also be.

A natural extension of ioco to real-time systems is obtained by using *timed trace inclusion* as a conformance relation meaning that the timed traces of the IUT must be included in the timed traces of specification. Like *ioco*, this relation ensures that the implementation has no behavior that is not allowed by the specification. In particular, the implementation may stay quiet as time passes only if the same time passage is possible in the specification without an output. Thus, timed trace inclusion offers the notion of time-bounded quiescence that—in contrast to ioco's eternal quiescence—can be observed in a real-time system.

The conformance relation forms the basis for a test verdict – *pass*, *fail* or *inconclusive*. Consider that we know the current state the implementation is in, then the intuition of a verdict is as follows:

fail is given if some output or time delay observed is not permitted by the specification, i.e. the output is not defined in the model of IUT for the current state, or the delay is not allowed by an invariant on the state;

inconclusive is given if some output observed was not expected by the model of environment, i.e. an unpredicted event happened which prevents reaching the goal of the test purpose. Such events may happen for example if you consider testing a communication protocol and the connection was lost;

pass is given if all outputs observed are allowed by the specification also with respect to the time instances at which the outputs occur. The passage of time implies that the IUT may stay quiet (not produce any output) only if the specification allows.

However, we can hardly be sure what state our black-box IUT is in. We use the reachability algorithms from model checking to track the current state as described in the next section.

3 Test Generation and Execution

Our testing method consists of the main test generation and execution algorithm described below which controls the test process as described in Section 3.1. The main algorithm uses the adopted version of UPPAAL reachability algorithms to track the state of the IUT as described Section 3.2.

3.1 The Main Algorithm

The test generation and execution algorithm is based on maintaining the current reachable state set Z representing all states that the test specification can possibly occupy after the timed trace observed so far. Knowing this state estimate allows us to choose appropriate test primitives and to validate IUT outputs. Initially Z contains the initial state $\langle \bar{l}_0, \bar{0} \rangle$

where $\bar{l}_0$ is the initial location vector of timed automata network and $\bar{0}$ is the initial clock assignment where all clocks are zero. The process of the testing tool is described in Algorithm 1.

The tester can perform two actions: either send an input to the IUT, or wait for an output for some time. The choice of input is based on the current symbolic state set and the system model. If the output or time delay is observed then the tester checks whether it is legal according the specification at that moment. The current symbolic state set is updated each time the input is offered or the output or delay is observed. Illegal occurrence or absence of the output is detected if the current symbolic state set update leads to an empty set which is the result if the observed trace is not in the specification.

Algorithm 1 Test generation and execution. Initially $Z := \{\langle \bar{l}_0, \bar{0} \rangle\}$.

```
while $(\mathit{more\ time\ for\ testing}) \wedge (Z \neq \emptyset)$ do switch(action, delay) randomly:
    action:                                          // offer an input
        $a := \mathit{ChooseAction}(\mathit{EnvOutput}(Z))$
        send $a$ to implementation
        $Z := \mathit{After}(Z, a)$
    delay:                                           // wait for an output
        $\delta := \mathit{ChooseDelay}(Z)$
        sleep for $\delta$ time units and wake up on output $o$
        if $o$ occurs at $\delta' \leq \delta$ then
            $Z := \mathit{After}(Z, \delta')$
            if $o \notin \mathit{ImpOutput}(Z)$ then return fail
            else if $o \notin \mathit{EnvInput}(Z)$ then return inconclusive
            else $Z := \mathit{After}(Z, o)$
        else                                         // no output within $\delta$ delay
            $Z := \mathit{After}(Z, \delta)$
if $Z = \emptyset$ then return fail
else return pass
```

The functions used in Algorithm 1 are:

- The function *After* computes a closure of states reachable after performing all potential internal (unobservable) transitions and one observable input, output or delay. Due to non-determinism in the specification this requires a representation of a set of states as opposed to a single state. *After* returns an empty set if the action or delay was not allowed by the specification. The underlying algorithms are described in Section 3.2.

