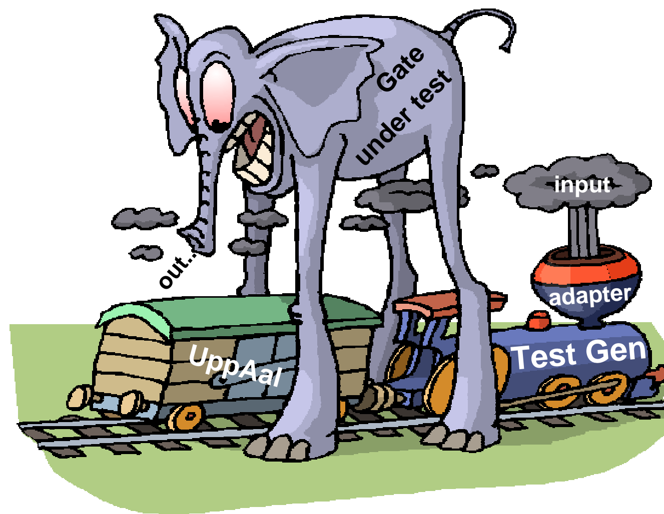
DEPARTMENT OF COMPUTER SCIENCE, AALBORG UNIVERSITY
FREDRIK BAJERS VEJ 7B, 9220 AALBORG ØST, DENMARK

# On-the-fly Testing Using UPPAAL

Master thesis




by
Marius Mikucionis and Egle Sasnauskaite

Supervisors: ass.prof. Brian Nielsen, co-supervisor prof. Kim G. Larsen

{ marius | eglese | bnielsen | kgl } @cs.auc.dk

June 11, 2003

**Title:**

On-the-fly Testing Using UPPAAL

**Subject:**

Methods and Tools for Validation

**Project group:**

SSE-4

**Participants:**

Marius Mikucionis
Egle Sasnauskaite

**Supervisors:**

Brian Nielsen
Kim Guldstrand Larsen

**Time of writing:**

The $3^{rd}$ February 2003 - the $11^{th}$ June 2003

**Copies:**

7

**Pages:**

87

**Abstract**

The goal of the project is to provide a test toolbox T-UPPAAL together with a sample randomized test algorithm for real time systems. A real time system model checker UPPAAL is an efficient symbolic state estimator and is chosen to be the base platform for the testing extensions. The test setup idea is inspired by the un-timed system testing tool TORX. T-UPPAAL allows different environments to be specified in order to test the implementation under different conditions.

# Preface

This Master thesis is submitted by a master group of Software Systems Engineering for the research unit of Distributed Systems and Semantics at the Department of Computer Science, the Faculty of Engineering and Science, Aalborg University, spring semester 2003. The purpose of the project is to investigate, design and integrate an automatic test generation algorithm into timed automata model checker tool UPPAAL.

Marius Mikucionis

Egle Sasnauskaite

## Resume

Testing is important subject in a software developing process for assuring and evaluating quality of a product. A testing theory for non time systems exists but testing of real time systems require special approach because of timing aspects. Therefore notations and formal methods used for testing should be reconsidered. This challenging task expands for one who want to develop or expand an existing tool for testing real time systems.

Our one year project was related to fields of semantics and verification and the main goal is to design and implement a test generation and execution algorithm for *on-the-fly* testing of real time systems. At the first step one should get familiar with the testing subject and we start with introduction to the main concepts of testing. They include purpose of testing, classification of testing strategies and uses as well as testing aspects of real time systems especially possibility of on-the-fly testing. In another step we introduce label transitions systems which are used for writing specifications and serve for expressing underlying formalism of the specification languages.

We introduce a part of the testing theory - conformance testing. The concept of the implementation relation is acquainted as a notation for correctness of an implementation towards a specification. The *ioco* implementation relation is used in a testing tool TORX and we present the main parts of the tools architecture and the test generation and execution algorithm used for non-timed testing.

We continued with formulating testing concepts for real time systems. Definitions and semantics of timed automata theory are introduced and extended with testing concepts. We add to the definition of LTS information for expressing time issues and introduce TLTS notations for using them later in time generation and execution algorithms.

We refer to the definition of a timed automata and explain semantics of a timed automata network. It is necessary to introduce symbolic techniques for representing infinite number of clock values in a compact way. Therefore we define the concept of a zone and operations on zones.

We need a correctness criterion for testing real time systems therefore *ioco* implementation relation is extended to the *rt-ioco* implementation relation. We explain the use of such a relation in pictures and later consider it in the test generation and execution algorithms.

The *on-the-fly* test generation and execution algorithm is presented and outlined in two ways. At first the algorithm is described for timed automata using timed labeled transition system states to get a pure theoretical background explaining the core of the testing process. However this algorithm is not applicable in practice and we introduc another one using symbolic states. The UPPAAL toolbox and its purpose were also described.

Having the fundamental background of testing theory and the notion of timed automata we continued with real time testing concepts. We described a physical system setup with a distinguished implementation and environment and the way they communicate through observable actions and shared variables. Test specification concepts for a UPPAAL automata network are introduced and the architecture of a T-UPPAAL tool is outlined as a part of a real time testing framework. We identify the assumptions and requirements for the real-time testing using timed automata network as specification. We propos ideas for designing and implementing the algorithm in the UPPAAL tool reusing the existing libraries.

Class diagrams were used to explain about UPPAAL tool design components: system model, symbolic state representation and algorithms. Those components were used as a base for a test specification, the algorithm implementation and the test configuration format.

Before implementing the test generation and execution algorithm we get familiar with a

pipeline paradigm used in UPPAAL. Later we introduce separate algorithm parts and describe their behavior and implementation details. Behavior of the algorithm is visualized in a message chart and explained in details referencing a test execution on a sample implementation. Implementation status shows what parts of code we needed to modify and create to implement the testing extensions.

We make several sessions of experiments with different implementations. Experiment models are used for testing the algorithm and for gathering the algorithm performance data. We also measure performance of the program using profiling the tool *gprof* and explain the results.

We introduce new ideas that we obtain during the project but could not implement due to project time constraints. After out work we conclude that real time system testing is a wide and interesting subject where research can be provided in several directions and this project gives an additional input into the field of this research.

# Acknowledgments

We thank our supervisors for proposing such an interesting and challenging mission with many various paths to be traversed, many things to learn and even to develop something peculiar of our own.

We owe the most of our success to Brian Nielsen for giving us the right directions during the project work as well as for help in finding proper information, for constant broadening our horizons and especially for patience in reviewing our drafts.

We are grateful to Kim G. Larsen for putting his interests in our work and sharing his experience, ideas and solutions to hot issues.

It is hard to imagine our work materializing into something substantial without the one of UPPAAL architects, Gerd Behrmann, who guided us through internals and was a great help when putting our hands into the source code of this magnificent tool.

# Contents

**1 Introduction** 8
1.1 Types of Testing . . . 9
1.2 Testing Strategies . . . 9
1.3 On-the-fly Testing for Real Time Systems . . . 10
1.4 Transition Systems . . . 11
1.5 Input Output Transition System . . . 12
1.6 Conformance Testing . . . 13
1.6.1 Test Purpose and Characteristics . . . 13
1.6.2 Test Execution . . . 14
1.6.3 Input Output Conformance Relation . . . 14
1.7 TorX . . . 15
1.8 Contributions . . . 17
1.9 Structure of the Report . . . 17

**2 Test Specification** 18
2.1 Timed Automata . . . 18
2.1.1 Timed Labeled Transition System . . . 18
2.1.2 Definition and Semantics . . . 19
2.2 Symbolic Techniques . . . 22
2.3 Implementation Relation for Real Time Systems . . . 24
2.4 Physical System Setup . . . 26
2.5 Real Time Testing Framework . . . 26
2.6 Input Enableness and Specification Completeness . . . 27
2.7 UppAal . . . 30
2.8 UppAal Network Specification . . . 31
2.9 Test Specification Grammar . . . 32

**3 Algorithms** 36
3.1 Test Execution on Reachable States . . . 37
3.1.1 Closure Algorithms . . . 38
3.1.2 Test Primitive Algorithms . . . 40
3.2 Test Execution on Reachable Symbolic States . . . 41

**4 Implementation** 47
4.1 Assumptions, Requirements and Desired Features . . . 48
4.2 UppAal Design . . . 49
4.2.1 System Model Representation . . . 50
4.2.2 Symbolic System State Representation . . . 52
4.2.3 Time Zone Representation . . . 53
4.2.4 Pipeline Architecture . . . 54
4.3 Specific Testing Extensions . . . 56
4.3.1 After Action Filter . . . 56
4.3.2 After Delay Filter . . . 57
4.3.3 Buffered Filter . . . 58
4.3.4 Driver Implementation . . . 58
4.3.5 Test Generation and Execution Algorithm . . . 61

4.4 Implementation Status . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**5 Experiments** **64**
5.1 Single Mouse Button . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.1.1 Model of the Implementation for the Mouse-button system . . . . . . . 66
5.1.2 Models of the Environment for the Mouse button . . . . . . . . . . . . 67
5.1.3 Sample Test Specification of the Mouse-button . . . . . . . . . . . . . 67
5.1.4 Implementation of the mouse button . . . . . . . . . . . . . . . . . . . 68
5.1.5 Multi-Button Mouse . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.1.6 Experiment Results of Mouse-click systems . . . . . . . . . . . . . . . 72
5.2 Train Gate Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2.1 Model of the IUT in the Train-gate system . . . . . . . . . . . . . . . . 74
5.2.2 Models of the Environment in the Train-gate System . . . . . . . . . . 75
5.2.3 Test Specification of the Train-Gate . . . . . . . . . . . . . . . . . . . 76
5.2.4 Implementation of the Train-Gate . . . . . . . . . . . . . . . . . . . . 77
5.2.5 Train-gate Experiment Results . . . . . . . . . . . . . . . . . . . . . . 79
5.3 Performance Issues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

**6 Epilogue** **83**
6.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85

**A Source Code** **87**

# 1 Introduction

Testing is a part of a software development process which includes the system execution with a goal of evaluating the quality of the product. One of the main product quality measures is the amount and importance of errors found. Testing is performed thoroughly in every mission critical project and typically consumes more than 30% project resources. Testing of real-time systems requires special attention to the timing aspects: the time at which events are supplied or expected and the timing relationship between separate events.

Our emphasis is on techniques for testing technical software systems such as communication software, control systems and embedded software. Those systems are event-driven systems in which stimulus/response behaviour is important as well as concurrency, distribution and non-deterministic behaviour [2].

*Formal methods* may be used to design detailed models, verify them and later derive implementations. However licensing constraints may not allow to walk through the source of implementation and/or physical nature of the implementation may still hide the potential errors. Therefore independent third-party testing is important to increase confidence in the quality of a computing system and formal methods have obvious advantages in system analysis when applying techniques from mathematics and logics. Formal languages are easier manageable to automatic processing. Formal methods assure preciseness and formal reasoning about systems or their relations, and protects from misinterpretation during the testing process. Therefore we use formal models as a specification for our implementations under test.

In our case testing is an operational way to check the correctness of a system implementation with respect to specification by experimenting with it. The correctness criterion is expressed by an implementation relation which formally defines when an implementation conforms to a specification. The specification is written in a formal language and formally defines functional behavior of a system. The specification prescribes what the system shall do, what should not do and in such a way constitutes the basis for any testing activity [2]. The success of testing depends also on correctness of the specification, i.e. a test is always as good as a specification, thus a formal verification of a specification is still needed.

The above suggests the testing scheme presented in Figure 1, where a test verdict *true* (test passed) or *false* (test failed) tells whether a system implementation conforms to a specification.

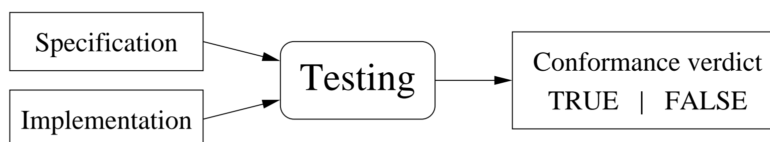



Figure 1: Testing framework.

In our project we propose an extension of automated test generation for model checking tool UPPAAL. We locate our testing aspects (type and strategy) in the Section 1.1 and Section 1.2 and land with the on-the-fly testing in Section 1.3. We define formal testing concepts in Section 1.4 and Section 1.5. Section 1.6 describes the idea of conformance testing which we use. We observe a closely related tool TORX in Section 1.7. The goal of the project is defined more precisely in the Section 1.8.

## 1.1 Types of Testing

A testing considers different aspects of system's behavior. Therefore, conformance, performance, robustness, stress testing, reliability, availability, security testing are different sort of testing and they reveal different properties and behavior of a system [2].

Two main accessibility strategies are used for testing. *White Box Testing* strategy tests whether an implementation conforms to a design since the internal structure of the system is know. *Black Box Testing* strategy tests functional aspects of a system under test and finds out whether an implementation conforms to a specification. Testing can be performed in different levels of a system and therefore we distinguish Unit, Integration and System levels. The variaty of types of testing is showed in Figure 2.

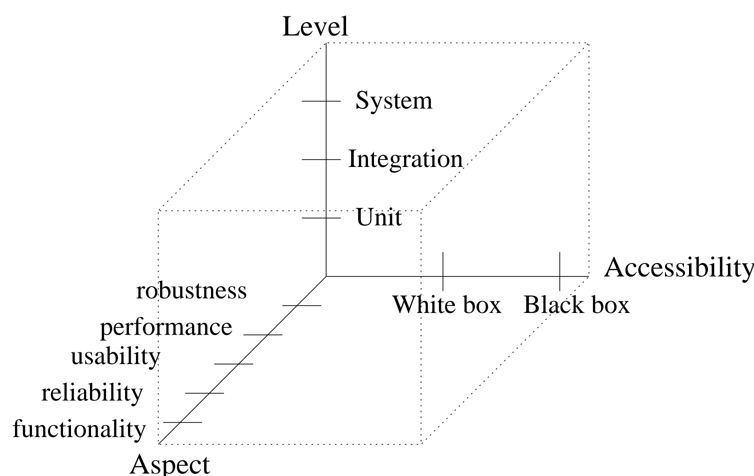



Figure 2: Testing types.

We are interested in testing the correctness of a system whose model is defined in a test specification, and we believe that the most of other criteria can be formulated in the test specification giving it an exact test purpose. Therefore our main focus is on functional or conformance system testing which refers to the *Black Box Testing* strategy.

## 1.2 Testing Strategies

Testing is an expensive activity in respect to time and resources of the project. Testing activities are repeated each time a system is modified. Therefore it is relevant to improve effectiveness of the test generation, execution and test coverage. There is a wide range of testing strategies which tend to implement a sound trade-off between coverage of testing and amount of time allocated to testing. We will give a short overview of dynamic versus static and batch testing versus on-the-fly.

*Static* testing encompasses program proving, symbolic execution, inspections and code walk-throughs. The purpose of the static strategy is to analyze whether the implementation code operates logically to the design decisions.

Different from static, *dynamic* testing requires that software is executed with test data. The key objective of the dynamic analysis is to experiment with the behaviour of the software in order to detect errors. Adequate test data sets are developed so that they activate errors and different strategies are used to develop such data sets.

*Batch* testing uses test cases that are generated completely and stored in a test notation language. Then the test case is executed against an implementation under test. The output is compared with the expected output and a decision is made according to the verdict of the test. Derivation of a test case may require exploration of all the state space and which is computationally expensive. *On-the-fly* testing is used to reduce complexity by combining test generation and test execution: tests are generated while they are executed. Instead of deriving a complete test case, the test derivation process only derives the next test event from the specification and this test event is immediately executed.

Figure 3 depicts the principle of on-the-fly testing. The *Tester* decides whether to trigger an input to *implementation under test* (*IUT*) or to wait for the output produced by *IUT*. In the case of triggering *IUT*, the *Tester* looks into the *Specification module* for a valid triggering stimulus. In the case of output observation the *Tester* checks whether the response is valid according to the specification. A time-out is observed when there is no output.

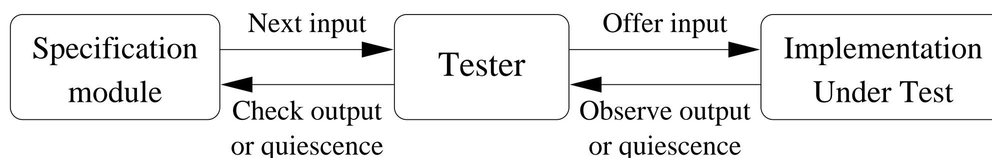



Figure 3: On-the-fly testing [2].

