Improvements on the Online Testing with T-UppAal: Coverage Measurement and Re-runs

Gunnar Hall
Piotr Kordy
Dalia Vitkauskaitė

Master Thesis

Software System Engineering
Department of Computer Science
Aalborg University, Denmark
June 2005

ACKNOWLEDGEMENTS

We wish to use the opportunity to thank out supervisor Brian Nielsen for his guidance and patience. His knowledge and encouragement has, without doubt, greatly improved this work

We would also like to thank Marius Mikucionis for his support.

Aalborg, 2nd June 2005

Gunnar Hall

Piotr Kordy

Dalia Vitkauskaitė

ABSTRACT

T-UppAal is an online testing tool for model-based black-box conformance testing of embedded real-time systems. The study of re-runs and coverage measurement with the tool had not been studied so far. This report gives a precise description of different re-runs criteria's for use with T-UppAal. To find out which of them would work in practice, a detailed analysis of T-UppAal in regards to re-runs was done. We also propose coverage measurements that can be used to determine the quality of a test. An industrial case study was done on both selective type of re-runs and coverage measurements. To help with creating test re-runs and coverage measurements, the tool Butler was made. It includes an array of features for creating re-runs and calculating coverage measurements.

CONTENTS

1. Introduction 4
1.1 Types of Testing 4
1.2 Testing with Formal Methods 5
1.3 Online Testing 7
1.4 T-UppAal 7
1.5 Project Motivation 8
1.5.1 Failure Diagnosis 9
1.5.2 Test Quality 10
1.6 Project Goal 11
1.7 Structure of the Report 11

2. Timed Automata and Testing 12
2.1 Timed Automata 12
2.2 Timed I/O Transition Systems 12
2.3 Example Specification – Light Controller 15
2.4 Relativized Time Conformance 18
2.5 UppAal 19
2.6 T-UppAal 20
2.6.1 Options in T-UppAal 21
2.6.2 The Adapter and The System Layers 24

3. Test Re-run 26
3.1 Variations of Test Re-runs 26
3.1.1 Available Options for Re-run 26
3.1.2 Specification and Environmental Constraints 27
3.1.3 Timing tolerance 27
3.1.4 Relative Position of Timing Constraints 28
3.1.5 Order of Actions 28
3.1.6 Examples of Re-runs 29
3.2 Timed Automata Trace 33
3.3 Possible Problems Concerning Re-runs 34
3.3.1 Platform Behaviour 34
3.3.2 Effect of Non-determinism 39
3.3.3 Global and Relative Time 40
3.3.4 Narrowing of Gap 40
3.3.5 Conversion of Time Units 44

4. The Coverage Measurements . . . 46
4.1 Idea of Implementation . . . 46
4.2 Coverage and Non-determinism . . . 47

5. Butler . . . 50
5.1 The Tool Requirements . . . 50
5.2 The Tool Implementation . . . 53
5.2.1 Reading and Representing the Options . . . 53
5.2.2 Parsing and Representing a Specification . . . 53
5.2.3 Parsing and Representing a Log File . . . 54
5.2.4 Library for Parsing input/output Actions . . . 54
5.2.5 Integrating the Trace Log into the Specification . . . 55
5.2.6 Adding Coverage Variables . . . 56
5.2.7 Scaling Specification Precision . . . 56
5.2.8 Actions with Parameters . . . 57
5.2.9 The Tool Status . . . 58
5.3 The Tool Options . . . 59
5.3.1 Usage . . . 60

6. Experiments . . . 62
6.1 System Delay Estimate . . . 62
6.1.1 Experimental Setup . . . 62
6.1.2 Results . . . 67
6.2 Local Scheduling . . . 70
6.2.1 Setup . . . 70
6.2.2 Results . . . 73
6.3 Scheduling Resolution . . . 73
6.3.1 Setup . . . 76
6.3.2 Results . . . 78
6.4 Conclusions . . . 80
6.5 Coverage experiments . . . 81

7. Industrial case study . . . 87
7.1 The ECK control objectives . . . 87
7.2 Model Structure . . . 88
7.3 Model Adaptation . . . 89
7.4 The Tools and The Options . . . 91
7.5 Results . . . 92
7.5.1 Re-run with Conformance to Specification and Sequence of Actions . . . 92
7.5.2 Re-run without Implementation Part . . . 93
7.5.3 Coverage Measurements . . . 94
7.6 Summary . . . 95

8. Conclusions and Future Work 96
8.1 Epilogue and Conclusions 96
8.1.1 Butler 96
8.1.2 Re-runs 97
8.1.3 Coverage Measurements 98
8.2 Future Work 98

References 99

1. INTRODUCTION

Success in a software industry is more than delivering product to the market on time and in an efficient manner. The quality of the system must also be taken into the consideration. As complexity of the software is rapidly growing, quality assurance activities can compound up to half of the software development process for normal system. It is higher for the life critical systems.

The errors can be made at any stage of software development - requirement analysis, design or coding. The testing role in the software quality assurance is to detect the faults in the product and to make sure that the final functionality meets customer's requirements.

However, most of the produced software has some errors left. The reasons could be following:

- Untested code was executed
- Statements during testing were executed in different order
- The combination of untested inputs was encountered
- User's operating environment was not tested [24]

Consequently, the testing itself should be observed and analysed to improve the quality of the product. In the following section we overview the main types of testing. Later we introduce testing using formal methods and online testing. Finally we describe the T-UppAal testing tool and our objectives.

1.1 Types of Testing

There are several ways of classifying tests. Different aspects of system behaviour can be tested using conformance, performance, robustness, stress reliability, availability or security testing.

Conformance testing is a process that verifies if a system meets specification requirements and performs correctly in its designed environment. A specification indicates how a system should behave. Tests are applied to the implementation under test (IUT). A verdict (pass or fail) about system correctness is given according to the observations performed during testing.

Depending on software accessibility appropriate testing strategies can be used. There are two main strategies: white box and black box testing. In *white box testing* the internal structure of the IUT should be known. Differently in

black box testing the tester may know only possible inputs and expected outputs but not how the program produces them [20].

The testing process can be classified as static or dynamic. *Static testing* combines all the ways of finding errors that can be made without executing the code (inspections, walk-throughs, reviews etc.). It is one of the most effective ways to find defects in early stages of a software development. By contrast the *dynamic testing* is running the program and comparing its behaviour with specification [20].

There are a number of different testing approaches starting with informal ad hoc testing and finishing with formal specified and controlled methods.

1.2 Testing with Formal Methods

A system specification written in natural language can be interpreted in different ways. Such uncertainty complicates entire software development process and the testing as a part of it. A solution is using formal methods. The main advantages of using formal methods are [20, 14]:

- The techniques from mathematics and logics make specifications precise, complete and unambiguous, which makes it a good means for communication among system designers, analysts and testers.
- A variety of model checkers can be used to validate a specification written in a formal language so errors can be found in earlier stages of the system development life cycle. The specification is clear, precise and complete when it reaches the test engineer.
- The automation of the process becomes more accessible. In the test generation phase test cases can be derived algorithmically from a formal specification. Automatic online testing allows executing tests while they are generated.
- A verdict assignment and a result analysis are simpler because a comparison of expected and observed responses is easier to perform with formal methods.

Conformance testing using formal methods is verifying the black-box IUT functional behaviour according to a formal specification. There are two main phases of the process: test generation and test execution, see Figure 1.1. A programmer develops the system according the specification. During testing conformance to the specification is checked. A test suite is made from the same specification which is executed on the IUT giving a pass or fail verdict.

A number of formal models and specifications exists [11]:

- State-Based Models – this includes for example Finite State Machines (FSM), Input/Output automata (also extended with time to I/O Timed automata) or Specification and Description Language (based on Extended

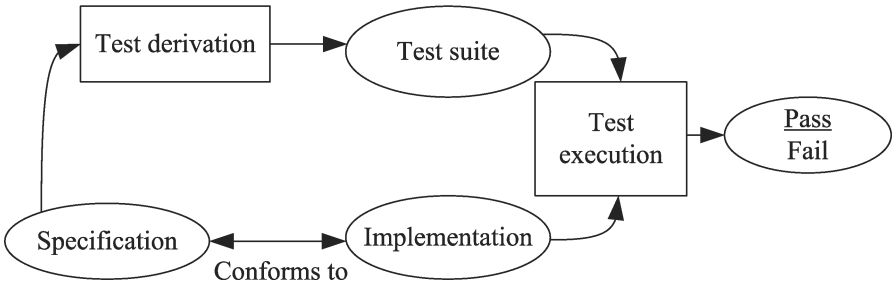


Fig. 1.1: Formal conformance testing process [20]

FSM) – the advantage of these models is that they usually provide the executable specifications, so they can be automatically converted into a simulator for the specified system

- Process Algebra Models – based on Communicating Sequential Processes (CSP) for example Timed Acceptances Model or Communicating Shared Resources
- Logic-Based Models – based on Temporal Logic, give possibility of asserting both safety and the liveness properties.
- Petri-Nets Models – a number of subclasses of Petri net models have been derived. Several models for timed Petri nets were proposed.

In our situation, we use *labeled transition systems* for describing behaviour of the system. A labeled transition system is a directed graph whose nodes are named as *states* and edges as *transitions*. One of the states is specified as *initial*. All transitions are labeled with a *action* indicating event that brings the system from one state to another.

A timed automata are a standard finite-state automata extended with a finite collection of real-valued clocks. That makes it an expressive formalism for modeling real-time systems.

The IUT specification is given as *a Timed Input-Output Automata* (TIOA), which is a timed automaton where the set of actions is partitioned into inputs and outputs. Where inputs are used to model actions performed by environment and outputs are under the system's control. TIAO is input and time-passage enabled.

We use *relativized timed conformance* to check if an implementation is correct according to a specification. The implementation is correct if it 1) produces the same output as a specification after the timed trace; 2) produces an output at a time when one is allowed by a specification; 3) omits to produce an output no longer than permitted by a specification [5].

1.3 Online Testing

During offline testing, test cases are generated completely and stored in a test notation language. The test cases are then executed on a system under test. The output is compared with the expected one and the decision is made according to the verdict of the test.

Online testing combines the test generation and execution (see Figure 1.2). One stimulus is generated at a time and immediately executed on a system under test. The produced output or quiescence is checked against the specification and a new test primitive is generated through all the testing process. online testing based on CPS has been proposed and applied in practice by J. Peleska [17]

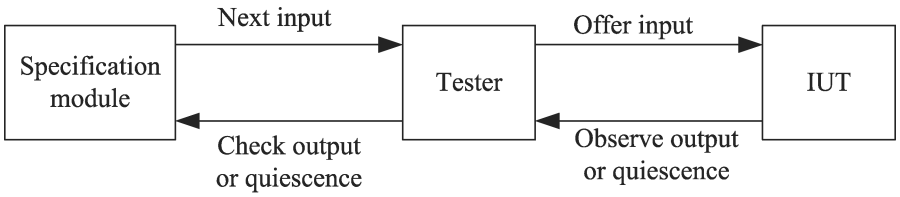


Fig. 1.2: Online testing [20]

The advantages of the online testing are [5]:

- A single test run can continue for a long time (hours or even days). Long and complicated test cases may be executed without a human interaction.
- In the offline testing the test steps must be precomputed. That reasons the huge size of a test suit. Contrarily, the online testing reduces the state explosion problem.
- It allows more expressive specification languages.

Non-determinism in the specification makes modeling flexible. If an implementation is too large and complex to interpret online in a real-time it can be changed to a more abstract one. So functional and/or timed behaviour is more important than the computation. If needed, the model can be a mixture of abstraction and precision [5].

The disadvantage of online testing could be the need to meet strict time constraints that is very difficult to assure for most of the systems. Also tools are relatively immature.

1.4 T-UppAal

T-UppAal is a testing tool for a model-based black-box conformance testing of embedded real-time systems. Tests are generated according to a formal timed automata model of the IUT and assumed operating environment, which combined, specifies the required and allowed behaviour of the IUT. The environment

and the model of a IUT are disjunct in T-UppAal. The reason is that it is easier to think of an environment and a IUT individually during the design process because they are separate in the real world. Another argument is that it is not simple to put TA instead of the environment when it is not specified explicitly.

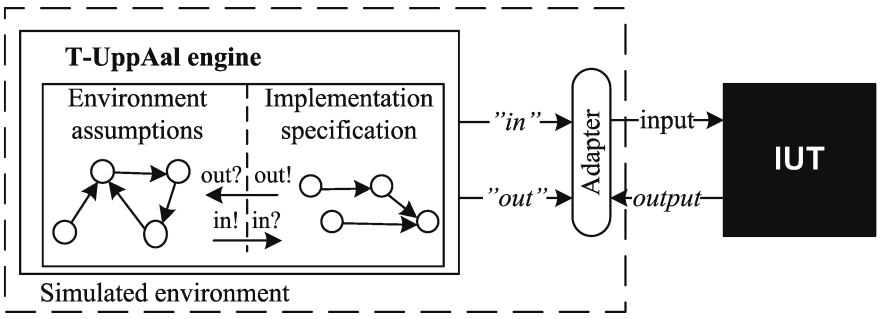


Fig. 1.3: T-UppAal

Since T-UppAal is an online testing tool it executes tests on the IUT as they are generated. The tool is based on the UppAal engine, which is a model-checker of real-time systems, modelled as networks of timed automata [23].

The IUT (Figure 1.3) is connected via an adapter and considered as a black-box. Communication with a system is via input/output channels. The user has to provide Timed Automata model of the IUT together with its environment. The latter is used to restrict the generated traces to realistic ones [22]. T-UppAal is explained with more detailed in Section 2.6.

1.5 Project Motivation

An online test of the IUT may potentially run for a long time at the same time generating a huge amount of data. A great value would be to obtain testing information that can be analysed and reused to achieve better testing performance:

- reproduce errors faster,
- test parts of the IUT that were not tested,
- easier find the cause of the error;

In some cases it is not easy to recall the scenario of how the error was found. We need to send exact sequence of inputs at the same time as it was during the test. If we had no data about time intervals when actions were send, we would need to try send several inputs in different time to repeat the failure. It is one of the reasons that makes process of reproducing the error longer and that could be eliminated by logging the test data.

When using randomly generated tests, a tester should be aware that some parts of the IUT could be left without testing. It could be that there are

number of conditions that must be satisfied in order to enter another part of the implementation. In this case testing process can be improved by combining random testing techniques and means of detecting which part of the IUT was not tested.

It can be the case, that the error was found after a long testing period. There is a minority of errors that needs all the trace to be repeated in order to reproduce them. Possibility to exclude the part of the trace that is only needed to reproduce the error would make testing process more efficient.

1.5.1 Failure Diagnosis

Consider the situation that the IUT was tested and a verdict *fail* was assigned. To report the failure, a test engineer must know the circumstances of its appearance. Moreover, a system developer should know how to reproduce the error in order to fix it.

Readability of the Data

The idea of model-based black-box conformance testing limits what we can observe during a test run to a sequence of input/output actions occurring at certain points in time. The information generated by T-UppAal about the executed test is written in a log file. The problem is that the file cannot be processed by any program and the format is not easily read by a human. Yet the hand-operated repeating of the test is not efficient.

As shown in Figure 1.4, to solve the problem we suggest to make a tool that builds a timed automata trace (TA trace) from the data stored in the log file (1). In order to re-run the test on the IUT automatically, the environment model should be replaced with our generated TA trace in T-UppAal. The TA trace could also be analysed in the UppAal model checker. The user could examine single steps or re-run the entire test on the model in order to check the states where the error occurred.

Repeatability of the Error

In order to reproduce the error we need to repeat the behaviour of the IUT. In some cases it is enough to send the same sequence of inputs as it was during the test run. However, to find the most difficult errors it may be necessary to repeat the exact behaviour of the T-UppAal and IUT.

There are many possible interpretations what does it mean to re-run a trace and also many technical problems connected (for more information see Chapter 3)

The problem is that we do not know what type of reruns we are able to perform using T-UppAal with TA trace.

The solution could be to analyse variants of a test reruns and to do experiments to find out if they are possible and what conditions should be satisfied in order to successfully reproduce the error.

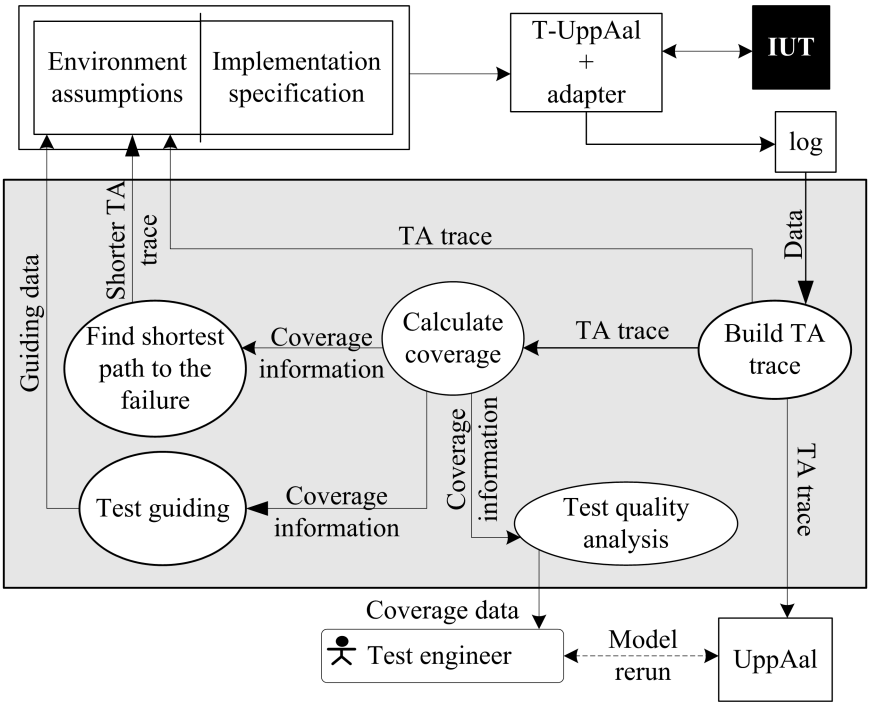


Fig. 1.4: System improvement proposal: suggested objects are in a grey area. Circles represent processes and arrows show data flow

1.5.2 Test Quality

Assume that the IUT was tested for some time and no fault was found. There could be two possibilities: there were no errors in the system or the test suite did not cover the part of the system that contained an error. The problem is that we do not know about the quality of the test generated by T-UppAal.

Our proposal is to calculate what part of the IUT was covered by the generated test and analyse the measurement. As shown in Figure 1.4 we will take data from the log file generated by T-UppAal and build a TA trace (1). The TA trace will be used to calculate how many times each location/edge was visited during the test (2). The obtained coverage information can be analysed (3) and given to a test engineer to evaluate the quality of the test.

Using coverage metric a tester can decide when to stop testing. It can be specified that test should stop after reaching target value of coverage measure.

Also there could be the situation that online test was running for a long time, and after certain time the coverage stopped increasing, but did not reach the target value. After analysis of such a data we could conclude about quality of test cases generated by the tool (maybe it never gives inputs to reach the certain part of the model).

1.6 Project Goal

We are examining the data of executed tests. Our aim is to discover analysis methods that makes it easier to search for a source of a failure. Our goal is to improve testing effectiveness by proposing following:

- Design an algorithm and a tool that creates a test trace according the data in the log file.
- Design an algorithm and create a tool that calculates the coverage.
- Use the implemented algorithm to simulate test the re-run in T-UppAal and try to indicate cause of the failure.
- Find the best possible ways of making a test re-runs.
- To evaluate performance of T-UppAal by analysing location, edge and code coverage.

1.7 Structure of the Report

The structure of the report is organized as follows. In Chapter 2 we introduce Timed Automata, TIOA network, relativized time conformance relation and give more details about T-UppAal. Variations of re-runs and problems concerning them are analysed in Chapter 3. The idea of coverage measure is presented in Chapter 4. Trace generation and coverage calculation algorithms and their usage are described in Chapter 5. Experimental results are presented in Chapter 6. Our experiment on industrial case study is described in Chapter 7 We outline conclusions and talk over future work in Chapter 8.

2. TIMED AUTOMATA AND TESTING

In this chapter we formally present our semantic framework. We introduce the Timed Automaton, the timed I/O transition system and the relativized input/output conformance relation. We give the semantics to the Timed Automaton using the timed I/O transition system. We stick to the definitions presented in [5].

2.1 Timed Automata

We assume that there is a set of actions Act that is partitioned into two disjoint sets of actions: the output actions A_{out} and the input actions A_{in}. We assume that there is a distinguished unobservable action $\tau \notin Act$. We denote by Act_τ the set $Act \cup \tau$.

One formal model for real-time systems is a Timed Automaton. Let X denote a set of non-negative real-valued variables called *clocks*. We let $\mathcal{G}(X)$ denote the set of *guards* on *clocks* being conjunctions of simple constraints of the form $x \bowtie c$, where $\bowtie \in \{\leq, <, =, >, \geq\}$ and $x \in X$ and $c \in \mathbb{N}$. Let $\mathcal{U}(X)$ denote the set of *updates* of clocks corresponding to sequences of statements of the form $x := c$, where $x \in X$, $c \in \mathbb{N}$.

Definition 2.1. *A Timed Automaton over (Act, X) is a tuple (L, l_0, I, E), where L is a set of locations, $l_0 \in L$ is an initial location, $I : L \rightarrow \mathcal{G}(X)$ assigns invariants to locations, and E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times Act_\tau \times \mathcal{U}(X) \times L$.*

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

2.2 Timed I/O Transition Systems

Definition 2.2. *A timed I/O transition system (TIOTS) $\mathcal{S}$ is a tuple $(S, s_0, A_{in}, A_{out}, \rightarrow)$, where S is a set of states, $s_0 \in S$ is the initial state, and $\rightarrow \subseteq S \times (A_\tau \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the usual timing constraints:*

- *time determinism – if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$ then $s' = s''$, $d \in \mathbb{R}_{\geq 0}$*
- *time additivity – if $s \xrightarrow{d_1} s'$ and $s \xrightarrow{d_2} s''$ then $s \xrightarrow{d_1+d_2} s''$; $d_1, d_2 \in \mathbb{R}_{\geq 0}$*

where $\mathbb{R}_{\geq 0}$ denotes non-negative real numbers

Definition 2.3. *Let* $a, b, ... \in Act_\tau, \alpha, \beta, ... \in Act_\tau \cup \mathbb{R}_{\geq 0}$, *and* $d, e, ... \in \mathbb{R}_{\geq 0}$.

- $s \xrightarrow{\alpha} =_{def} \exists s' : s \xrightarrow{\alpha} s'$
- $s \stackrel{a}{\Rightarrow} s' =_{def} s \xrightarrow{\tau}{}^* \xrightarrow{a} \xrightarrow{\tau}{}^* s'$
- $s \stackrel{d}{\Rightarrow} s' =_{def} s \xrightarrow{\tau}{}^* \xrightarrow{d_1} \xrightarrow{\tau}{}^* \xrightarrow{d_2} \xrightarrow{\tau}{}^* ... \xrightarrow{d_n} \xrightarrow{\tau}{}^* s'$ *where* $d = d_1 + d_2 + ..d_n$
- $s \stackrel{a}{\Rightarrow} =_{def} \exists s' : s \stackrel{a}{\Rightarrow} s'$

We extend $\Rightarrow$ to sequences of actions and delays in the usual manner.

Definition 2.4. *The timed I/O transition system* $\mathcal{S}$ *is strongly input enabled iff it specifies all input action transitions for all possible states:*

$$\forall s \in S \;\; \forall i \in A_{in} : \; s \xrightarrow{i}$$

Strongly input enabled I/O transition system will accept any input in any state.

Definition 2.5. *The timed I/O transition system* $\mathcal{S}$ *is weakly input enabled iff it specifies all input action transitions for all possible states within an arbitrary number of internal* τ *action transitions:*

$$\forall s \in S \;\; \forall i \in A_{in} : \; s \stackrel{i}{\Rightarrow}$$

Definition 2.6. *Let* $d_1 \dots d_{n+1} \in \mathbb{R}_{\geq 0}$ *and* $o_1 \dots o_n \in A_{out}$ *The timed I/O transition system* $\mathcal{S}$ *is non-blocking iff*

$$\forall s \in S \;\; \forall t \in \mathbb{R}_{\geq 0} \;\; \exists \sigma = d_1 o_1 ... d_n o_n d_{n+1} : \; s \stackrel{\sigma}{\Rightarrow} \wedge \sum d_i \geq t.$$

Definition 2.7. *The timed I/O transition system* $\mathcal{S}$ *is deterministic iff*

$$\forall s \in S \;\; \forall a \in Act_\tau \cup \mathbb{R}_{\geq 0} : \;\; if \left(p \xrightarrow{a} p' \wedge p \xrightarrow{a} p'' \right) then \; p' = p''$$

We assume that the timed I/O transition system $\mathcal{S}$ is strongly *input enabled* and *non-blocking*. Therefore $\mathcal{S}$ will not block time in any input enabled environments.