- The *EnvInput*, *EnvOutput* and *ImpOutput* functions compute the applicable input and output actions for the model of respectively the environment and the model of implementation:

$$\mathit{EnvInput}(Z) = \{a \in A_{out} \mid \langle \bar{l}, z \rangle \in Z, \langle \bar{l}, z \rangle \xrightarrow{a?}\} \quad (1)$$

$$\mathit{EnvOutput}(Z) = \{a \in A_{in} \mid \langle \bar{l}, z \rangle \in Z, \langle \bar{l}, z \rangle \xrightarrow{a!}\} \quad (2)$$

$$\mathit{ImpOutput}(Z) = \{a \in A_{out} \mid \langle \bar{l}, z \rangle \in Z, \langle \bar{l}, z \rangle \xrightarrow{a!}\} \quad (3)$$

- The functions *ChooseDelay* and *ChooseAction* respectively selects a delay and an action applicable to IUT when expected to occupy some state mentioned in Z. The *ChooseAction* and *ChooseDelay* functions currently selects actions or delays at random.

Different strategies can be applied to guide the test generation to “interesting” or uncovered states by changing the model of environment, “choose” functions and adopting them to a particular test purposes. The best results are expected in symbiosis with [5], [7] and other future works in this area.

3.2 Symbolic State-set Computation

We use symbolic constraint solving techniques to represent sets of clock valuations compactly. In particular, since we use real-valued clocks they cannot be represented explicitly. A *symbolic state* is a pair $\langle \bar{l}, z\rangle$ consisting of a location vector $\bar{l}$ and the clock constraint system z denoting a set of clock valuations, i.e. a symbolic state represents a set of concrete states: $\langle \bar{l}, z\rangle = \{(\bar{l}, \bar{v}) \mid \bar{v} \in z\}$.

We assume the following operations on clock constraints systems: conjunction $z \wedge z'$, future $z^{\uparrow} = \{\bar{v} + \delta \mid \bar{v} \in z, \delta \in \mathbb{R}_{\geq 0}\}$, clock x assignment to c value $z_{x:=c}$, containment check $z \subseteq z'$. The symbolic transition relation $\rightarrowtail$ between symbolic states denotes the possibility of taking a transition from a (concrete) state in the source symbolic state to a one in the destination. It is computed as follows:

$\langle \bar{l}, z\rangle \overset{\gamma}{\rightarrowtail} \langle \bar{l}', (z \wedge g)_r \wedge I(\bar{l}')\rangle$ if $\bar{l} \xrightarrow{g,\gamma,r} \bar{l}'$ is an (internal or synchronized) γ-action transition.

The required reachability algorithms for on-the-fly testing are similar to those used for model checking[8] except that only states up to a certain time limit needs to be computed. This is most easily accomplished by introducing an auxiliary clock t that is set to zero whenever an observable action occurs. Again due to non-determinism it is necessary to represent the state-set as a set Z of symbolic states.

Algorithm 2 computes the $Closure_{\delta\tau}(Z, d)$ function collecting the reachable symbolic state set within a delay from 0 to d. The predicate $Contains(Z, \langle \bar{l}, z\rangle)$ tests whether a symbolic state z is included in the state set Z.

Algorithm 2 $Closure_{\delta\tau}(Z, d)$ $passed := \emptyset$, $waiting := Z$..

```
while $waiting \neq \emptyset$ do
    $waiting := waiting \setminus \{\langle \bar{l}, z\rangle\}$          // pick a symbolic state
    $z := z^{\uparrow} \wedge (t \leq d) \wedge I(\bar{l})$                    // limited delay
    $passed := passed \cup \{\langle \bar{l}, z\rangle\}$
    for each symbolic transition $\langle \bar{l}, z\rangle \overset{\tau}{\rightarrowtail} \langle \bar{l}', z'\rangle$ where $\tau \notin \mathcal{I} \cup \mathcal{O}$
        if not $Contains(passed, \langle \bar{l}', z'\rangle)$ then $waiting := waiting \cup \{\langle \bar{l}', z'\rangle\}$
return $passed$.
```

The algorithm for the $Closure_{\tau}(Z)$ function computing the reachable symbolic state set after making all possible internal transitions in zero delay is similar to Algorithm 2.