There are advantages and disadvantages for both *Batch* and *On-the-fly* testing. A fiew are mentioned below:

- Batch testing is suited for manual and for semi-automated suite preparation. "Humans are good at test selection but they are not fast enough to do it at run-time which is required for on on-the-fly testing" [11].
- In batch testing the tester encodes all the mapping details from the abstract test cases into concrete test cases and test implementation becomes easier. In the on-the-fly approach translation has to be done at a runtime and test cases should be generic enough to allow reuse [11].
- The on-the-fly testing requires to do all computations at run-time, i.e. a run-time translator must meet timing constraints. The batch testing allows some of the work to be moved to compile time and then it is easier to satisfy the run-time requirements [11].
- In the batch testing precomputing of the test steps leads to test-suites of enormous size. The amount of pre-computation work and the storage demand should also be considered. On-the-fly testing help to avoid those problems [11].
- On-the-fly testing does not require to explore all the states but consider only the actual responses from the IUT. However, on-the-fly test execution requires more run-time resources of a processor for computing a set of reachable states from a given set compared to batch testing.

## 1.3 On-the-fly Testing for Real Time Systems

The above mentioned problems for batch testing are even more actual for testing real time systems. Specifications of systems behavior should cover real time constraints and then the

specifications become very complex. Moreover continuity of time leads to infinite states set when a tested real time system is modeled as timed automata.

The automatic test generation and execution allows to reduce manual work which is used for preparing test cases. On-the-fly execution of tests combines automatic test generation and execution and just a part of state space has to be explored. On-the-fly testing reduces the number of states to be explored at a certain event of the system but does not solve the problem of state explosion completely. Therefore symbolic techniques are also used along with the on-the-fly testing. The following four sections describe how the untimed on-the-fly testing is done in TORX.

## 1.4 Transition Systems

A *labeled transition system* (LTS) is used to write specifications that capture the behavior of implementations, even tests cases and serve an underlying formalism for many specification languages. A LTS consist of nodes and transition between nodes that are labeled with actions. Definition 1.1 describes formally what is a labeled transition system.

**Definition 1.1** *A labeled transition system* over $L$ is a 4-tuple $\langle S, s_0, Act_\tau, \rightarrow\rangle$ where

- S is a (countable), non-empty set of *states*,
- $s_0 \in S$ is the *initial state*,
- $Act$ is a set of observable actions, and $Act_\tau = Act \cup \{\tau\}$ the actions including the distinguished internal action $\tau$,
- $\rightarrow \subseteq S \times Act_\tau \times S$ is the *transition relation*;

□

The special action $\tau \notin Act$ represent an *unobservable* (or internal, or silent) action. An element $\rightarrow$ is called a *transition*. A transition system is *rigid* if it can not do a silent action.

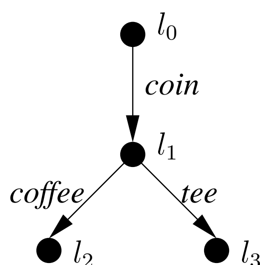



Figure 4: LTS of a coffee vending machine.[4]

Figure 4 gives an example of a simple LTS for a coffee vending machine:

$$\langle\{l_0, l_1, l_2, l_3\}, l_0, \{coin, coffee, tea\}, \{\langle l_0, coin, l_1\rangle, \langle l_1, coffee, l_2\rangle, \langle l_1, tea, l_3\rangle\}\rangle$$

We denote the class of LTS over $Act$ as $\mathcal{LTS}(Act)$. Transition systems without infinite sequences of transitions with only internal actions are called *strongly converging*. For technical reasons we restrict $\mathcal{LTS}(Act)$ to strongly converging transition systems.

A *computation* is a (finite) sequence of transitions:

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} ... \xrightarrow{\alpha_{n-1}} s_{n-1} \xrightarrow{\alpha_n} s_n$$

A *trace* $\sigma$ is a sequence of observable actions of a computation; that notation defines the observable aspect of a computation. The finite set of all sequences over a set of actions $Act$ is denoted by $Act^*$ with $\epsilon$ denoting the empty sequence. If $\sigma_1, \sigma_2 \in Act$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of $\sigma_1$ and $\sigma_2$.

Definition 1.2 presents some handy notations over LTS which we will used further in the definition of Timed Label Transitions in Section 2.1.

**Definition 1.2** A *LTS notation* for a given LTS $\langle S, s_0, Act, \rightarrow\rangle$ where $s, s' \in S$, $S' \subseteq S$ and $a_i \in Act$, $\alpha_i \in Act_\tau$, $\sigma \in Act^*$.

$$
\begin{array}{lll}
s \xrightarrow{\alpha} s' & =_{def} & (s, \alpha, s') \in\rightarrow \\
s \xrightarrow{\sigma} s' & =_{def} & \exists s_0 ... s_n \colon s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_n} s_n \\
 & & \text{where } s = s_0,\ s' = s_n,\ \sigma = (\alpha_1\alpha_2 \ldots \alpha_n) \\
s \xrightarrow{\sigma} & =_{def} & \exists s' : s \xrightarrow{\sigma} s' \\
s \not\xrightarrow{\sigma} & =_{def} & \not\exists s' : s \xrightarrow{\sigma} s' \\
s \stackrel{\epsilon}{\Rightarrow} s' & =_{def} & s = s' \text{ or } s \xrightarrow{\tau ... \tau} s' \\
s \stackrel{a}{\Rightarrow} s' & =_{def} & \exists s_1, s_2 \colon s \stackrel{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\epsilon}{\Rightarrow} s' \\
s \stackrel{\sigma}{\Rightarrow} s' & =_{def} & \exists s_0, s_n \colon s_0 \stackrel{a_1}{\Rightarrow} s_1 \stackrel{a_2}{\Rightarrow} \cdots \stackrel{a_n}{\Rightarrow} s_n, \\
 & & \text{where } s = s_0,\ s' = s_n,\ \sigma = (a_1a_2 \ldots a_n) \\
s \stackrel{\sigma}{\Rightarrow} & =_{def} & \exists s' : s \stackrel{\sigma}{\Rightarrow} s' \\
\mathbf{traces}\,(s) & =_{def} & \{\sigma \in Act^* \mid s \stackrel{\sigma}{\Rightarrow}\} \\
s\,\mathbf{after}\,\sigma & =_{def} & \{s' \mid s' \in S',\ s \stackrel{\sigma}{\Rightarrow} s'\}
\end{array}
$$

□

In a non-formal way the first notation in the Definition 1.2 must be read as: "when the system is in the state $s$ it performs the action $\alpha$ and goes to $s'$".

In Figure 4 we have:

$\mathbf{traces}\,(l_0)$=$\{\varepsilon, \text{coin}, \text{coffee}, \text{tea}, \text{coin} \cdot \text{coffee}, \text{coin} \cdot \text{tea}\}$

$l_0\,\mathbf{after}\,coin$=$\{l_1\}$ and $l_0\,\mathbf{after}\,coin \cdot tea$=$\{l_3\}$

## 1.5 Input Output Transition System

An *input/output transition system* (IOTS) is used to model systems for which the set of actions $Act$ can be partitioned into *input actions* $Act_{in}$ and *output actions* $Act_{out}$: $Act = Act_{in} \cup Act_{out}$ and $Act_{in} \cap Act_{out} = \emptyset$. We consider our system as IOTS because we need to distinguish inputs and outputs between the implementation and the environment, and the inputs and outputs are the only observable events when we consider the implementation as a black box. In Section 2.5 we explain how the implementation and the environment communicate through input and output actions.

**Definition 1.3** *An input-output transition system* $\mathcal{P}$ is a LTS where $p$ is initial state, the set of actions partitioned into input actions $Act_{in}$ and output actions $Act_{out}$ ($Act = Act_{in} \cup Act_{out}$ and $Act_{in} \cap Act_{out} = \emptyset$) and all inputs are enabled in any state [2]:

$$\text{whenever } p \stackrel{\sigma}{\Rightarrow} p' \text{ then } \forall\, a \in Act_{in} \colon p' \stackrel{a}{\Rightarrow}$$

IOTS allows input enabling via internal transitions (as opposed to strong input enabling $p' \xrightarrow{a}$). The class of IOTS over $Act_{in}$ and $Act_{out}$ is denoted by $\mathcal{IOTS}(Act_{in}, Act_{out}) \in \mathcal{LTS}(Act_{in} \cup Act_{out})$. □

We note that a certain kind of traces are distinguished in IOTS. A trace, which ends in a *quiescent* state from which no outputs or internal actions are possible, is called a *quiescent trace*.

The special action $\delta \notin Act$ indicates the absence of an output action, i.e. it becomes an observable event when there are no output from a state.

If $\delta$ can appear in a trace at any place, then we have traces with repetitive quiescence where outputs are refused and inputs after such outputs can occur. Such traces are called *suspension traces*.

We introduce definitions of a *quiescent* state and a *suspension* trace in the Definition 1.4. The concepts are formulated using [2].

**Definition 1.4** Let $p \in \mathcal{IOTS}(Act_{in}, Act_{out})$, a quiescence action $\delta \notin Act$ and $s \in S$:

$$
\begin{aligned}
s \xrightarrow{\delta} s \quad &=_{def} \quad \forall \alpha \in Act_{out} \cup \{\tau\}\colon s \stackrel{\alpha}{\not\rightarrow} \\
\mathbf{Straces}\,(s) \quad &=_{def} \quad \{\sigma \in (Act \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Rightarrow}\} \\
&\qquad \text{where } \stackrel{\sigma}{\Rightarrow} \text{ includes } \delta \text{ transitions } s \xrightarrow{\delta} s.
\end{aligned}
$$

□

We use IOTS for introducing conformance relation in Section 1.6.3, but before we present conformance testing concepts.

## 1.6 Conformance Testing

Conformance testing is used for testing the functionality of a system with respect to systems specification. A test is used to define whether an implementation conforms to the specification by performing experiments on the implementation and observing reactions. “The specification of such an experiments is called a *test case*, and the process of applying a test to an implementation under test is called *test execution*”[2].

In the next three sections we formalize testing concepts and definitions in details.

### 1.6.1 Test Purpose and Characteristics

A test case is derived from the test specification which depicts the behavior of a tester. The test is then executed on the implementation. A test verdict - *pass* or *fail* - indicates conformance (or not) of the implementation to the specification. The verdict *pass* indicates that the test execution did not reveal non-conformance; *fail* indicates that an error occurred during the test execution [2] (we will define *fail* more precisely in next section).

A test case must be deterministic, therefore a choice of at most one input action to be offered and one edge for each output action is determined. A test should also have a finite behavior and last for a finite time [2]. A test case *determinism* and *finite behavior* are required formally in Definition 1.5. Definition 1.5 defines a test case over $LTS(Act_{in} \cup A_{out} \cup \{\delta\})$ where the absence of outputs is noted as quiescence $\delta$.

**Definition 1.5** [2] A *test case* $\theta$ is a LTS $\langle S, s_0, A_{in} \cup A_{out} \cup \{\delta\}, \rightarrow \rangle$ such that:

- $\theta$ is deterministic and has finite behavior
- $S$ contains the only permitted terminal states $\mathbf{pass}$ and $\mathbf{fail}$ where $s = \mathbf{pass}$ or $s = \mathbf{fail}$ and $\nexists a$: $s \xrightarrow{a}$
- for any state $t \in S$ of the test case, $t \neq \mathbf{pass}, \mathbf{fail}$ if $t \stackrel{a}{\Longrightarrow}$ for some $a \in A_{in}$, or $a \in A_{out} \cup \{\delta\}$
- A test suite $\Theta$ is a set of test cases.

□

### 1.6.2 Test Execution

A test run of an implementation with a test case is modeled by the synchronous parallel execution of the test case with the implementation under test. Execution continues until no more interaction is possible. Absence of an interaction indicates a deadlock, i.e. a test case is in a terminal state $\mathbf{fail}$ or $\mathbf{pass}$.

An implementation passes a test run if the test run ends in the state $\mathbf{pass}$. A test case have several test runs if the implementation behaves nondeterministically. Different runs might lead to different terminal states and different verdicts. Therefore a test case must be executed several times to ensure better model coverage. An implementation passes a test case if all test runs from a test suite end in a state $\mathbf{pass}$. A test suite is *sound* if correct implementations and possibly some incorrect implementation will pass the suite while any erroneous implementation is indeed non conforming. A test suite is *complete* if no erroneous implementation can pass it.

We refer to the $\sigma$ trace when we discuss a test run representation in order to have definitions for a formal testing framework.

**Definition 1.6** [2] Let $\theta$ be a test case from a test suite $\Theta$, a state of a test case $t \in \theta$, an implementation $i \in LTS(Act)$ and $\theta\rceil|i$ is a synchronous parallel composition whose execution leads to a terminal state of $\theta$, then:

$\sigma$ is a test run $=_{def}$ $\exists i'$: $\theta\rceil|i \stackrel{\sigma}{\Longrightarrow} \mathbf{pass}\rceil|i'$ or $\theta\rceil|i \stackrel{\sigma}{\Longrightarrow} \mathbf{fail}\rceil|i'$

$i\,\mathbf{passes}\,\theta$ $=_{def}$ $\forall\sigma, \forall i'$: $\theta\rceil|i \stackrel{\sigma}{\not\Longrightarrow} \mathbf{fail}\rceil|i'$

$i\,\mathbf{passes}\,\Theta$ $=_{def}$ $\forall\theta \in \Theta$: $i\,\mathbf{passes}\,\theta$

$i\,\mathbf{fails}\,\Theta$ $=_{def}$ $\exists\theta \in \Theta$: $i\,\mathbf{pa\not{s}ses}\,\theta$

□

During test execution the verdict is drawn from observations. Occuring events and responses are observed and logged.

### 1.6.3 Input Output Conformance Relation

The aim of the conformance testing is to define the correctness of implementation with the respect to a specification. An *implementation relation* is the notation of correctness which has a sound theoretical background for expressing what is a correct implementation of a specification [5]. The intuition behind the implementation relation is that an implementation may show only behavior which is specified in the specification, i.e any possible behavior of an implementation refers to a possible behavior of the specification.

We focus on the **ioco** implementation relation (also called a suspension or delay conformance relation [2]). The argument for using such a relation definition is that the relation only

requires an implementation to react correctly to the traces that are explicitly specified in the specification. It leaves freedom to an implementation to react in any manner to traces not specified in the specification.

The behavior of a system can be expressed in term of traces of observable actions. Thus the implementation relation can be expressed through trace preorder between an implementation and a specification. We will consider suspension traces in the definition 1.7 of the **ioco** implementation relation.

This relation assumes that the specifications exists as a LTS with distinguished input and output (but not necessarily as an IOTS). The implementation behavior is modeled as an IOTS (Section 1.4): **ioco** $\subseteq \mathcal{IOTS}(Act_{in}, Act_{out}) \in \mathcal{LTS}(Act_{in} \cup Act_{out})$.

We define the **ioco** relation which uses collected outputs that a system may produce after a *suspension trace*, including quiescent actions. The **ioco** definition requires that any possible output of the implementation should be possible output in the specification after any suspension trace of the specification.

To avoid misundestanding we use notation $p$ for a state instead of $s$ and $P$ stands for a set of states.

**Definition 1.7** [2] Let $p \in P$ be a state of a LTS, an implementation $\mathcal{I} \in \mathcal{IOTS}(Act_{in}, Act_{out})$ with initial state $i$ and a specification $\mathcal{S} \in \mathcal{LTS}(Act_{in} \cup Act_{out})$ with initial state $s$, then:

$$
\begin{aligned}
out(p) &=_{def} \{\alpha \in Act_{out} \cup \{\delta\} |\, p \xrightarrow{\alpha}\} \\
out(P) &=_{def} \bigcup \{out(p) |\, p \in P\} \\
\mathcal{I}\,\mathbf{ioco}\,\mathcal{S} &=_{def} \forall \sigma \in Straces(s) : out(i\,\mathbf{after}\,\sigma) \subseteq out(s\,\mathbf{after}\,\sigma)
\end{aligned}
$$

□

The **ioco** relation restricts inclusion of *out*-sets to suspension traces of the specification and it is used as a formal notation of what a correct implementation should and should not do. The intuition behind the **ioco** implementation relation implies what a correct implementation $\mathcal{I}$ of a specification $\mathcal{S}$ is:

- after the same suspension trace an implementation $\mathcal{I}$ may only produce outputs which are allowed by a specification $\mathcal{S}$, i.e. any output produced by $\mathcal{I}$ must be producible by $\mathcal{S}$ [2];
- if $\mathcal{S}$ refines some output event then $\mathcal{I}$ must produce some output, i.e. if $\delta \notin out(s\,\mathbf{after}\,\sigma)$ then $\mathcal{I}$ must produce some output. [2].

The tool TORX employs the *ioco* relation for test derivation and integrated execution. In the next section we introduce the tool and the algorithm that it used for testing.

## 1.7 TORX

The tool TORX is developed in the Dutch Côte de Resyste project. The tool is based on a formal model of conformance testing. TORX accepts specifications written in the formal specification languages LOTOS, PROMELA and SDL. The semantics of the languages is expressed in terms of labeled transition systems. The test generation algorithm of the TORX tool can be used for on-the-fly or manually driven testing [2].