The semantics of a Timed Automaton T is defined by associating a TIOTS S with T. The states of a Timed Automaton are of the form $s = (l, \bar{v})$, where $l \in L$ is a location and $\bar{v} \in \mathbb{R}_{\geq}^{|X|}$ is a clock valuation satisfying the invariant of current location $l : \bar{v} \models I(l)$. Intuitively there are two kinds of transitions: delay transitions and discrete transitions.

Definition 2.8. *The Transitions for Timed Automata System S:*

- *let* $d \in \mathbb{R}_{\geq 0}$. *We say that* $(l, \bar{v}) \xrightarrow{d} (l, \bar{v}')$ *is a* delay transition *iff* $\forall d' \leq d$: $\bar{v} + d' \models I(l)$ *and* $\bar{v}' = \bar{v} + d$

- *let $\alpha \in Act$. We say that $(l, \bar{v}) \xrightarrow{\alpha} (l', \bar{v}')$ is a discrete transition iff there exists an edge $e = (l, g, \alpha, u, l')$ such that $\bar{v} \models g, \bar{v}' = u(\bar{v})$ and $\bar{v}' \models I(l')$.*

In delay transitions the values of all clocks of the automaton are incremented by the amount of the delay, d. Discrete transitions correspond to execution of edges (l, g, α, u, l') for which the guard g is satisfied by $\bar{v}$. The clock valuation $\bar{v}'$ of the target state is obtained by modifying $\bar{v}$ according to updates u and satisfies the invariants on l'.

Let us have two input enabled, non-blocking TIOTS $\mathcal{S} = (S, s_0, A_{in}, A_{out}, \rightarrow)$ and $\mathcal{E} = (E, e_0, A_{out}, A_{in}, \rightarrow)$. We call $\mathcal{E}$ the *environment* for the specification $\mathcal{S}$. E and S are correspondingly the sets of states for $\mathcal{E}$ and $\mathcal{S}$. e_0 and s_0 are the initial states. Any output action of $\mathcal{E}$ is the input action for $\mathcal{S}$ and vice verse.

Definition 2.9. *The parallel composition of $\mathcal{S}$ and $\mathcal{E}$ form a* closed system *$\mathcal{S}\|\mathcal{E}$ whose observable behaviour is defined by the timed I/O transition system $\langle S \times E, (s_0, e_0), A_{in}, A_{out}\rangle$ where we define $\rightarrow$ as:*

$$\frac{s \xrightarrow{a} s' \quad e \xrightarrow{a} e'}{(s,e) \xrightarrow{a} (s',e')} \qquad \frac{s \xrightarrow{\tau} s'}{(s,e) \xrightarrow{\tau} (s',e)} \qquad \frac{e \xrightarrow{\tau} e'}{(s,e) \xrightarrow{\tau} (s,e')} \qquad \frac{s \xrightarrow{d} s' \quad e \xrightarrow{d} e'}{(s,e) \xrightarrow{d} (s',e')} \quad [5]$$

where $a \in Act$ and $d \in \mathbb{R}_{\geq 0}$

In a similar way we can present the specification $\mathcal{S}$ as a Timed Automata Network $\mathcal{N} = (\mathcal{S}_1 || \ldots || \mathcal{S}_n || \mathcal{E}_1 || \ldots || \mathcal{E}_m)$ as a collection of concurrent Timed Automata composed by a parallel composition:

Definition 2.10. *Let assume that there exist a set of unobservable actions A_{intern} where $A_{intern} \cap (A_{in} \cup A_{out}) = \emptyset$ over which we allow timed automata to synchronize internally. Let $A_{1\_in} \cup \cdots \cup A_{n+m\_in} \cup A_{1\_out} \cup \cdots \cup A_{m+n\_out} \subseteq A_{intern}$. Let assume also that we have a number of TIOTS:*

$$\begin{aligned}
\mathcal{S}_1 &= (S_1, s_{i_1}, A_{in} \cup A_{1_in}, A_{out} \cup A_{1_out}, \rightarrow) \\
&\vdots \\
\mathcal{S}_n &= (S_n, s_{i_n}, A_{in} \cup A_{n_in}, A_{out} \cup A_{n_out}, \rightarrow) \\
\mathcal{E}_1 &= (E_1, e_{i_1}, A_{out} \cup A_{n+1_in}, A_{in} \cup A_{n+1_out}, \rightarrow) \\
&\vdots \\
\mathcal{E}_m &= (E_m, e_{i_m}, A_{out} \cup A_{n+m_in}, A_{in} \cup A_{n+m_out}, \rightarrow)
\end{aligned}$$

A parallel composition of $\mathcal{S}_1 \ldots \mathcal{S}_n \ldots \mathcal{E}_1 \ldots \mathcal{E}_m$ form a Timed Automata Network $\mathcal{N}$ whose observable behaviour is defined by the timed I/O transition system

$$(\mathcal{S}_1 \times \cdots \times \mathcal{S}_n \times \mathcal{E}_1 \times \cdots \times \mathcal{E}_m, (s_{i\_1}, \ldots, s_{i\_n}, \ldots, e_{i\_m}), A_{in}, A_{out}, \rightarrow)$$

where we define transition $\rightarrow$ as:

- *action transition:*

$$\frac{s_i \xrightarrow{a} s_i' \quad e_j \xrightarrow{a} e_j'}{(s_1, \ldots, s_i, \ldots, e_j, \ldots, e_m) \xrightarrow{a} (s_1, \ldots, s_i', \ldots, e_j', \ldots, e_m)}$$

  where $a \in A_{in} \cup A_{out}$, $1 \leq i \leq n$ *and* $1 \leq j \leq m$

- *synchronization transition*

$$\frac{s_i \xrightarrow{b} s_i' \quad s_j \xrightarrow{b} s_j'}{(s_1, \ldots, s_i, \ldots, s_j, \ldots, e_m) \xrightarrow{\tau} (s_1, \ldots, s_i', \ldots, s_j', \ldots, e_m)}$$

  or

$$\frac{e_k \xrightarrow{c} e_k' \quad e_l \xrightarrow{c} e_l'}{(s_1, \ldots, e_k, \ldots, e_l, \ldots, e_m) \xrightarrow{\tau} (s_1, \ldots, e_k', \ldots, e_l', \ldots, e_m)}$$

  where $b \in A_{i\_in} \cap A_{j\_out}$, $c \in A_{n+k\_in} \cap A_{n+l\_out}$, $1 \leq i, j \leq n$ *and* $1 \leq k, l \leq m$

- *delay transition*

$$\frac{s_1 \xrightarrow{d} s_1' \ldots s_n \xrightarrow{d} s_n', e_1 \xrightarrow{d} e_1' \ldots e_m \xrightarrow{d} e_m'}{(s_1, \ldots, s_n, e_1, \ldots, e_m) \xrightarrow{d} (s_1', \ldots, s_n', e_1', \ldots, e_m')}$$

  where $d \in \mathbb{R}_{\geq 0}$

One of the extensions that T-UppAal supports is *committed locations*. [15] An automata network is forced to perform the next action from that location, i.e., a committed location must be left immediately that is only the transitions from that location can be taken. Another extension is the use of integer variables. The variables can be updated and checked for value by guards in a similar way as clocks. The difference is that the values of the variables can be changed only through the updates but not by delay transitions.

2.3 Example Specification – Light Controller

In order to better understand specification as Timed Automata Network we give an example of such a specification. The specification is a controller for some smart lamp. When the switch is closed for very short time the impulse is ignored. If the switch is held for a short time then the light is turned on/off. If the switch is held for longer time then the level of light should be increased or decreased depending on the time for how long the switch was closed.

The specification consists of the network of four Timed Automata. We call them: *Interface* (Figure 2.1), *Dimmer* (Figure 2.2), *Switch* and *Adjust Light* (Figure 2.3). The environment is just one Timed Automaton (Figure 2.4). The locations are marked as circles and the transitions as arrows. The initial location is marked with double circle. The committed locations are marked

with C inside a circle. Some of the locations have invariants. The invariants are the text placed near the given location. If the label belongs to the input set of labels then the name of the transition ended with question mark. In the other case name ends with exclamation mark. The set of visible input actions consists of *grasp* (which corresponds to closing a switch or holding a wire) and *release* (which opens the switch). The set of visible output action consists of *setLevel_1...setLevel_10* (which corresponds to setting light to certain level).

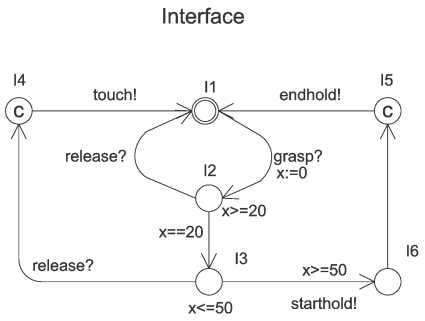


Fig. 2.1: Interface Timed Automaton of the Light Controller Specification

The first automaton *Interface* (Figure 2.1) is responsible for interpreting a sequence of *grasp* and *release* actions. Depending on the length between the consecutive grasp and release actions, they are ignored (if the delay was less than 20 time units) they generate the *touch* action (if the delay was between 20 and 50 time units) or they are treated as holding the wire (for the delay bigger than 50 time units)– the sequence of the *starthold* and *endhold* actions is generated. For measuring the time delay timed automaton uses the x clock variable.

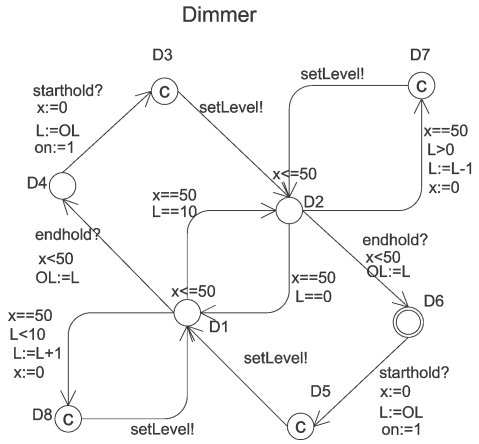


Fig. 2.2: Dimmer Timed Automaton of the Light Controller Specification

The second automaton *Dimmer* (Figure 2.2) interprets the sequence of *starthold* and *endhold* actions generated by the *Interface*. Depending on how long the delay is between those two actions, the appropriate actions are generated. The current level of light is stored in the L variable. When the automaton is in location *D2* the light level is decreased every 50 time units, and when in *D1* it is increased. This is done by updating the value of the L variable and generating the *setLevel* action. The delay is measured using the local clock variable x. Here we can see how the committed locations (*D7* and *D8*) can be used to make two transitions atomic. When we start holding the wire (close the switch) we set the light level to the stored old value of the light level ($L{:=}OL$). At the same time we set the variable *on*, which represents whether the light is on. We also need to generate the *setLevel* action so we use again the committed locations (*D5* and *D3*). When we end holding the wire (*endhold* action) we store the current value of light in the OL variable.

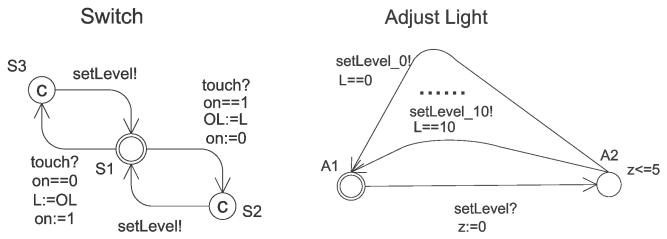


Fig. 2.3: Switch and *Adjust Light* Timed Automata of the Light Controller Specification

The next automaton *Switch* (left in Figure 2.3) is responsible for interpreting the *touch* action. When the *touch* action is received, it turns on/off the light by setting the *on* variable and storing/restoring the light level to/from OL variable. Also the *setLevel* action is generated.

The last automaton *Adjust Light* (right in Figure 2.3) is used to translate the *setLevel* action into the output actions *setLevel_0...setLevel_10*. Which action is generated depends on the current value of the variable L. A short delay is allowed between receiving the *setLevel* action and sending the output action. This delay can be up to 5 time units.

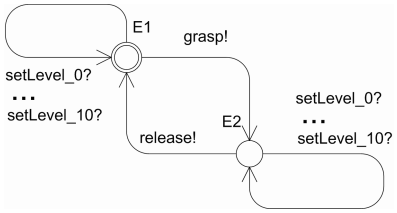


Fig. 2.4: Environment for the Light Controller Specification

To make the specification a closed system we need an environment. An

example model is shown in Figure 2.4. The environment presented here is very permissive. It allows any sequence of the input/output actions except that after the *grasp* action must follow the *release* action, i.e. it is not possible to generate two consecutive *release* or *grasp* actions.

2.4 Relativized Time Conformance

In this section we define the notion of a relativized conformance [5] between timed I/O transition systems. The notion is derived from the input/output conformance relation (ioco) of Tretmans [21] by taking time and environment constraints into account. This relation ensures that the implementation has no behaviour that is not allowed by the specification:

- it is not allowed to produce an output at a time that is not allowed by the specification
- it is not allowed to omit producing an output when one is required by the specification by delaying more than allowed.

To define the relativized timed conformance relation (rtioco) we need to introduce some terms. Let $\sigma \in (A_{in} \cup A_{out} \cup \mathbb{R}_{\geq 0})^*$ be an observable *timed trace* of the form $\sigma = d_1 a_1 d_2 \dots a_k d_{k+1}$.

Definition 2.11. *The observable timed traces TTr(s) of the state s is:*

$$TTr(s) = \left\{ \sigma \in (A_{in} \cup A_{out} \cup \mathbb{R}_{\geq 0})^* \,|s \stackrel{\sigma}{\Rightarrow} \right\}$$

Definition 2.12. *For a state s (and the subset $S' \subseteq S$) and a timed trace σ, s After σ is :*

$$s \text{ After } \sigma = \left\{ s' | s \stackrel{\sigma}{\Rightarrow} s' \right\}, \qquad S' \text{ After } \sigma = \bigcup_{s \in S'} s \text{ After } \sigma$$

s After σ is the set of states that can be reached after σ.

Definition 2.13. *For a state s (and the subset $S' \subseteq S$) the set Out(s) is:*

$$Out\,(s) = \left\{ \alpha \in A_{out} \cup \mathbb{R}_{\geq 0} | s \stackrel{\alpha}{\Rightarrow} \right\}, \qquad Out(S') = \bigcup_{s \in S'} Out(s)$$

$Out(s)$ is a set of observable outputs or delays that can occur in that state s. Now we are able to define the rtioco relation formally:

Definition 2.14. *Given an environment $\mathcal{E}$ with the initial state e_0, a system $\mathcal{S}$ (called implementation) with the initial state s_0 and a system $\mathcal{T}$ (called specification) with the initial state t_0, the $\mathcal{E}$-relativized timed input/output conformance relation $rtioco_{\mathcal{E}}$ between systems $\mathcal{S}$ and $\mathcal{T}$ is defined as:*

$$\mathcal{S}\ rtioco_{\mathcal{E}}\ \mathcal{T} \quad iff \quad \forall \sigma \in TTr(e_0): \ \ Out\,((s_0, e_0)\ After\ \sigma) \subseteq Out\,((t_0, e_0)\ After\ \sigma)$$

Whenever $\mathcal{S}\ rtioco_{\mathcal{E}}\ \mathcal{T}$ we will say that $\mathcal{S}$ is a correct implementation of the specification $\mathcal{T}$ under the environmental constraints expressed by $\mathcal{E}$.

2.5 UppAal

UppAal is a tool for modelling, verification and simulation of real time systems modelled as networks of timed automata. The tool consists of a graphical user interface and a model-checking engine (Figure 2.5).

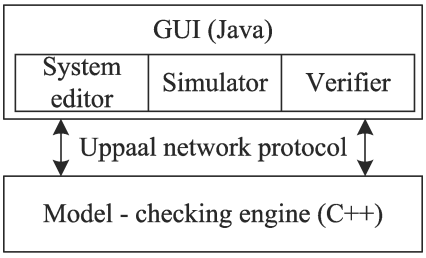


Fig. 2.5: UppAal architecture

There are three parts in the user interface : system editor, simulator and verifier.

In the system editor the user can model a real time system as a network of timed automata. These automata are designed using the UppAal modelling language that extends timed automata with additional features [3].

The simulator is a validation tool, which allows the user to examine the possible executions of the system during early design stages, and enables the fault detection prior to the verification by the model-checker. The user can control the simulation and select the (symbolic) state or transition to be visualized or perform a simulation step-by-step. The values of the data and the clock variables in the current state or transition selected are displayed. The current control points of each automaton of the selected automata edges are marked to indicate the current transition. A message sequence chart view of the generated trace is also showed in the simulator[23].

The verifier allows the user to enter the properties to be verified. The user can specify the system requirements and indicate the options according to which the verification is going to be performed. The requirement specification is expressed in a simplified version of computation tree logic. The result can be: the truth value of the property is unknown, the property is satisfied, and the property is not satisfied. The result of the model checking may include a diagnostic trace if the property is satisfied [23].

A command line version of UppAal exists. Given a specification and a set of properties it checks whether these properties are satisfied. It also gives the possibility of generating a trace that satisfies given property. The trace generated can be of the following type: 1) some , 2)shortest or 3)fastest. This diagnostic information is generated on the standard output. The advantage of using the command line tool is that we can re-direct the output to a text file. This allows to automate generating test and batch of experiments can be made more efficiently.

2.6 T-UppAal

As we mentioned before, T-UppAal is a testing tool for model-based black-box conformance testing of embedded real-time systems. The specification for the algorithm is in the form of two TIOTSs $\mathcal{S}||\mathcal{E}$. One represents the IUT and the other the environment. The interaction between $\mathcal{S}$ and $\mathcal{E}$ is done by explicitly declaring set of observable actions called the test primitives. An adapter can translate those primitives to the real interactions with the IUT. The randomized online testing algorithm [5] that T-UppAal uses maintains the current reachable set of states $\mathcal{Z} \subseteq \mathcal{S} \times \mathcal{E}$. It represents all the states the specification can occupy after the observed timed trace. Here $\mathcal{S}$ is a set of states and $\mathcal{E}$ the set of environment states. This information allows T-UppAal to choose the proper test primitive and validate the outputs from the IUT.

T-UppAal randomly performs one of three actions: send an input to the IUT, wait for the output, or reset the IUT. The state set is updated when an input is offered or an output or a delay is observed. The presence of improper output from IUT is detected when the set of current states becomes empty.

The functions used in Algorithm 1 are defined as:

$$EnvOutput(\mathcal{Z}) = \left\{ a \in \ A_{in} | \exists\, (s,e) \in \mathcal{Z}.e \xrightarrow{a} \right\}$$

$$Delays(\mathcal{Z}) = \left\{ d \,\middle|\, \exists\, (s,e) \in \mathcal{Z}.e \stackrel{d}{\Rightarrow} \right\}$$

$$ImpOutput(\mathcal{Z}) = \left\{ a \in \ A_{out} | \exists\, (s,e) \in \mathcal{Z}.s \xrightarrow{a} \right\}$$

The *EnvOutput* is the set of input actions that are allowed by the environment in the current state set, and is empty if environment model has no outputs to offer. If the environment must produce an input at a certain time, *Delays* must pick a real number from the interval that fulfils those constrains. To compute *After* the reachability algorithm from [2] is used. *ImpOutput* is the set of output actions that is allowed according to IUT specification in the current state set. The function $\mathcal{Z}\,After\,\alpha$ computes the set of states $\mathcal{Z}$ reachable after the action $\alpha \in A_{in} \cup A_{out}$. A_{in} denotes the set of input actions. A_{out} is the set of output actions. The function $\mathcal{Z}\,After\,\delta$ computes the set of states $\mathcal{Z}$ reachable after the time delay δ. The concrete realization of the Algorithm 1 is presented in [12].

The Main idea of the algorithm is the following. Start where the $\mathcal{Z}$ contains the initial states in both the environment and the specification. Then continually compute the set of states $\mathcal{Z}$ that the specification can be in after the observed test run so far. This is done until either $\mathcal{Z}$ is empty (no legal states) and a fail verdict is assigned, or it has reached the defined number of iterations (MaxNoIterations) and a passed verdict is assigned. One of the three basic actions T-UppAal does is:

1. Send an input (enabled output in the environment) randomly chosen among legal inputs according to the $\mathcal{Z}$. Then update $\mathcal{Z}$, according to $\mathcal{Z}\,After\,\alpha$(line 3-6).

2. Randomly choose how long it will wait for the output. If it observes the

output before the chosen time has passed, it updates the $\mathcal{Z}$ according to how long it have waited. It checks also if the output is a legal one according to $\mathcal{Z}$. If it is a legal output, $\mathcal{Z}$ is updated, else a fail verdict is assigned. If no output is observed during sleeping the $\mathcal{Z}$ is updated according to the passing of time (line 7-19).

3. Third, reset the IUT and restart the algorithm(line 21-22).

Algorithm 1: Test Generation and Execution

```
1:  Initially Z := {(s_0, e_0)}
2:  while Z ≠ ∅ ∧ NoIterations ≤ MaxNoIterations do
3:     action:
4:     randomly choose a ∈ EnvOutput(Z)
5:     send a to IUT
6:     Z := Z After a
7:     delay:
8:     randomly choose δ ∈ Delays(Z)
9:     sleep for δ and wake up on output o
10:    if o after δ' ≤ δ then
11:       Z := Z After δ'
12:       if o ∉ ImpOutput(Z) then
13:          return fail
14:       else
15:          Z := Z After o
16:       end if
17:    else
18:       Z := Z After δ
19:    end if
20:    restart:
21:    Z := {(s_0, e_0)}
22:    reset IUT
23: end while
24: if Z = ∅ then
25:    return fail
26: else
27:    return pass
28: end if
```

2.6.1 Options in T-UppAal

The T-UppAal tool itself is a command line programme. It does not have any GUI interface. The specification of the IUT must be built using other tools, such as UppAal. There are several command line arguments that can be specified for a test run. Below we describe the one we consider most relevant to re-runs and coverage measurements, other options are described further in [22]:

- I *filename*. References the specified dynamic library containing the binary adapter code to an implementation;
- b Use breadth-first search of the state space;
- d Use depth-first search of the state space;
- P delay
    - o eager: chose possible transition as soon as possible,
    - o lazy: chose possible transition as late as possible,
    - o random: delay within bounds specified by model,
    - o lower, upper: try random delay from the specified interval;
- u (inDelay,inRes,outDelay,outRes). Observation uncertainty intervals in microseconds:
    - o inDelay: the least delay that takes to deliver input,
    - o inRes: possible additional delay for delivering input,
    - o outDelay: the least delay that takes to observe output,
    - o utRes: possible additional delay for observing output;
- F future. The amount of future in model time units to be precomputed;

The -u and -F options

As the -u and -F options effects are subtler than the other options, we describe them as they are implemented in the current version of T-UppAal [16]. We use the following notions for describing them:

1. **Delay uncertainties:**
    - δ: The delay in sending *action* to the IUT (the source of the delay is later described in Section 3.3.1). The least delay is δ_{min} and the max delay is δ_{max}
2. **Time-stamps:**
    - t: current real-world time.
    - t_{tgt}: the time we want T-UppAal to deliver the input.
    - t_{try}: the time T-UppAal tries to send the input to the IUT.
    - t_{done}: the time input was delivered.
3. **Intervals:**
    - $\langle l, u \rangle$: real time interval where l is the lower bound and u is the upper.
    - $\langle L, U \rangle$: model time interval where L is the lower, U is the upper.

4. **Constants**

   - T: One model time unit corresponds to T amount of real time.

The algorithm 4 gives a more detailed description of line 4, 5 and 6 in the algorithm 1 in conjunction with the -u options and the -F options. These options are used to compensate for uncertainties in adapter delay by increasing the state set which the IUT can be in. For simplicity we omit in the algorithm preemption because of an output o received from the IUT. If there is an output from the IUT, the output is processed according to $\mathcal{Z}After\ o$ and then T-UppAal may try to send another input $a \in EnvOutput(\mathcal{Z})$.

Algorithm 3 and 2 which are used by the algorithm 4, show how model time units are converted to real time units and vice verse. Detailed description of how it is actually done is omitted for clarity and only the idea described. T-UppAal did not support $\delta^{in}_{min} > 0$ at the time of this writing. We assume $\delta^{in}_{min} = 0$ and do not include it in the algorithm.

The idea of algorithm 2 is to change the interval (l, u) (which should be a real time interval) to model time. As T-UppAal wants to take into account the observation uncertainties. The $delta^{in}_{max}$ is added to the upper interval, thereby making it larger.

The idea of algorithm 3 is very similar to the previous one. The input is still an interval except now it's in model time (L, U). The real time interval should not have any timing uncertainties(as the uncertainties are only used in computing of the state set$\mathcal{Z}After\ a$) and therefore $delta^{in}_{max}$ is subtracted from the real time $(T * U)$.