Given the closure functions, the actual algorithms for computing $\mathit{After}(Z,\delta)$ and $\mathit{After}(Z,a)$ functions become trivial:

$$\begin{aligned}
\mathit{After}(Z,a) &= \mathit{Closure}_\tau\Big(\{\langle\bar{l}',z'\rangle \mid \langle\bar{l},z\rangle \in \mathit{Closure}_\tau(Z),\ \langle\bar{l},z\rangle \overset{a}{\rightarrowtail} \langle\bar{l}',z'\rangle\}\Big) \quad (4)\\
\mathit{After}(Z,\delta) &= \Big\{\langle\bar{l},z'\rangle \mid \langle\bar{l},z\rangle \in \mathit{Closure}_{\delta\tau}(Z,\delta),\ z' = \big(z \wedge (t == \delta)\big)_{t:=0}\Big\} \quad (5)
\end{aligned}$$

Note that the outer $\mathit{Closure}_\tau$ operation is redundant if we use $\mathit{After}(Z,a)$ only in a context of Algorithm 1 and is omitted in actual computations. We may also consider not to reset the clock t and count the absolute testing time with some minor adjustments which in turn would even more lessen the amount of expensive computations.

3.3 Implementation

We have implemented our algorithms in a prototype tool by extending the UPPAAL model-checker tool. UPPAAL, besides a graphical editor for timed automata, provides an efficient implementation of the needed symbolic constraint solving operations, and symbolic state successor computation. Unlike UPPAAL, our implementation does not store the reached state space, but only the current symbolic state set.

4 Experiments

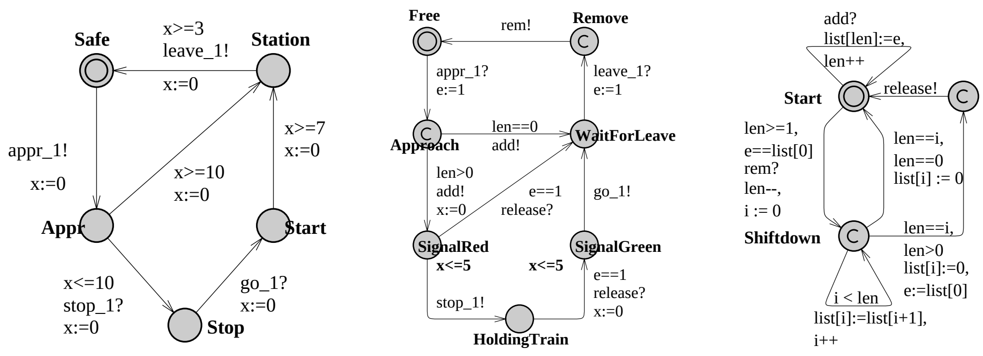
This section presents the results of a first set of experiments using our prototype implementation to test an implementation. The purpose of the experiments is to give some first indications of the feasibility of our technique in terms of applicability, error detection capability, and performance in terms of state-set size and computation time.

4.1 Test Specification

The experiment is based on a non-trivial version of a rail-road intersection controller that controls trains on a set of rail-road tracks with a shared track segment, e.g. a train-station. The main objective of the controller is to ensure that only one train occupies the shared segment at a time, and that they are granted access in arrival order.

Throughout the paper we use UPPAAL syntax to illustrate timed automata, and the figures are direct exports from UPPAAL. 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 as the next transition taken by the system. Finally, bold-faced clock conditions placed under locations are location invariants.