Figure 5 depicts the main subparts of the tool TORX. The *Explorer* is a component, dependent on the specification language, which offers functions to explore a specification. The *Primer* computes test primitives - the events of the tests that has to be executed. The primer drives the test generation algorithm using the state exploration functions of the *Explorer*. The

*Explorer* and the *Primer* manipulate specification to generate the tests. The *Driver* keeps the on-the-fly test running and controls the flow of a testing process in real-time. The *Adapter* is an implementation specific part which encodes and decodes abstract primitives of test events from and to application specific formats. The *adapter* provides communication between TORX and an implementation. The encoding and decoding functions have to be written manually once for every implementation. The *Implementation* is a real system that should be tested - a piece of software, hardware, or a combination of both [2].

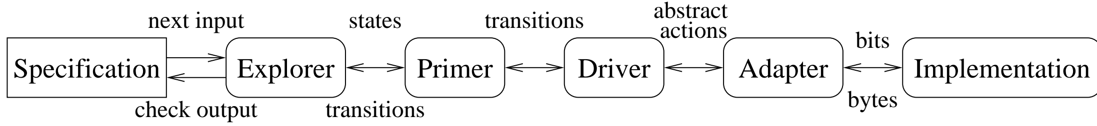



Figure 5: TORX - automated testing tool.

TORX uses the **ioco** relation for test derivation and execution for un-timed systems. Later we adapt this idea for testing real-time systems, which are modeled by timed automata. Therefore we describe the test generation algorithm used by TORX and introduced by Tretmans [2].

The algorithm provides on-the-fly test generation and execution. "The aim of on-the-fly testing is to reduce a number of states and transitions to be considered by using the actual responses of the implementation under test. From a certain state of the specification we need to derive the possible input actions, the expected output actions and the possibility of quiescence"[6]. These are called *test primitives*.

**Definition 1.8** Let S be a set of states in which the specification may be after a particular partial test run, $a \in Act_{in} \cup Act_{out} \cup \{\tau\}$. A *test primitive* is any member of the set $\{a \mid s \in S, s \xrightarrow{a}\} \cap Act_{in} \cup\ out(s)$. □

Algorithm 1 describes the test generation and execution in TORX. Let $\mathcal{S}$ be a specification with initial state $s_0$ and $s_0\,\mathbf{after}\,\epsilon$ is a non empty set of states. Initially the set of $\mathcal{S}$ states is $S = \{s_0\}$. An implementation under test is $\mathcal{I} \in IOTS(Act_{in}, Act_{out})$ and $i, i'$ are states of $\mathcal{I}$.

**Algorithm 1** Test generation and execution in TORX: $TestGenExe(S)$.

- While not (TERMINATE or FAILURE) the tester has choices:
  - Offer an input $a \in \{a \mid s \in S, s \xrightarrow{a}\} \cap Act_{in}$: when $i \xrightarrow{a} i'$ then:
    - $S := S\,\mathbf{after}\,a$ and $i := i'$
    - if $S = \emptyset$ then FAILURE:=true
  - React on quiescence: when $i \xrightarrow{\delta} i$ then
    - if $\delta \in out(S)$ then $S := S\,\mathbf{after}\,\delta$
    - else FAILURE:=true
  - React on an output: when $x \in Act_{out}$ and $i \xrightarrow{x} i'$ then
    - if $x \in out(S)$ then $S := S\,\mathbf{after}\,x$ and $i := i'$
    - else FAILURE:=true
  - Stop testing: TERMINATE:=true
- if FAILURE then return FAIL else return PASS

Algorithm 1 performs on-the-fly testing: it derives test primitives from the specification and at the same time executes these actions on the attached implementation under test. A test case is generated and executed by selecting one of the choices: to offer an input, to react on quiscence or to produce an output.

An implementation is assumed to be **ioco**-conforming to the specification when the algorithm terminates with FAILURE=false. If the algorithm terminates with FAILURE=false then we have a test run which did not produce test failure, i.e. the generated test suite can test for non-conformance, but can not assure conformance.

The algorithm reduces the state space because only the part of the test case used during test execution is derived during on-the-fly testing. The state space although increase during the derivation of test primitives.

We skip the procedures describing how to obtain test primitives and states after input or output events while performing $\delta$-, input-, output-transitions. For the procedures refer to [6]. We describe the corresponding functions and algorithms for state and test primitive computation for timed LTS and symbolic states in 3.2.

## 1.8 Contributions

The following list presents our objectives in this project:

- Generalize testing concepts for real-time systems.
- Extend the idea of automated test generation and execution to be suitable for timed automata. We refer to the TORX tool which provides the test generation and execution based on the *ioco* implementation relation for non-timed system.
- Design a test generation and execution algorithm for on the fly testing for real-time systems.
- Extend the UPPAAL model-checker toolbox with functions for automated test generation for timed automata.
- Adopt and present several examples of testing to assess the test algorithm functionality in a simulated environment.
- Evaluate and present test algorithm performance measurements based on our test examples.
- Discuss the problems of the first test generation and execution algorithm prototype.

## 1.9 Structure of the Report

The remainder of the report is organized as follows. In Section 2 we discuss the test specification structure, properties and theory necessary for the real-time on-the-fly testing. We describe algorithms used in test generation and execution process in Section 3. In Section 4 we discuss the implementation details and issues. Section 5 presents the experiments on chosen virtual implementations under test together with their model descriptions. Finally we summarize our work by outlining the conclusions of this project and further ideas for future in Section 6.

# 2 Test Specification

In this section we extend the timed automata theory with testing concepts. However the testing theory for timed automata is not developed yet and is still an un-opened research issue [12]. We explain how we interpret the **ioco** implementation relation for timed systems in Section 2.3. A system setup is introduced in Section 2.4 for a timed automata before continuing with concepts of testing framework for UPPAAL tool. We propose the real time testing framework in Section 2.5, discuss the input enableness assumptions for the systems to be tested in Section 2.6, discuss the testing specific extensions for timed automata networks, and present the test specification language as an extended UPPAAL timed automata specification language. The on-the-fly test generation and execution algorithms are presented in Section 3 for non-symbolic and symbolic states.

Our work is closely related to TORX and UPPAAL. TORX is an automated test generator for labeled transitions systems. However TORX does not take into account any timing requirements. On another hand UPPAAL deals with timed systems but does not have a capability of generating tests for them.

## 2.1 Timed Automata

Timed automata are used to model finite state real time systems [3]. In this section we present the definition of the timed automaton, explain the semantics of it and introduce a parallel composition operation on timed automata and networks. We stick to the definitions presented in [1].

In order to give the semantics of timed automaton we need to extend LTS (Definition 1.1) with time. Definition 2.1 defines an infinite state timed labeled transition system where the progress of time is modeled by a set of special *delay actions* $\delta \in \mathbb{R}_+$. Execution of delay $\delta$-action means the passage of $\delta$ time units.

### 2.1.1 Timed Labeled Transition System

**Definition 2.1** *Timed Labeled Transition Systems*[1] (TLTS) is a tuple $\langle S, s_0, Act_{\tau\delta}, \rightarrow\rangle$, where

1. $S$ is the set of states,
2. $s_0 \in S$ is the initial state,
3. $Act$ is the set of observable actions, and $Act_{\tau\delta} = Act \cup \{\tau\} \cup \{\delta \mid \delta \in \mathbb{R}_+\}$ ($Act_\delta = Act \cup \{\delta \mid \delta \in \mathbb{R}_+\}$) is the action set with the additional internal $\tau$ and delay $\delta$ actions.
4. $\rightarrow \subseteq S \times Act_{\tau\delta} \times S$ is the transition relation satisfying the following consistency constraints:

   **Time Determinism:** whenever $s \xrightarrow{\delta} s'$ and $s \xrightarrow{\delta} s''$ the $s' = s''$.

   **Time Additivity:** $\forall s, s'' \in S$, $\exists s' \in S$: $s \xrightarrow{\delta_1} s' \xrightarrow{\delta_2} s''$ iff $s \xrightarrow{(\delta_1+\delta_2)} s''$.

   **Null delay:** $\forall s, s' \in S$. $s \xrightarrow{0} s'$ iff $s = s'$.

5. We assume that $Act$ is equiped with a mapping $^-: Act \mapsto Act$ such that for all actions $\bar{\bar{a}} = a$. $\bar{a}$ is said to be the complementary action of $a$.

□

We lift the notation given in Definition 1.1 for LTS to apply to timed LTS, with notable additions. We define several notations that we use to express the state estimation functions for timed labeled transition system.

**Definition 2.2** *TLTS notation* for given TLTS $\langle S, s_0, Act_{\tau\delta}, \rightarrow\rangle$, where $a \in Act$, $\alpha \in Act_{\tau\delta}$, $d \in Act_\delta$ and $S' \subseteq S$:

$$
\begin{array}{lll}
s \xrightarrow{\alpha} s' & =_{def} & (s, \alpha, s') \in \rightarrow \\
s \xrightarrow{\alpha} & =_{def} & \exists s' \colon s \xrightarrow{\alpha} s' \\
s \xrightarrow{\sigma} s' & =_{def} & \exists s_1, s_2 \ldots s_n \colon s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots \xrightarrow{\alpha_n} s_n \text{ and } s_n = s', \\
 & & \text{where } \sigma = \alpha_1 \cdot \cdots \cdot \alpha_n,\ \alpha_i \in Act_{\tau\delta} \\
s \stackrel{\delta}{\Longrightarrow} s' & =_{def} & s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n \text{ such that } s_n = s', \\
 & & \forall i \in [1, n] \colon \alpha_i = \tau \vee \alpha_i = \delta_i, \text{ and } \delta = \sum_{i \mid \alpha_i = \delta_i} \delta_i \\
s' \,\mathbf{after}\, \delta & =_{def} & \{s \mid s \in S : \ s' \stackrel{\delta}{\Longrightarrow} s\} \\
S' \,\mathbf{after}\, \delta & =_{def} & \{s \mid s \in S,\ s' \in S',\ s' \stackrel{\delta}{\Longrightarrow} s\} \\
\xrightarrow{\tau}{}^{*} & =_{def} & \text{the reflexive and transitive closure of } \xrightarrow{\tau} \\
s \stackrel{\epsilon}{\Longrightarrow} s' & =_{def} & s = s' \text{ or } \tau \xrightarrow{\tau}{}^{*} s' \\
s \stackrel{a}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 \in S \colon s \xrightarrow{\epsilon} s_1 \xrightarrow{a} s_2 \xrightarrow{\epsilon} s' \\
s \stackrel{\sigma}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 \ldots s_n \in S \colon s \stackrel{d_1}{\Longrightarrow} s_1 \stackrel{d_2}{\Longrightarrow} s_2 \cdots \stackrel{d_n}{\Longrightarrow} s_n \text{ and } s_n = s', \\
 & & \text{where } \sigma = d_1 \cdot \cdots \cdot d_n \text{ and } d_i \in Act_\delta \\
s \stackrel{\sigma}{\Longrightarrow} & =_{def} & \exists s' \in S \colon s \stackrel{\sigma}{\Longrightarrow} s', \text{ where } \sigma \in Act_\delta{}^* \\
s \stackrel{\sigma}{\not\Longrightarrow} & =_{def} & \nexists s' \in S \colon s \stackrel{\sigma}{\Longrightarrow} s', \text{ where } \sigma \in Act_\delta{}^* \\
s' \,\mathbf{after}\, \sigma & =_{def} & \{s \mid s \in S : \ s' \stackrel{\sigma}{\Longrightarrow} s\}, \text{ where } \sigma \in Act_\delta{}^* \\
S' \,\mathbf{after}\, a & =_{def} & \{s \mid s \in S : \ \exists s' \in S',\ s' \stackrel{a}{\Longrightarrow} s\}
\end{array}
$$

□

### 2.1.2 Definition and Semantics

Informally a timed automaton is an automaton extended with a concept of a clock which defines the timed behavior of the automaton. *Clocks* have positive real *valuations* which evolve at the same rate in the system. However any clock valuation can be reset to a positive integer value. The set of clocks which valuations are to be reset is specified by the set of assignments $R(X)$ of the form $x := c$ where $x \in X$ and $c$ is a non-negative integer. We denote a new clock valuation after a reset $r \subseteq R(X)$ by $\bar{v}' = r(\bar{v})$, where $\bar{v}$ and $\bar{v}'$ are the valuation vectors of all clocks: $\bar{v} = \langle v_{x_1}, \ldots, v_{x_n}\rangle$ and $v' = \langle v'_{x_1}, \ldots, v'_{x_n}\rangle$ where $v'_{x_i} = c_i$ if $(x_i := c_i) \in r$ and $v'_{x_i} = v_{x_i}$ otherwise. We also use a notation $\bar{v}' = \bar{v} + \delta$ to update clock valuations when $\delta$ time passes: $\langle v_{x_1} + \delta, \ldots, v_{x_n} + \delta\rangle = \langle v_{x_1}, \ldots, v_{x_n}\rangle + \delta$.

Timed automaton uses *guards* $G(X)$ over a set of clocks $X$ to allow specification timing constraints. A guard $g \in G(X)$ is specified by grammar $g ::= \gamma \mid g \wedge g$ where $\gamma$ is a constraint of the form $x_1 \sim c$ or $x_1 - x_2 \sim c$ with $\sim \in \{<, \leq, =, \geq, >\}$.

**Definition 2.3** A *timed automaton* $T$ over actions $A$ is a tuple $(L, l_0, X, E, I)$, where [1]:

- $L$ is a non-empty finite set of locations;
- $l_0 \in L$ is the initial location;
- $X$ is a finite set of real-valued clocks that evolve at the same rate;

- $E \subseteq L \times G(X) \times A \times 2^{R(X)} \times L$ is a super set of directed edges, where $G(X)$ is the set of guards, $A$ is a set of actions and $R(X)$ is the set of assignment operations. We write $l \xrightarrow{g,a,r} l'$ if $\langle l, g, a, r, l' \rangle \in E$ to represent a transition from location $l$ to location $l'$ with guard $g \in G(X)$, action $a \in A$ and assignments $r \subseteq R(X)$.

- $I : l \mapsto G(X)$ is the location invariant mapping that gives an invariant $g \in G(X)$ for each location $l \in L$.

- Let $\bar{a}$ denote the complementary action of action $a \in A$ such that $\bar{a!} = a?$ and $\bar{a?} = a!$

□

The semantics of a timed automaton $T$ is defined by associating a timed label transition system $S_T$ with $T$. A *state* $s$ of a timed automaton is a pair $\langle l, \bar{v} \rangle$ where $l{\in}L$ is a location and $v$ is a valuation of all clocks in $X$. The valuation $v$ must always satisfy the invariant constraints in automatons current location $l$: $\bar{v} \models I(l)$. There are two types of transitions in $S_T$:

**Definition 2.4** The *Transitions* for timed automata system $S_T$:

- let $\delta \in \mathbb{R}_+$. We say that $\langle l, \bar{v} \rangle \xrightarrow{\delta} \langle l, \bar{v}' \rangle$ is a $\delta$-*delay* transition, if and only if $\bar{v} + \delta' \models I(l)$ $\forall \delta' \leq \delta$ and $\bar{v}' = \bar{v} + \delta$.

- let $a \in A$. We say $\langle l, \bar{v} \rangle \xrightarrow{a} \langle l', \bar{v}' \rangle$ is an $a$-*action* transition, if and only if an edge $e = \langle l, g, a, r, l' \rangle$ exists such that $\bar{v} \models g$, $\bar{v}' = r(\bar{v})$ and $\bar{v}' \models I(l')$.

□

A *network* of timed automata $N = (T_1 \| \ldots \| T_n)$ is a collection of concurrent timed automata $T_i$ composed by a *parallel composition*. Next we give a semantical meaning to parallely composed networks. A *state of a network* is modeled by a configuration $\langle \bar{l}, \bar{v} \rangle$. The first component is a location vector $\bar{l} = \langle l_1, \ldots, l_n \rangle$ where $l_i$ is a location of automaton $T_i$. We write $\bar{l}[l_i'/l_i]$ to denote the location vector where the $i$-th element of $\bar{l}$ has been replaced by $l_i'$. The second component $\bar{v} \in \mathbb{R}_+^{|X|}$ is the current clock valuation. The *invariant on a location vector* is the conjunction of the invariants on the individual locations: $I(\bar{l}) = \bigwedge_{1 \leq i \leq n} I(l_i)$. The evaluation of a location vector invariant with clock valuation $\bar{v}$ is written $\bar{v} \models I(\bar{l})$. The initial state of the network is $\langle \bar{l_0}, \bar{0} \rangle$, where $\bar{l_0}$ is the vector of initial locations, and $\bar{0}$ is the clock valuation with all clocks being zero.

We define rules for three types of transitions in a network of timed automata in Definition 2.5. Note, that we introduce an internal action $\tau \in A$ for each pair of synchronized transitions, which is important when computing a set of reachable states. We will return to explanation about internal actions after an action definition.

**Definition 2.5** The *Transitions* for timed automata network $N = (T_1 \| \ldots \| T_n)$ are defined by:

- *Action*: If $l_i \xrightarrow{g,a,r} l_i'$ is an action transition in the $i$-th automaton with $g(\bar{v})$, $\bar{v}' \models I(\bar{l}')$ and $a \in A$ then $\langle \bar{l}, \bar{v} \rangle \xrightarrow{a} \langle \bar{l}', \bar{v}' \rangle$ is an action transition in $N$, where $\bar{l}' = \bar{l}[l_i'/l_i]$ and $\bar{v}' = r(\bar{v})$.