Algorithm 2: Simplified Real to Model Time

```
1: function ⟨L, U⟩ R2M(l, u)
2:   return ⟨(l/T, (u + δ^in_max)/T⟩
3: end function
```

Algorithm 3: Simplified Model to Real Time

```
1: function ⟨l, u⟩ M2R(L, U)
2:   return ⟨T * L, T * L − δ^in_max⟩
3: end function
```

As stated before, the algorithm 4 is a detailed description of what happens in line 4 to 6 in algorithm 1 with timing uncertainties. T-UppAal starts by getting the current real time. This time (after changed to model time units) is the lower value of the interval to compute $\mathcal{Z}After\ \tau([L, U])$. The upper value is determined by the - **F** options, and is the amount of future time we want to be precomputed. T-UppAal randomly chooses $a \in EnvOutput(\mathcal{Z})$ with the interval in which a should be sent to the IUT. The next step is to determine when a should be sent. As some time has been used for choosing a, T-UppAal

Algorithm 4: Working of T-UppAal with timing uncertainties

1: $t :=$ **getTimeNow()**
2: $\langle L, U\rangle :=$ **R2M**$(t, t + F)$
3: $\mathcal{Z}$ *After* $\tau([L, U])$
4: $a\,\langle L, U\rangle :=$ *randomly choose* $a \in$ *EnvOutput*$(\mathcal{Z})$
5: $\langle l, u\rangle :=$ **M2R**(L, U)
6: $t :=$ **getTimeNow**()
7: *randomly choose* $t_{tgt} \in (max(l, t), u)$
8: *wait until* t_{tgt} *time*
9: offer a to IUT
10: $t_{try} :=$ **getTimeNow()**
11: receive confirmation a sent to IUT
12: $t_{done} :=$ **getTimeNow()**
13: $\langle L, U\rangle :=$ **R2M**(t_{try}, t_{done})
14: $\mathcal{Z}$ *After* $\tau([L, U])$

needs to get the current time. The higher of l and t must be chosen. t is the current time and T-UppAal cannot send a at a time before time t. A random time from that interval is chosen to determine when a is sent. At the target time the action a is then sent to the IUT. Next T-UppAal stores the interval in which it tried to send it and to the time it got acknowledgement from the adapter that the actions has been sent. This interval is then used to compute $\mathcal{Z}$ *After* $\tau([L, U])$.

2.6.2 The Adapter and The System Layers

An important system component that is needed for testing is the adapter. The adapter must be programmed by "hand". The adapter is strongly dependent on the *IUT* and must be created (or adopted) for each one individually. Whenever there is a communication between the IUT and T-UppAal, the message (action) is sent first to the adapter before either going to the IUT or to the T-UppAal engine. The adapter translates then the action into IUT specific message and ensures that the IUT receive it. The adapter also handles the time stamping of when an action is performed.

One of the results of testing is a log file. We will call this file *trace log*. The log contains every input and output primitive that was sent by T-UppAal to the IUT and the delays that occurred between them. This log file can help the analysis of the quality of the testing or diagnose the cause of the error. All this is done in a separate part of the adapter called *the driver*.

The layers in T-UppAal that the actions pass, can be seen in Figure 2.6 for input actions and in Figure 2.7 for output actions. In Figure 2.6 we see the path for the input action α. It starts in the *engine* where T-UppAal has decided to send it according to T-UppAal algorithm 1. The action α is then passed trough the *driver* to the *adapter*. When α has been sent to the *IUT*,

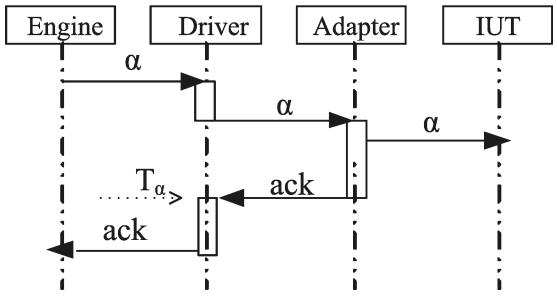


Fig. 2.6: Where the action α is time stamped

the *adapter* confirms that the action has been consumed to the *driver*. When the *driver* receives the *ack*, it time stamps the event by T_α. The *engine* then receives an update that the action α has been sent to the IUT at time T_α. Then *driver* writes this info to the log file.

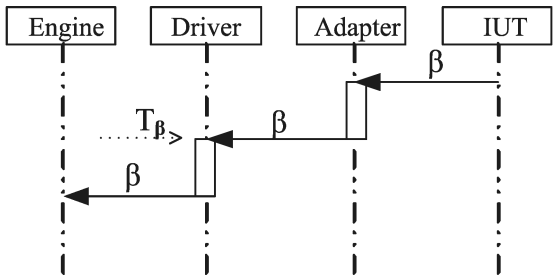


Fig. 2.7: where the action β is time stamped

The process is very similar for when output actions are received from the IUT. Instead of the action originating from the *engine*, it now comes from the *IUT*. The *IUT* communicates to the *driver* via the *adapter* that the output action β has occurred. As before, it is time stamped in the driver, sent to *engine* and then written by the *driver* to the log (with its time stamp).

3. TEST RE-RUN

The idea of test re-runs is important in testing and debugging. Whenever we encounter some error we would like to know if we are able to reproduce the occurrence of the error. Another use is to verify that the error was removed. In this chapter we will try to discuss different issues connected to the problem of making re-runs.

3.1 Variations of Test Re-runs

The idea of model-based black-box conformance testing limits what we can observe during a test run to a sequence of input/output actions occurring at certain points in time. In order to make a re-run of the test and reproduce the error we want to repeat the exact behaviour of the IUT. To do this we need:

- send the same input at the same time,
- check if received output conforms to the one received during the test run and that it was produced at exactly the same time;

Due to various factors discussed later in this chapter we are not able to specify exactly at what time the input action will occur. We are able to specify some bounds when the action will occur. We have a similar situation with output action. It may occur at slightly different moment as last time. These facts require that we define more specifically what it means to re-execute a trace.

3.1.1 Available Options for Re-run

In the following sections we will try to identify number of independent factors that may change how the actual re-run is done. We split them into four sections:

- Specification and Environmental Constraints
- Timing tolerance
- Relative position of timing constraints
- Order of the actions

3.1.2 Specification and Environmental Constraints

When making a run T-UppAal, creates a log file in which the time of occurring input/output actions is stored. When doing re-run this information is our basis for making a re-run. Additionally we have the specification according to which the run was done. This specification is divided into two parts: one describing environment and the other describing behaviour of the IUT.

When doing the re-run we have three options:

1. We may take the original specification constraints into account. Taking into account the original specification makes the re-run timing much more restrictive. This is because we are at the same time checking that during the re-run the IUT is conforming to the specification. The resulting time tolerance on action will be conjunction of constraints resulting from interpreting information stored in the driver log file and the fact that the IUT must conform to the specification.

2. We may take into account only part of the specification describing the behaviour of IUT. This way we still check if the IUT conforms to the specification and is as restrictive about when the actions should occur. The environment specifies constraints on the input actions. Without the environment we may happen to test the part that was not possible to test when using the original environment.

3. We may skip specification constraints. We take into account only the information stored in the driver log file. This way we only check if the re-run is *similar* to the original run, but we are not checking if the IUT still conforms to the specification during the re-run. This kind of re-run is the least restrictive of all three possibilities.

3.1.3 Timing tolerance

We can specify the timing tolerance in several ways:

- Specify one global tolerance for all actions. The size of timing tolerance will be the same for all actions.

- Specify timing tolerance separately for input actions and separately for output actions.

- Specify timing tolerance individually for each action. By this we mean that timing tolerance will be individual for each action, but all actions with the same label will have the same tolerance. There is also the possibility of not specify timing constraints at all for some actions (e.g. output actions).

In general it is difficult to say which type of specifying tolerance is best. It may happen that specifying one tolerance for all action is good enough, but for some cases it may happen that we need to specify individual tolerance for each action. It seems that having separate tolerance for input and output, but not to specify explicitly the tolerance for each action, is good enough for most cases.

3.1.4 Relative Position of Timing Constraints

Given information stored in the driver log file we need to specify the relative position of timing bounds when this action should occur. We can do this in several ways:

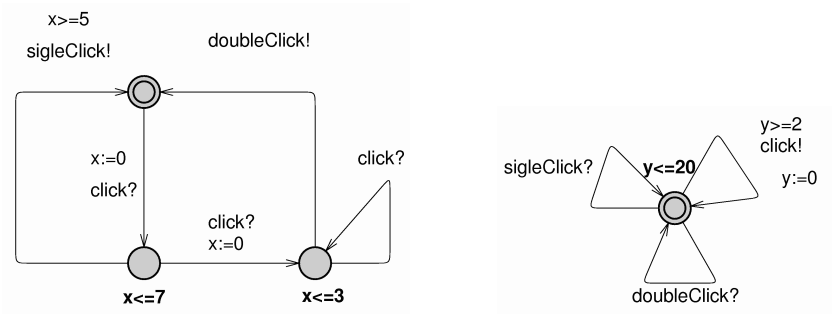
- Specify the timing constraints relative to the beginning of the run. It means that if the previous action occurred at different time as previously, it does not affect the time bounds the next action. This type of specifying timing bounds we will refer in this report as using *global time*.
- Specify the timing constraint relative to the time when the previous action had occurred. By this we mean that the action time bounds will be shifted by the difference between times when previous action occurred in the run and in the re-run. In our report we will refer to this type of specifying bounds as *relative time*.
- Specify the timing constraints of the action relative to the time when the action with the same label/type occurred. The timing bounds on the action will be shifted by the difference between the time when action with the same label occurred during the run and the re-run.

Which type of specifying bounds is the best one, depend on the behaviour of IUT. Basically we do not know how the implementation treats its time. It may start counting from the beginning after each interaction or always use the global time. It may also be a mixture of both local and global time. Thus the small differences between run and the re-run may or may not accumulate over time. In theory there can be a situation when no solution is good.

3.1.5 Order of Actions

Sometimes it may happen that if two actions occur quickly one after another. If the tolerance on actions is bigger that the time between two consecutive actions then the timing bounds of those actions will overlap. So another option is whether allow actions to occur in different order. We have three choices:

- All actions should happen in the same order as in the run. This is the strictest choice.
- Allow change of order only in certain situations. For example we may allow actions of different type to be swapped - if input action occurs next to the output action during re-run their order may be different.
- Do not pay any attention to the order of action as long as the timing constraints are satisfied. This option gives most freedom.



(a) Specification (b) Environment

Fig. 3.1: Specification and environment used to illustrate the examples

3.1.6 Examples of Re-runs

When we combine the parameters we can get many different types of re-runs. In this section we give examples how the re-run constraints can be constructed.

In all the examples we will be using the same specification. It is a specification of what can be a simple mouse controller. It can be seen in Figure 3.1(a). It accepts actions *click* and depending how quickly they are pressed after each other, a *singleClick* or a *doubleClick* action is produced. In Figure 3.1(b) we see the environment for that specification. We accept anything from the IUT and we try to input *click* actions not too fast and not too slowly – between 2 and 20 model time units.

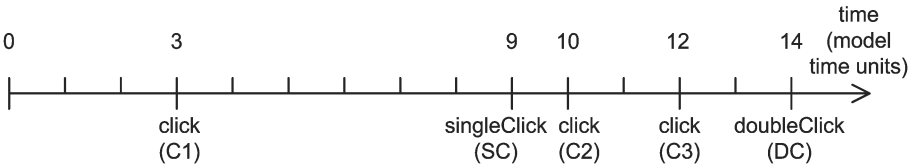


Fig. 3.2: The time line showing actions during exemplary run

In Figure 3.2 we can see an exemplary run that we will refer in the examples. The action *click* happened at the time of 3 model time units, then it was followed by the *singleClick* action and then we have 2 more *click* actions and one *doubleClick* action.

To make clearer description of generation of the re-run constraints, we need to present the constraints that were imposed by the specification and the environment in each step. We will present them in the form of intervals. To shorten notation and avoid ambiguity we will use alternative names for actions. The first action *click* will be represented by C1,*singleClick* by SC, second and third *click* by C2 and C3 and *doubleClick* by DC. Timing bounds may depend on

when some action occurred before. To represent the time when an action occurred we will use t letter superscripted with an action name. For example t_{C1} represents the time when the first *click* action occurred. We will use the same notation when presenting the constraints for the re-runs.

Action name	Specification constraints	Environment constraints
C1	$[0, \infty]$	$[2, 20]$
SC	$[t_{C1} + 5, t_{C1} + 7]$	$[0, \infty]$
C2	$[t_{SC}, \infty]$	$[t_{C1} + 2, t_{C1} + 20]$
C3	$[t_{C2}, t_{C2} + 7]$	$[t_{C2} + 2, t_{C2} + 20]$
DC	$[t_{C3}, t_{C3} + 3]$	$[0, \infty]$

Tab. 3.1: Constraints imposed by the environment and the specification

In Table 3.1 we can see the constraints imposed by the specification IUT and the environment. The first *click* action can happen according to the specification at any time. The environment imposes the interval between 2 and 20 model time units. The *singleClick* action can occur between 5 and 7 model time units measured relative to the first click. The environment does not impose anything. When we look at the specification and Table 3.1 it may seem that there is slight inconsistency in the bounds for the C2 (second *click*) action. The specification does not have any bounds. We do not specify $[0, \infty]$ interval because this *click* must follow the *singleClick* action.

When looking at the row with C3 action it may seem that specification is not input enabled (specify constraint on input action). This is not true. We just omit possibilities for actions that would result in different run. For example in the second row we have not only possibility to perform *singleClick* action but also another *click* action. But this would lead to different run.

Example 1 - Environment with Specification and Global Time

The first example that we present is quite restrictive. We include environmental and specification constraints. The timing tolerance for all actions is the same. It is ± 2 model time units no matter if it is input or output action. The position of timing constraints is measured relative to the beginning of the run. We do not allow the actions to be mixed between each other. In Table 3.2 we can see the constraints derived from the log file. These are created following options we specified for this example. The additional part (in all rows except the fist one e.g. $[t_{C1}, \infty]$) is added because we want actions to occur in the same order as during the run. The final version of the constraints is the conjunction of the constraints from Table 3.2 and Table 3.1. It can be seen in the third column of Table 3.2. The only situation when environment affect the final interval is for the action $C3$. We can notice that the actual interval when action occurs depends how we choose the previous actions to occur. For example if the first *click* action occur after 1 model time unit then the allowed interval for *singleClick* action

Action name	Log file constraints	Final constraints: $Env \wedge Spec \wedge Log$
C1	$[1, 5]$	$[1, 5]$
SC	$[7, 11] \cap [t_{C1}, \infty]$	$[7, 11] \cap [t_{C1} + 5, t_{C1} + 7]$
C2	$[8, 12] \cap [t_{SC}, \infty]$	$[8, 12] \cap [t_{SC}, \infty]$
C3	$[10, 14] \cap [t_{C2}, \infty]$	$[10, 14] \cap [t_{C2} + 2, t_{C2} + 7]$
DC	$[12, 16] \cap [t_{C3}, \infty]$	$[12, 16] \cap [t_{C3}, t_{C3} + 3]$

Tab. 3.2: Example 1 - Constraints for the re-run derived from the log file

is $[7, 8]$ (in model time units), but if the *click* action occurs after 5 model time units then the interval for *singleClick* action is $[10, 11]$

Example 2 - Specification and Relative Time

In this example we set the options to little less restrictive values. We will include only specification constraints. The timing tolerance is individual for each action. For *click* it will be ± 1 mode time unit, for *singleClick* ± 2 model units and for *doubleClick* ± 3 model time units. The timing constraints will be positioned relative the time when action with the same label occurred last time. We additionally allow actions to change the order - the sequence of actions may be different in the re-run as long as the timing constraints are satisfied.

Action name	Log file constraints	Final constraints: $Spec \wedge Log$
C1	$[2, 4]$	$[2, 4]$
SC	$[7, 11]$	$[7, 11] \cap [t_{C1} + 5, t_{C1} + 7]$
C2	$[t_{C1} + 7 - 1, t_{C1} + 7 + 1]$	$[t_{C1} + 6, t_{C1} + 8]$
C3	$[t_{C2} + 2 - 1, t_{C2} + 2 + 1]$	$[t_{C2} + 1, t_{C2} + 3]$
DC	$[11, 17]$	$[11, 17] \cap [t_{C3}, t_{C3} + 3]$

Tab. 3.3: Example 2 - Constraints for the re-run derived from the log file

In Table 3.3 we can see the constraints derived from the log file for this example. The final constraints that take into account the environmental constraints can be seen in the last column. Similarly as in the first example the timing constraints for some actions depend on the time when the previous action occurs. For example action *singleClick* will have the timing tolerance $[8, 10]$ if the first *click* action occurs at 3 or $[9, 11]$ if the *click* action occurs at 4.

Example 3 - Input and Output Actions

In the third example the options are the less restrictive. We will not include any part of the specification constraints. We specify separately the tolerance separately for the input and output actions. For input actions it is ± 2 model time units and for the output actions it is ± 3 model time units. The position of

timing constraints will be relative the previous action occurred(*relative time*). We allow the order of action to change. For example in the re-run action SC is allowed before action $C1$ if the timing constraints are satisfied.

Action name	Log file constraints
C1	$[1, 5]$
SC	$[t_{C1} + 6 - 3, t_{C1} + 6 + 3]$
C2	$[t_{SC} + 1 - 2, t_{SC} + 1 + 2]$
C3	$[t_{C2} + 2 - 2, t_{C2} + 2 + 2]$
DC	$[t_{C3} + 2 - 3, t_{C3} + 2 + 3]$

Tab. 3.4: Example 3 - Constraints for the re-run derived from the log file

In Table 3.4 we can see the constraints derived from the log file for this example. These are the final constraints at the same time. As in the previous examples the timing constraints depend on the time when the previous action should occur, but only the relative position can change. The size of the *gap* is the same and depends only on the tolerance we specified.

Example 4 - Input Actions Only

In the last example the options are the least restrictive of all examples. We will not include any part of the specification constraints. We put constraints only on input actions. We allow output actions to appear at any time in any order. The tolerance for input actions is ± 3 model time units. The position of timing constraints will be relative the beginning of the run (*global time*). The order of input action should not change. In Table 3.5 we can see the constraints derived

Action name	Log file constraints
C1	$[0, 6]$
SC	$[0, \infty]$
C2	$[6, 9]$
C3	$[7, 10] \cap [t_{C2}, \infty]$
DC	$[0, \infty]$

Tab. 3.5: Example 4 - Constraints for the re-run derived from the log file

from the log file for this example. These are the final constraints also. The time when the action should occur is independent on the time when other actions occur.

Summary

We can see that there can be many ways how we can create constraints for the re-run. The biggest impact has the fact if we include specification constraints

or not. If we do include specification constraint, then we must be careful with choosing other parameters. The gap for the actions may be smaller than intended. In the extreme case including the specification constraints may result in deadlocks in the model. Not including the environment is much safer. We always have the same tolerance gap as we intended. The problem with this kind of re-run may be that we are not sure if the behaviour of the IUT still conforms to the model.

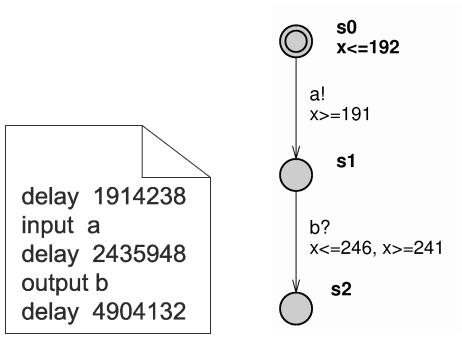
3.2 Timed Automata Trace

The idea behind Timed Automata trace is to take information stored in the driver log, decide which options we want to use during re-run (see Section 3.1) and create the Timed Automata. This Timed Automata (which we will call Timed Automata trace or TA trace in short) should be generated in a way that it can be easily merged with the specification. In general it is possible to create Timed Automata with any value of the option specified in Section 3.1. However we present here only a subset of the many possibilities. We assume that we will replace the environment with the TA trace we create. Thus we have no possibility of including the environmental constraints. We discuss later in this chapter why including environmental constraints would create problems. First we describe the way we create the TA trace and then tell what possibilities it gives us.

As mentioned in Section 2.6 T-UppAal writes all test primitives to the trace log. The data stored in the log was kept to minimum so not to affect the test run. The information stored there is straightforward. It stores labels of the test primitives, parameters to the labels and the delays between them.

The process of changing the debug trace into the TA is in principle quite straightforward one. For each test primitive in the log we create a location in the TA (except the initial location which we just include). Then we create the transition leading from the last added location to the newly added one, and on that transition a proper label is added (the input or output label). We also consider the delay by adding proper guards and invariants. In order to choose those guards we must know how many microseconds one model time unit is. We call this value *precision*. The precision value can be different from the one used in the original run. Care must be taken to avoid rounding errors and the specification itself must be modified. See Section 5.2.7 for the details. To take into account *value passing* we must update the variables or put additional guards to take the parameters into account (for the details see Section 5.2.8).

In Figure 3.3(a) we can see short driver log file. In Figure 3.3(b) we can see how the example TA trace could look like for this log file. In this example the order of actions must be the same as during the original run. We also assume that the timing constraints are relative to the beginning(*global time*) of the run. We assume that one model time unit was 10 000 microseconds. The tolerance on input was 0, so it was as small as possible. The gap for the input action to happen is therefore one model time unit. The tolerance for output was ± 2



(a) Definition of input output actions

(b) Driver log file

Fig. 3.3: The driver log and the TA trace

model time units.

3.3 Possible Problems Concerning Re-runs

Sometimes the re-run of the test can be difficult or even impossible. In this section we will discuss several factors that can influence the reruns. We divide them into four parts:

- Unpredictable platform behaviour ,
- Non-determinism,
- Global and relative time;
- Conversion of time units;

The following sections describe them in more detailed.

3.3.1 Platform Behaviour

To make re-runs it is important that the system is in the same initial state as it was during the original run.

Furthermore do we need to perform the same actions as we did in the original run at the precisely the same time. Because of the system delays that is rarely possible. The cause of the delay can be traced to computation, communication and scheduling.

Computation

During a test run, the set of states reached after the event trace must be computed in real time. The correctness of output/input sent to T-UppAal is validated based on this state set. The number of stored states depends on the size of model and if it is deterministic or non-deterministic. In more deterministic model the count of states remains relatively lower. Concurrent models can interleave events in many different orders so the state set may grow large for non-deterministic models. We can therefore expect more computation delay when dealing with non-deterministic models because computation time tends to be longer.

We can also expect the computation time to be different during a re-run than in the original run as the model is different.

Communication

The communication itself also takes time. A message needs to travel trough various operating and protocol a stack layers. Each layer handles specific part of the procedure of delivering a message over the net, before the message is sent over the physical wire. The International Standards Organization (ISO) defines seven layers [9]. We depict them and the route that the message from T-UppAal must traverse to get to the IUT in Figure 3.4. We are also aware that there are other ways that the adapter for the T-UppAal application could communicate with the IUT. For example if the IUT and T-UppAal are on the same computer, the adapter and IUT could use mutual exclusion and shared variables. Whatever method that is used between the IUT and T-UppAal's adapter, we must acknowledge that the time it take can vary and be unpredictable, thereby possibly affect the ability to make re-runs.

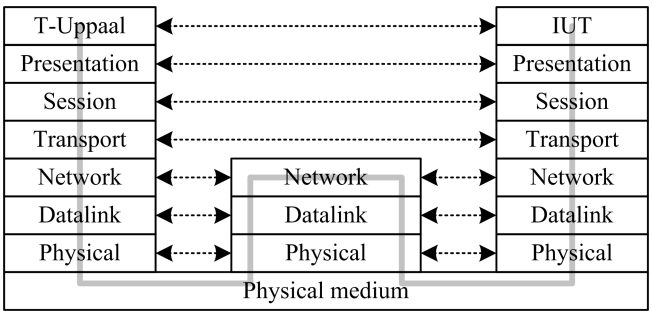


Fig. 3.4: ISO 7 layers

Like message takes time in getting to the physical medium(cable, radio, fiber, etc.) we also have a delay corresponding to the time in which the message is on the physical medium. This depends on bandwidth (how much information can be transferred over a connection in a given period of time) and a latency (how much time it takes for a response to return from a request) of the net.

Scheduling

Process scheduling in OS is the key to multiprogramming. The role of process scheduling is to assign which processes should execute on the CPU over time. In many multitasking systems the processor scheduling operates on three levels. They are differentiated by the time scale at which they perform their operations.