In the concrete setup we assume 4 tracks, and for simplicity 1 train per track at a time. Trains on track i signals the controller when they approach and leave the station using signals appr_i and leave_i respectively. When train i approaches an occupied station the



(a) Train 1 of 4. (b) Controller 1 of 4. (c) Queue

Figure 3: Test specification for train controller: trains as environment, controller and queue as implementation.

controller is required to issues a $stop_i$ signal within 5 time units, and is similarly required issue a go_i signals within 5 time units after the station becomes free.

The environment assumption model consists of 4 concurrent timed automata each modeling the assumed behavior a train. The model for train 1 is shown in Figure 3(a); the remaining trains are identical except for the train-id. In location $safe$ the train is outside the area of the controller. After signaling $appr_1$ it is in location in $Appr$. If it then receives a $stop_1$ signal before 10 time units has elapsed it stops and enters location $Stop$; after 10 time units the train may enter the station and thus change to location Station, and it may be too late to stop it (perhaps depending on the speed and weight of the train). A stopped train that has been resumed (thus in location $Start$) takes at least 7 time units to start and enter the station. It takes a train at least 3 time units to clear the station.

The model of the IUT requirements consists of 4 concurrent train control automata tracking the position of each potential train, and one queue automaton tracking their arrival order. Train id's are added and removed from the queue using signals add and rem and a shared integer variable e. Signal $release$ is issued by the queue to indicate the front of the queue. The shared integer variable len contains the number of elements stored in the queue. An approaching train (location $Approach$) is first enqueued, but immediately allowed to enter the station if no other trains are waiting ($len == 0$). The controller keeps the train in location $WaitForLeave$ until it is signaled that the train has left the station after which the controller dequeues it. If other trains were waiting, the controller issues a $stop$ signal within 5 time units as ensured by the invariant in location $SignalRed$, maintains and then holds the train in location $HoldingTrain$ until the front of the queue indicates that this is the next train to be granted access. Once release, the train is given the go signal within 5 time units.

Figure 3(b) depicts the control automaton for train 1, and Figure 3(c) depicts the queue automaton ($list$ is an array of integers, and i is an index into the array).

The complete test specification is reasonably large and nontrivial for a first experiment: it consist of 9 concurrent timed automata, 8 clocks, and a sequential queue data structure.

4.2 Implementation Under Test

The system under test is implemented as an approximately 100 line C++ program following the basic structure of the specification. It uses POSIX Threads and POSIX locks and condition variables for multi-threading and synchronization. It consists of one thread per train, and queue data structure whose access is guarded by mutual exclusion and condition variables. In the experiment, the IUT runs in the same address space as the T-UPPAAL tool, and input and output actions are communicated to and from the driver/adapter via two single place bounded buffers.

In addition we have created a number of erroneous mutations based on the assumed correct implementation (**M0**):

M1: The $stop_3$ signal is issued 1 time unit too late.

M2: The controller issues $stop_1$ instead of $stop_3$.

M3: The controller never issues $stop_3$

M4: The controller uses a bounded queue limited to 3 trains. Thus, the fourth train overwrites the third train in the queue.

M5: The controller uses LIFO queue instead of FIFO.

M6: The controller ignores $appr_3$ signals if a train arrives before 2 time units after entering the location $Free$.

4.3 Error Detection Capability

The purpose of this experiment is to determine that our technique can in fact detect basic timing and implementations errors, and to determine how fast this can be done.

The experiments are run on a 8x900 MHZ Sun Sparc Fire v880R workstation with 32 GB memory running Sun Solaris 9 (SunOS 5.9). T-UPPAAL runs on one CPU whereas the IUT may run on one or more of the remaining. T-UPPAAL it self does not require these extreme amount of resources, and it runs well on a standard PC, but the fact a multiprocessor is used in the experiment allows T-UPPAAL and the IUT to run in parallel as they would normally do in a black-box system level test.