- *Synchronization*: if $l_i \xrightarrow{g_1,a,r_1} l_i'$ and $l_j \xrightarrow{g_2,\bar{a},r_2} l_j'$ is synchronized transitions in $i$-th and $j$-th ($i \neq j$) automata with $\bar{v} \models (g_1 \wedge g_2)$ and $\bar{v}' \models I(\bar{l}')$ then $\langle \bar{l}, \bar{v} \rangle \xrightarrow{\tau} \langle \bar{l}', \bar{v}' \rangle$ is an internal action transition in $N$, where $a, \tau \in A$, $\bar{l}' = \bar{l}[l_i'/l_i, l_j'/l_j]$ and $\bar{v}' = (r_1 \cup r_2)(\bar{v})$.

- *Delay*: if $\delta \in \mathbb{R}_+$ is a delay with condition $\forall d < \delta : (\bar{v}+d) \models I(\bar{l})$ then $\langle \bar{l}, \bar{v} \rangle \xrightarrow{\delta} \langle \bar{l}, \bar{v}+\delta \rangle$ is a $\delta$-delay transition in $N$.

□

The network synchronizes with the environment via a set of *observable actions* $A_O \subseteq A$. The set of observable actions $A_O$ is partitioned into input and output actions sets: $A_O = A_{in} \cup A_{out}$ and $A_{in} \cap A_{out} = \emptyset$. The observable action set $A_O$ together with partitioning $A_{in}$ and $A_{out}$ is called an *interface* of the timed automata network. The network synchronizes internally only via hidden *unobservable action* $\tau \notin A_O$ and, i.e. no internal communication is permitted over external, observable actions. Note that the smallest network contains a single automaton and its interface matches all the automaton observable actions.

UPPAAL distinguishes *urgent* and *non-urgent* actions in a network of timed automata. Two automata synchronize over urgent actions immediately whenever the automata are ready for synchronization. Non-urgent actions synchronize at an undefined time, i.e. the time may pass, unless invariants trigger synchronization earlier. As we can see later, urgent actions are not used in our examples (Section 5), we consider urgent action support a low priority and the urgent action implementation in T-UPPAAL remains untested in this project.

Another UPPAAL feature of a timed automaton is *committed locations*. An automata network is forced to perform next action from that location, i.e. a committed location must be left immediately without any interruptions of other transitions. Committed locations are useful for modeling atomic sequences or atomic multi-channel synchronizations.

We denote $T \in N$ if $T = T_i$ for some $i = 1 \dots n$, i.e. automaton $T$ is in network $N$ if and only if there exist equal (by specification) automaton $T_i$ which participates in parallel composition of network $N$.

We extend the parallel composition for timed automata networks. Given $n$ timed automata networks $\{N_i \mid i = 1 \dots n\}$ where $N_i = (T_{1,i} \| \dots \| T_{m_i,i})$, we define the following as *parallel composition on timed automata networks* which is also a timed automata network:

$$N = (N_1 \| \dots \| N_n) = (T_{1,1} \| \dots \| T_{m_1,1} \Big\| T_{1,2} \| \dots \| T_{m_2,2} \Big\| \dots \Big\| T_{1,n} \| \dots \| T_{m_n,n})$$

In general the set of observable actions for a new network is a union of all observable actions: $A_O = \bigcup_i^n \bigcup_j^{m_i} A_O^{ij}$ where $A_O^{ij}$ is the set of observable actions for timed automaton $T_{i,j}$. One may want to hide some observable actions by declaring them unobservable, especially those which are already paired with their complements inside the composition of network: $A_O = (\bigcup_i^n \bigcup_j^{m_i} A_O^{ij}) \setminus A_h$ where $A_h$ is a set of hidden actions which became unobservable after parallel composition. Such explicit action hiding is not supported by UPPAAL and the idea of hidden actions remain only in the theoretical level of the model. We do not introduce a new function for separating the hidden actions after parallel composition, but instead we require the observable actions to be declared in test specification (further discussion in Section 2.9).

We say that a network $N$ is *closed* if all observable actions and their complements are handled inside the network, i.e. $\forall a, \bar{a} \in A_O : \exists \langle l_a, g, a, r, l'_a \rangle \in E_{T_a}$ and $\langle l_{\bar{a}}, g, \bar{a}, r, l'_{\bar{a}} \rangle \in E_{T_{\bar{a}}}$ in some automata $T_a, T_{\bar{a}} \in N$. Such network closing does not necessarily mean that all observable actions must be hidden. Normally you are interested in modeling the closed timed automata networks in UPPAAL, since *open* action synchronizations (only action or its complement is handled inside network) can never be triggered without its complement.

## 2.2 Symbolic Techniques

A timed automata network model has an infinite number of states because of dense real valued clocks. We can see in Figure 6 that an automata passes many states with different clock values and we are interested in all clock values between 0 and 2. Therefore we need compact finite

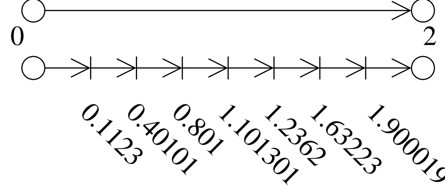



Figure 6: Explanation about a dense clock values.

structures to represent infinite number of clock values. The *symbolic techniques* are used to handle the problem of the infinite state space. For reachability analysis UPPAAL uses *symbolic states* of the form $\langle \bar{l}, z \rangle$, where $\bar{l}$ is a location vector $\langle l_1, \ldots, l_n \rangle$ of $n$ timed automata and $z \subseteq \mathcal{Z}^1$ is a set of clock valuations called a *zone*. A finite set of automata states is called *symbolic states set* $Z = \{\langle \bar{l_1}, z_1 \rangle, \ldots, \langle \bar{l_n}, z_n \rangle\}$.

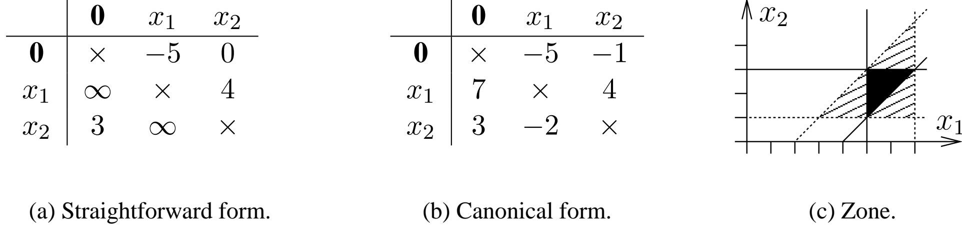
Informally zone is a solution area for the set of inequalities that are the clock constraint representation. Graphically viewed, the zone can be represented by a $n$-dimensional *polyhedron*, where $n$ is the number of clocks. The polyhedron on a finite set of variables is a set of clock valuations. A formal definition of a zone is given in Definition 2.6.

**Definition 2.6** Let $X = x_1, \ldots, x_n$ be a set of clocks. A *zone* $z$ [1] over clocks in $X$ is a constraint system consisting of conjunctions of clock constraints of the following form: $\{x_i - x_j \prec c_{ij} \mid i, j \leq n\} \cup \{a_i \prec x_i\} \cup \{x_i \prec b_i\}$, where $\prec \in \{<, \leq\}$, $c_{ij}$, $a_i$, $b_i$ are integers $\mathbb{Z} \cup \{\infty\}$, and $x_i, x_j \in X$. □

Note that zones are always convex since all constraints cut the area by straight line and the final zone is constructed using intersection of convex zones which gives also a convex zone.

Zones can be represented and manipulated effectively by a *difference bound matrix* (DBM), first applied by Dill in [9]. A DBM represents a clock difference constraints by $(n+1) \times (n+1)$ matrix, where $n$ is the number of clocks. Constraints are in the form $x_i - x_j \leq c_{ij}$, where $x_i, x_j$ are clocks and $c_{ij}$ is a constraint over $x_i$ and $x_j$ difference. A special zero clock **0** with constant value of 0 is used in DBM. Constraints with zero clock represent constraints of the form $x_i \leq c_{i0}$. Hence, the representation of $x_i \leq c_{i0}$ is $x_i - \mathbf{0} \leq c_{i0}$. Note that the lower bound constraints of the form $x_i - x_j \geq c_{ij}$ are rewritten as $x_j - x_i \leq -c_{ij}$ to fit into DBM. Similarly, $x_i \geq c_{0i}$ is rewritten as $\mathbf{0} - x_i \leq -c_{0i}$. Figure 7 shows an example DBM for given constraints on clock values. Closer inspection of the inequalities reveals that other constraints in Figure 7(a) can also be strengthened. We require that identical zones should have identical DBMs to be able to compare zones, such DBMs are called *canonical* and they are unique in a set of DBMs representing the same zone. The canonical DBM constraints have the tightest bounds for all constraints, but still represent the same solution set. Figure 7(b) shows a canonical form for the same constraint and Figure 7(c) shows the filled zone bounded by actual constraints (solid lines) and striped zone bounded by additional constraints (dotted lines) in canonical form.

[1] $\mathcal{Z} \subseteq \mathbb{R}^{|X|}$ is the maximum zone used for extrapolation in reachability analysis. This maximum is limited by the equipment used in verification.



|  | **0** | $x_1$ | $x_2$ |
|---|---|---|---|
| **0** | $\times$ | $-5$ | $0$ |
| $x_1$ | $\infty$ | $\times$ | $4$ |
| $x_2$ | $3$ | $\infty$ | $\times$ |

(a) Straightforward form.

|  | **0** | $x_1$ | $x_2$ |
|---|---|---|---|
| **0** | $\times$ | $-5$ | $-1$ |
| $x_1$ | $7$ | $\times$ | $4$ |
| $x_2$ | $3$ | $-2$ | $\times$ |

(b) Canonical form.



(c) Zone.

Figure 7: DBM representation of the constraint $z = [(x_1 - x_2 \leq 4) \wedge (x_2 \leq 3) \wedge (x_1 \geq 5)]$.

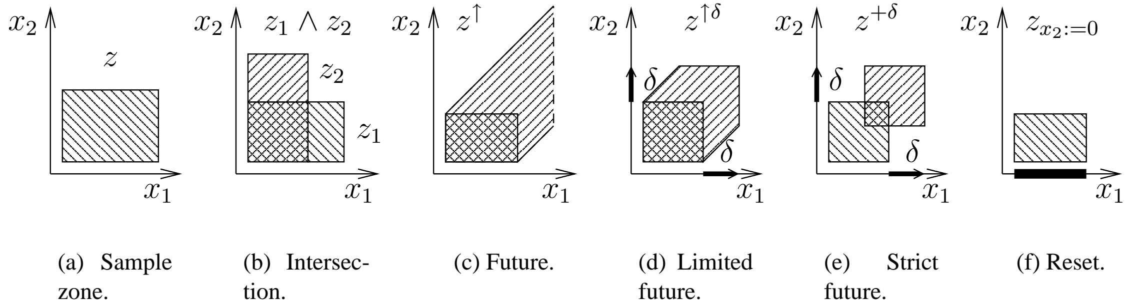
We specify several operations over zones in Definition 2.7 in order to be able to apply symbolic actions on them. The operation $z \wedge z'$ is simply an *intersection* of two zones. $z^{\uparrow}$ contains the *future* of $z$, i.e. the clock value that eventually can be reached from $z$ after unlimited delay. The *limited future* operation $z^{\uparrow d}$ gives a zone which corresponds to reachable clock valuations within a bounded delay $d$. The *strict future* operation $z^{+d}$ gives the expected clock valuations after a given bounded delay $d$. The *reset* of zone $z$ on a clock $x_i$ is a new zone $z' = (z_{x_i:=0})$ where $x_i$ is set to zero and other clocks are unchanged. UPPAAL also supports a clock reset to any non-negative value for convenience purposes.

**Definition 2.7** Let $\bar{v}$ be the automatons current clock valuation of clocks in $X$ and $z, z' \in \mathbb{R}^{|X|}$ be solution zones of constraints over clocks in $X$, then:

$$
\begin{array}{rrll}
\text{Intersection:} & z \wedge z' & =_{def} & z \cap z' = \{\bar{v} \mid \bar{v} \in z \ \wedge \ \bar{v} \in z'\} \\
\text{Future:} & z^{\uparrow} & =_{def} & \{\bar{v} + \delta \mid \bar{v} \in z,\ \delta \in \mathbb{R}_{+}\} \\
\text{Bounded future after } \delta \in \mathbb{R}_{+}\text{:} & z^{\uparrow \delta} & =_{def} & \{\bar{v} + d \mid \bar{v} \in z,\ d \leq \delta\} \\
\text{Strict future after } \delta \in \mathbb{R}_{+}\text{:} & z^{+\delta} & =_{def} & \{\bar{v} + \delta \mid \bar{v} \in z\} \\
\text{Reset:} & z_r & =_{def} & \{r(\bar{v}) \mid \bar{v} \in z\} \text{ where } r \subseteq R(X) \\
\text{Containment:} & z \subseteq z' & =_{def} & \forall \bar{v} \in z,\ \bar{v} \in z' \\
\text{Emptiness:} & z = \emptyset & =_{def} & \nexists \bar{v} \in \mathbb{R}^{|X|} \text{ such that } \bar{v} \in z
\end{array}
$$

□

All operations except bounded future and strict future are already efficiently implemented in UPPAAL. Figure 8 illustrates these operations over zones for $n = 2$ clocks as operations on a 2-dimensional polyhedra.





(a) Sample zone. (b) Intersection. (c) Future. (d) Limited future. (e) Strict future. (f) Reset.

Figure 8: Operations on zones.

We define the semantics of transitions over a symbolic states $\langle \bar{l}, z \rangle$ in Definition 2.8. We refer to [8] for the proof of correctness and soundness of symbolic action transition semantics.

**Definition 2.8** *Symbolic transitions* for timed automata network $N$ with $\gamma \in A$ and $\delta \in \mathbb{R}_+$:

Action: $\langle \bar{l}, z\rangle \overset{\gamma}{\rightarrowtail} \langle \bar{l}', (z \wedge g)_r \wedge I(\bar{l}')\rangle$ if $\bar{l} \xrightarrow{g,\gamma,r} \bar{l}'$ is an (internal, observable or synchronized) $\gamma$-action transition in $N$, $z \wedge g \neq \emptyset$ and $(z \wedge g)_r \wedge I(\bar{l}') \neq \emptyset$.

Delay: $\langle \bar{l}, z\rangle \overset{\delta}{\rightarrowtail} \langle \bar{l}, z^{+\delta} \wedge I(\bar{l})\rangle$ if $z^{+\delta} \wedge I(\bar{l}) \neq \emptyset$.

where $I(\bar{l})$ is the invariant condition on the location vector $\bar{l}$. □

Later in Section 3.2 we use the notations on symbolic states defined in Definition 2.9. The first two notations helps to define next four. The **after** notations are used to denote the reachable symbolic states after observable event in the test generation and execution algorithm and *closure* functions are needed to compute **after** functions.

**Definition 2.9** Let $\langle \bar{l}, z\rangle$ and $\langle \bar{l}', z'\rangle$ be symbolic states and $Z$ be a set of symbolic states:

$\langle \bar{l}, z\rangle \overset{a}{\Longrightarrow} \langle \bar{l}', z'\rangle =_{def} \exists \langle \bar{l}_1, z_1\rangle, \langle \bar{l}_2, z_2\rangle : \langle \bar{l}, z\rangle \overset{\tau_1}{\rightarrowtail} \cdots \overset{\tau_m}{\rightarrowtail} \langle \bar{l}_1, z_1\rangle \overset{a}{\rightarrowtail} \langle \bar{l}_2, z_2\rangle \overset{\tau_{m+1}}{\rightarrowtail} \cdots \overset{\tau_n}{\rightarrowtail} \langle \bar{l}', z'\rangle$, where $\tau_i \in A_U$ and $a \in A_O$.

$\langle \bar{l}, z\rangle \overset{\sigma}{\Longrightarrow} \langle \bar{l}', z'\rangle =_{def} \exists \langle \bar{l}_0, z_0\rangle \ldots \langle \bar{l}_n, z_n\rangle$ such that $\langle \bar{l}_0, z_0\rangle = \langle \bar{l}, z\rangle$, $\langle \bar{l}_n, z_n\rangle = \langle \bar{l}', z'\rangle$ and $\langle \bar{l}_{i-1}, z_{i-1}\rangle \overset{a_i}{\Longrightarrow} \langle \bar{l}_i, z_i\rangle$, where $\sigma = a_1 a_2 \ldots a_n$ with $a_i \in \mathbb{Z}_+$ or $a_i \in A_O$.

$Z\,\mathbf{after}\,a =_{def} \{\langle \bar{l}', z'\rangle \mid \langle \bar{l}, z\rangle \in Z,\ \langle \bar{l}, z\rangle \overset{a}{\Longrightarrow} \langle \bar{l}', z'\rangle\}$.