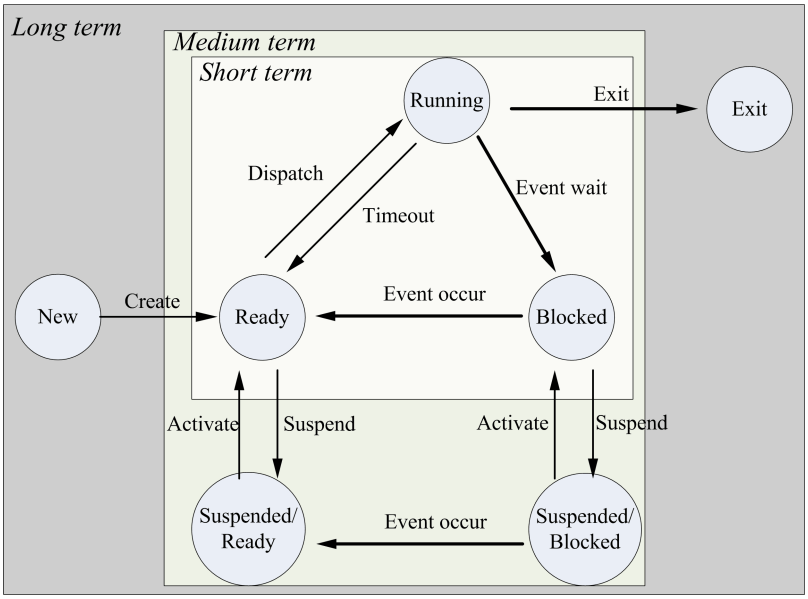


Fig. 3.5: Process states

The states that the processes can be in can be seen in Figure 3.5. We use the more common one, a seven state process model [18] to explain the example effects of scheduling on the ability of being able to re-running a trace.

A process is decided by the long-term scheduler if it is added to the system for processing. If admitted it is put into the ready queue and its control handed over to the short-term scheduler. The short-term scheduler decides – according to performance criteria [1] – to either: admit it to the CPU, continue to keep it in ready state or moves it to the medium-term scheduler. The short time scheduler is invoked because of pre-emption to favour another process or when an event (example: clock interrupts, I/O interrupts, operating system calls, signals [18]) occurs that leads to suspension of the current process.

In many systems each process is assigned a priority and the short-term scheduler always chooses a process with the highest priority. If two processes have equal priority, one is chosen in accordance to scheduling policy (First come first served, Round robin, etc. [1]). In Linux/Unix the priority is dynamic and depends on the CPU Utilization history.

In a view of process scheduling, we need to make sure that T-UppAal has access to the CPU at the time it is either delivering or receiving an action.

Failure to do so will cause a delay of the same length as it took other preceding processes to finish and to do the context switch. If the only process running is T-UppAal running or T-UppAal is having higher priority than other processes, then the possibility of this delay occurring is lower. On the other hand it is impossible to have complete control over non real-time operating systems. The system can initiate I/O or use the file system. There are also parts in UppAal that require the use of the file system and the need to do I/O. The I/O operations are usually done by kernel code. In most non real-time systems kernel code is not pre-emptive thus T-UppAal will not run (even with highest priority) until those operations finish execution.

For some operating system the CPU is given a fixed time (quantum) to work on a process. After each quantum the short-term scheduler checks for processes to be scheduled. If the need to switch a task arises anywhere in a quantum time frame the actual task switch would happen only at the end of the current quantum.

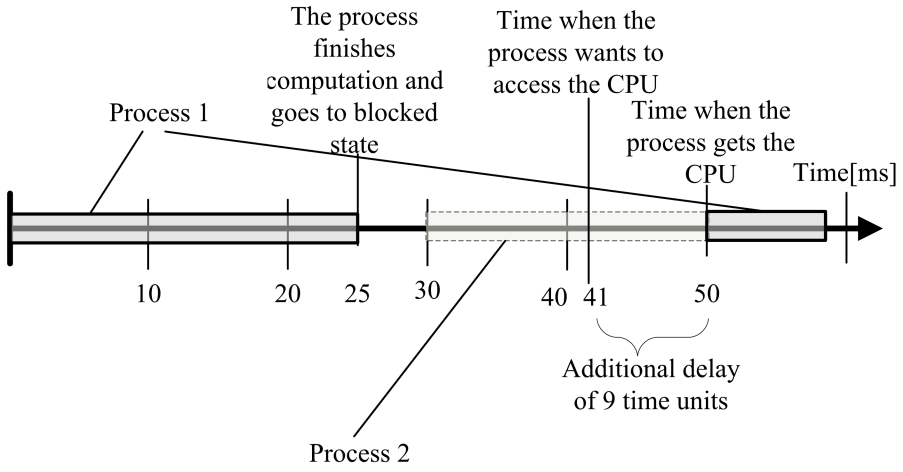


Fig. 3.6: Origin of a delay example

The example showing how quantum concept can create additional delay is shown in Figure 3.6. We assume that the quantum is 10 ms. Lets say that process (e.g. T-UppAal) needs 25ms of CPU time to finish its computation. After that it goes into blocked state. We assume that process wants to wake up 16ms later. Even though no process is occupying the CPU the scheduler does not check for process to run until the end of quantum (at 40ms). Therefore we can get a delay as big as the quantum time gap plus the dispatcher latency (the time it takes to stop one process and to start another).

A different situation would be in Real Time Operating System (RTOS). In Figure 3.7 we can see an example of priority-based pre-emptive scheduling used in most RTOSes. A task with higher priority will be run first. If low a priority

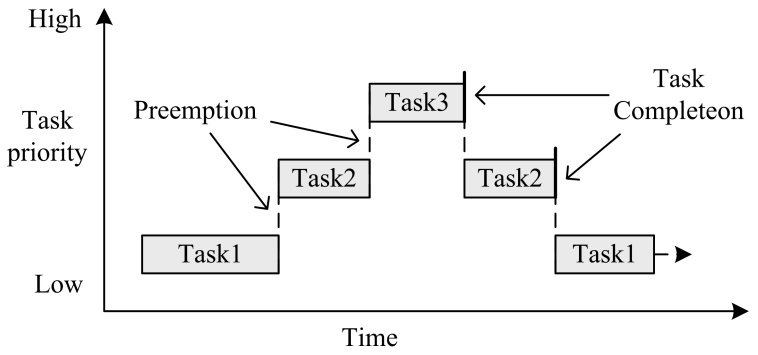


Fig. 3.7: Scheduling in Real Time OS [13]

task already started to run (task 2 in Figure 3.7) and a high priority task became ready (Task 3), the scheduler will immediately stop execution of the low priority task allowing the higher priority task (task 3) to run. After the higher priority task finished its work the lower priority task can continue executing. At the time of writing this report, the version(s) of T-UppAal released so far did not support RTOS.

Timing of actions

Because of the system delays, we cannot choose the exact moment in time when a *action* should occur. We therefore propose an interval in time (*tolerance*) on when actions should occur. This is both needed for input actions and output actions. The input tolerance is because of the delay in T-UppAal and the output tolerance for the IUT delay.

Validity

By having a tolerance on when action should occur introduces another problem, *validity*. That is when constructing the TA re-run trace, there is a possibility of that the new TA that replaces the *environment*will allow actions to happen that were not allowed according to the original *environment*.

In Figure 3.8 we can see the example of this. There is the original *environment*, TA trace, and the trace log (driver log), which we want to make a re-run from. The TA trace was constructed with tolerance of ± 3 ms on the input actions and ± 5 on the output actions. The re-run TA allows action happen within time period [695, 701] (698 ± 3). If T-UppAal offers action between 695 and 700 we will have no problems. But if the time is equal to 701 we could generate the a! action at a time that was not possible according to the original *environment*.

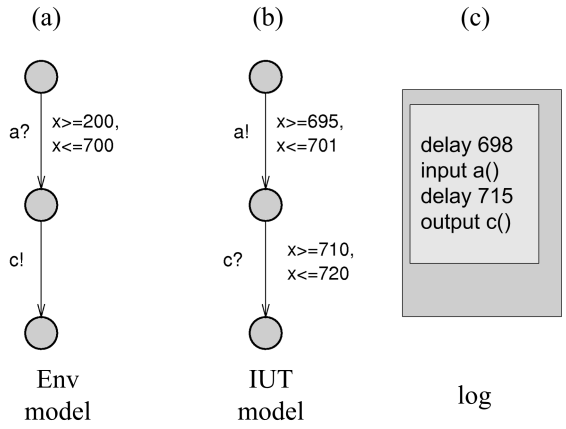


Fig. 3.8: Example problem of validity after having generated TA generated from the trace log to the right

3.3.2 Effect of Non-determinism

We identify two sources of non-determinism in the TA: 1) in the possible of states after an action (Action non-determinism), and 2) in timing of actions(Time non-determinism).

Action non-determinism

In Figure 3.9 (a) an example of a non-deterministic specification is shown. There are two transitions with the same action a going from the initial location. The problem is that during a test re-run any transition can be traversed by the IUT and we cannot control the IUT in which one it should traverse. The path chosen could differ from those we wanted to according to the re-run. A variation of this problem is when time determines which branch of the specification can be traversed and both time intervals have common time. We must therefore accept that we will not be able (or at least difficult), to make re-runs for IUT with high degree of action non-determinism.

Time non-determinism

We also have non-determinism because of timing of actions as allowed by location invariants and guards(tolerance). We can see an example of non-determinism because of timing in Figure 3.9(b). According the specification the a action can happen in time interval [3, 5]. In the test re-run the action a can be sent on time 3, 4 or 5 so we cannot be sure which time the IUT will choose to produce the output.

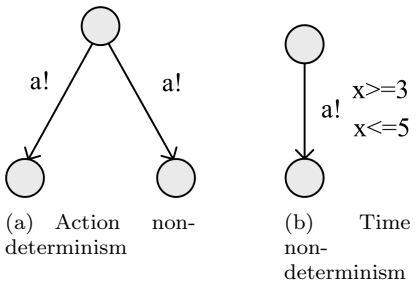


Fig. 3.9: Non-determinism examples

3.3.3 Global and Relative Time

When generating the TA from the trace log, we have two main ways on how the flow of time can be represented. We can use absolute/global time so that action happen relative to when we started the test. Or we can use local time so the action occurs relative to the time of previous action performed. In Figure 3.10 an example of the two approaches can be seen.

Both approaches have their pros and cons. The features of using a global-time are:

- Actions in the re-run happens at the same time as during the original run
- The relative time between any two actions can change only by amount allowed by tolerances of both actions.

We would therefore rather use global time when we are having an IUT where timing of actions is dependent on other actions. For example: a 5 min after action α action β should occur(where number actions between those action can occur).

The attributes of local time are:

- If the IUT consistently produces actions too late or too early the error does not accumulate itself as we measure relative to the last action.
- The relative time between consecutive actions can change only by amount allowed by tolerance of the latter action.

When timing of actions in the IUT is only dependent on when previous action occur, local time should be chosen.

3.3.4 Narrowing of Gap

A problem we call the *narrowing of gap*, is related to timing of actions as described in the previous section about global and relative time. The environment

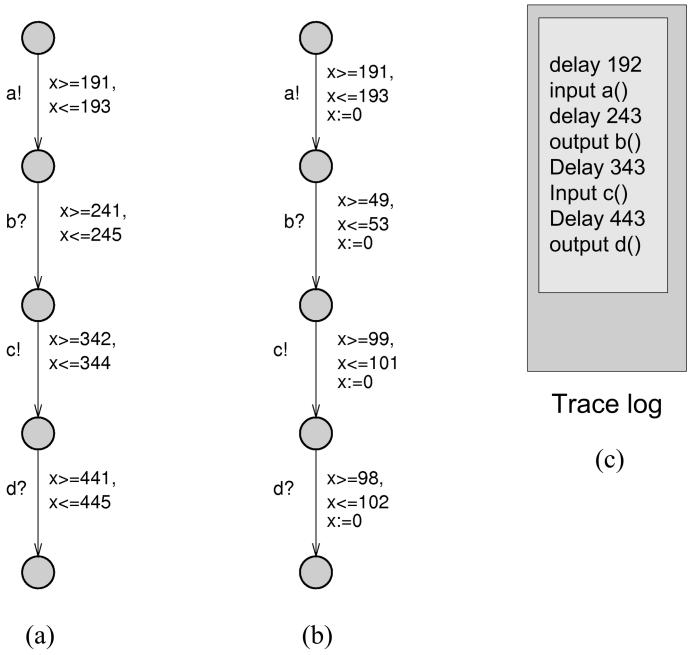
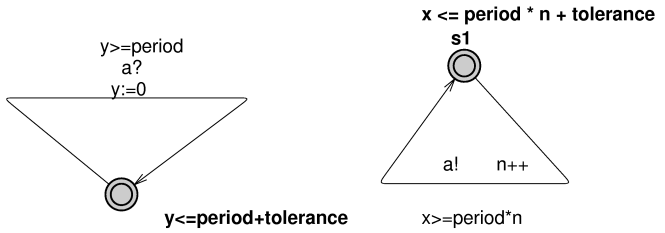


Fig. 3.10: Example of global (a) and local (b) time generated from a trace log (c) to the right

and the specification of the IUT constraints are conjuncted in T-UppAal. Because of this conjunction, a different timing of actions in the environment and the model can lead to that the tolerance (gap), in which the action is allowed to happen, to become to small for T-UppAal to handle.



(a) Specification (b) Environment

Fig. 3.11: Model that causes the gap to narrow

An example model that suffers from this kind of problem can be seen in Figure 3.11. The model consists of two simple templates. The template in Figure Figure 3.11(a) consumes an action a periodically, while the template in Figure 3.11 (b) sends the same action a periodically. We use *relative time* for the specification and *global time* for the environment. The specification consumes an action between *period* and *period + tolerance*. The other one (the Environment), sends an action at the time between *period*n* and *period*n+tolerance*. As the time progress and T-UppAal chooses action a to happen at a specific time. Because of the separation of specification and the environment, the allowed interval for following actions a becomes smaller.

A time-line of a run from the above example with a period of 20 and tolerance of 10 can be seen in Figure 3.12. We start where T-UppAal can perform the action between 20 and 30 model time units. In the next cycle, the intersection of the environment and the specification has narrowed to 42 and 50 time units, which T-UppAal can choose from. This continues until the gap becomes the size of 1. This could lead to T-UppAal being unable to perform the action in time because of real time constraints(which will result in a *inconclusive* verdict).

Narrowing gap re-run example

We show another example to demonstrate how this could occur when we are having a re-run. In this example we have the specification input enabled (as it should be according to our assumption of Timed I/O specification in Chapter 2). In Figure 3.13 we have an in (a) an environment that can consume an a action at any time, and it sends b action periodically between 5 and 7 model time units after each send. In (b) we have specification that sends a action periodically between 4 to 8 model time units one after other. The specification is also input enabled and can receive b at any time.

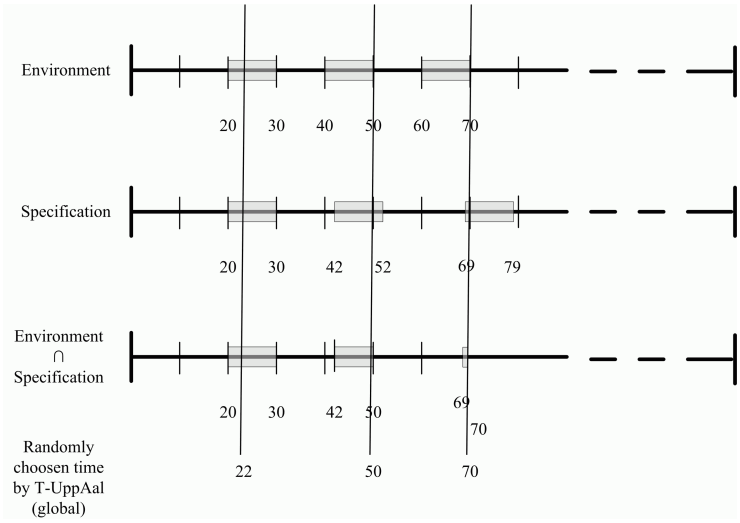
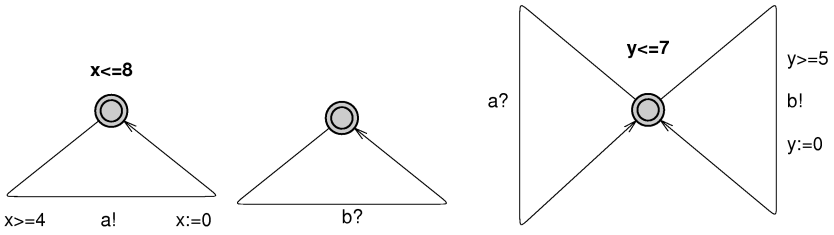
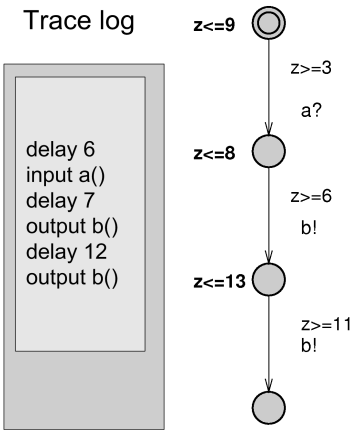


Fig. 3.12: Example of a *narrowing gap* using the model in Figure 3.11



(a) Specification (b) Environment

Fig. 3.13: An example environment and a specification model



(a) Trace (b) TA Environment

Fig. 3.14: The TA environment generated from a trace of an example run from the model in Figure 3.13

In Figure 3.14(a) we can see an example trace generated using the specification from Figure 3.13. In Figure 3.14(b) we see TA generated from the trace in Figure 3.14(a). The input actions have *tolerance* of $\pm$ 1 time units and the *tolerance* on output actions of $\pm$ 3. Then we use it and replace environment in Figure 3.13 according to our idea of making re-runs.

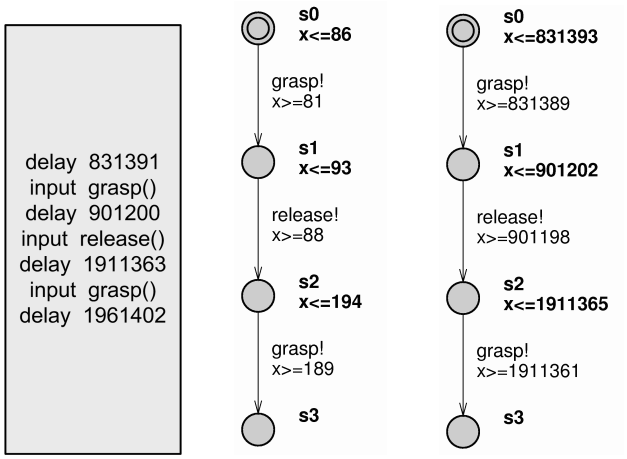
At the first step, the intersection between the environment and the specification gives us that we can receive the a action at an interval from 4 to 8 model time units. Let us assume we receive it at 8. But now we can send the b action only at 8 as the allowed interval for action b is from 6 to 8. We get a gap of 0 and that can be impossible for T-UppAal to handle.

3.3.5 Conversion of Time Units

As noted in Section 3.2, when we create the TA trace for re-run, we use the *trace log* and the specification used for that trace. The level of precision in the model time units in the specification corresponds to that what is needed to accurately model the IUT and the environment model(at least according to the author of the specification).

After a run we get a very precise trace (in microseconds) of when actions happened. When generating the TA from the trace log, one idea would be to convert microseconds in the trace log to the model time units. This approach has some shortcomings. Because of rounding, some information will be lost. Because of that the interval cannot be smaller than one model time unit. This leads potentially to *validity* problems. It may be the case, that T-UppAal is able to deliver action at much higher precision than one model time unit. In that

case a better approach could be to generate the TA according to the precision in the trace log and then modify the specification so it uses the same time units as the TA.



(a) Driver log file

(b) Converted with precision 10000

(c) Converted with precision 1

Fig. 3.15: Conversion of time units

Figure 3.15 shows TA generated using both of the approaches. First the driver log file shown in Figure 3.15(a) is converted using precision 10 000 and the tolerance on the input ± 2. The result is in Figure 3.15(b). In Figure 3.15(c) we can see the same driver log file converted using the precision of 1 and same tolerance.

For specification with high model time units, for example hours, having a precision in microseconds is excessive of what is needed. Thereby the best option could be, to allow the tester to specify the precision in the new model.

4. THE COVERAGE MEASUREMENTS

As test-runs can possibly take hours to days, the log generated by T-UppAal can become quite large. In order to better examine what part of the IUT were tested and to what extend they were tested a coverage measure can be useful. We created the tool that can facilitate the calculation of the coverage of the specification. We believe that with the use of our tool the analysis of the test run is easier and gives more understanding what was happening during the test-runs.

4.1 Idea of Implementation

Our proposal is to create a tool that can facilitate the calculation of the coverage of the model. We believe that with the use of the tool the analysis of the test run is easier and gives more understanding what was happening during the test-runs.

The idea how to measure the coverage is to include in the specification a set of auxiliary variables that will be used as counters. Each time an edge leading to a location is visited the corresponding counter for that location is incremented by one. Similarly we can calculate also how many times the edges have been visited by including the variables for transitions. We need to modify the specification to do this. As explained in section 2.6, the model consists of two parts: the environment and the model of the IUT. We replace the environment part with TA generated from the trace log. The model of the IUT is extended with one with auxiliary variables. Having that we can ask UppAal (off-line) for some path that will lead to the last state of the TA generated from trace log. By checking the values at that stage we can calculate the coverage and check what parts of the specification were visited during test re-run.

In Figure 4.1 we see an example taken from the light controller, showing specification with added variables. It shows the specification of the "interface". The "interface" generates nothing if we press it for a short time, action *touch* if we hold it for longer time and a pair of actions *starthold* and *endhold* when we hold it for a long time. In Figure 4.1 we see added variables for location and transition coverage. Those variables are in slanted font. The variables starting with *cLoc* are used to count when a location is covered and they are added to all incoming transitions. The variables starting with *cTrans* are used to count the transitions. For example *cLocIntFace[0]* corresponds to the initial state *S1* and *cTransIntface[3]* to transition from state *S4* to *S1*. The variable of transition

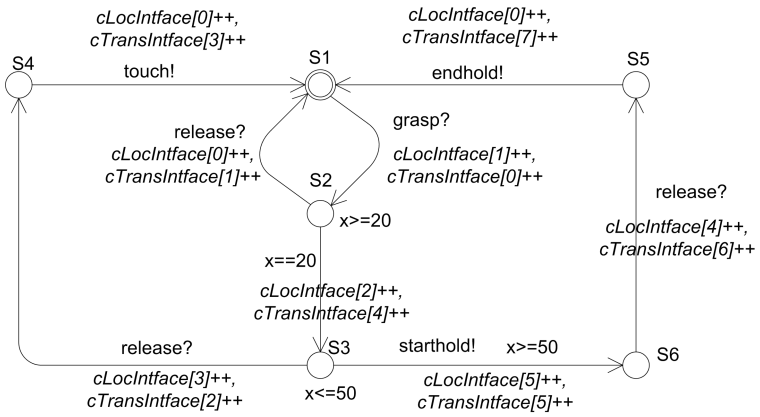
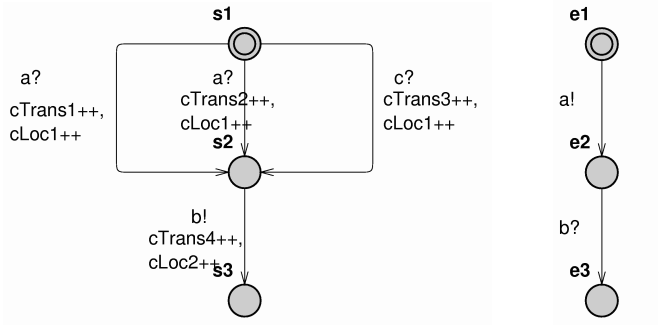


Fig. 4.1: TA of an Interface from a light controller specification where variables for coverage measurement have been added.

coverage is incremented every time the transition is traversed. The location coverage variable is increased by one every time the transition leading to that location is passed.

4.2 Coverage and Non-determinism

Sometimes it can be the case that there exists more than one path leading to the last location TA from trace log. This can only happen if the specification is non-deterministic. The problem is illustrated in Figure 4.2 (a).



(a) Specification

(b) Environment

Fig. 4.2: Example of non-deterministic specification

We can see that in the specification there are two transitions going from the

initial location $s1$ that accepts action a and one transition that accepts action c. They all leads to the location $s2$. From $s2$ to $s3$ there is transition producing output b. The specification is not restricted by any guards. $cTran$ variables are added on every transition and $cLoc$ variables are added to every transition that leads to the corresponding location.In Figure 4.2 (b) there is a test run TA trace example. The environment is sending action a and then accepting action b.

We can ask UppAal to find some path to the last location of the TA trace and examine values of variables in the last state. The problem is that there are three paths leading to the state. Therefore for some edges we will be not able to check if they were covered. We can distinguish three cases:

1. transition was ***possibly*** traversed,
2. transitions was ***certainly not*** traversed,
3. transition was ***certainly*** traversed

Using query asking about the path to the last state we get information only that some transition/location could be possible covered. The solution could be to ask query about specific transition/location. Using UppAal reachability properties we can ask if there exists the path starting at the initial state such that property φ is eventually satisfied along that path Asking two queries is enough to classify a transition/location into one of enumerated categories. If we assume that the last location's name in the trace is *final* and that the name of transition of our interest is *cTrans[x]* then the queries could look as follows:

1. $E <> Environment.final \ and \ cTran[x] > 0$
2. $E <> Environment.final \ and \ cTran[x] == 0$

Depending on the results of those queries we can classify transition as follows:

1. If both results of the queries are true then the transition was ***possibly traversed***.
2. If the first query is false and the second true then the transition was ***certainly not*** traversed.
3. If the first query is true and the second false then the transition was ***certainly traversed***.