To allow for faster and more experiments and reduce potential problems with real-time clock synchronization between the engine and IUT, the experiments are run using a simulated clock. Preliminary experiements with testing in real-time (with one model time unit corresponding to 10 ms on a single CPU linux host) we observe in about 5% of the (especially longer) runs that the clocks become unsynchronized.

The experiment is conducted by testing each mutant 1100 times. For each test run the number of communicated observable actions and total running time until the error is found is recorded. An upper bound of 10.000 time units is placed on each experiment. The number of pass, fail, and inconclusive verdicts together with the minimum, maximum, and average running time and number of used input actions are summarized in Table 4.

The results show that all erroneous mutants are killed surprisingly quickly using less than 22 input actions and less than 700 (model) time units. The assumed correct implementation is not killed in the available 10.000 time units using in average 377 input actions,

Table 4: Error Detection Capability:

Mu-	Number of verdicts				Input actions			Duration (time units)		
tant	Pass	Fail	Incon	Crash	Min	Avg	Max	Min	Avg	Max
M1	0	1100	0	0	2	5.0	18	6	72.7	359
M2	0	1099	0	1	2	4.6	12	3	66.7	370
M3	0	1100	0	0	2	4.8	12	6	80.2	389
M4	0	1100	0	0	6	8.6	22	37	163.4	641
M5	0	1099	0	1	4	5.7	14	17	92.0	435
M6	0	1100	0	0	2	3.9	14	6	62.8	349
M0	1077	3	10	10	99	376.0	442	2408	9951.1	10000

except 23 cases. In 3 cases T-UppAal wrongly gives a fail verdict. In the remaining 20 cases the our prototype produced wrongly an inconclusive verdict (meaning that the environment model received an unexpected output) or it crashed (segmentation fault). We do not know the exact reason for these problems, but they typically occur after long sequences of events on the correct IUT, and we believe that they are caused by a bug in our prototype implementation. Running this part of the experiment on a single CPU Linux host we neither observed the false negatives nor the crashes.

In conclusion, the results indicate that real-time on-the-fly testing may be a very effective technique, but that are memory leaks or thread synchronization bugs in T-UppAal that sometimes prevent very long test runs. Also the real-time clock synchronization between IUT and tester implemented in the prototype should be improved.

4.4 Performance

The purpose of this experiment is to evaluate the performance of the symbolic state-set exploration technique. Two metrics are important: the number of symbolic states (size) of the state-sets measured, and the amount of computation time to compute the state-set successors after a delay ran observable action.

Based on the same experimental setup as described in Section 4.3 we instrumented the T-UppAal tool to record the size of the symbolic state-set (i.e. the number of symbolic states in the state-set) as the test was executed, and to record the amount of CPU time used to compute the next state-set after a delay and an observable action. Because the test generation algorithm is randomized and because its behavior depends on the responses of the IUT, we made a total of 1100 test runs against each mutant **M0** resulting in more than 2 million data points for $After(delay)$ and 1.5 million $After(action)$.

Table 5 summarizes the results. The state-set size is in average only 2-3 symbolic states per state-set, but it varies a lot, up to 36 states. The state-set sizes reached after performing an action appear slightly larger than after a delay.

In average it costs 2.7 ms to compute the successor state-set after a delay, but less that 0.2 ms after an action. The $After(delay)$ algorithm is the most complex and computational demanding of the two, and it is not surprising that it cost an order of magnitude more.

Table 5: Execution Performance

Mu-tant	Execution time, μs				State-set size,			
	After(delay)		*After*(action)		*After*(delay)		*After*(action)	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
M1	2720.8	6739.3	123.3	762.2	2.31	17	2.65	34
M2	2783.9	6744.6	131.0	759.9	2.38	19	2.76	30
M3	2770.9	6640.2	125.9	755.5	2.38	20	2.68	30
M4	2696.1	6666.1	106.5	750.2	2.91	31	3.04	36
M5	2771.0	6830.4	129.6	731.2	2.94	31	3.26	32
M6	2814.6	6660.7	130.2	810.3	2.07	16	2.50	32
M0	2573.8	7066.6	78.0	722.4	2.91	32	2.83	44

Again observe that the computation time varies a lot, e.g. between 2 ms and up to 7 ms for $After(delay)$.