$Z\,\mathbf{after}\,\delta =_{def} \{\langle \bar{l}', z'\rangle \mid \langle \bar{l}, z\rangle \in Z,\ \langle \bar{l}, z\rangle \overset{\sigma}{\Longrightarrow} \langle \bar{l}', z'\rangle,\ \forall \sigma = \gamma_1\gamma_2\ldots\gamma_n : \gamma_i \in A_U \vee \gamma_i \in \mathbb{Z}_+ \text{ and } \delta = \sum_{\gamma_i \in \mathbb{Z}_+} \gamma_i\}$.

$\tau\text{-}closure(Z) =_{def} \{\langle \bar{l}', z'\rangle \mid \langle \bar{l}, z\rangle \in Z,\ \langle \bar{l}, z\rangle \overset{\tau_1}{\rightarrowtail} \cdots \overset{\tau_n}{\rightarrowtail} \langle \bar{l}', z'\rangle,\ \tau_i \in A_U\}$.

$\delta\tau\text{-}closure(Z) =_{def} \{\langle \bar{l}', z'\rangle \mid \langle \bar{l}, z\rangle \in Z,\ \langle \bar{l}, z\rangle \overset{\sigma}{\Longrightarrow} \langle \bar{l}', z'\rangle,\ \forall \sigma = \gamma_1\gamma_2\ldots\gamma_n : \gamma_i \in A_U \vee \gamma_i \in \mathbb{Z}_+\}$.

□

Using symbolic states we get a finite set of symbolic states after an action or a delay. We can explain that as follows:

- we get finite traces $\sigma$ as they consists of delays which are sums of integer valued intervals $\gamma$;
- we do not allow Zeno traces, i.e. during a bounded time interval a system can perform a bounded number of actions.

Note that the "$Z$ **after**" definition for a symbolic state set is compatible with the "$S$ **after**" definition for a state set, i.e. if the symbolic state set $Z$ includes the state set $S$ – $\forall \langle \bar{l}, \bar{v}\rangle \in S\ \exists \langle \bar{l}, z\rangle \in Z.(\bar{v} \in z)$ – then $(Z\,\mathbf{after}\,\delta)$ includes $(S\,\mathbf{after}\,\delta)$ and $(Z\,\mathbf{after}\,a)$ includes $(S\,\mathbf{after}\,a)$.

## 2.3 Implementation Relation for Real Time Systems

Testing of real time systems requires a different approach of testing theory than the one for non-timed systems, because of additional time sensitive constraint in a model. We describe a testing relation for a timed automata network and illustrate the properties for the implementation relation that would allow us to discriminate real time systems for testing them.

An *implementation relation* is the correctness criterion used in automated testing theory. It defines the correctness for an implementation with respect to a given specification. We interpret the implementation relation as follows: an implementation $i \in \mathcal{IOTS}(Act_{in} \cup Act_{out})$ is only

allowed the behavior, in terms of timed traces of observable actions, which is prescribed by the specification $s \in \mathcal{TLTS}(Act \cup \{\delta\})$.

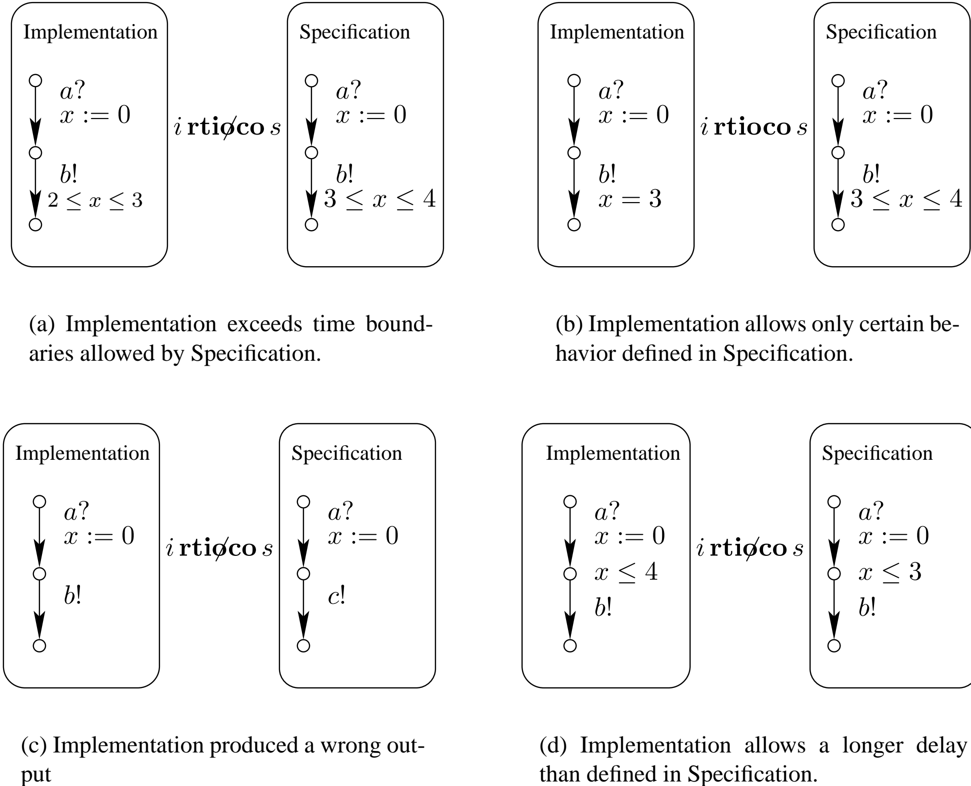
In the untimed systems quiescence can be approximated and implemented as a finite time-out [2]. In the timed case we can only observe that the implementation was quiescent for a bounded amount of time, corresponding to observing the passage of finite delays. We define timed traces and *rt-ioco* implementation relation expressed in terms of output sets after timed traces where a delay $\delta \in \mathbb{R}_+$ is non-negative real number.

**Definition 2.10** Let $p \in S$ be a state of a TLTS, an implementation $i \in \mathcal{TIOTS}(Act_{in}, Act_{out})$ and a specification $s \in \mathcal{TLTS}(Act_{in} \cup Act_{out} \cup \{\delta\})$ where $\delta \in \mathbb{R}_+$ is a time delay, then

$$
\begin{array}{lll}
ttraces(p) & =_{def} & \{\sigma \in (Act \cup \{\delta\})^* \mid p \stackrel{\sigma}{\Rightarrow}\} \\
out(p) & =_{def} & \{\alpha \in Act_{out} \cup \{\delta\} \mid p \stackrel{\alpha}{\rightarrow}\} \\
out(S) & =_{def} & \bigcup \{out(p) \mid p \in S\} \\
i\,\mathbf{rtioco}\,s & =_{def} & \forall \sigma \in ttraces(s)\colon out(i\,\mathbf{after}\,\sigma) \subseteq out(s\,\mathbf{after}\,\sigma)
\end{array}
$$

□

From the definition 2.10 follows that $\sigma \in ttraces(p)$ iff $\delta \in out(p\,\mathbf{after}\,\sigma)$ or $out(p\,\mathbf{after}\,\sigma) \neq \emptyset$. We show through the examples in Figure 9 how the $\mathbf{rtioco}$ implementation relation can be applied for real time systems as a relation between timed traces of the implementation and the specification. Any erroneous behavior is non-conforming and any correct is not necessary conforming.




(a) Implementation exceeds time boundaries allowed by Specification.

(b) Implementation allows only certain behavior defined in Specification.

(c) Implementation produced a wrong output

(d) Implementation allows a longer delay than defined in Specification.

Figure 9: Examples with $\mathbf{rtioco}$ relation for TLTS.

In Figure 9(a) we illustrate that the sample implementation does not conform to the specification. The specification require to produce the output $b$ when $x \in [3; 4]$ but in the im-

plementation the output $b$ is permitted only when $x \in [2;3]$. The implementation can do an output transition earlier than it is defined in the specification, e.g. $out(i\,\mathbf{after}\,a.\delta(2)) \not\subseteq out((s\,\mathbf{after}\,a.\delta(2))$. In Figure 9(b) there is a timed trace when output $b$ is produced when $x = 3$ and $out(i\,\mathbf{after}\,a.\delta(3)) \subseteq out((s\,\mathbf{after}\,a.\delta(3))$ therefore the implementation is in the $\mathbf{rtioco}$ relation with the specification.

In Figure 9(c) the implementation does not conform to the specification because the implementation does not produce the permissible output. In Figure 9(d) the relation is non-conforming because the implementation can produce the output $b$ when $x = 4$, while the specification requires to produce the output not later than $x \leq 3$.

Based on the examples we declare that we consider $\mathbf{rtioco}$ implementation relation, which implies that:

- an output from the implementation is erroneous if the specification does not allow the output;
- timed traces of an implementation with delay transitions which violates the maximum or minimum delay defined in the specification show non-conforming behavior of the implementation.

The above implications assure that the implementation behaves safe, i.e. does not do more than required by the specification and the outputs required by a specification are also produced by an implementation. Before continuing with testing framework concepts we describe a physical system setup in the next section.

## 2.4 Physical System Setup

Figure 10 shows a real system setup where an implementation fits in its environment. The implementation communicates with its environment through observable actions: input $A_{in} \subseteq A_O$ and output $A_{out} \subseteq A_O$, where $A_O = A_{in} \cup A_{out}$ and $A_{in} \cap A_{out} = \emptyset$. We name actions as input or output from the implementations point of view if it is not noted otherwise. In our case the *output actions* are the actions controlled by the implementation ($A_{out}$ in Figure 10) and the *input actions* are controlled by the environment ($A_{in}$). In practice environment and implementation automata and variables are isolated from each other (if they exist at all) and access to them is limited. The only way to communicate with the implementation are $A_{in}$ and $A_{out}$ channel synchronizations combined with shared global variables. To make this communication real, we extend the UPPAAL notion of an action with data transmition in Section 2.9.

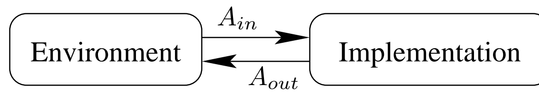



Figure 10: Real physical system setting.

The goal of the testing is to verify the implementation conformance to the specification by using these input/output actions together with the timing specification, i.e. by simulating the behavior of the environment.

## 2.5 Real Time Testing Framework

An implementation under test may have an unspecified environment. If the test specification omits the model of the environment we assume a fully permissive environment. A *fully per-*

*missive* environment may offer any input defined in the implementation interface at any time. We also consider an option to model a specific environment since we might be interrested in the implementation behavior only under specific environmental conditions. The model of the environment may also be used to guide a test to some specific situations like a specific use case of the system.

The above implies the UPPAAL tool extension to the on-the-fly test generation tool which we call T-UPPAAL (Testing-UPPAAL) with a following test setup shown in Figure 11:

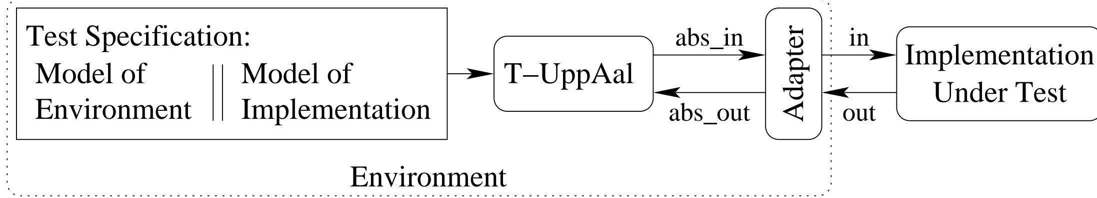



Figure 11: Automated test generation setup with T-UPPAAL.

1. The *Test Specification* is a model of the environment and the implementation written in the UPPAAL timed automata's language with test specific extensions which are discussed further in this section.

2. The T-UPPAAL reads the test specification and controls the automated testing: offers an input and receives an output to and from adapter in abstract encoded form, e.g. character strings "insert coin", "give coffee". T-UPPAAL corresponds to *Explorer*, *Primer* and *Driver* in TORX (see Figure 5).

3. The *Adapter* is an implementation dependent event encoder/decoder which immediately translates the abstract encoded actions (e.g. character string "insert coin") to a physical input to the implementation (e.g. inserts a coin) and translates a physical output from the implementation (e.g. coffee outcome) to their abstract encoded representation (e.g. character string "give coffee").

4. The *Implementation Under Test* (IUT) is a physical subsystem or a device to be tested, e.g. coffee machine. The IUT model is given in the test specification, several examples are presented in Figure 9.

## 2.6 Input Enableness and Specification Completeness

In this section we discuss our assumptions on the implementation model properties. We assume that the implementation is always capable of synchronizing with an input offered by the environment. The first requirement comes that a model of implementation should not contain internal action cycles, which would allow infinite loops ignoring the input action transitions. In other words we require that a timed labeled transition system for implementation is strongly converging. The formal definitions for stable state and strongly converging TLTS are given in Definition 2.11 and Definition 2.12.

**Definition 2.11** Let $N$ be a timed automata network and $\mathcal{TLTS}(N) = \langle S, s_0, Act_{\tau\delta}, \rightarrow \rangle$ is a TLTS for $N$. A state $s \in S$ is called *stable* if no internal $\tau$ transition is specified for it. The set of *stable states* is denoted by $StableStates(S) = \{s \in S \mid s \stackrel{\tau}{\not\rightarrow}\}$. □

**Definition 2.12** TLTS $\langle S, s_0, Act_{\delta\tau}, \rightarrow\rangle$ is *strongly converging* if only a finite sequence of internal transitions is possible to take beginning with any state, i.e. a stable state $s'$ is reached eventually: $\forall s \in S\ \exists \tau_i \in A_U.\ s \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} s'$ and $\nexists \tau \in A_U.\ s' \stackrel{\tau}{\nrightarrow}$. □

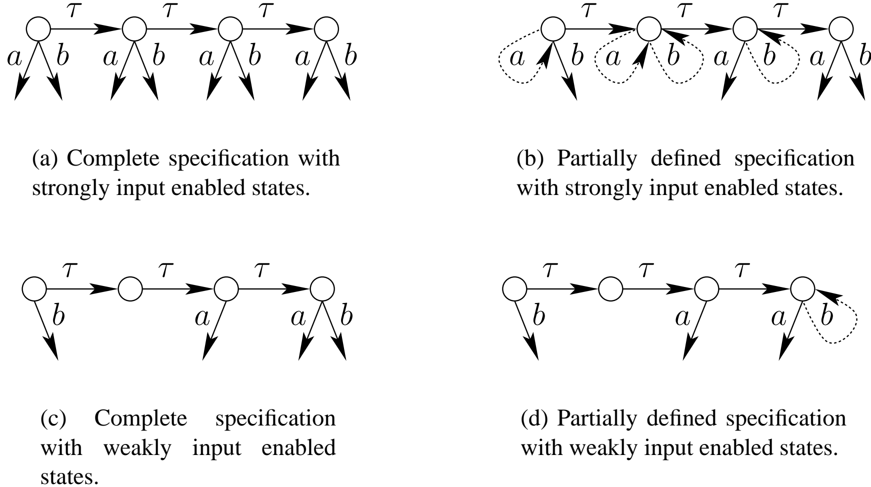
**Definition 2.13** Let $N$ be a timed automata network and $\mathcal{TLTS}(N) = \langle S, s_0, Act_{\tau\delta}, \rightarrow\rangle$ is a TLTS for $N$. The network $N$ is *strongly input enabled* if it specifies all input action transitions for all possible states: $\forall s \in S, \forall a \in A_{in}.\ s \xrightarrow{a}$. □

**Definition 2.14** Let $N$ be a timed automata network and $\mathcal{TLTS}(N) = \langle S, s_0, Act_{\tau\delta}, \rightarrow\rangle$ is a TLTS for $N$. The network $N$ is *weakly input enabled* if it specifies all input action transitions for all possible states within an arbitrary number of internal $\tau$ action transitions: $\forall s \in S, \forall a \in A_{in}.\ s \stackrel{a}{\Longrightarrow}$. □

**Theorem 2.1** Let $N$ be a timed automata network and $\mathcal{TLTS}(N) = \langle S, s_0, Act_{\tau\delta}, \rightarrow\rangle$ is a strongly converging TLTS for $N$. The network $N$ is *weakly input enabled* if and only if all input action transitions are specified for all stable states: $\forall s \in StableStates(S), \forall a \in A_{in}.\ s \xrightarrow{a}$. □

However not all specifications are (strongly or weakly) input enabled, but there are techniques to make them input enabled. We say that timed automata network specification is *complete* if it is (strongly or weakly) input enabled. To lower the requirement we allow the partially defined specifications. We say that timed automata network is *partially* defined if the specification is not (strongly or weakly) input enabled but there are rules how to make such specification complete.

Figure 12 shows four combinations of different input enabliness assumptions for TLTS where $\tau$ is internal transition and $a, b \in A_{in}$ are input actions which are not bounded to any data or clock variables:



(a) Complete specification with strongly input enabled states.

(b) Partially defined specification with strongly input enabled states.

(c) Complete specification with weakly input enabled states.