When analysing whether a property was certainly satisfied, the combination of the queries can be changed to the one equivalent query: $A <> Environment.final \rightsquigarrow cTran[x] == 0$

In the example (figure 4.2) the following queries will be satisfied:

- Transition was ***possibly*** traversed:

  $E <> Environment.e3 \ and \ cTrans1 > 0$

- Transition was ***certainly not*** traversed:
  $E <> Environment.e3$ *and* $cTran3 == 0$
- Transition was ***certainly*** traversed:
  $A <> Environment.e3 \rightsquigarrow cTrans4 == 1$

5. BUTLER

In this chapter we present the tool we created: *Butler*. First we specify requirements for the tool we intended to create. Next we give some details about how the tool was implemented. In the end of this chapter we present the available options for the tool and example of usage.

5.1 The Tool Requirements

The idea of behind *Butler* is to provide user with a tool that is able to convert the driver log file into Timed Automata trace (as described in Chapter 3). In order to make the tool more usable it should be able to merge the created TA trace in the specification automatically. There are number of possibilities how the TA trace can be created. (see the Section 3.1) It should be possible to specify how the TA trace is generated: the time tolerance for the input and output actions, if we use relative or global time, what is the precision of the model time units. This way we can easily generate TA trace that is best for our needs. Additionally it should be able to add coverage variables as described in Chapter 4. It should run on different platforms. The detailed requirements are listed in the Table 5.1

No.	Description	Priority
1	The tool should be able to generate TA from the trace log and be able to handle action value passing (as parameters)	A
2	The tool should be able to generate TA in XML format.	A
3	The tool should be able to generate TA that can be imported into the UppAal tool.	A
4	The user should be able to specify 1. The name of the log to read 2. The name of the output file that the tool generates 3. The name of the file with the specification (to merge with TA trace or add coverage variables) 4. The name of the configuration file.	A
5	The user should be able to specify if the time is global or relative (relative time - clock value on the nodes in the TA should reset after each input/output action.)	A
6	The user should be able to specify the precision that the model time unit will have (specify how many microseconds one model time unit lasts)	A
7	The user should be able to specify: 1. *time tolerance* (Error bounds)that should be placed on the clock value for the input action in the TA. 2. *time tolerance* (Error bounds) that should be placed on the clock value for the output action in the TA.	A
8	The user should be able to get help by typing "-h" to see what option the tool provides.	A
9	The tool should be able to run on the following platforms: 1. Linux on Intel 2. MS Windows 3. Sun Solaris	1-A, 2-B, 3-B

No.	Description	Priority
10	The tool should be able to generate the TA in other than XML format.	B
11	User should be able to choose if he wants auxiliary variables on transitions that are initiated to 0 and increment by one when: 1. a transition is traversed (transition coverage) 2. a transition leading to a location is visited (location coverage)	A
12	User should be able to choose to have the tool generate a query. The query is of the form: $E <> trace.sX$ where *trace* is the name TA trace and sX is the last location in that trace. This query can be used by UppAal to check if there is a path to the last state in the TA generated from the trace log. This query should be saved in a file specified as a input parameter by the user.	A
13	A T-UppAal specification outputted by the tool should be formatted, in such a way that a new line character is inserted in certain places to make the XML format more readable.	B
14	The tool should read from configuration file all the options that are available by specifying a command line option (except configuration file name). In addition user should be able to change the default name of the template TA trace.	A

Tab. 5.1: Butler requirements continued

5.2 The Tool Implementation

In this section we describe some interesting details and some problems regarding implementation of the tool. The tool we have created is written using C++. To facilitate parsing XML files we used Xerces C++ Parser [25]. The tool supports value passing using parameterised input/output actions.

In order be compatible with the UppAal we used the UppAal Timed Automata Parser Library (*libutap*) to parse the specification. We extended this library with the possibility of writing the specification. Both XML and XTA format are supported.

We created also the library for reading definition of input and output actions. So far those definitions were hard coded into the source of the adapter. Every change required recompilation of the adapter.

The problems with implementation can be split into several sub problems:

- reading and representing the options
- parsing and representing a specification
- parsing and representing a log file
- integrating the trace log into the specification
- adding coverage variables
- printing output to the file

We describe each of these problems separately in the following sections. The section 5.2.9 describes the current status of the tool and some features that were designed but not implemented.

5.2.1 Reading and Representing the Options

The program as it is has quite many options. We need easy access to those options from almost any class in the project. Because of that and to make adding new options easier we created a separate class just for storing option for the program. Each option has dedicated variable in the class. These variables are initialized when the options are read first from the file and then from the command line. The only exception is the option to specify the name of the file with the options, which from obvious reasons must be read from the command line first. The format of the file where the options are stored is in the XML format, to make changes to that file possible using any text editor or even better – an XML dedicated editor.

5.2.2 Parsing and Representing a Specification

In order to stay compliant to the format of the specification that UppAal uses, we decided to use the UppAal Timed Automata Parser Library (*libutap*). Butler is able to parse both formats supported by UppAal: Extensible Markup Language

(XML) and XTA. If the specification format changes, it will be easier to update Butler. The price for that was that we needed to use the data structures used in *libutap*. The *libutap* checks specification for syntactic and semantic mistakes and is quite complicated. The representation of the specification is also quite complicated but it allows doing many complex operations. What *libutap* library was missing was writing the specification. In UppAal all changes are done by hand or by GUI part of the program. The *libutap* is used only to check the correctness of the specification. We extended *libutap* with the possibility of writing the structure. The supported format of the output is XML and XTA.

5.2.3 Parsing and Representing a Log File

We assumed that the log file for the test run could be quite large since test may be running for quite long time. Because of that we did not represented the trace automaton using the same structures as for representing the specification. Instead we used more compact representation. We store all of the labels appearing in the log file in a separate table. Next for each two rows of the log file we store only two integers: the length of the delay and the pointer to the label. This way long names for the input/output action will not cause program to increase its memory requirements very much.

5.2.4 Library for Parsing input/output Actions

In order to decide which part of the specification is the environment, we need explicit definition of the actions that are input and output. Those are not defined in the file containing the specification. So far it was the responsibility of the adapter to tell T-UppAal which actions are outputs and which inputs. In the same way the precision of model time unit and timeout for testing was defined. Our solution is to put those definitions in a separate file. We wrote a library that parses this file and stores the results in a class that can be easily used in the adapter. This way there is no need to recompile the adapter if we want to change for example the timeout for testing. The explicit definition of input/output actions is also necessary for proper interpretation of the log file created by the driver.

Structure of the file

The file containing definition of input/output actions can contain following lines:

- line containing definition of input actions. The line starts with *input* keyword followed by comma separated list of names actions. The line should end with semicolon. The list of parameters for the actions is inside pair of brackets that follows the action name. This list is a comma separated sequence of parameters names.
- line containing definition of output actions. The line starts with *output* keyword and, in similar way as line defining input actions, is followed by

comma separated list of actions names. The list of parameters is inside a pair of brackets that follows action name. The line ends with semicolon.

- line containing definition of the *timeout* for the test. The timeout is assumed to be in the Model Time Units. It defines after what time the test will finish with success. The line starts with *timeout* keyword followed by number (the value of *timeout*). The line ends with semicolon.
- line containing definition of the *precision*. The precision tells how many microsecond is one Model Time Unit. The line starts with *precision* keyword followed by number (value of *precision*). The line ends with semicolon.

The example of the file can be seen in Figure 5.1(a).

5.2.5 Integrating the Trace Log into the Specification

We tried to make the integration of the trace log into the specification in a way that would require no human intervention. In order to do this it is not enough to read the specification and trace log from the files. We need to determine which part of the specification is the environment. After that we substitute only the environment part with the newly created trace log. The problem increases by the possibility that the specifications can be in the form of templates. A template is a specification of one timed automaton but with parameters. This way creating similar Timed Automata is simplified greatly. This caused detection of between environment and IUT more difficult for butler. In the extreme case it may happen that one template is instantiated once as timed automaton specifying environment, and the other time to represent IUT. Butler detects such situations and handles them by assuming, that one instantiation cannot be used at the same time to represent the environment and the specification. We use the following strategy:

1. We read the definition of input/output from a separate file (see section 5.2.4).
2. For each process we check if it produces input or output labels. To do this first we must check labels in the original template and check if the labels not passed as parameters. Then we classify process as environment or as specification (or left undefined)
3. In the last step we change the system definition to contain only instantiations that represent specification or are undefined. Next we add the trace log to the system definition.

Using this strategy it may happen that we do not delete all parts of the environment. This will be the case if there is an instantiation of a template being part of the environment that does not produce any actions defined in the separate file. The *lonely* process should be not the problem, because it will not produce any actions defined as input/output actions. Those actions should be the only

means for communication between environment and IUT. The rare case where manual deletion of a template could be required is, for example, when there is some invariant on initial location in this template.

5.2.6 Adding Coverage Variables

In our tool we have the possibility to add auxiliary variables to measure coverage. The measure can be for locations or for edges. The idea is quite similar for both. The difference is that for edge coverage we have one variable for each transition and for location coverage we have one variable for each location. The algorithm for adding coverage variables is as follows:

1. We classify all instantiations as described in the section 5.2.5
2. We declare all necessary variables in the global declaration section. We add one variable for each location/transition for each instantiation that is not classified as environment. We declare them as arrays of integers– one array for each instantiation. We include instantiation name in the name those arrays.
3. We modify the definition of each template that is used to create instantiation that is not classified as environment. The modified template increases the appropriate variables at each transition (as described in the section 4.1).
4. We modify the declarations of the templates and the instantiations to pass the needed array variables.

The example of the specification with added coverage variables can be seen in Figure 4.1. The slanted texts in Figure are the variables added by Butler. The variables starting with *cLoc* are used to measure location coverage and variables starting with *cTrans* are used to measure the transition coverage.

5.2.7 Scaling Specification Precision

Our tool has possibility of scaling the default precision of the specification. It means that we have the possibility to change the meaning of model time units. For example if one model time unit is 100 ms we can change it to be 10 ms. We do not allow scaling the precision the other way round - it is not possible to change precision from 10 ms to 100 ms. The algorithm looks as follows:

1. First we get the number by which we need to scaling the precision. It is the ratio: $r = \lfloor \frac{old\ precision}{new\ precision} \rfloor$ For example when changing from 100ms to 10ms then $r = 10$. As UppAal does not support the rational numbers as update for clock the r number must be a natural number and we can do only scaling.
2. For every comparison or assignment to a clock variable in every guard or invariant:

- If the right side does not contain clocks then multiply everything by r.
- Otherwise split the right side into parts until we get a single clock or part without clock. Multiply the parts without clocks by r.

For example let us assume that we want to change the guard:

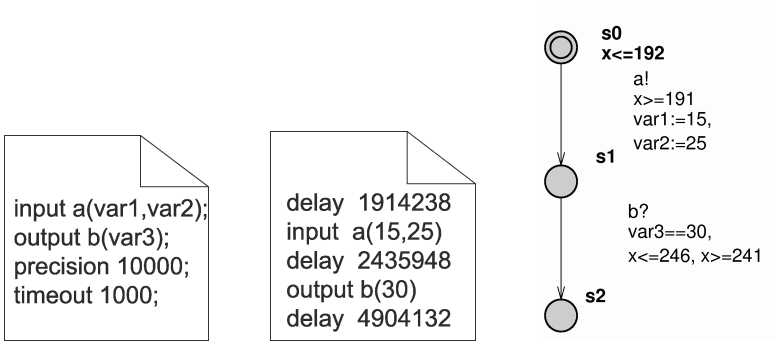
$$a >= 30 + delay \;\&\&\; t >= delay + x$$

We assume that a is an integer, $delay$ is constant and t and x are clocks. The guard after parsing by our tool and changing the precision ($r = 10$) would look like:

$$a >= 30 + delay \;\&\&\; t >= 10 * delay + x$$

5.2.8 Actions with Parameters

Recently the actions that are defined as input or output were extended with a possibility of having parameters. It means that when we send or receive an action from the IUT, we can have parameter for that action. The actions and the parameters are defined in a separate file (see section 5.2.4)



(a) Definition of input output actions

(b) Driver log file

(c) Timed Automata trace

Fig. 5.1: Definition of input/output actions, driver log and resulting TA trace

An example of such definition can be seen in Figure 5.1(a). As input we have one action a with two parameters *var1* and *var2*. The b action is the only output and it has one parameter *var3*. The precision is 10 000 (one model time unit is 10 000 microseconds) and timeout for testing is 1000 model time units. The example driver log file with a and b actions defined in above way can be seen in Figure 5.1(b). First we delay for 1914ms then we send input action a with two parameters: *15* and *25*. After delaying more we receive output action b with parameter *30*. T-UppAal does not support the actions with parameters directly. We need to make use of guards and updates to compensate for that.

The process of generating the TA trace taking parameter of actions into account looks as follows:

1. For every pair of delay and input/output interaction create one transition with guards taking into account the delay and action synchronizing over this transition.

2. For every *input* action and for every parameter for that action take the corresponding value from the driver log file and *assign* it to the variable defined in the definition of this input action

3. For every *output* action and for every parameter for that action take the corresponding value from the driver log file and *compare* it to the variable defined in the definition of this output action.

In Figure 5.1(c) we see the resulting TA trace created from the driver log from Figure 5.1(b) and actions defined as in Figure 5.1(a). For input action a we assign values 15 and 25 from the driver log to the correspondingly to the variables *var1* and *var2*. For the output action b we check in the guard if it is equal to the value of the variable *var3*.

5.2.9 The Tool Status

All of the requirements with priority of A were implemented and works (see the Table 5.1). From the B priority requirements writing in XTA format is not functioning (specifically there are some errors when generating trace in XTA format). Workaround could be to run Butler two times: first generate TA trace in XML format and then use Butler to convert it to XTA format.

Butler does not compile under Solaris platform. Under Windows operating system it is possible to compile it using Cygwin environment. The compiled version for Windows was not tested as thoroughly as the version compiled for Linux.

We tried to make a documentation of code at the same time as implementation was being done, but it was only partially done. Butler does support generating the queries needed to do the detailed coverage measurement as described in section 4.2.

Butler source code consists of total 9168 lines. They can be divided into following parts:

- The extension of UppAal Timed Automata Parser Library (*libutap*). We added support for writing of the structures to the file in XML or XTA format. This code is new compared to version from the first semester – 1919 lines – 10 files

- The library for parsing the file with input/output actions. (see the Section 5.2.4) – 3 files were a source for tools that generate code automatically (GNU Bison, GNU Flex, GNU Gperf). From 264 lines of code 4674 lines of C++ code were generated. New code in this semester.

- All other files. This includes files for manipulation on the data structures from *libutap* and converting the driver log file into TA trace. This *main* part of Butler. It was changed heavily from the first semester to support value passing and several other extensions. It consists of 16 files containing 2035 lines of code.

5.3 The Tool Options

The options for Butler are as follow:

- **-h, –help** - shows the help for the program
- **-O file name** - tells the program to read options from the specified file. The command line options override the options read from the file. By default no option file is specified.
- **-d file name** - this option allows user to specify the file where the trace log is stored. If this option is not specified no trace log is merged with specification.
- **-s file name** - tells the program to read the specification from the file. When no parameter is specified no specification is merged with TA trace.
- **-o file name** - this option specify in what file should the output be stored. When no parameter is specified then the standard output is used.
- **-c file name** - name of the file from which input/output channel definitions will be read. This file must be specified if trace log file is processed.
- **-q file name**- this option specify file into which queries should be written to. The queries include the query to the last location of the TA trace. If the coverage variables are added then the file will contain also the queries to do the detailed coverage measurements. If file is not specified queries are not generated.
- **-f [xml—xta]** - this option allows to specify format in which output is produced. By default the format is the same as the format of the specification. There exists no option to specify the format of the specification. Butler detects the format by analysing the content of the file.
- **-u number** - allows setting up the time tolerance for the output actions. This way we can allow SUT to send the output within some error bounds compared to the previous run.
- **-n number** - allows feeding the SUT with input within some bounds. This option should be used with care, because providing input at different time may cause the SUT to interpret the same input with differently.

- **-p number** - this option allows to specify how many microseconds is one model time unit. If this number is different from the one specified in the file with definition of input/output actions then all clocks assignments/comparisons are modified as described in the section 5.2.7
- **-l** - include location coverage in the specification. This option causes our program to add coverage variables and updates to measure how many times each location has been visited.
- **-t** - include transition coverage (similar to *-l* option - location coverage)
- **-e** - generate the trace that allows any *output* action from IUT.
- **-g** - tells the program to generate TA where clock is not reset after each transition (*global time*). By default the *relative time* is used.
- **-i** - by default the upper limit for output action is specified as a guard for transition. If output does not arrive in time we are allowed to stay in that location. When this option is specified then the upper bound is specified as invariant. If the output does not arrive in time we will be not allowed to stay in that location.

The user can specify in the configuration file all the options that can be specified in the command line. If the option is specified both in command line and in the configuration file, then the command line option will override the option specified in the configuration file. Additionally in the configuration file we can specify the name of the template in which the trace is placed. This can be used to avoid conflicts with names of existing templates.

5.3.1 Usage

In using the tool various combinations of options can be chosen depending on what the user needs. The examples below show the most general cases of usage. In the examples we assume that *driver.log* is the name of the trace log generated by T-UppAal. *output.xml* or *output.xta* is the name of the output file, *specification.xml* is the name of the file with specification we are working with and *definition.act* is the file containing the definitions of input and output actions.

Example 1 - TA trace generation

The simplest example of usage would be to create the TA trace from the log file:

```
butler -s driver.log -c definition.act
```

The generated output will be displayed on the screen.

Example 2 - merging TA trace into specification

In this example we construct a TA from a trace and add it to the specification. We have specify the tolerance on input actions to 2 and on output actions to 3:

```
butler -d driver.log -o output.xml -s specification.xml -n 2 -u 3
    -c definition.act
```

The *output.xml* file contains now the specification from specification.xml where the TA generated from the trace log has replaced the environment part. It is ready for opening with UppAal where we can analyse it and simulate the run.

Example 3 - coverage measurement

Here is a command to add the additional coverage variables to the specification:

```
butler -d driver.log -o output.xml -s specification.xml -l -t
    -q query.q -c definition.act
```

The auxiliary variables are added in the generated file *output.xml*. They are used as counters. The user can open *output.xml* in UppAal and simulate the re-run to examine how often a state or transitions have been visited during a certain run. Using the command line based tool of UppAal can also be used to do the measurement. For this purpose the -q option is used and the result is *query.q* file. This file contains the query asking if there exists a path to the last state of the TA trace. Additionally it contains the queries (two queries for each location and transition) to check the detailed coverage of transition and locations as described in Chapter 4

Example 4 - precision changing

Here is an example command to change the default precision of the specification:

```
butler -d driver.log -o output.xml -s specification.xml
    -c  definition.act -p 10
```

We assume that the precision specified in the *definition.act* file is 100. The result will be specification combined with the trace log, with bigger precision. In the new specification one model time unit will be 10 microseconds instead of 100 microseconds as before. All comparisons and assignments with clock variables will be updated accordingly. It is the user responsibility to update the *definition.act* file (or the adapter source code) so during the re-run the new precision is used.

6. EXPERIMENTS

To be able better understand what factors affect possibilities of re-runs, experiments focusing on T-UppAal and the adapter were needed. Our aim is to find the precision of determining when an action should happen. That is, if we want an action to happen at a certain time, what will be the difference between when the actions happens and the time when we wanted this action to happen. This information is critical to check if the test re-run is possible and to what extend. We trace the origin of the *delay* to three sources. *Computation delay*, *communication delay* and the delay connected to the scheduling policy of the system (*scheduling delay*)(Section 3.3.1). When testing an IUT using T-UppAal, all communication goes trough the adapter(we consider the *driver* as a part of the adapter). These layers as described in Section 2.6.2 all add to delay, and thereby lessening our control in timing of events.

To identify these delays we do three experiments. The first one is to give us a broad view of timing in a model with an IUT and T-UppAal and the effect of the delay. By that we get a measure of the total *delay* and the behaviour of the model. In the second experiment, we put our focus on T-UppAal and the effect of the *scheduling delay*. We do this by removing a part of the model, namely the IUT. For the last experiments in this category, we look at different OS. We do this to check what impact OS has scheduling precision and thereby timing of events.

6.1 System Delay Estimate

In this experiment we try to measure the delay we can expect in T-UppAal, the adapter and the IUT, and in the communication between them.

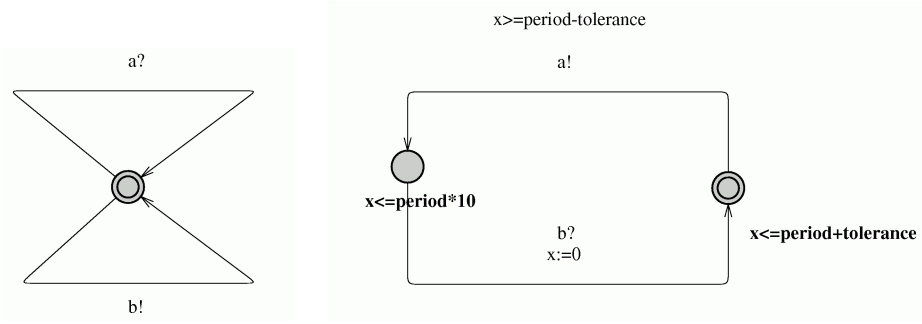
To do this we made a simple Java program (for the IUT) that waits for an input, sleeps a certain amount of time and sends an output back. We try to force T-UppAal to send input periodically by restricting the environment. By measuring the actual time when T-UppAal sends an input and receives an output from the intended one, we will be able to estimate the size of the delay.

6.1.1 Experimental Setup

The experiments were performed on two computers connected directly to each other a via cross cable. This way we ensure that the network delay is minimal and not affected by other network traffic. The following computers were used:

PC1: AMD Athlon 1.8 GHz, 256MB on RAM running Red Hat Linux 2.4.27
PC2: Pentium 1.3 GHz, 512MB of RAM running Red Hat Linux 2.4.27

The model used for experiments is shown in Figure 6.1.



(a) Specification of theIUT (b) Environment

Fig. 6.1: The Model

The specification of the IUT has one location and it accepts an input a and sends an output b at any time (Figure 6.1(a)). This gives our Java program complete freedom when to send the output back. The environment model (Figure 6.1(b)) is more complex. It sends an input a to the IUT in a time interval: [period-tolerance; period+tolerance]. After sending an input it waits for an output for max *period*10* time units. This constraint could have been omitted but we wanted to have time-out when no action is sent from the IUT. The values of *period* and *tolerance* were 500 and 50 ms respectively during all of the experiments.

All communication between T-UppAal and the IUT goes trough the adapter. T-UppAal and the adapter were running on PC1 and the IUT on PC2. The time was measured in both the IUT and the adapter. In Figure 6.2 we see the sequence of actions and moments when the time is measured.

T-UppAal sends an action a to the adapter. The adapter records the time $t_1^{[n]}$ right after it receives the input. Then the action a is sent to the IUT and time $t_2^{[n]}$ is measured. Acknowledgement that input was forwarded to the IUT is sent to the T-UppAal. The time $t_3^{[n]}$ is measured right after this event.

The IUT records time twice: right after receiving the input $t_{IUT1}^{[n]}$ and before sending a b output $t_{IUT2}^{[n]}$. In between these actions the IUT sleeps for the defined period of time t_{sleep}.

The adapter measures time $t_4^{[n]}$ after receiving the output from the IUT and after sending the output to T-UppAal $t_5^{[n]}$.

Then T-UppAal sends the next input a to the adapter at the time $t_1^{[n+1]}$. The process is repeated until a specified number of iterations.