To examine these variations in greater detail and the dependency of computation time on state-set size, we created the scatter plot in Figure 4.4, also including the regression line of the mean. The figure shows the distribution of $After(delay)$ successor computation time of as function of state-set size. The figure shows graphically by far that most of the population of state-set sizes are concentrated below 5 symbolic states, and that very few are larger than 25. We found a similar, but less dispersed, pattern for $After(action)$ successor computation time.

Figure 7 plots the average successor state computation time as function of the state-set size. The $After(delay)$ computation time appear to depend linearly on the state-set size, where as $After(action)$ appear even sub-linear. But this conclusion is uncertain because only few measurements points are available for large state-set sizes.

In conclusion, based on this fairly large example, the performance of our technique looks very promising and appear to be fast enough for many real-time systems. Obviously, many more experiments with different model of varying size and complexity are needed to obtain firm conclusions.

5 Conclusions and Future Work

The current work on the on-the-fly test generator has resulted in the first prototype of T-UPPAAL (Testing-UPPAAL) [1], which show promising results in generating and executing test from non-trivial system models, such as the train-gate controller [4]. The concept is realizable, functional and the performance of the symbolic state set computation algorithms appear fast enough for many realistic real-time systems. However, further work is needed to evaluate the behavior and performance of the algorithms in details.

The future T-UPPAAL prototype development includes enhanced test generation and execution algorithm with further test event selection strategies. We also aim at improving

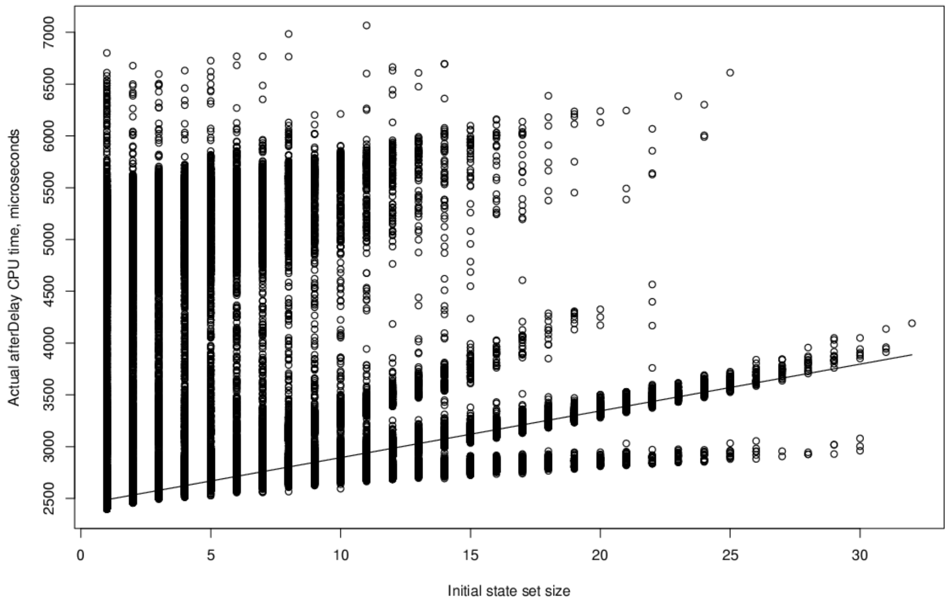
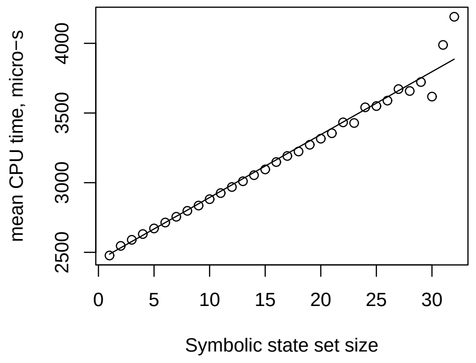
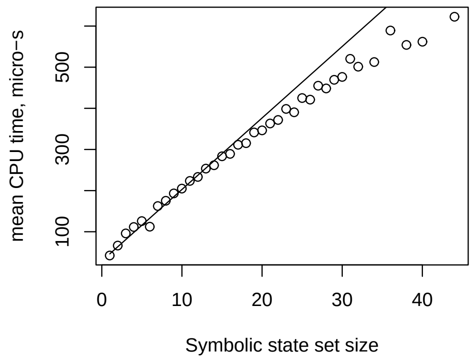