(d) Partially defined specification with weakly input enabled states.

Figure 12: Four combinations of assumptions about TLTS states.

a) If we require the specification to be strongly input enabled then a complete specification has to include all input action transitions for every possible state (see LTS example in Figure 12(a)).

b) If we do not define all the input action transitions for all possible states we assume that automaton ignores the input and stays in the same state (dotted transitions in Figure 12(b)).

c) If we require just weak input enabliness then the complete specification is required to have input action transitions defined in stable states (Figure 12(c)).

d) And for the partially defined specification with weak input enabliness we assume that only stable states ignores the unspecified input actions (dotted transition in Figure 12(d)).

In partial specifications we assume that the implementation is ignoring the input if the input action transition is not specified: the implementation makes the input action transition to the same location as it was before and only the certain variable values are overwritten with the values transfered together with the input action. If the input actions are bounded to some clock or data variables then the implementation will be forced to move to another state with the same location vector and updated variable values (which is not the case in Figures 12(b) and 12(d)).

An implementation can not refuse input actions offered by the environment. In general the implementation may ignore the input offered and no response is also a legal response. We assume that the implementation synchronizes its input action transition immediately with an input offered by the environment and we assume that implementation is complete and strongly input enabled. Notice that any partially defined model of implementation can be easily converted into complete strongly input-enabled one by adding a timed automaton which handles input actions independently from the rest of the implementation. These additional automata can be used to model monitors or display screens, but they should not be used in general since they introduce additional non-determinism.

Models of implementation are rarely input enabled out of the box, therefore it is desired that we can easily adapt them for a test specification. There are two choices for doing that according to what kind of input enableness we want:

1. If we consider a model to be partially defined then
   - we may choose methods shown in Figure 12(b) to make it strongly input enabled;
   - we may use methods shown in Figure 12(d) if we want a weakly input enabled model.
2. If we assume a model to be complete then the model creator should consider reviewing the model when something goes wrong during testing (tool prompts that there are no observable actions enabled). Model verification could be used to check whether some or all inputs are enabled in all states.

We assume that the environment network synchronizes immediately with output produced by the implementation. We allow the environment model not to be (strongly or weakly) input enabled, i.e. there might be a state of the environment where a transition on some implementation output action is not specified. In practice, the test is *inconclusive* if the implementation gives an output which cannot be synchronized with an action transition in the environment model current state at a current moment. Inconclusiveness may occur due to as-synchronization of clocks between an environment and the implementation therefore we assume that clocks are synchronous. The verdict *inconclusive* means that the goal of the test was not reached due to unexpected (by the environment) conditions, but it does not mean that implementation succeeded or failed to comply with the specification.

## 2.7 UPPAAL

UPPAAL[7] is a toolbox for symbolic simulation and automatic verification (via automatic model checking) of real time systems modeled as networks of extended timed automata.

The tool provides reachability analysis (described in Section 2.2) for automatic verification of properties for real-time systems. It contains a number of additional features including graphical interfaces for editing and simulating system models.

The UPPAAL model checker tool consists of the programs (Figure 13): graphical user interface (GUI) and the model checker engine (server). UPPAAL GUI provides cross-platform user

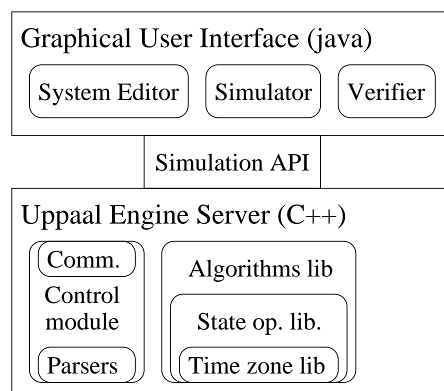



Figure 13: UPPAAL component architecture.

interaction build on Java. UPPAAL server provides an efficient computation of a system symbolic state after given transition and verification result for a given property. The UPPAAL GUI (re)starts the server program whenever user chooses to update simulator and/or verifier with a new system model. The GUI and the server communicate through TCP/IP socket connection established automatically after the server start. The UPPAAL GUI has three parts:

- The **System Editor** allows the user to describe and edit the timed-automata system. The system timed automata consists of global declarations, a timed-automaton templates, process assignment and system definition sections.
- The **Simulator** allows the user to virtually interact with the system described. The simulator shows the system state by displaying the states of compound automata and the values of variables. The simulator allows the user to choose enabled transitions manually or randomly. It also has a feature of displaying the history of events in sequence chart.
- The **Verifier** accepts the user formulated properties to be verified on a particular timed-automata model, and displays the result of verification: true or false depending on whether the property was satisfied or not, and an event trace example if the property proof requires one.

UPPAAL server consists of many libraries, but in particular our project overlaps with the following:

- DBM library provides efficient operations on time zones.
- System state operation library used for timed automata network state and transitions representation.

- Reachability algorithms used for property verification.
- Parser library provides dynamic loading and saving of the system specification to be extended with test specific additions.
- Communication (API) library used to connect GUI and transfer the system specification with verification properties. The API library is to be reused for communication with real implementation under test.

UPPAAL uses templates to construct a network of timed automata. A UPPAAL template resembles a timed automaton with additional features: integer data variables and arrays of such variables, urgent channels and committed locations.

Data variables [8] do not change their values at the delay-transitions as the clock variables do; they can only be assigned to values from finite domains (bounded integers in our case), and therefore they do not cause infinite-stateness. Data variables form *non-clock constraints* which are similar to *clock constraints*. $G(D)$ is a set of *non-clock constraints*, where $D$ is a set of data variables. A *non-clock constraint* $g$ is defined by grammar $g ::= v(d) \sim k \mid v(d) * v(b) \sim k \mid g_1 \wedge g_2$, where $v$ is the value of the data variable $b, d \in D$, $k \in \mathbb{Z}$, $\sim \in \{<, \leq, =, \neq, \geq, >\}$ and $* \in \{+, -, \times, /\}$. We denote $d \in g$ when variable $d \in D$ participate in guard $g \in G(D)$. A set of assignment operations $R$ is also extended for data variables. We denote $d \in r$ when variable $d \in D$ participate in assignment operation $r \in R$.

## 2.8 UPPAAL Network Specification

A UPPAAL timed automata specification describes *closed* timed automata networks where the environment and the implementation models are embedded in a parallel composition. Both the environment and the implementation models can also be timed automata composed in parallel, forming separate networks even if the separation is not always obvious when looking at the UPPAAL model specification. Figure 14 gives an example of UPPAAL timed automata network

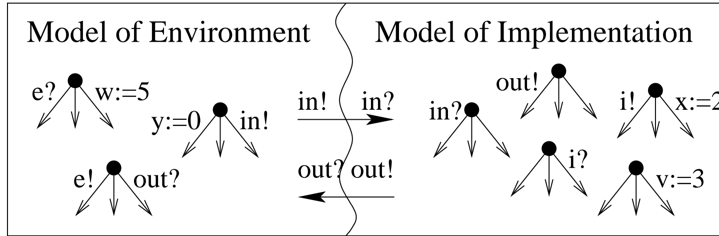



Figure 14: Environment and implementation models embedded into a single network model.

model where the model of the environment and the model of the implementation are both integrated into a single network. The environment consists of three parallely composed timed automata and the implementation consists of five automata. Both the environment and the implementation networks synchronize through $A_{in}$ and $A_{out}$ sets of channels and communicate through shared global variables forming a complete system network model.

The concepts involved in a UPPAAL timed automata network model specification are shown in a class diagram in Figure 15. Timed automata are treated as process definitions in UPPAAL, since they are used to model processes. A network specification consists of four sections: 1) global variable, clock and channel declarations, 2) the process template section eases the modeling task and lets user to specify a class of localy equal automata 3) the process instantiation

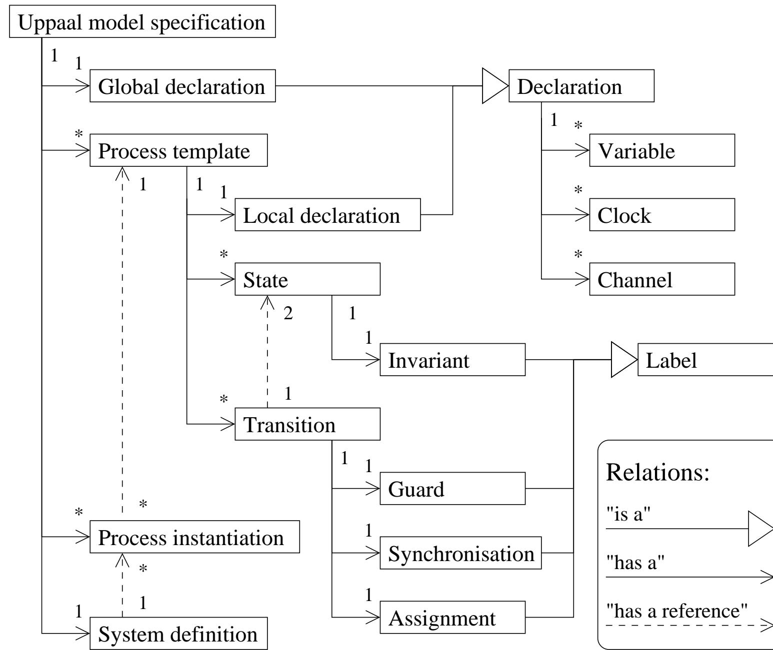



Figure 15: UPPAAL timed automata network specification class diagram.

which defines the final automata design and connectivity and finally 4) the system definition tells what automata are involved in network.

Channels denote the actions which "connect" and synchronize automata. A process template may have its own variables and clocks declared in the local declaration, however local channels do not have meaning since local items are not accessible from outside of the automaton. The process template specifies automaton's states with invariants and transitions between them with guards, action synchronizations and assignment statements. The process instantiation section references the global declaration items and process templates and defines concrete automata. System definition references constructed automata from the process instantiation and defines the construction of a final automata network.

## 2.9 Test Specification Grammar

We assume that the only way to communicate with the implementation is its interface: action sets $A_{in}$ and $A_{out}$. The clocks and variables of the implementation are isolated from those of the environment and values can be exchanged only by observable actions $A_O = A_{in} \cup A_{out}$. In our case each action is a channel synchronization associated with some variable values to be transfered to the implementation and back to allow value passing. Having the assumptions above we conclude that a test specification contains:

1. UPPAAL network specification as a parallel composition of implementation and environment networks.

2. *Input* and *output channel* synchronizations which form an interface of the implementation to the environment.

3. Data variables associated with *input* and *output channels*.

```
       Network ::= <Declaration>* <Inst>* <System>
          Inst ::= ID '=' ID '(' <ArgList> ')' ';'
        System ::= 'system' ID (',' ID)* ';'
   Declaration ::= <VariableDecl> | <ProcDecl>

  VariableDecl ::= <Type> <DeclId> (',' <DeclId>)* ';'
        DeclId ::= ID [ '=' <Expression> ]
          Type ::= <Prefix> ('clock' | 'int' | 'channel')
        Prefix ::= ( [ 'urgent' ] [ 'broadcast' ] | ['const'] )

      ProcDecl ::= 'process' ID <ParameterList> '{' <ProcBody> '}'
      ProcBody ::= <VariableDecl>* <States> [<Commit>] [<Urgent>]
                   <Init> [<Transitions>]

        States ::= 'state' <StateDecl> (',' <StateDecl>)* ';'
     StateDecl ::= ID [ '{' <Expression> '}' ]

        Commit ::= 'commit' StateList ';'
        Urgent ::= 'urgent' StateList ';'
     StateList ::= ID (',' ID)*

          Init ::= 'init' ID ';'

   Transitions ::= 'trans' <Transition> (',' <TransitionOpt>)* ';'
    Transition ::= ID '->' ID <TransitionBody>
 TransitionOpt ::= Transition | '->' ID <TransitionBody>
TransitionBody ::= '{' [<Guard>] [<Sync>] [<Assign>] '}'

         Guard ::= 'guard' <Expression> ';'
          Sync ::= 'sync' <Expression> ('!' | '?') ';'
        Assign ::= 'assign' <ExprList> ';'
```

Figure 16: UPPAAL network specification language core rules.

A UPPAAL specification includes global variables declaration without telling which variables actually belong to the implementation neither specifies which automata are used in the implementation. To be able to simulate the environment we need to specify it explicitly to separate the implementation from the environment. We split the UPPAAL network model $N = (T_1 \| \ldots \| T_n)$ into following parts:

1. *Implementation network* $N_I$ is a parallel composition of $i$ timed automata which specify the model of the implementation under test: $N_I = (T_{I_1} \| \ldots \| T_{I_i})$.

2. *Environment network* $N_E$ is a parallel composition of $j$ timed automata which specify the model of the environment for the implementation to be tested in: $N_E = (T_{E_1} \| \ldots \| T_{E_j})$.

3. *Implementation variables* $D_I$ are controlled (assigned) by the implementation automata: $D_I = \{d \mid \exists T = (L, l_0, X, D, E, I) \in N_I,\ \exists e = \langle l, g, a, r, l' \rangle \in E :\ (d := exp) \in r\}$, where $exp$ is an integer expression, i.e. the implementation variable is the variable $d$ for which there exists an automaton $T$ in the implementation network of timed automata with an edge $e$ containing an assignemt to $d$.

4. *Environment variables* $D_E$ are controlled (assigned) by the environment automata: $D_E = \{d \mid \exists T = (L, l_0, X, D, E, I) \in N_E,\ \exists e = \langle l, g, a, r, l' \rangle \in E :\ (d := exp) \in r\}$, where

$exp$ is an integer expression, i.e. the environment variable is the variable $d$ for which there exists an automaton $T$ in the environment network of timed automata with an edge $e$ containing an assignemt to $d$.

Notice that so far we do not care about the relations between sets $N_I$ and $N_E$, $D_I$ and $D_E$, we consider it later when defining what is a separable UPPAAL network.

We have separated the implementation and the environment models and now we need to make sure that shared variable values are synchronized in the implementation and the environment. There are two alternative approaches to synchronize the data values between the implementation and the environment:

1. *Shared memory approach* suggests to synchronize variable values as soon as possible.i.e. right after value assignment. For that purpose we attach a separate output channel synchronization for each variable assignment inside the implementation model and send the new variable value to the environment together with the output channel synchronization. The same rule applies for the implementation model attaching an additional input channel synchronization. This case explained in the train-gate example in Section 5.2.

2. *Value passing approach* suggests to postpone data variable synchronization until next channel synchronization and for each channel attach sets of variables which values are to be transmitted just before synchronization. Look for explanation in Section 5.2.

The shared memory approach makes sure that variable values in the implementation and the environment are the same at all times. It is not always possible to achieve that in practice (variables may exist only in models) and the communication efficiency is considered to be poor comparing with the second approach. The message passing approach minimizes the amount of communications and enables only the final computation values to be transmitted.

Having discussed the options we choose to implement the second method. We are ready to define what input and output data is actually transmitted during synchronizations.

1. *Output variables* is a function $\Delta_{out} : A_{out} \mapsto 2^{D_I}$ which assigns a subset of implementation variables to an observable output action from a set $A_{out} \subseteq A_O$.

2. *Input variables* is a function $\Delta_{in} : A_{in} \mapsto 2^{D_E}$ which assigns a subset of environment variables to an observable input action from a set $A_{in} \subseteq A_O$.

Figure 17 describes the data transmission protocol during input action synchronization in four basic steps:

1. The environment network of timed automata $N_E = T_E = (\{l_0^E, l_1^E\}, l_0^E, \{d_a\}, \{l_0^E \xrightarrow{in!} l_1^E\}, \{I(l_0^E) = true, I(l_1^E) = true\})$ and implementation network of timed automata $N_I = T_I = (\{l_a^I, l_b^I\}, l_a^I, \{d_a\}, \{l_a^I \xrightarrow{in?} l_b^I\}, \{I(l_a^I) = true, I(l_b^I) = true\})$. are in the initial states. Variable $d_a$ is bounded to the channel $a$, i.e. $\Delta(a) = \{d_a\}$.

2. The environment decides to take a transition $l_0^E \xrightarrow{a!} l_1^E$ and offers an input $a(d_a = 2)$.

3. The implementation receives action $a(d_a = 2)$, updates the variable $d$ value to 2 and decides which transition to take.