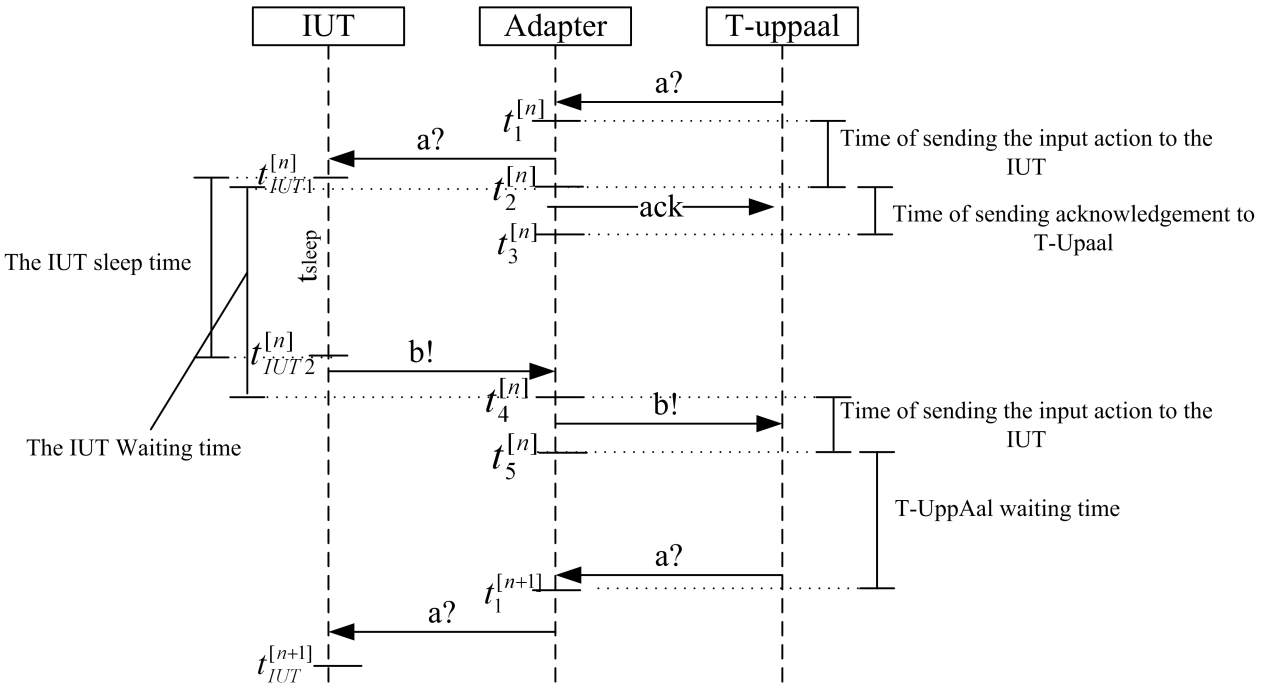


Fig. 6.2: The sequence of actions and the measured time

The pseudo code for the IUT can be seen in algorithm 5. The IUT enters the loop first. In this loop it waits for an input. After receiving input it behaves as denoted in Figure 6.2. When it is not specified otherwise, the intended sleeping time of the IUT is 250ms.

Algorithm 5: IUT

```
1: n := 0
2: while (true) do
3:    getMessage ()
4:    t_IUT1^[n] := getTime()
5:    sleep(wait_time)
6:    t_IUT2^[n] := getTime()
7:    sendMessage(msg)
8:    n++
9: end while
```

The pseudo code of the adapter is shown in the algorithm 6. It consists of two functions *perform()* and *threadExecute()*. T-UppAal calls the function *perform()* to pass *the message* to the adapter. The adapter sends the message received from T-UppAal further to the IUT using *sendMessageToIUT(msg)*. The acknowledgement to T-UppAal is sent with the function *inputDelivered()*. After that the function *perform()* returns.

The receive part of the adapter is running a separate thread, which is supposed to handle messages coming from the IUT. The function *threadExecute()* is a main function for this thread. Inside *threadExecute()* we enter a loop. In this loop we wait for the message from the IUT. The message is sent to T-UppAal by function *tryReport(msg)*. The counter n is incremented after sending one input and receiving one output.

The moments, when values of current time are assigned to variables, are explained in Figure 6.2.

We performed number of experiments using this setup. The summary of the parameters that were used is presented in Table 6.1.

We ran T-UppAal with two options for choosing when to send the input: the *Eager* and *Random* (described in Section 2.6). As these option affect the timing of when actions occur, we wanted to check how T-UppAal works with them. We expect that the *Eager* option is better for the re-run as it gives more control of when events happen.

Additionally we specified different lengths of the observation uncertainties. The numbers used in the Table 6.1 are specified in microseconds. For the *Eager* parameter we vary from 0 to 100 000 microseconds. We do not expect actions to occur at different time. The option causes to accept actions that arrive later than normally. For *Random* parameter we used the same numbers, but for smaller values we were not able to obtain a longer runs.

We decided also to do experiments with the IUT sleeping a random time and with not waiting at all, because we experienced an interesting behaviour of

Algorithm 6: Adapter for experiment with input and output actions

```
1: n := 0
2: perform(msg)
3:     t_1^[n] := getTime()
4:     Mutex_Lock(msg)
5:     sendMessageToIUT(msg)
6:     t_2^[n] := getTime()
7:     inputDelivered()
8:     Mutex_UnLock(msg)
9:     t_3^[n] := getTime()

10: threadExecute()
11: while (!fail) do
12:     waitForMessagefromIUT()
13:         Mutex_Lock(msg)
14:     msg := getMessage()
15:     t_4^[n] := getTime()
16:     tryReport(msg)
17:         Mutex_UnLock(msg)
18:     t_5^[n] := getTime()
19:     n := n + +
20: end while
```

-P	-U	T_{sleep} [ms]
Eager	0, 0	250
Eager	10, 10	250
Eager	100, 100	250
Eager	1000, 1000	250
Eager	10 000, 10 000	250
Eager	100 000, 100 000	250
Random	0,0	250
Random	1000, 1000	250
Random	100 000, 100 000	250
Random	1000, 1000, 1000, 1000	250
Random	100 000, 100 000, 100 000, 100 000	250
Eager	0, 0	no sleep
Eager	0, 0	Random
Eager	10 000, 10 000	no sleep

Tab. 6.1: Overview of experiments

the T-UppAal (explained later in this chapter).

In the description of experiments we are using:

- *T-UppAal waiting time*:
  The time T-UppAal waits before sending an input after the output was received.
  $(t_1^{[n+1]} - t_5^{[n]})$
- *IUT waiting time*:
  The time IUT waits before sending an output after the input was received.
  $(t_4^{[n]} - t_2^{[n]})$

6.1.2 Results

Our goal in this experiment was to find out the *delay* we could expect in T-UppAal, adapter and the IUT. The results for all the runs were quite similar. We present here only chosen results that seemed most interesting.

Waiting Time of the IUT and T-UppAal, and the effect of T-UppAal options

In Figure 6.3 *T-UppAal waiting time* and *IUT waiting time* is shown. X-axis represents number of steps n. Y-axis delineates time in microseconds.

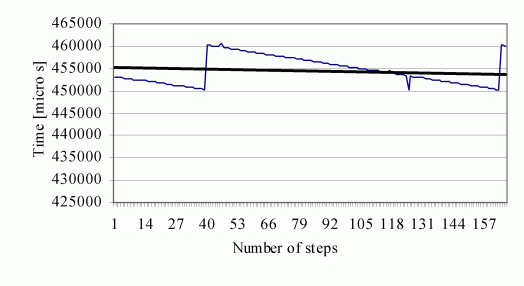
According to the environment specification 6.1 (b), a waiting time after receiving *b actions* should be 500± 50 (period ± tolerance). By using only the *eager* option we expect the optimal case *T-UppAal waiting time* to be 450 ms. We can see in Figure 6.3(a) that the average *T-UppAal waiting time* is around 455 ms when -u option is set to 0,0, and gives us the average delay of 5 ms.

When -u option is 100 000 microseconds (Figure 6.3(b)) the average of T-UppAal waiting time remains 445 ms. This difference is much smaller than specified -u option (100ms) so either the -u option does not have the effect on when T-UppAal sends the message or this effect is minimal.

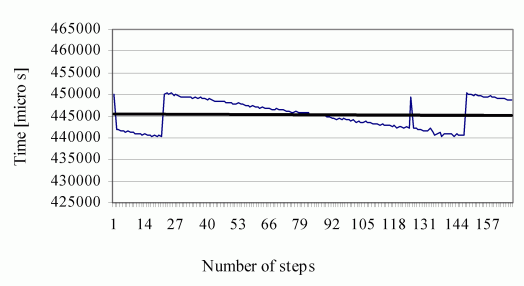
In the IUT the T_{sleep} was set to 250 ms. The IUT behaviour is almost the same in both Figures (6.3(c) and (d)) average waiting time is 255 ms, giving us again *delay* of 5 ms. This is quite reasonable, as the options specified for T-UppAal should not affect the waiting time of the IUT.

Peaks

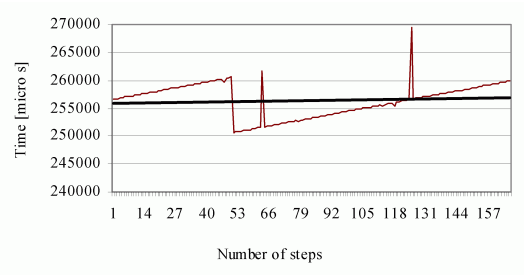
When looking at graphs 6.3 (c) and (d), we can see that we have upwards and downwards peaks. As they happen at the same steps 61 and 125 for both of the runs (with the options -u 0,0 and -u 10.000, 10.000), we speculate that it is T-UppAal or the adapter that initiate some process that takes control of the CPU rather than a random scheduling disturbance. One reason could be that T-UppAal or the adapter needed to perform some I/O or to access the file system.



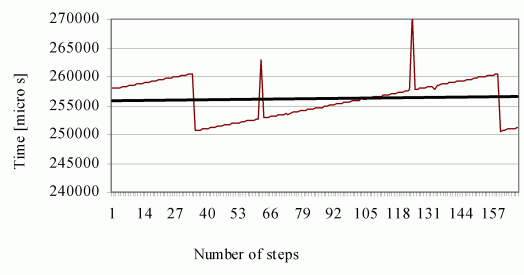
(a) T-UppAal waiting time. -u 0,0



(b) T-UppAal waiting time. -u 100 000,100 000



(c) IUT waiting time. -u 0,0



(d) IUT waiting time. -u 100 000,100

Fig. 6.3: T-UppAal and IUT waiting time. -P eager. OS Linux 2.4

The rest of the runs with *Eager* option looked almost identical to the one showed in Figure 6.3, except it was shifted up to 10 ms.

Network Delay

By having the t_{sleep}, and the time between the adapter sending an a action to the IUT and getting a b action back from the IUT. We can calculate the network delay. The formula becomes $(\frac{t_4^{[n]} - t_1^{[n]} - t_{sleep}}{2})$.

The maximal value for this was 0.5 ms. An important factor in the calculation is that the max precision in Java that is possible when timing t_{sleep} is in milliseconds. Because of that, we know that even tough we got the maximal value of 0.5 ms, the actual value can be much smaller.

Dependency between T-UppAal and the IUT Waiting Time

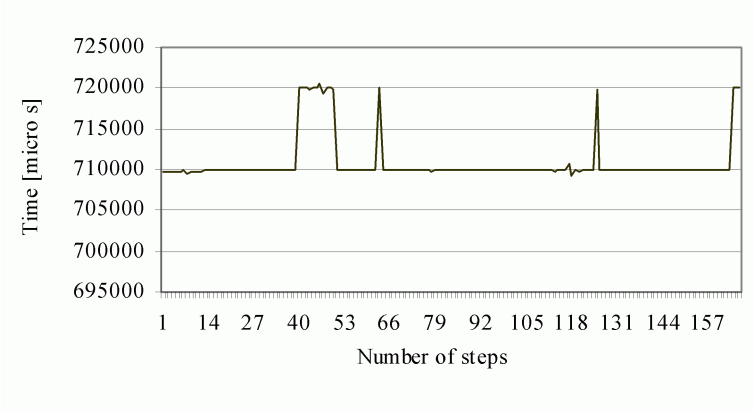


Fig. 6.4: 1st experiment. Sum of T-UppAal and IUT waiting time. Options: -u 0,0; -P eager. OS Linux 2.4

An interesting fact is the zigzag behaviour of the graphs with the *Eager* option. In theory there should be no dependency between the waiting time of T-UppAal and the waiting time of IUT. But obviously, when we add those two graphs, we obtain a straight line with 10ms jumps as seen in Figure 6.4.

We found out that this behaviour is caused by the scheduler of Linux kernel 2.4. It appears that the sleeping process can be woken up only at a certain periodic points in time. Specifically the process can be woken up every ten milliseconds. In Figure 6.5 we see a time line for several runs. Each time the waiting time of the IUT (as in experiments) increases a little. Next we can see the time T-UppAal wants to sleep. At the end there is a dispatch latency.

We assume that the main source of latency is the fact that the scheduler checks if there is a new process to be scheduled every 10ms. So each time the waiting time of the IUT increases, the latency decreases and so does the actual

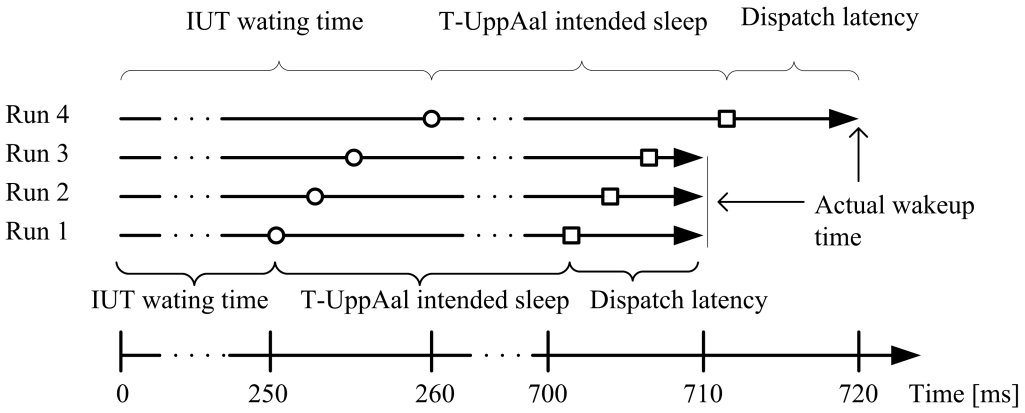


Fig. 6.5: T-UppAal and IUT waiting time

waiting time of T-UppAal. When we cross to certain critical point (between Run 3 and Run 4) we will see 10ms *jumps*. That can explain why we have *jumps* in T-UppAal waiting time.

In Figure 6.4 (b) and (d) we have similar problem. We do not have *jumps* in T-UppAal waiting time when we have *jumps* in IUT waiting time. This can be explained using the same mechanism. If we compare Run 1 and Run 4, we see that IUT waiting time is different by 10ms. At the same time the dispatch latency has not changed, thus that waiting time of T-UppAal does not change significantly also.

6.2 Local Scheduling

In these experiments we wanted to measure the characteristic of T-UppAal when sending an action periodically. By that we will be able to find what the precision to expect from T-UppAal. We send one action periodically to the adapter and do not expected a response from it. We also do not reset the clock after each repetition to avoid accumulation of errors thus reproducing a perfect periodic signal.

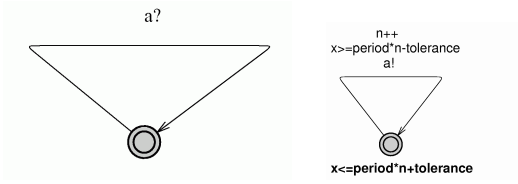
6.2.1 Setup

Experiments were performed on:

Linux computer: AMD Athlon 1.8 GHz, 256MB of RAM running Red Hat Linux with kernel 2.4.27 and Debian Linux with kernel 2.6.11

Solaris computer: Fire v880R, 8x900 MHz CPU, 32 GB RAM Solaris 9 (SPARC) with Sun OS 5.9

The specification of the IUT has one location and it accepts input a at any time but does not send output at all (Figure 6.6 a). The environment model (Figure 6.6 b) has one location and it sends input a to the IUT in a time interval [period*n-tolerance; period*n+tolerance] where n is a counter incremented after every action.



(a) Specification of the IUT (b) Environment

Fig. 6.6: Strict, periodic Model

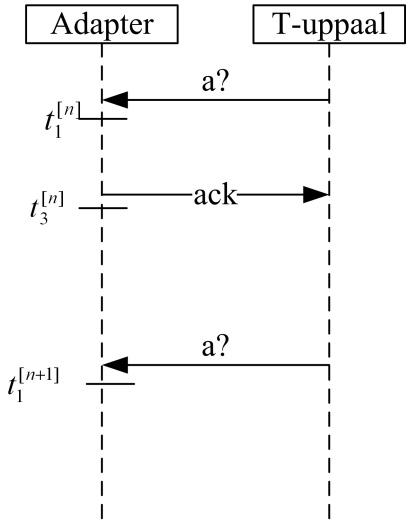


Fig. 6.7: Sequence of actions and measured time

In Figure 6.7 we can see the sequence of actions and moments when the time was measured. The pattern differs from the one explained in 6.2. There are no IUT and output actions. Time $t_2^{[n]}$ is not measured. From previous set of experiments (see Section 6.1) we realized that difference between $t_2^{[n]}$ and $t_3^{[n]}$ is small enough to be ignored.

The adapter for these experiments differs from the previous one. It does not have the function that sends action to IUT and function *threadExecute()* is empty. The pseudo code can be seen in Algorithm 7.

In Table 6.2 we can see the parameters used in the experiments on Linux

Algorithm 7: Adapter for experiment only with input actions

1: $n := 0$
2: **perform(msg)**
3: $\quad t_1^{[n]} := getTime()$
4: $\quad Mutex\_Lock(msg)$
5: $\quad inputDelivered()$
6: $\quad Mutex\_UnLock(msg)$
7: $\quad t_3^{[n]} := getTime()$
8: $n := n++$
9: **threadExecute()**

-P	-U	Operating System
Eager	0, 0	Linux kernel 2.4
Eager	10.000, 10.000	Linux kernel 2.4
Eager	100.000, 100.000	Linux kernel 2.4
Random	15.000, 15.000	Linux kernel 2.4
Random	100.000, 100.000	Linux kernel 2.4
Eager	0, 0	Linux kernel 2.6
Eager	10.000, 10.000	Linux kernel 2.6
Eager	0, 0	Solaris
Eager	10 000, 10 000	Solaris

Tab. 6.2: Overview of parameters for experiments on Linux kernel 2.4, Linux kernel 2.6 and Solaris (right)

kernel 2.4 and 2.6 and Solaris. Additionally we did runs with high scheduling priority on Linux 2.4 and 2.6, to compare if it had any influence. We were not able to perform runs with high priority on Solaris, because we were not able to obtain root privileges needed for that. The experiments run on Solaris were run on a public machine, so the processes run by other users could bias the results.

6.2.2 Results

In Figure 6.8 we see how practical time measured during experiments differs from actual time on different Operating Systems. In the graphs we used deviation from perfect periodic expected output time calculate using formula:
$(t_1^{n+1} - t_1^n) - n * 500.000$, $n = 1, .., N$ where k is the number of steps.

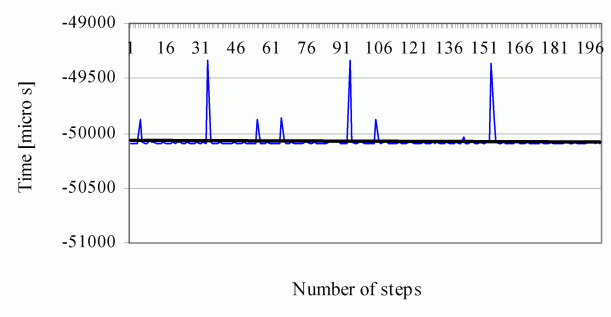
In Figure 6.8 (a) we can see the runs for the Linux with kernel 2.4. The plot average value is almost 50 ms with seldom peaks reaching 49.5 ms. It seems that the precision at which T-UppAal is sending the input is pretty good. There are only 0.5ms disturbances, probably caused by some background process. In Figure 6.8 (c) we can see the same run but for Solaris system. The disturbances on this graph are smaller than for Linux with kernel 2.4. They seem to be smaller than 0.2ms. In Figure 6.8 (b) we can see the results for the last operating system we tried, Linux with kernel 2.6. The disturbances here are around 1ms. From these results it seems that Linux 2.6 is the worst kernel to perform the experiments on. This is not exactly true, which we will show in the next experiments (Section 6.3).

In Figure 6.9 (a) we see the runs with a high priority (-19) on Linux with kernel 2.4. The plot is very similar to the one with normal priority on the same kernel (see Figure 6.8). We can notice that still we have similar peaks, but now they are quite regular and fewer. In Figure 6.9 (b) we can see the runs with a high priority but on Linux with kernel 2.6. This time we do not see any apparent difference between run with normal priority and run with high priority. We can see that high priority does help, but the improvement is minimal. The reason for that could be, that we performed the runs on a dedicated machine.

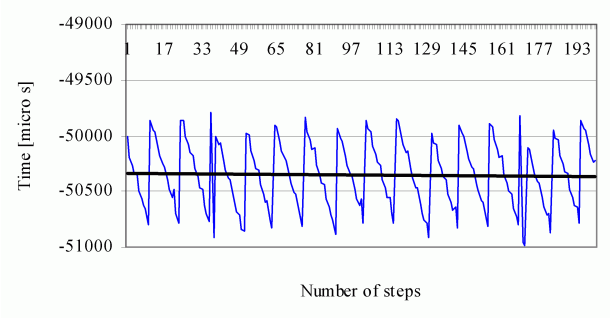
We also did runs using the *random* option. All of the operating systems gave us similar results to the one seen in Figure 6.10. We can see from that graph, that T-UppAal has good random distribution. The range was in accordance to specification ranging from 450 ms to 550 ms. We noticed that in all of the experiments the effects of uncertainty were minimal to none.

6.3 Scheduling Resolution

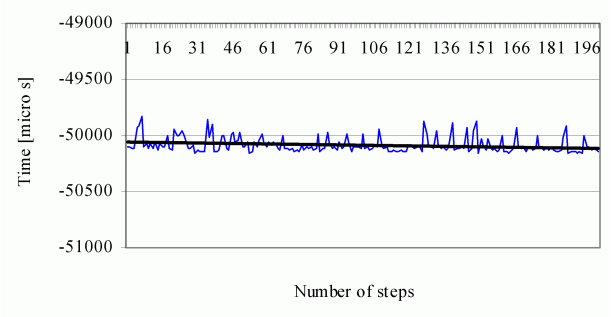
This set of experiments was motivated by the fact, that the results in Section 6.1 (dependency between T-UppAal and IUT waiting time) still had no explanation and experiments from Section 6.2 showed that Linux with kernel 2.4 and Solaris OS are better than the Linux with kernel 2.6. This is in contradiction with the fact that scheduler in the kernel 2.6 was improved since kernel 2.4. The idea



(a) Linux 2.4

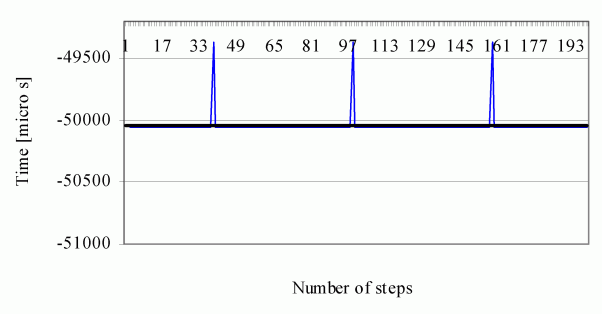


(b) Linux 2.6

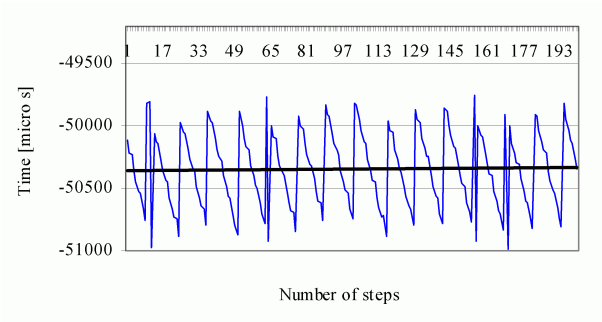


(c) Solaris

Fig. 6.8: -P eager. -u 0,0



(a) High priority - Linux 2.4



(b) High priority - Linux 2.6

Fig. 6.9: High priority -P eager. -u 0,0

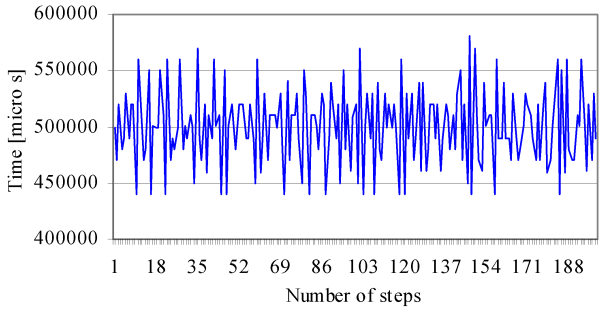


Fig. 6.10: Linux 2.4 -P random. -u 15.000, 15.000

of the experiment is the same as in Section 6.2, but this time we the adapter sleeps certain amount of time after receiving input. After waking up it sends back the output to T-UppAal.

6.3.1 Setup

The experiments were performed on the same set of computers used in previous experiment (6.2). The model used in this experiment is the same as used in Section 6.1.