Figure 6: Distribution of *After*(delay) state-set successor computation on state-set size.



(a) *After*(delay)



(b) *After*(action)

Figure 7: The scatter plots of average CPU time per state-set size with regression line.

clock synchronization and time-stamp the events observed by an interval rather than by specific time points, and use absolute time measurements of events avoiding resets of the auxiliary t-clock in the algorithms.

The *data value passing* protocol described in [1] must be implemented to enable the data value communication between the tool and the IUT. Also it should be applied to realistic embedded and real-time systems and protocols.

References

[1] Marius Mikucionis and Egle Sasnauskaite. On-the-fly Testing Using UPPAAL. Aalborg University. Distributed Systems and Semantics unit, Department of Computer Science. Master thesis, June 2003. URL: `http://www.cs.auc.dk/ marius/master.pdf`

[2] R. de Vries, J. Tretmans, A. Belinfante, J. Feenstra, L. Feijs, S. Mauw, N. Goga, L. Heerink, and A. de Heer. Côte de Resyste in PROGRESS. In STW Technology Foundation, editor, PROGRESS 2000 - Workshop on Embedded Systems, pages 141-148, Utrecht, The Netherlands, October 13 2000.

[3] Rene de Vries, Jan Tretmans. On-the-Fly Conformance Testing Using Spin. University of Twente. Formal Methods and Tools group, Department of Computer Science. P.O. Box 217, 7500 AE Enschede, The Netherlands.

[4] Wang Yi, Paul Pettersson and Mats Daniels. Automatic Verification of Real-Time Communicating Systems by Constraint Solving. In Proceedings of the 7th International Conference on Formal Description Techniques, pages 223-238, North-Holland. 1994.

[5] Anders Hessel, Kim G.Larsen, Brian Nielsen, Paul Pettersson, Arne Skou. Time-optimal Real-Time Test Case Generation using UPPAAL. In the 3rd International Workshop on Formal Approaches to Testing of Software (FATES'03). Montreal, Canada,

[6] Ahmed Khoumsi and Thierry Jéron and Hervé Marchand. Test Cases Generation for non-deterministic real-time systems In the 3rd International Workshop on Formal Approaches to Testing of Software (FATES'03). Montreal, Canada,

[7] Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. In proc. TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems (Tiziana Margaria and Wang Yi Eds.), 343–357, Genova, Italy, April 2–6, 2001.

[8] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal Implementation Secrets. In Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02), 2002. URL: `http://www.uppaal.com/`

[9] 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, 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196. Kluwer Academic Publishers, 1999.

[10] Teruo Higashino, Akio Nakata, Kenichi Taniguchi, and Ana R. Cavalli. Generating Test Cases for a Timed I/O Automaton Model. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS), September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 197–214. Kluwer, 1999.

[11] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In J.-P. Katoen and P. Stevens, editors, *TACAS 2002*, pages 327–341. Kluwer Academic Publishers, April 2002.

[12] Thierry Jéron and Pierre Morel. Test Generation Derived from Model-Checking. In *International Conference on Computer Aided Verification (CAV'99)*, July 7–10 1999. Italy.

[13] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.