4. The environment and implementation state after the synchronous transition.

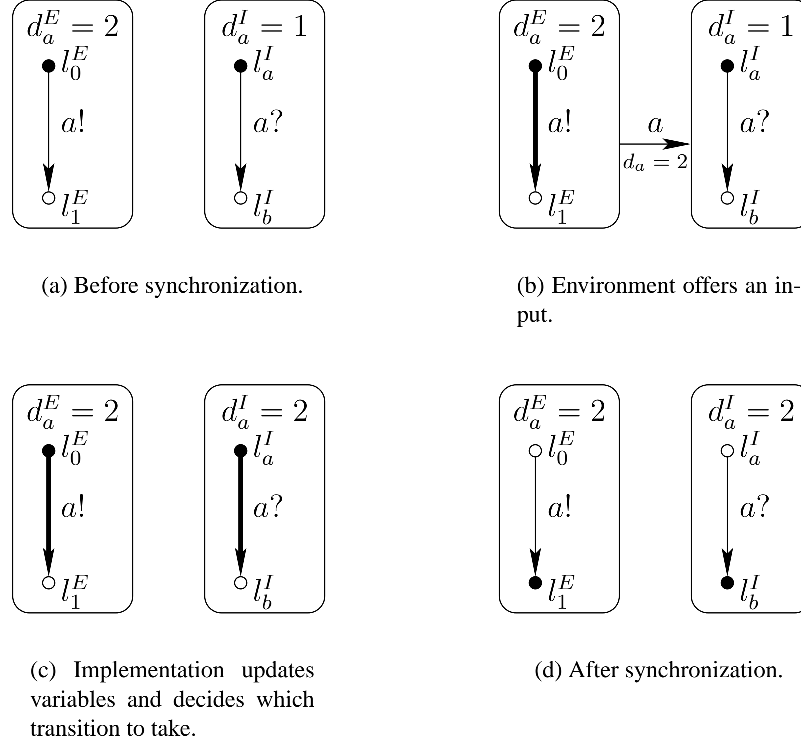



Figure 17: Protocol of channel synchronization with data transmission.

The separation of the environment and the implementation becomes obvious after having discussed the ownership of variables in a composed specification. Definition 2.15 summarizes the conditions for the specification to be separable into models of the environment and the implementation. The first two requirements are obvious: we want timed automata and data variables to be strictly distinguished between models. The next two requirements restricts the use of the implementation variables in the model of environment and vice a versa, i.e. in the model of the environment the implementation variables can be used only in synchronized output action transitions, otherwise we would risk to operate on the environment model with outdated implementation variable data, the same must hold for the model of the environment.

**Definition 2.15** We say that the UPPAAL network $N$ is *separable* into the implementation $N_I$ and the environment $N_E$ sub-networks if the following criteria are fulfilled:

1. UPPAAL network is closed $N = (N_I \| N_E)$ and automata belong either to the implementation or the environment: $N_I \cap N_E = \{T \mid T \in N_I \wedge T \in N_E\} = \emptyset$,
2. UPPAAL global variables belong exclusively to either the implementation or the environment: $D_I \cap D_E = \emptyset$.
3. The environment variable value can be accessed through dedicated action synchronization. The same applies to the implementation variables.
4. The implementation network does use the value of the variable belonging to environment, except during the dedicated action synchronization. The same applies to environment in respect to the variables belonging to implementation.

□

**Theorem 2.2** UPPAAL network $N$ is *separable* into implementation $N_E$ and environment $N_I$ networks if and only if:

1. the value of a data variable $d$ is transmited from implementation during $a$-action synchronization, i.e. $\forall d \in D_I$ and $\forall e = \langle l, g, a, r, l' \rangle \in E_E$ where $d$ is used in the guard $g$, the invariant $I_E(l')$ or the assignment $r$ of the edge $e$ if and only if $d \in \Delta_{out}(a)$.
2. $\forall d \in D_E$, $\forall e = \langle l, g, a, r, l' \rangle \in E_I$ where $d$ is used in the guard $g$, the invariant $I_I(l')$ or the assignment $r$ of the edge $e$ if and only if the value of $d$ is sent to implementation during $a$-action synchronization: $d \in \Delta_{in}(a)$.

□

Once we have the separable model of the implementation and the environment we are ready to define what is a test specification:

**Definition 2.16** A *test specification* TestSpec for a given separatable timed automata network $N = (N_E \| N_I)$ is a tuple $\langle N, A_{in}, A_{out}, \Delta_{in}, \Delta_{out}, \mu, \Omega \rangle$ where:

- $A_{in} \cup A_{out} = A_O$ and $A_O$ is an observable action set for $N$,
- $\mu \in \mathbb{Q}_+$ is the smallest time value in seconds, which corresponds to one time unit. We allow the implementation to have up to $\pm\frac{\mu}{2}$ seconds deviation in test execution.
- $\Omega \in \mathbb{N}$ is the amount of $\mu$ time units for a timeout.

□

We define the $TestSpec$ language by extending the UPPAAL timed automata network language to contain the test information in Figure 18.

```
 TestSpec ::= <Network> <Interface> <Precision> <Timeout>
Interface ::= 'input' [ <Action> (',' <Action>)* ] ';'
              'output' <Action> (',' <Action>)* ';'
   Action ::= <Channel> '(' [ <Variable> (',' <Variable>)* ] ')'
  Channel ::= ID
 Variable ::= ID
Precision ::= 'precision' Rational ';'
  Timeout ::= 'timeout' Integer ';'
```

Figure 18: Test specification language grammar.

# 3 Algorithms

Given the test generation and execution algorithm for TORX in Section 1.7 and the description of TLTS and timed automata in Section 2.1 we continue with the test generation and execution algorithm for real time systems.

Our goal is to generate test runs that would effectively check whether the implementation conforms to the specification, i.e. the implementation and the specification are in the **rtioco** implementation relation.

We present two versions of the test generation algorithms: 1) computing the reachable states of TLTS (Section 3.1) and 2) computing the reachable symbolic states of timed automata network. Actually the first version of the test generation and execution algorithm is not computable in practice, but might be used to prove the correctness of the second one.

## 3.1 Test Execution on Reachable States

We introduce reachable state *estimation functions*. The state estimation functions compute a set of reachable states after applying a sequence of observable or unobservable actions, or delays. In Figure 19 we show how a set of reachable states is computed in an LTS expressed as a graph. Initially a set of states $S_0$ is computed before any observable action is observed considering $\tau$ transitions $S_0 = s_0 \textbf{ after } \epsilon = \{s_0, s_1\}$. The next set of reachable states $S_1$ is derived applying an observable action $a$ to the present set $S_0$, $S_1 = S_0 \textbf{ after } a = \{s_3, s_2, s_4\}$.

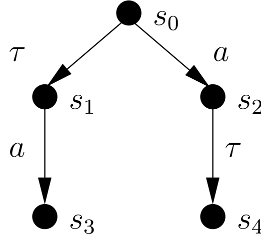



Figure 19: A graph of a LTS.

A state $s_n$ is *reachable* from s if there exists a finite test run $\sigma$, which starts at s, goes through a set of reachable states and ends in $s_n$.

We note $s \textbf{ after } \sigma = \emptyset$ if no states can be reached from s, i.e. $s \stackrel{\sigma}{\not\Longrightarrow}$ (Definition 2.2 ).

We present the test generation and execution algorithm for TLTS and afterwards we define several functions and algorithms for computing test primitives (possible input or expected output actions) and a set of reachable states $S' = S \textbf{ after } x$ after a particular action event $x$ which can be a delay $\delta \in \mathbb{R}_+$ or an action $a \in Act$.

Algorithm 2 presents the steps for a test case generation and execution. There are two choices depending whether the tester wants to send some output $a$ to the implementation or the tester want to delay for some time units. For both choices a state set is estimated as described by the state estimation functions later. We select the choices of test events randomly.

If we choose a delay then the function $ChooseDelay()$ calculates a tester dependent delay $\delta = ChooseDelay()$. An algorithm for the function is presented later in Section 3.2. We wait for an output for $\delta$ time units. If an output occurs before $\delta$ passes, i.e. the tester receives an output at $\delta' \leq \delta$ then it calculates an estimated set of reachable states and checks the validity of the output. If after $\delta$ time units it observes no output from the implementation model and the estimated set of reachable states is empty then test generation and execution fails, otherwise it proceeds on the estimated set of reachable states.

Theoretically, test cases should cover all transitions of a specification, but in practice it is an infeasible task, because long delays complicate testing process. Therefore, we introduce a time-out for test execution to obtain a finite test.

In the second step of the second choice we check whether the output $o \in A_{out}$ produced by the implementation is allowed by a specification to produced such an output. If it is not allowed then implementation does not conform to a specification and testing fails. The test

generation is inconclusive if the output is not in the set of inputs for the environment, i.e. we can not state conformance nor non-conformance. The test generation proceeds with a new input if the output is in both sets. Input: a set of states for the environment model $\mathcal{TLTS}(E)$ and the implementation model $\mathcal{TIOTS}(S)$.

**Algorithm 2** . $TestGenExec(S, E)$.

- Choose $a \in EnvOutput(E)$
  - send $a$ to implementation
  - $TestGenExec$($S$ **after** $a$, $E$ **after** $a$)
- Choose $\delta$ time units. $ChooseDelay(S)$
  1. wait and listen for $\delta$ time units
  2. if $o$ occurs at $\delta'$ then compute $S' =S$ **after** $\delta'$, $E' =S$ **after** $\delta'$, where $\delta' \leq \delta$
     - if $o \in ImpOutput(S')$ and $o \in EnvInput(E')$ then $TestGenEx$($S'$ **after** $o$, $E'$ **after** $o$)
     - if $o \notin ImpOutput(S')$ then FAIL
     - else if $o \notin EnvInput(E')$ then INCONCLUSIVE
  3. else compute $S' =S$ **after** $\delta$, $E' =S$ **after** $\delta$
     - if $S' = \varnothing$ or $E' = \varnothing$ then FAIL
       else $TestGenEx(S', E')$

In Algorithm 2 we compute a set of reachable states after a test run from a particular state, $S =s$ **after** $\sigma$, where $\sigma \in Act^*_\delta$. The test primitives at a set of states S are $out(S) \cup Act_{in} \cup \{\delta\}$. The next set of states $S'$ after an action $\alpha \in Act$ is noted as $S' =S$ **after** $\alpha$ (see Definition 2.2).

**Definition 3.1** a state $s, s' \in S$, $S' \subseteq S$ and $a \in Act_{in} \cup Act_{out}$:

$$
\begin{array}{lll}
Closure_\tau(S) & =_{def} & \{\exists s' |\ s \xrightarrow{\tau}{}^* s'\} \\
Closure_{\delta\tau}(S) & =_{def} & \{s' |\ \delta \in \mathbb{R}_+,\ s \stackrel{\delta}{\Longrightarrow} s'\} \\
EnvOutput(S') & =_{def} & \{a \mid a \in A_{in}, \exists s \in S, s \xrightarrow{a!}\} \\
ImpOutput(S') & =_{def} & \{a \mid a \in A_{out}, \exists s \in S,\ s \xrightarrow{a!}\} \\
EnvInput(S') & =_{def} & \{a \mid a \in A_{out}, \exists s \in S, s \xrightarrow{a?}\}
\end{array}
$$

□

### 3.1.1 Closure Algorithms

In the beginning we present Algorithm 3 for computing $Closure_\tau(S)$, i.e. a set of states after $\tau$-closure $s \stackrel{\epsilon}{\Longrightarrow}$. The $\tau$-closure is computed before and after an observable action transition. Let $s \in S$ be a state, $passed, waiting \subseteq S$ and $\tau \in \epsilon$.

**Algorithm 3** $Closure_\tau(S)$

> $passed := \emptyset$; $waiting := S$
>
> While $waiting \neq \emptyset$
>
> chose a state $s \in waiting$
>
> $passed := passed \cup \{s\}$, $waiting := waiting - \{s\}$
>
> for all $s \xrightarrow{\tau} s'$
>
> if $s' \notin passed$ then $waiting := waiting \cup \{s'\}$
>
> return $passed$

$Closure_{\delta\tau}(S, \delta)$ computes a set of states where a system can be after some delay $S$ **after** $\delta$ and internal actions as outlined in Algorithm 4. S is a set of states initially $S = \{s_0\}$, $\delta \in \mathbb{R}_+$, and $\delta = \sum_{d_i \in \mathbb{R}_+} d_i$ where $d_i$ is the value of variable $d$ in $i$-th recursive call level.

Global $passed := \emptyset$

**Algorithm 4** $Closure_{\delta\tau}(S, \delta)$

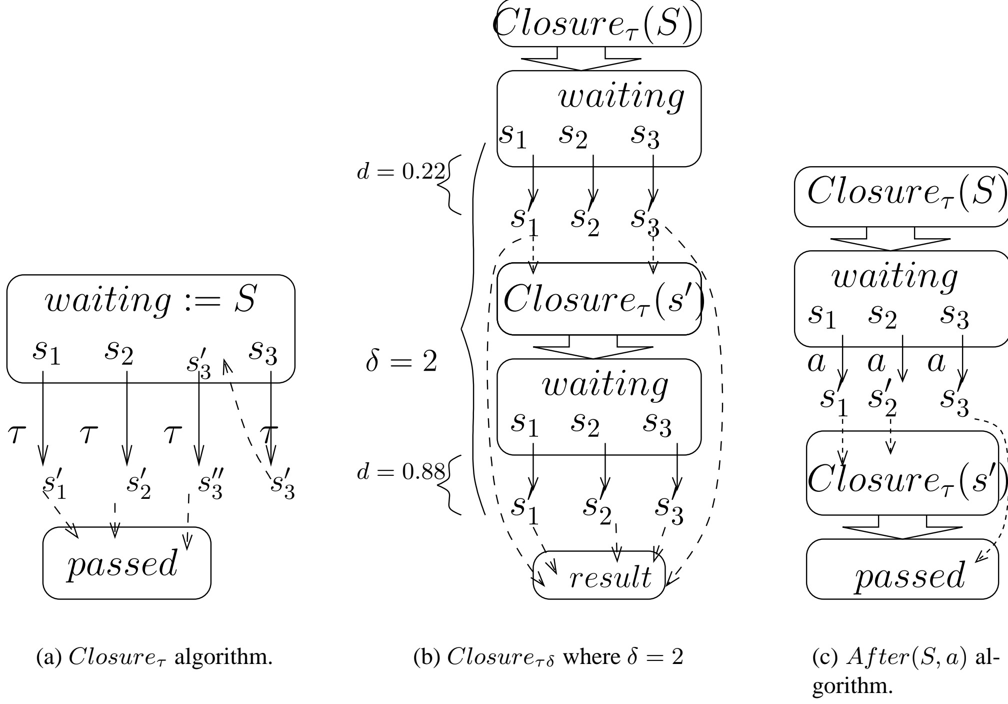
> $waiting := Closure_\tau(S)$, $passed := \emptyset$, $result := \emptyset$
>
> if $\delta = 0$ then return $waiting$
>
> While $waiting \neq \emptyset$
>
> choose $s$: $waiting := waiting \backslash \{s\}$, $passed := passed \cup \{s\}$
>
> for all $d \in (0, \delta]$
>
> compute state $s'$ after delay transition: $s \xrightarrow{d} s'$
>
> if $s' \notin passed$ then $result := result \cup Closure_{\delta\tau}(\{s'\}, \delta - d)$
>
> return $result$

$S'$ **after** $a$ is s set of states $S'$ after an action transition $s \xrightarrow{a} s'$ and is computed using Algorithm 5: let the $s \in S$ be a state, a set of states after an input or output event $passed \subseteq S$, $a \in Act_{in} \cup Act_{out}$.

**Algorithm 5** $\mathbf{After}(S, a)$

> $passed := \emptyset$; $waiting := Closure_\tau(S)$
>
> While $waiting \neq \emptyset$
>
> $waiting := waiting \backslash \{s\}$
>
> for all transitions $s \xrightarrow{a} s'$
>
> if $s' \notin passed$ then $passed := passed \cup Closure_\tau(s')$
>
> return $passed$

In Figure 20 we visualize the closure algorithms. The $Closure_\tau(S)$ algorithm in Figure 20(a)is used in the next two algorithms. We did not display the condition for checking whether a state has been reached or not, i.e. whether it is in the $passed$ list. That is why the state $s_3'$ gets to the waiting list and a set of states is calculated after $\tau$ transitions.




(a) $Closure_\tau$ algorithm.

(b) $Closure_{\tau\delta}$ where $\delta = 2$

(c) $After(S, a)$ algorithm.

Figure 20: Closure algorithms.

The $Closure_{\tau\delta}(S, \delta)$ algorithm is exemplified in Figure 20(b) with $\delta = 2$ and with only two recursive levels of real valued delays. After such a delay successor states are included into the $result$ list and in the next step a $closure_\tau$ is calculated from the successors. If a state is already in the $passed$ list then it is skipped as, for example, the state $s_2'$ which might has be reached before from the state $s_1$. As we can see there can be infinitely many recursive iterations as every delay $d$ requires an iteration and there can be infinitely many such real valued delays and states to which they lead.

In the $After(S, a)$ algorithm in Figure 20(c) recursive iterations are in the $Closure_\tau(S)$ computations but not while computing a concrete action transition. If a state is already in the $passed$ list then $Closure_\tau$ is not calculated for that state as for example in the state $s_3'$.

### 3.1.2 Test Primitive Algorithms

$EnvOutput(S')$ function computes test primitives which are obtained on every transition from a set of states after an input to the implementation from the environment where $a \in Act_{in}$ and $S' := Closure_\tau(S)$. The algorithm for the function is in Algorithm 6. Let the $s \in S$ be a state.