The sequence of actions is shown in Figure 6.11. The pattern of the experiment is similar to the one in Figure 6.7. The difference is, that the adapter waits for t_{adp} (value depending on the experiment) and then sends output b to T-UppAal. The time $t_4^{[n]}$ is additionally measured.

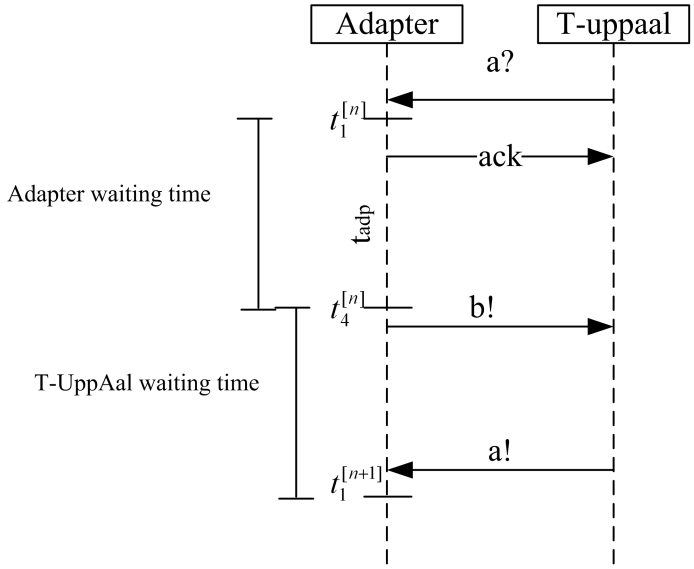


Fig. 6.11: Sequence of actions and measured time

The algorithm 8 shows how the adapter for this set of experiments behaves. As there is no IUT in this experiment, *perform()* function just wakes up another thread by sending a signal. When *threadExecute()* function gets the signal it starts to sleep for k time units. We subtract time from k because we want the total sleep to be as close to k as possible. As already some time has passed before we encounter the sleep statement, we subtract that time ($getTime() - t_1^{[n]}$). In every step the k value is increased by l which can be from 0 or 1000 depending on the experiment. After waking up local time is stored to $t_4^{[n]}$ variable and the message is sent to T-UppAal by calling function *tryReport(msg)*.

In the result analysis we used:

- *T-UppAal waiting time*:
  The time T-UppAal waits before sending an input after the output was received.
  $(t_1^{[n+1]} - t_4^{[n]})$
- *Adapter waiting time*:
  The time T-UppAal waits before sending an output after the input from T-UppAal was received.
  $(t_4^{[n]} - t_1^{[n]})$

Algorithm 8: Adapter for experiment without IUT

```
1: n := 0
2: perform(msg)
3:     t_1^[n] := getTime()
4:     inputDelivered()
5:     signalThead()

6: threadExecute()
7: k := sleepTime
8: while (!fail) do
9:     waitForSignal()
10:    sleep(k − (getTime() − t_1^[n]))
11:    k := k + step
12:    t_4^[n] := getTime()
13:    tryReport(msg)
14:    n := n + +
15: end while
```

-P	-U	sleep	step
Eager	0,0	0	0 μs
Eager	0,0	0	10 μs
Eager	0,0	0	100 μs
Eager	0,0	0	1000 μs

Tab. 6.3: Parameters - Linux with kernel 2.4.27 and Solaris

In Table 6.3 we can see the summary of the parameters that were use in experiments on Linux with kernel 2.4. We decided to do all the experiments with -u 0,0 parameter, because we saw in Section 6.1 that it did not had significant influence on the results. The *Eager* option was also used, because the results are easier to interpret. The parameter that was changing was adapter's waiting time before sending back the input. We experimented with no sleeping time and with an increasing the sleeping time. We did the same set of experiments

also on the Solaris machine. Again it was a public machine, so results may be biased.

-P and priority	-U	sleep	step
Eager, normal	0,0	200 ms	0
Eager, normal	0,0	0	10 μs
Eager, normal	0,0	0	100 μs
Eager, normal	0,0	0	1000 μs
Eager, high	0,0	0	100 μs
Eager,high	0,0	0	1000 μs

Tab. 6.4: Parameters - Linux with kernel 2.6.11

In Table 6.4 we can see the parameters for the Linux with kernel 2.6. They are the same as for Linux with kernel 2.4. The only difference is that we did the experiment with constant sleeping time equal to 200 ms. We also did some experiments with higher priority.

6.3.2 Results

We do not show all the results that we obtain, but again only the most interesting ones.

The runs without sleeping delay in adapter on Linux with kernel 2.4 gave us *adapter waiting* time of between 160 μs and 250 μs.

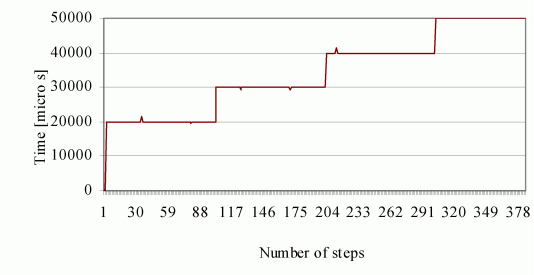
T-UppAal waiting time for the same set of experiments was between 460.0 ms and 459.6 ms. For Solaris the adapter delay was between 0.99 ms and 0.96 ms. T-UppAal waiting time was between 45.0 ms and 46.0 ms with the average equal to 45.1 ms. Solaris seems worse at immediate response time, but it may be affected by the fact that it is a public machine. T-UppAal sleeping time is nearer the desired value on Solaris though.

The runs with *adapter waiting time* increasing by 100 μs gives new information about the problem. The Adapter and T-UppAal waiting time is shown in Figure 6.12 for Linux with kernel 2.4, in Figure 6.13 for Linux with kernel 2.6 and in Figure 6.14 for Solaris.

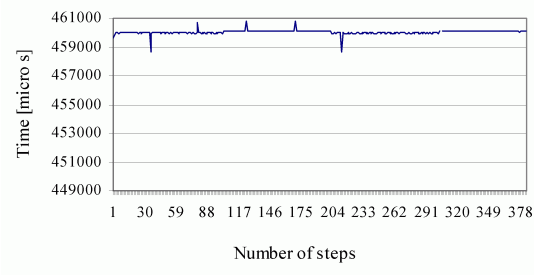
The X-axis represents number of steps (one step is sending one input and receiving one output). Y-axis delineates time in microseconds.

In Figure 6.12 (a) the most apparent is the fact that the waiting time increases in 10 ms steps. The obvious conclusion is that the Linux kernel 2.4 is not capable of getting better scheduling precision than 10 ms when waking up the process. (in the C++ code we used nanosleep). In Figure 6.12 (b) we see the T-UppAal waiting time for the same set of experiments. We can notice similar effect, as in Section 6.1, that the small peaks complement each other.

In Figure 6.13 (a) we see the waiting time of the adapter but this time on Linux with kernel 2.6. The plot here is quite different. We no longer see 10 ms steps. The size of steps is 1 ms and this is the best resolution scheduling with kernel 2.6. This is actually the resolution of the 8254 Programmable Interval

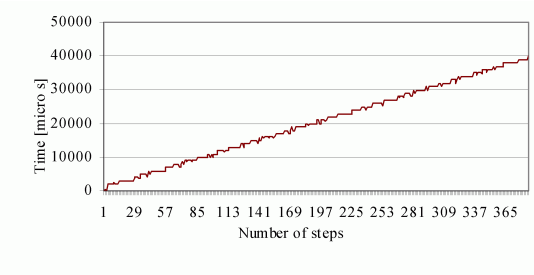


(a) Adapter waiting time

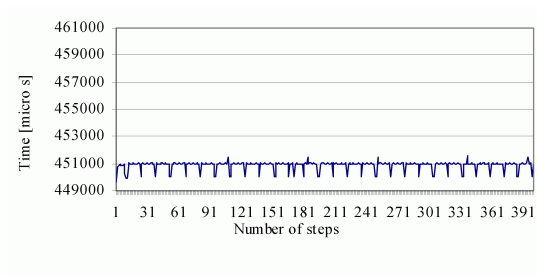


(b) T-UppAal waiting time

Fig. 6.12: Linux 2.4. -u 0,0. Increasing from 0 by 100 μs



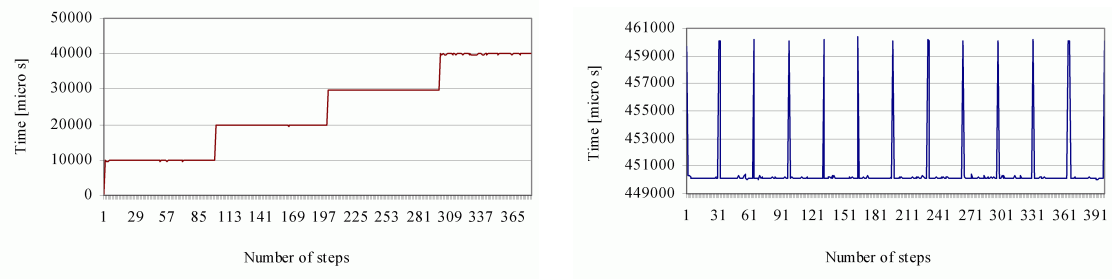
(a) Adapter waiting time



(b) T-UppAal waiting time

Fig. 6.13: Linux 2.6. -u 0,0. Increasing from 0 by 100 μs

Timer used on most uni-processor PC [19]. This can explain why the results from the experiments from Section 6.2 could show that the kernel 2.4 is better than 2.6. We simply did not change the time, so we never experienced the 10 ms jump. In Figure 6.13 (b) we see the waiting time for T-UppAal. From this graph we can see more clearly that 1 ms is the limit of scheduling resolution. The plot is almost a straight line with 1 ms jumps.



(a) Adapter waiting time

(b) T-UppAal waiting time

Fig. 6.14: Solaris. -u 0,0. Increasing from 0 by 100 μs

In Figure 6.14 (a) we can see the waiting time of the adapter on Solaris. The graph looks very similar to the one in Figure 6.12 (a). We can also see 10 ms jumps, so the resolution for schedule for Sun OS is the same as for Linux with kernel 2.4. The same fact can be seen in Figure 6.14 (b), where T-UppAal waiting time is plotted. We see 10 ms also here. The resolution of 10 ms for scheduler is the base of our explanation for the first experiment (Section 6.1).

6.4 Conclusions

Our first set of experiments (Section 6.1) showed that the network latency is minimal compared to the *scheduling delay*. The first and second experiments (Section 6.1 and 6.2) showed us that the limit in the precision we could achieve when we are deciding when an action happened or when we want T-UppAal to perform an action. This gives us the knowledge that the tolerance gap (Section 3.3.1) on when input actions should be performed by T-UppAal can in no cases be less than the scheduling delay (10 ms for Linux 2.4). Therefore having a specification of higher precision than milliseconds is not applicable. We found out that the problem of precision depends on the OS. In our experiments Linux kernel 2.6 gave the best precision.

We also see that even tough we have a certain precision in scheduling, we sometimes get a longer waiting time (peaks) that can be explained by other processes needing the CPU. We also noted that the effect of the uncertainty parameter -u option in T-UppAal is minimal on the *IUT waiting time* and the *T-UppAal waiting time*.

6.5 Coverage experiments

In our previous work [7] we made experiments involved coverage calculation. We chose to calculate two types of the implementation model coverage:

- Location coverage measure how many (and how often) locations of the selected TA-components were visited when test was executed on the model.
- Transition coverage indicates how many (and how often) transitions of the selected TA-components were traversed when test was executed o the model. [8]

There were two specifications used for experiments:

1. Light Controller specification (2.3)
2. Modified Light Controller specification (without integer variables) [7].

For experiment we used mutation testing [10]. Mutants with two types of errors were made for the experiments:

1. Timed error. It occurs when system does not meet time constraints (10 mutants).
2. Logical error. It is usually observed when the system produces unexpected results (12 mutants).

Coverage measure

We wanted to check the hypothesis what was *the dependency between the coverage required to kill a mutant and a way the coverage was calculated*. We were aware that different measures of the coverage produce different results. Therefore we wanted to find out which measure gives the greatest confidence (when it reaches value 100%) that all errors have been found.

Coverage was measured using different methods:

1. Counting the number of visited locations and divide by the number of all locations;
2. Specifying the target number n of visits for each transition. When each transition is visited at least n times then the coverage is equal to 100%;
3. Calculating transition coverage using the modified specification;

Form experimental results we conclude that location coverage calculated on original specification does not give good assurance, because most of the times maximum coverage value reached 100% and the average value was about 80%.

We expected transition coverage measure to be more confident than location coverage. The experiments on the original specification showed that average value was shifted down by about 20%. Still for many mutants the maximum values reached 100%.

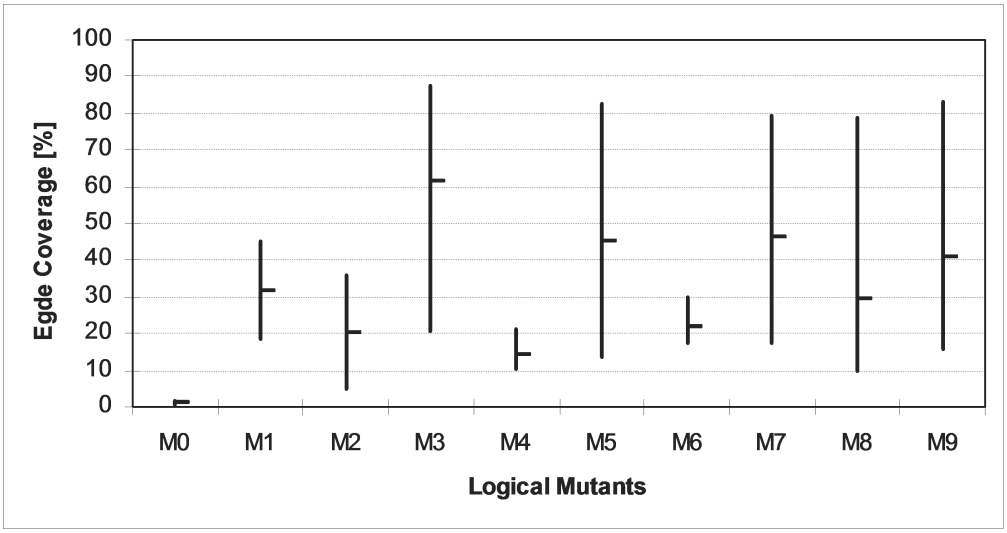


Fig. 6.15: The transition coverage of the modified specification for mutants with logical error(maximal, average and minimal values)

Our experiments showed that the coverage measure with target did not give much better assurance than the ordinary ones. Additionally it causes unnecessary stretch between the minimal and maximal value.

The transition coverage measure calculated for modified specification (see Figure 6.15 gave the best assurance. We show minimum, maximum and average coverage values for each mutant. This time maximum values are below 90%. And for example to kill M6 mutant we needed about 20% of the model to be covered. Compared with location coverage calculated on original specification, the same mutant (M6) was killed when average coverage value was 80%.

We can conclude that coverage required to kill a mutant depends on the way it was calculated and that using transition coverage on modified specification we can have much bigger confidence that most of errors were found when the coverage reached 100%.

Dependency between coverage and time

Another thing we wanted to check was the *dependency between time and the coverage*. We expected that the coverage increase with a time faster at the beginning. The longer test is executed the slower coverage increases. The coverage is not reaching 100 percent. The longer test was executed the bigger part of the IUT was examined.

Experiments were performed:

- with IUT without any errors introduced.

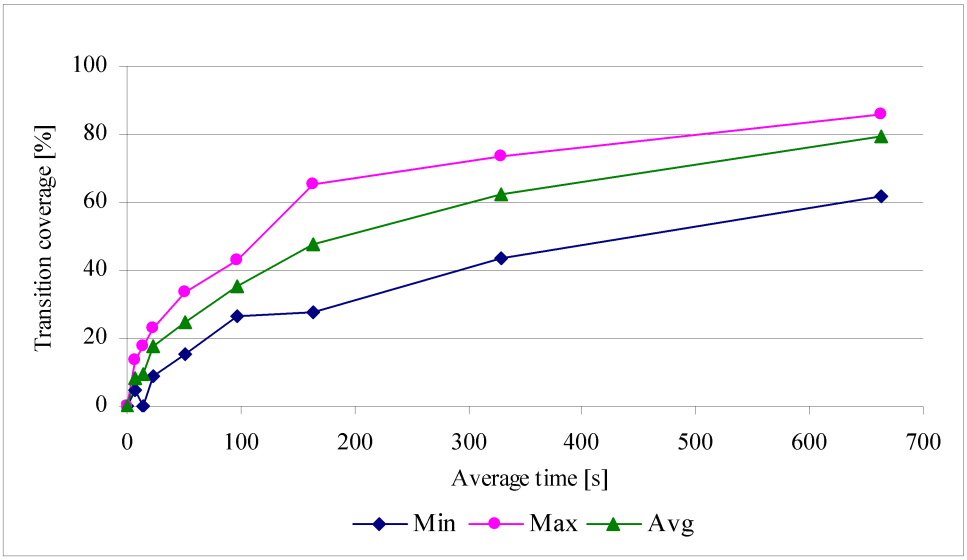


Fig. 6.16: Transition Coverage vs. Time. Modified Specification.

- using original and modified specifications
- run the test for different intervals of time

The examined data was:

- time of a test run;
- location and transition coverage.

In Figure 6.16 we show transition coverage versus time calculated using modified specification. In the graph we have minimum, maximum and average coverage value. In X axis we show average time in seconds and in y axis we delineate coverage value in percentage. It was the most detail coverage measure, so coverage value was increasing slower compared with other setups (*e.g.*. location coverage calculated on original specification). But still the result supported the hypothesis.

As we expected, the coverage value increased with the time faster at the beginning. The longer test was executing the slower coverage was growing. According the hypothesis coverage value should not reach 100%. This did not proved out with original specification. However with modified specification the 100% coverage was not reached.

Error detection

Another hypothesis we were tying to prove was that *the same error is detected in certain location/transition or set of locations/transitions.*

We believed that when some error is introduced into a program, it is usually detected when passing though certain part of the specification (transition or location). The possibility of encountering this error may depend on other factors like the values of variables or the values of clocks (or factors not captured by the formal specification due to abstraction in modelling).

To check the hypothesis we made:

- 10 runs for each mutant described above;
- timeout for T-UppAal 1000 seconds.

The examined data was in which locations/transitions the error was observed (in the case where it is detected).

In the bottom part of Figure 6.17 we see mutants and their transitions, where the error was detected. The number in the x axis represents amount of times the error was found on the particular transition.

We see that for mutants M0, M4, M7 the error was discovered always when traversing the same transition. For mutants M5 it was in the same transition in nine out of ten runs. Mutants M8 and M9 did have a different last transitions but the error seemed to almost always be detected when leaving state A1.

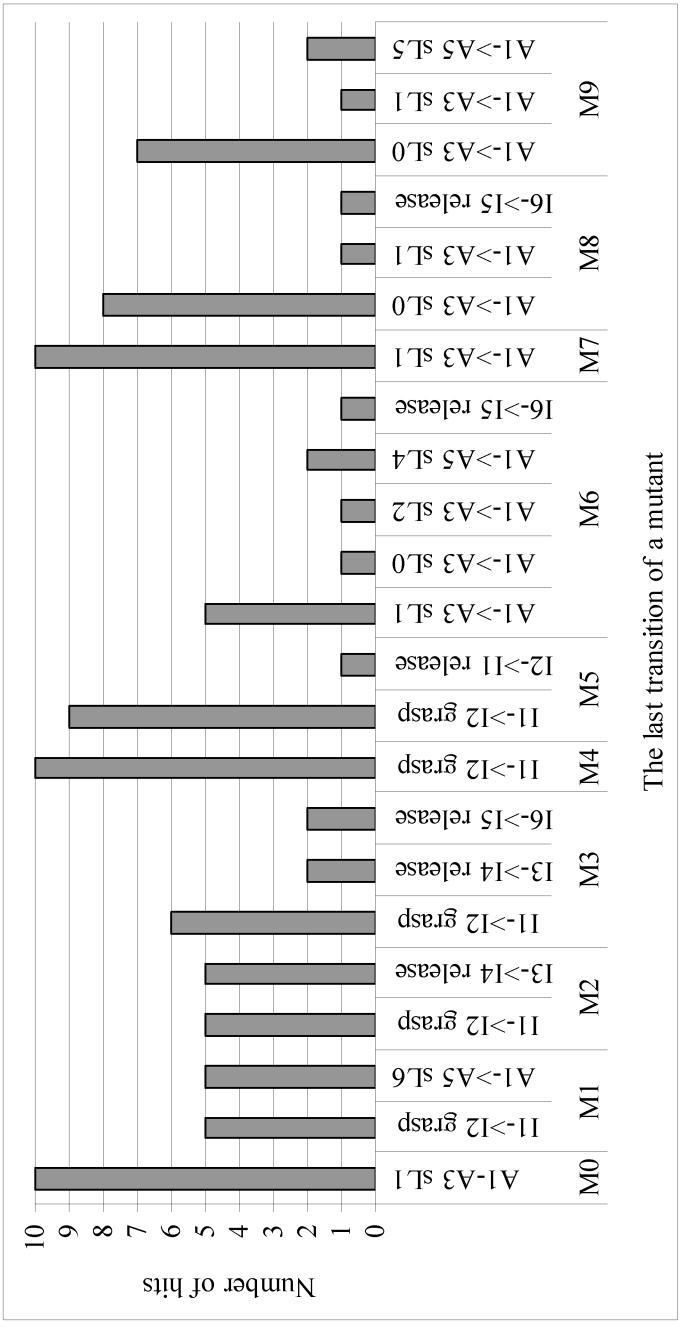


Fig. 6.17: Number of hits for the last transition(s). Logical errors

If several test runs were done for a long time, it is likely that different paths would be traversed. After examining last transitions we realized that no matter how long was the test run, we usually find error in the same transition/location or in a constant set of them. For some mutants T-UppAal needed only one transition and for others it was from one to three. This implies that when getting many different last transitions, the mutant can be discovered on many transitions, but not that many transitions are needed to discover the mutant. For timed mutants the results were similar. It was frequently the case that T-UppAal found the error on the same transition or when traversing from a certain location.

The experiment gave promising results that shortening of test run was possible.

7. INDUSTRIAL CASE STUDY

Having analysed the factors that could affect re-runs we believed that we were ready to try out re-runs on the industrial case study. As our IUT we used EKC 201/301. The EKC 201/301 is an advanced electronic thermostat regulator. It is made by Danfoss, a Danish company recognized world-wide for its Refrigeration and Air Conditioning, Heating, Water and Motion Controls [4]. It has been sold worldwide and is used to control and monitor the temperature of cooling plants such as freezer rooms and large supermarket refrigerators. The T-UppAal (TRON) tool was used by [6] to examine an industrial case study and EKC refrigerator was used as IUT. We extend the ideas proposed in [6] by checking the possibility on test re-run on the same IUT (EKC refrigerator controller). Additionally we do coverage measurements of one run using the ideas presented in Chapter 4. The modelling of the IUT to TA had already been done by [6] and was therefore ready to be used in our experiment. For keen readers the model can be viewed at http://www.cs.aau.dk/∼bnielsen/compressor.xml. The time constants in the specification were in the order of seconds , but some ranged to hours.

7.1 The ECK control objectives

The main control objective of EKC unit is to turn on/off a compressor when the refrigerator room it is monitoring gets too hot/cold. The temperature should be maintained between user defined *set-point* and *set-point+differential* degrees. The regulation is based on the temperature T_n which is the average room temperature. The temperature T_n is recalculated periodically using the readings from a sensor. The new temperature is a weighted average (equation 7.1) where the old temperature is weighted by 80% and the new sample by 20%

$$T_n = \frac{T_{n-1} * 4 + T}{5} \tag{7.1}$$

The simplified control objective can be seen in Figure 7.1. There should be a minimal delay before the compressor can restart and the minimum duration when compressor should remain on. If the temperature is above/below *high Alarm Limit/low Alarm Limit* for *alarm delay* time units an alarm must sound.

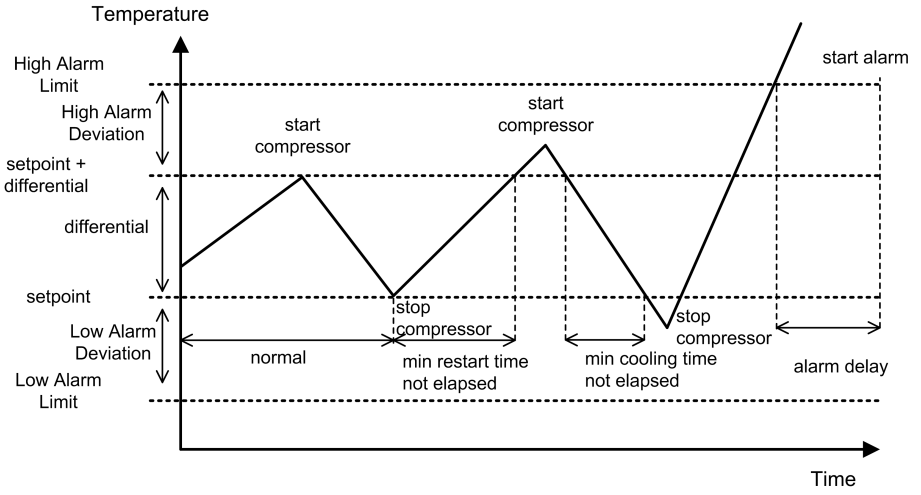


Fig. 7.1: EKC Main Control Objective

7.2 Model Structure

The functionality of the EKC unit was model by [6] as the network of TA: its main components handling, basic temperature regulation, alarm monitoring, and defrost modes with manual and automatic controlled (fixed) periodic defrost (de)activation. The input actions that are sent to the EKC (IUT) unit can be seen in Figure 7.2 with the output actions. The tolerance and timing uncertainties of the IUT and those caused by adaptation software were modelled explicitly in the model.

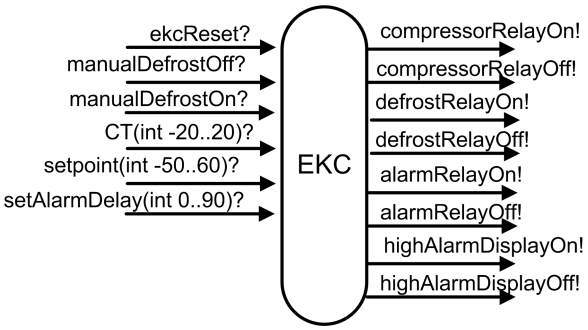


Fig. 7.2: Model input/output actions

The model itself is quite complicated even though not all of the functionality of the controller was modelled. It consists of thirteen main components (counting components of the environment) The model is also non-deterministic as it gives much tolerance for IUT when the action can occur.

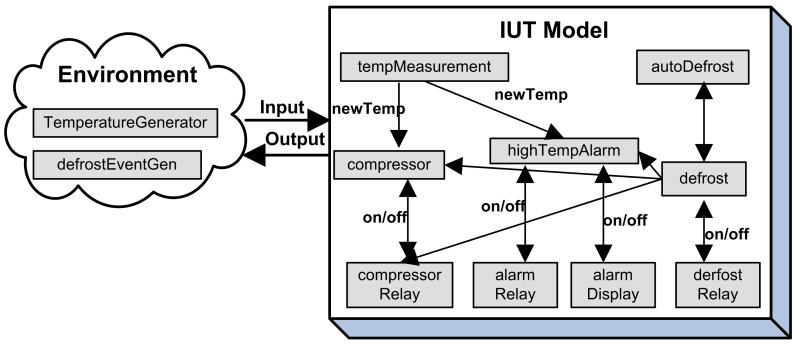


Fig. 7.3: Model components dependencies

Figure 7.3 depicts the main components and their dependencies. The figure and the following description is taken directly from [6] as their explanation of the model we find complete and descriptive.

"The **Temperature Measurement** component periodically samples the temperature sensor and calculates a new estimated room air temperature. The **Compressor** component controls the compressor relay, based on the estimated room temperature, alarm and defrost status. The **High Temperature Alarm** component monitors the alarm state of the EKC, and triggers the alarm relay if the temperature is too high for too long. The **Defrost** component controls the events that must take place during a defrost cycle. When defrosting the compressor is disengaged, and alarms are suppressed until *delayAfterDefrost* time units after completion. Defrosting may be started manually by the user, and is engaged automatically with a certain period. It stops when the defrosting time has elapsed, or when stopped manually by the user. The **Auto Defrost** component implements automatic periodic time based defrosting. It automatically engages the defrost mode periodically. The **Relay** component models a digital physical output (compressor relay, defrost relay, alarm relay, alarm display) that when given a command switches on (respectively off) within a certain time bound. The **Temperature Generator** is a part of the environment that simulates the variation in room temperature, currently alternatingly increases the temperature linearly between minimum and maximum temperature, and the reverse. Finally, the **Defrost Event Generator** environment component randomly issues user initiated defrost start and stop commands."

7.3 Model Adaptation

The setup was a little more complicated than in the previous experiments as the inputs and outputs of the EKC are stored in a parameter database internally in the EKC. To access that database, we were provided by Danfoss a Visual Basic (VB) API monitoring software that runs on a MS Windows XP PC. As T-UppAal only exists in UNIX version, a second computer was needed to host it and connect to the VB interface using a TCP/IP connection. The connection

from the VB host to the EKC gateway was via RS-232 and from it to the EKC unit by a LON network.

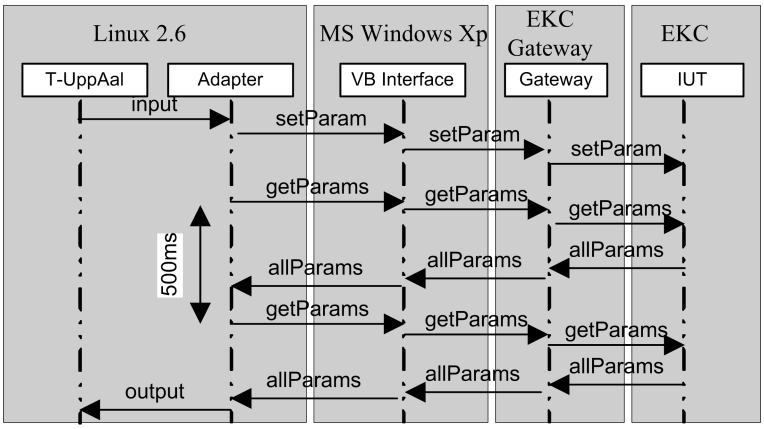


Fig. 7.4: Test Adaptation

In Figure 7.4 we can see the path that input and output message must travel between T-UppAal and the IUT. When T-UppAal decides to send input it is forwarded to the adapter. In the adapter this is converted to the *set parameter* command for the VB component. The VB component updates the internal parameter database in the EKC through the EKC Gateway. In order to receive an output from IUT, the adapter periodically (every 500ms) asks for the values of all the parameters. It compares the received values with the previous readout and translates the changed parameters to the outputs that are understood by T-UppAal. As the result the timing uncertainty is quit large. We have no grain control when event should happen or even no exact information when the event happened.

The computers that were used:

1. PC1:(used alternatively)
   (a) AMD Athlon 1.8 GHz, 256MB on RAM running Red Hat Linux 2.6
   (b) Dual Xeon, 2x2,8 GHz CPU, 3072 MB RAM Interactive use RedHat Enterprise Linux 3 (Intel x86)
2. PC2:Pentium 1.3 GHz, 512MB of RAM running Windows XP

We used two computers for hosting T-UppAal (one at a time) and one for the VB application. The more powerful one (PC1 b) is a computer situated at Aalborg University network for running applications. When using that computer, other processes could be running that could affect our results. Therefore we choose also to have another - not as powerful one (PC1a), which we could dedicate only for running the experiments.

7.4 The Tools and The Options

When making a re-run there are number of parameters that both butler and T-UppAal can use that affects the re-runs. We divide them into two parts. The one used in generating the re-run TA with Butler, and the options that T-UppAal were run with.

The Butler options (see Section 5.3) used during this experiments were:

- Input tolerance
- Output tolerance
- Precision
- Output enabled
- Global/Local clock

The *input* and *output tolerance* specify the gap in which actions are allowed to happen (see Section 3.3.1). The *precision* specifies how many microseconds one model time unit lasts in the new specification generated by Butler. 5.2.7. For example the value of 1000 would make the re-run specification where each model time unit would correspond to 1000 microseconds. *Output enabled* option creates specification where all output actions coming from the IUT are valid. Any output action can arrive at any time (only original model constraints are taken into account). *Global/Local clock* parameter decides if a global or a local clock should be used in the TA re-run (see Section 3.3.3).

For coverage measurement we used Butler with following options:

- The input and output tolerance set to zero - we used most restrictive tolerances in order to minimise state set explosion and reduce time needed to give results to the query
- The global clock was used. We wanted the actions to be allowed to happen at exactly the same time. This is possible as we do not the actual re-run, just model checking.
- We increased the precision from 100 000 to 10 000. This the rounding errors had no impact.
- We added the transition coverage variables. We did not included the location coverage as the results should be similar. The other factor was time limitation.

Butler saves the specification with coverage variables and TA trace included in the separate file. It generates also the file with queries for the UppAal. We used command line version of UppAal called *verifyta*. We used depth first search.

The T-UppAal options used:

- -u inRes,outRes (in microseconds)
- -P delay (eager, random)

The -u options is used to specify the observation uncertainty intervals in microseconds. This tells T-UppAal how long it may take the action to be delivered. The -P option controls how T-UppAal should choose to delay before sending a action. When *random* option is specified, the action is sent at a random time chosen from the allowed time interval. With *eager* the action is sent at the earliest possible moment (as early as allowed by the time interval). To be able to make re-runs we need as much control as possible when the input actions should occur. Because of that we have chosen to set the -P delay option to *eager* in our experiments.

7.5 Results

We split the results into three sections. One where we tried a very restrictive type of re-runs where:

- The sequence of the actions had to be the same.
- Both the input and the output actions were with *tolerance* which decided when they should happen.
- Tolerance was specified separately for the input and the output.
- The input and the output was checked against conformance to a specification.

The second type of re-run was made less restrictive by removing the implementation part from the specification.

The last section presents the results for the coverage measurements.

7.5.1 Re-run with Conformance to Specification and Sequence of Actions

Our first attempts allowed us to find and fix minor bugs in Butler: One regarding invariants wrongly defined to be a valid T-UppAal XML specification, the other being the channels not being in right order. The previous experiments were done on much simpler specifications and therefore we did not encounter those errors. After having fixed these errors we continued. Over 50 runs were made from two recorded test runs of length of 7 and 10 minutes. The number of actions for these runs was 63 and 128 respectively. The summary of all the options can be seen in Table 7.1

The length of these runs lasted from 40 to 273 seconds (average of 27 actions). The majority of them stopped after around 126, some even sooner. The failures of the run can be split into three categories:

- "TEST INCONCLUSIVE: IUT failed to offer output in time"

Computer used	Number of runs.	Input tolerance range in ms	Output tolerance range in ms	Precision in μs	T-UppAaal parameter range on -u
PC1 a)	41	25 to 400	10.000 to 20.000	1.000	0,0 to 5.000, 5.000
PC1 b)	13	25 to 1.200	5.000	1.000	5.000, 5.000 to 20.000, 20.000

Tab. 7.1: Overview of parameters used in the experiment

- "TEST INCONCLUSIVE: Model contains Deadlock(s)"
- "TEST INCONCLUSIVE: Input executed too late. CPU too slow or load too high"

The inconclusive verdict of "IUT failed to offer output in time". To solve this problem we increased the tolerance on output actions. After we increased it these verdict stopped to appear.

More of a mystery was the fact that T-UppAal was reporting deadlock(s) in the specification. This was rather unexpected, as when the specification was checked for deadlocks using the UppAal model checker tool, it showed none. By generating a TA trace of the re-run using *Butler*, we managed to analyse where the deadlock occurred. We created simple specification in which UppAal did not showed deadlock where it should. This appeared to be a critical bug in the UppAal model checker. Because of the bug a new version of UppAal (with a bug fix) was released.

Many times we encountered the "Input executed too late..." verdict. Our first response was to increase the tolerance on when input actions could occur. When increasing the tolerance we managed to get a little longer runs as seen in Figure 7.5. But increasing tolerance more than 400ms did not increased the time of re-run. With tolerance much higher than 400ms the re-runs were either stopping very shortly, after 15 sec or around 250 sec. One explanation can be that T-UppAal is encountering state explosion. Therefore it needs more powerful computer to be able to fulfill the action deadlines. When trying on more powerful computer PC1 b), it gave similar results. After further investigation, it appeared that in some of the cases, the state set used by T-UppAal for computing the possible state of the IUT, was rather low (88 states). After thorough analysis (accompanied by simple experiments) we found possible reason of the problem. The result is the concept of *narrowing gap* described in Section 3.3.4.

7.5.2 Re-run without Implementation Part

We experimented with re-runs that have less restriction on actions. For this experiment we used the same setup as for the previous experiment except only computer PC1 a) was used.

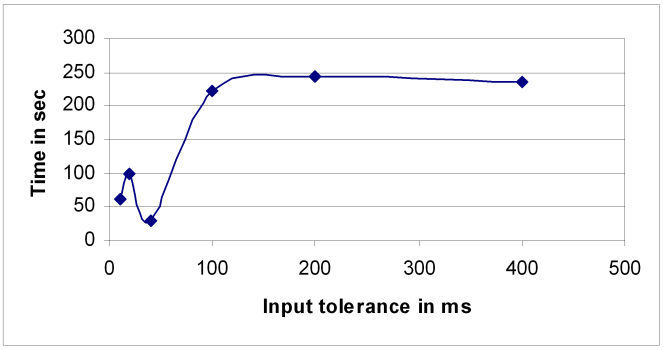


Fig. 7.5: Re-runs length in seconds(Y-axis) for different input tolerance (X-axis

The parameters used in the experiment can be viewed in Table 7.2. The difference in this experience from the last one is, that now we remove the implementation part in the specification. This gives us a specification where we only have TA trace of the run we want to simulate. This means that T-UppAal verdict will only conform that the re-run is in accordance to the TA trace, but not check if it is valid according to the implementation part (as it is now removed). We generated four TA (specification) from four different runs of length of 20 min. The number of actions in these runs was 169, 187, 190 and 201.

Computer used	Number of runs.	Input tolerance range in ms	Output tolerance range in ms	Precision in μs	T-UppAaal parameter range on -u
PC1 a)	4	50	10.000	1.000	0,0

Tab. 7.2: Overview of parameters used in the experiment

The result was that: For all of the four runs, we encountered success and managed to do a complete re-runs (of 20 min.).

7.5.3 Coverage Measurements

We performed the coverage measurements for the 10 min run with 128 actions we obtained in the previous experiments. We generated two queries for each edge as described in Section 4. It was in total 260 queries.

During our first trials we experienced a problem with a state space explosion. UppAal was using more than 1.5GB of memory and still was not able to provide answer to half of the queries. The solution to the problem was changing the range of coverage variables from $[0, 32762]$ to $[0, 1]$ and the assignments from increasing value of coverage variable by one to setting value to one. After such modification we managed to obtain answer to all queries within 30 minutes when

using -T option with UppAal (reuse state space).

Template instantiation	Certainly traversed	Certainly not traversed	Possibly traversed
IUT_TemperatureReceiver	2	0	0
IUT_TemperatureMeasurementErr	5	0	0
IUT_Compressor	8	2	0
IUT_compressorRelay	5	7	0
IUT_ActionHandler	4	6	0
IUT_alarmRelay	1	10	1
IUT_highAlarmDisplay	1	10	1
IUT_HighTemperatureAlarm	8	7	4
IUT_AutoDefrost	1	1	0
IUT_Defrost	11	4	1
IUT_defrostRelay	5	7	0
IUT_INIT Count	6	0	0
Sum	57	54	7

Tab. 7.3: Results of coverage measurements - number of covered edges

The summary of the results of the queries can be seen in Table 7.3. We can see that more than half of the transitions were covered. Seven out of 118 transitions were *possibly covered*. We have no possibility of checking if they were actually visited during the run using data we have. We can see that the templates *IUT_AutoDefrost*, *IUT_highAlarmDisplay* and *IUT_alarmRelay* were not covered at all except the first transition. From that we see that the features associated with the alarm and the auto defrosting were not tested the run.

7.6 Summary

The first experiment 7.5.1 showed us that with this kind of restriction on the re-run, It is difficult (almost impossible) to make a re-runs with a fairly complicated specification. One of the reason for that, we explain in Section 3.3.4 *narrowing gap*. By knowing why the more restrictive type of re-run failed, we choose a less restrictive one 7.5.2 where the problem of *narrowing gap* should not occur. In that we had success for all of the runs which confirmed, that the *narrowing gap* had the most effect on why we failed with the more restrictive ones. We managed to do the coverage measurements using technique described in Chapter 4. The care must be taken to avoid state space explosion when doing the coverage measurements.

8. CONCLUSIONS AND FUTURE WORK

8.1 Epilogue and Conclusions

Our main goal of the project was to obtain testing information, analyse it and reused in order to achieve a better testing performance. The data logged during a test run needs to be used to reproduce errors faster by repeating exact behaviour of the test tool. Additional metrics to the test run information should be applied to discover untested parts of the systems.

In order to make such an improvement we put following tasks for us:

- To transform test information and use it for a re-run,
- To find a technique of model coverage measurement.

In the next sections we conclude and sum up the results obtained through out our work.

8.1.1 Butler

The problem of not readable test run information was solved by creating a Butler tool, that constructs TA trace from a trace log.

One of the features is that a tool automatically generates a new specification, where the environment part is replaced with the TA trace that can be used for test re-runs. Also using UppAal and the new specification, simulation and reachability analysis can be performed regarding the run.

Another helpful thing is that definitions can be separated from the code of the adapter. In order to achieve it we created a library for parsing a file with the definitions of input/output actions.

The need to calculate test coverage can also be satisfied by the tool. Butler has an option to include in the specification a set of auxiliary variables used as counters. Using UppAal and the modified specification with counters and TA included, we are able to obtain the information of how many times each transition/location was traversed/visited in the run.

The tool is compatible with future releases of UppAal, because we used *libutap* library for parsing the specification. We extended this library with possibility of writing the specification that has been parsed (and possibly modified).

We conclude that Butler is a helpful tool that gives the tester an array of possibilities that he can use for testing and diagnosis of runs.

8.1.2 Re-runs

Problems regarding Re-runs

To be able to make re-runs, it is crucial to understand the factors that affect re-runs and problems concerning them. We identified following problems:

- Platform behaviour
- Effect of non-determinism
- Global and relative time
- Narrowing of the gap
- Conversion of time units

We proposed solutions to some of them. In order to achieve that a detailed analysis of T-UppAal performance with re-runs was done.

When analysing behavior of platforms in context of a test re-run, we came up with conclusions:

- The effect of network latency is small (less than 500 microseconds) compared with a scheduling delay that is at least 1 ms in Linux kernel 2.6 and 10 ms in Linux kernel 2.4.
- The tolerance gap (section 3.3.1) on when input actions sent by T-UppAal can in no cases be less than the scheduling delay (10 ms for Linux 2.4), otherwise T-UppAal would not be able to deliver actions in time.
- Having a specification of higher precision than milliseconds is not applicable, because scheduling resolution is in ms. It may be applicable in Real time OS.
- T-UppAal should be run on RTOS in order to perform precise reruns, because RTOS gives much better control over the scheduling.

The problem of time non-determinism we partially solve by specifying an *eager* option to T-UppAal during a test re-run.

When analysing usage of a global and a local time, we infer that the choice is IUT specific. The global time should be used when we have the IUT where timing of actions is dependent on other actions. When timing of actions in the IUT are only dependent on when previous action occur, local time should be chosen.

The narrowing gap problem we solve by removing the constraints imposed by specification.

To deal with a situation that T-UppAal is able to deliver action at much higher precision than one model time unit, we decided to generate the TA and the specification according to a precision specified by the user. The user can increase by choosing a natural number (decrease is not possible) the precision of the model, to a max value of microseconds (as that is the precision in the trace log from T-UppAal).

Re-runs Criteria's

In our analysis of re-run, we identified different types of test re-runs criteria's. We ordered them according to the difficulty:

- Specification and environmental constraints
- Timing tolerance
- Relative position of timing constraints
- Order of the actions

We identified the border which of them are feasible using T-UppAal and which are not. In our analysis we focused our work on a restrictive type. When we found out that it is not possible to make a re-run using that type, a detailed analysis of possible reasons followed.

After detail examination of the criteria's, we conclude that they are IUT specific. In our case (industrial case study) the best was to remove specification, use global time and have bigger tolerance on output actions than on input. We successfully did re-runs using this approach.

8.1.3 Coverage Measurements

The coverage calculation experiments were done in our previous work [7]. In this project we proposed another methodology for coverage measure. Using this methodology transitions (or locations) can be classified as:

- possibly traversed,
- certainly not traversed,
- certainly traversed

We added a feature to Butler so that it generated the queries automatically for every transition in the specification. Using this feature we did a coverage measurements on an industrial case study. We conclude that our tool and method are feasible, and can be used in practice to give the tester more understanding of what was tested during a run.

8.2 Future Work

We successfully applied Butler to an industrial case study, however bugs in Butler were found and corrected. This could indicate that there still exist undiscovered bugs in Butler. A more thorough testing of Butler on different specifications with the use of different options could be useful.

Also some improvement of Butler can be made:

- possibility to reorder actions,

- setting individual tolerance for actions
- extension with a graphical interface for better usability.

The coverage measure opens several possibilities:

- Guiding - It would be useful to generate the TA traces that would cover the parts of the specification that were not covered during previous runs
- Graphical representation of results - at the moment the coverage measurements are obtained in the form of answers to queries or the values of variables. It would be useful to present them in some more readable format. For example the specification can be coloured according to coverage results.

More study of the different types of re-runs than used in our experiments could prove useful. This could include to show in more detail which types of re-runs are usable for different kind of testing (specifications).

Runs can continue for a long time. Making a re-run of the complete trace is at the moment also equally long. Our results indicated that path shortening of TA trace to the error is possible [7]. One idea to do this would be to generate TA trace from the trace log, remove unnecessary parts from the path to the error and then replace the environment with the new shorter TA trace.

BIBLIOGRAPHY

[1] G. Gagne A. Silberschatz, P. Galvin. *Applied Operating System Concepts, First Edition*. John Wiley and Sons, Inc, New York, 2000.

[2] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002*, volume 2469 of *LNCS*, pages 3–22. Springer, 2003.

[3] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] Danfoss internet webpage, http://www.danfoss.com/.

[5] Kim G.Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal: Status and future work. Department of Computer Science, Aalborg University, 2004.

[6] Kim G.Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron an industrial case study. Center of Embedded Software Systems, CISS, 2005.

[7] Gunnar Hall, Piotr Kordy, and Dalia Vitkauskaite. Experiments and Future Work for T-Uppaal. Technical report, Department of Computer Science, Aalborg university, 2005.

[8] Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal Test Cases for Real-Time Systems. In *3rd International Workshop on FORMAL APPROACHES TO TESTING OF SOFTWARE (FATES 2003)*, Montréal, Québec, Canada, October 2003. In affiliation with the 18th IEEE International Conference on AUTOMATED SOFTWARE ENGINEERING (ASE 2003).

[9] Iso - international organisation for standardization, http://www.iso.org.

[10] Stephen D. Lee Jeff Offutt. An empirical evaluation of weak mutation. In *IEEE Transactions on Software Engineering, 20(5)*, pages 37–344, May 1994.

[11] Rinat Khoussainov and Ahmed Patel. Formal modelling in embedded system design: a case study. In *Proc. of Formal methods and telecommunications: International Workshop*, pages 40–59, September 1999.

[12] Kim Larsen, Marius Mikucionis, and Brian Nielsen. Online Testing of Real-time Systems using Uppaal. In Jens Grabowski and Brian Nielsen, editors, *International workshop on Formal Approaches to Testing of Software*, Co-located with IEEE Conference on Automates Software Engineering 2004, Linz, Austria., September 2004.

[13] Qing Li and Carolyn Yao. *Real-Time Concepts for Embedded Systems*. ISBN:1578201241. CMP Books, 600 Harrison Street, San Francisco, CA 94107 USA, 2003.

[14] Peter A. Lindsay. A tutorial introduction to formal methods. In *Technical report No. 98-25*. Software verification research centre, school of information technology, the university of Queensland, October 1998.

[15] Egle Sasnauskaite Marius Mikucionis. On-the-fly testing using uppaal. Master's thesis, Department of Computer Science, Aalborg University, 2003.

[16] Marius Mikucionis. About uppaaal tron simulation. 2005.

[17] Jan Peleska and Cornelia Zahlten. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *Sixth Biennial World Conference on Integrated Design and Process Technology*, Pasadena, California, June 2002.

[18] William Stallings. *Operating Systems: Internals and Design Principles. Fourth Edition*. Prentice-Hall, 2001.

[19] Guidelines for providing multimedia timer support, September 2002. http://www.microsoft.com/whdc/system/cec/mm-timer.mspx.

[20] Jan Tretmans. Testing techniques, formal methods and tools group. Faculty of Computer Science, University of Twente, The Netherlands, 2002.

[21] J. Tretmant. Testing concurrent system: A formal approach. In *CONCUR'99-10th Int. Conference on Concurrency Theory, volume 1664 of Lecture Notes in Computer Science*, pages 46–65. Spriner-Verlag, 1999.

[22] T-uppaal homepage, www.cs.aau.dk/˜marius/tuppaal.

[23] Uppaal homepage, www.uppaal.com.

[24] J.A. Whittaker. What is software testing? and why is it so hard? In *IEEE Software 17, January/February*, 2000.

[25] Xerces c++ parser, http://xml.apache.org/xerces-c/.