**Algorithm 6** $EnvOutput(S')$

$S' := Closure_\tau(S); \Lambda := \emptyset$,

for all $s \in S'$

for all transitions

if $s \xrightarrow{a!}$ and $a \in Act_{in}$ then $\Lambda := \Lambda \cup \{a\}$

- return $\Lambda$

$ImpOutput(S')$ procedure computes output actions produced by the implementation on every transition from a set of states S' (Algorithm 7). Let the $s \in S$ be a state, $a \in Act_{out}$.

**Algorithm 7** $ImpOutput(S')$

$S' =: Closure_\tau(S); \Lambda := \emptyset$

for all $s \in S'$

for all transitions $s \xrightarrow{a!}$

if $a \in Act_{out}$ then $\Lambda := \Lambda \cup \{a\}$

- return $\Lambda$

$EnvInput(S')$ procedure computes test primitives which are obtained on every transition from a set of states S' (Algorithm 8). Let the $s \in S$ be a state, $a \in Act_{in}$.

**Algorithm 8** $EnvInput(S')$

$S' =: Closure_\tau(S); \Lambda := \emptyset$

for all $s \in S'$

for all transitions $s \xrightarrow{a?}$

if $a \in Act_{out}$ then $\Lambda := \Lambda \cup \{a\}$

- return $\Lambda$

## 3.2 Test Execution on Reachable Symbolic States

In this section we repeat the test generation and execution algorithm rewriten for symbolic states. At first we describe the algorithm for computing test primitives and additional functions used and the test algorithm at the end, since the test algorithm structure is similar to the algorithms presented in Section 3.1.

First of all we need an efficient predicate $Contains(Z, \langle\bar{l}, z\rangle)$ algorithm which computes $true$ if the symbolic state $\langle\bar{l}, z\rangle$ is contained in the symbolic state set $Z$ and $false$ otherwise. This predicate will be used almost in every algorithm dealing with sets of symbolic states. Algorithm 9 outlines such predicate description: the algorithm assumes that it is possible to

lookup for symbolic states for given location vector $\bar{l}$ via constant time (possibly having hash index on location vectors), thus limiting the complexity of the algorithm even less than $O(|Z|)$. However this algorithm is unable check the zone containment within the union of several given zones (it is not always possible to compute a DBM for the union of zones). This leaves a gap for further optimizations but does not harm the correctness of the algorithms where this predicate is used.

**Algorithm 9** $Contains(Z, \langle\bar{l}, z\rangle)$

for each symbolic state $\langle\bar{l}, z'\rangle \in Z$

  if $z \subseteq z'$ then return $true$

return $false$

We need to compute a $\tau$-closure before and after computing the reachable symbolic states while making an observable action. Algorithm 10 provides $Closure_\tau(Z)$ function which computes $\tau$-closure for a given symbolic state set $Z$, i.e. $Closure_\tau(Z) = \{\langle\bar{l}', z'\rangle \mid \langle\bar{l}, z\rangle \in Z,\ \langle\bar{l}, z\rangle \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_n} \langle\bar{l}', z'\rangle,\ \tau_i \in A_U,\ n \in \mathbb{N}\}$.

**Algorithm 10** $Closure_\tau(Z)$

$passed := \emptyset$, $waiting := Z$

while $waiting \neq \emptyset$

  choose symbolic state $\langle\bar{l}, z\rangle \in waiting$

  $waiting := waiting \backslash \{\langle\bar{l}, z\rangle\}$, $passed := passed \cup \{\langle\bar{l}, z\rangle\}$

  for each symbolic transition $\langle\bar{l}, z\rangle \overset{\tau}{\rightarrowtail} \langle\bar{l}', z'\rangle$ where $\tau \in A_U$

    if not $Contains(passed, \langle\bar{l}', z'\rangle)$ then $waiting := waiting \cup \{\langle\bar{l}', z'\rangle\}$

return $passed$.

*After*$(Z, a)$ function computes reachable symbolic states after observing action $a \in A_O$ having initial states in $Z$, i.e. *After*$(Z, a) = (Z\,\mathbf{after}\,a)$. Algorithm 11 assumes that the given set of symbolic states is closed under internal transitions, computes the $a$-action transition for each symbolic state and returns the $\tau$-closure of the computed symbolic states.

**Algorithm 11** *After*$(Z, a)$

$passed := \emptyset$, $waiting := Closure_\tau(Z)$

for each symbolic state $\langle\bar{l}, z\rangle \in waiting$

  for each symbolic transition $\langle\bar{l}, z\rangle \overset{a}{\rightarrowtail} \langle\bar{l}', z'\rangle$

    if not $Contains(passed, \langle\bar{l}', z'\rangle)$ then $passed := passed \cup \langle\bar{l}', z'\rangle$

return $Closure_\tau(passed)$

We present $Closure_{\delta\tau}(Z, d)$ function in Algorithm 12. Algorithm computes $\delta\tau$-*closure*

for a given symbolic state set $Z$, i.e. all symbolic states reachable within $d \in \mathbb{N}$ time units. UPPAAL uses extrapolation technique to handle unbounded delay, which may also be reused here for computing unbounded delay closure, but we recommend not to waist the computing power for looking far ahead ($\delta$ time units is enough in our case), since the symbolic state space may grow very large and testing needs exact clock values anyway even if they are reasonably large. Both Algorithm 12 and Algorithm 13 assume an additional clock $t$ which is not defined in the specification and used only for computational purposes.

**Algorithm 12** $Closure_{\delta\tau}(Z, d)$

$passed := \emptyset, waiting := \emptyset$

for each symbolic state $\langle \bar{l}, z \rangle \in Z$

$z := (z_{t:=0})^{\uparrow} \wedge (t \leq d) \wedge I(\bar{l})$
if not $Contains(waiting, \langle \bar{l}, z \rangle)$ then $waiting := waiting \cup \{\langle \bar{l}, z \rangle\}$

while $waiting \neq \emptyset$

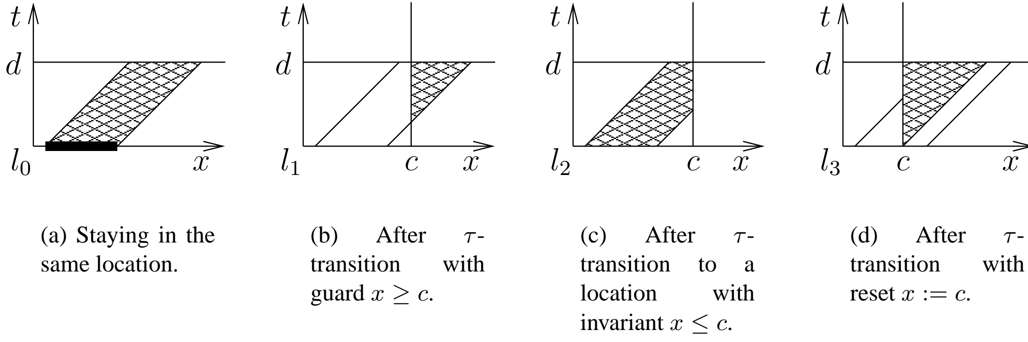
choose symbolic state $\langle \bar{l}, z \rangle \in waiting$
$waiting := waiting \backslash \{\langle \bar{l}, z \rangle\}, passed := passed \cup \{\langle \bar{l}, z \rangle\}$
for each symbolic transition $\langle \bar{l}, z \rangle \stackrel{\tau}{\rightarrowtail} \langle \bar{l}', z' \rangle$ where $\tau \in A_U$
$z' := z'^{\uparrow} \wedge (t \leq d) \wedge I(\bar{l}')$
if not $Contains(passed, \langle \bar{l}', z' \rangle)$ then $waiting := waiting \cup \{\langle \bar{l}', z' \rangle\}$

return $passed$.

The idea behind the algorithm is to propagate a wave to all direction by making all enabled internal transitions to achieve the set of symbolic states which are reachable within $d$ time units. Later the **after** $d$ algorithm just cuts out the edge of zones with $t = d$ from computed set of reachable symbolic states forming a set of symbolic states which are reachable precisely after $d$ time units.

Figure 21 demonstrates four variations of $\delta$ and $\tau$ transitions computed by $\delta\tau$-closure operation: a) automaton stays in the same location $l_0$, thick segment shows the initial zone where clock $x$ has some constraints and $t$ is set to zero, thus zone is expanding toward clock valuation flow in both clocks in parallel limited by the $t \leq \delta$ constraint; b) the zone is achieved after making an internal transition $l_0 \xrightarrow{x \geq c, \tau, \emptyset} l_1$ where the zone on location $l_0$ is taken from Figure 21(a); c) shows an internal transition from $l_0$ to $l_2$ with invariant $I(l_2) = (x \leq c)$ which limits the zone from right side; d) demonstrates an internal transition $l_0 \xrightarrow{true, \tau, x:=c} l_3$ where the zone from Figure 21(a) is projected into line $x = c$ and expanded until $t = d$.

*After*$(Z, \delta)$ computes the symbolic states reachable after $\delta \in \mathbb{Z}_+$ time units having the current reachable symbolic state set $Z$. $t$ is an additional clock in the system. A special constraint $t = c$ used in the algorithm can be reformulated as $(t \leq c) \wedge (-t \leq -c)$. As mentioned earlier, the idea of Algorithm 13 is to compute $\delta\tau$-closure within $\delta$ time units and then require the precise moment of **after** $\delta$. Figure 22 shows an example computation result of *After*$(Z, \delta)$ for corresponding zones in Figure 21.

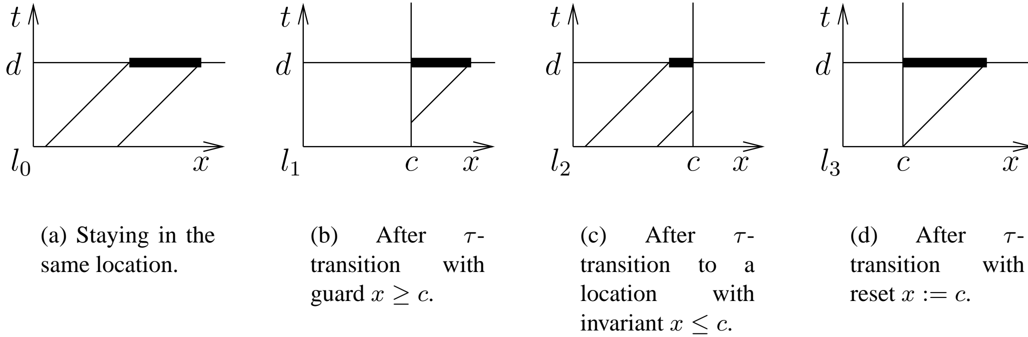



(a) Staying in the same location.
(b) After $\tau$-transition with guard $x \geq c$.
(c) After $\tau$-transition to a location with invariant $x \leq c$.
(d) After $\tau$-transition with reset $x := c$.

Figure 21: $\delta\tau$-$closure$ computation on a sample symbolic state with two clocks.

**Algorithm 13** *After*$(Z, \delta)$

> $t := 0$, $Q := Closure_{\delta\tau}(Z, \delta)$, $Q' = \emptyset$
>
> for each symbolic state $\langle \bar{l}, z \rangle \in Q$
>
> $\quad Q' := Q' \cup \{\langle \bar{l}, z \wedge (t = \delta) \rangle\}$
>
> return $Q'$




(a) Staying in the same location.
(b) After $\tau$-transition with guard $x \geq c$.
(c) After $\tau$-transition to a location with invariant $x \leq c$.
(d) After $\tau$-transition with reset $x := c$.

Figure 22: $Z$ **after** $\delta$ computation on a sample symbolic state from $\delta\tau$-$closure$.

Algorithm 15, Algorithm 14 and Algorithm 16 compute possible observable actions for current reachable symbolic state set $Z$. $EnvOutput$ returns the set of output actions controlled by the environment, $EnvInput$ returns the set of actions acceptable by the environment and $ImpOutput$ returns the set of actions controlled by the implementation. The algorithms actually compute the set $\{a \mid \langle \bar{l}, z \rangle \in Z,\ \langle \bar{l}, z \rangle \stackrel{a}{\Longrightarrow}\}$ assumming that $Z$ is closed under $\tau$-closure.

**Algorithm 14** $EnvOutput(Z)$

$actions := \emptyset$

for each symbolic state $\langle \bar{l}, z \rangle \in Z$

for each symbolic transition $\langle \bar{l}, z \rangle \stackrel{a!}{\rightarrowtail} \langle \bar{l}', z' \rangle$, where $a \in A_{in}$.

$actions := actions \cup \{a\}$

return $actions$.

**Algorithm 15** $EnvInput(Z)$

$actions := \emptyset$

for each symbolic state $\langle \bar{l}, z \rangle \in Z$

for each symbolic transition $\langle \bar{l}, z \rangle \stackrel{a?}{\rightarrowtail} \langle \bar{l}', z' \rangle$, where $a \in A_{out}$.

$actions := actions \cup \{a\}$

return $actions$.

**Algorithm 16** $ImpOutput(Z)$

$actions := \emptyset$

for each symbolic state $\langle \bar{l}, z \rangle \in Z$

for each symbolic transition $\langle \bar{l}, z \rangle \stackrel{a!}{\rightarrowtail} \langle \bar{l}', z' \rangle$, where $a \in A_{out}$.

$actions := actions \cup \{a\}$

return $actions$.

Now we are able to compute reachable states and actions allowed in symbolic form after any event, i.e. we have algorithms for $Z$ **after** $a$ and $Z$ **after** $\delta$ where $a$ and $\delta$ is an action and delay respectively. We present the procedure of the test generation and execution by performing operations on symbolic states in Algorithm 17.

**Algorithm 17** $TestGenExec(TestSpec)$ returns $\{pass, fail, inconc\}$.

$Z = \{\langle \bar{l}_0, \bar{\mathbf{0}} \rangle\}$, $timeout := ChooseTimeout(TestSpec)$

while $timeout > 0$ do switch to $ChooseEvent(Z)$

$action$: // offer an input

$a := ChooseAction(EnvOutput(Z))$
send $a$ to implementation
$Z := After(Z, a)$

$delay$: // wait for output

$\delta := ChooseDelay(Z)$
sleep for $\delta$ time units and wake up on output $o \in A^I_{out}$
if $o$ occurs at $\delta' \leq \delta$ then
$Z := After(Z, \delta')$
if $o \notin ImpOutput(Z)$ then return $fail$
else if $o \notin EnvInput(Z)$ then return $inconc$
else $Z := After(Z, o)$
$timeout := timeout - \delta'$
else
$Z := After(Z, \delta)$
if $Z = \emptyset$ then return $fail$
$timeout := timeout - \delta$

return $pass$

$Choose$ functions mentioned are test strategy dependent in Algorithm 17. The choice decision might be based on the parameters passed to a function. To make the algorithm complete we present sample randomized algorithms for $TestSpec = \langle N, A_{in}, A_{out}, \Delta_{in}, \Delta_{out}, \mu, \Omega \rangle$:

- $ChooseTimeout(TestSpec)$: $random(\{k \cdot \Omega, \ldots, (k+1) \cdot \Omega\})$, where $k = random(\mathbb{N})$.
- $ChooseEvent(Z)$: $random(\{action, delay\})$.
- $ChooseAction(A)$: $random(A)$, where $A \subseteq A_{in}$.
- $ChooseDelay(Z)$: see Algorithm 18.

The universal function $random$ takes any type of set as a parameter and returns a member of a set with equal probabilities.

**Algorithm 18** $ChooseDelay(Z)$

$min := \infty$, $max := 0$, $dim := |X|$

for each symbolic state $\langle \bar{l}, z \rangle \in Z$

for each $i \in \{0, \ldots, dim\}$ consider constraints $c_{i0}$ and $c_{0i}$ in $z$:
if $max < c_{i0} - c_{0i}$ then $max := c_{i0} - c_{0i}$
if $min > c_{i0} - c_{0i}$ then $min := c_{i0} - c_{0i}$

if $min - 1 < \Omega$ then $min := 1$ else $min := min - \Omega$

if $max > \Omega$ then $max := \Omega$

return $random(\{min, \ldots, max\})$

Algorithm 18 aims to select time delays close to zone boundaries. The idea of Algorithm 18 is based on the length of intervals of reachable clock values. The function tries to pick delay values wich are close to clock zone limits. The minimum value of delay to be chosen can be 1, which means that Algorithm 17 will make a progress in time if a delay event was chosen. The algorithm is expected to run on $\delta\tau\text{-}closed$ zones with additional clock $t$ which tracks how much time elapsed after the last observable action.

# 4 Implementation

The UPPAAL tool is constantly under development. Therefore it is very hard to keep up with the newest and best code written for UPPAAL. This documentation is based on a January 2003 development source snapshot and there is a slight difference between the class names used in that code and the concepts we have described. Hopefully the class names will be changed in the future, but for now we keep them as they were in the beginning of the project implemention phase.

The general idea for implementing the test generation and execution algorithm in UPPAAL is to extend the UPPAAL library shown in Figure 13 with test specific extensions shown in Figure 23:

- We reuse the time zone and symbolic state libraries intact as described in Section 4.2.
- We add structures and functions to handle sets of symbolic states since the test generation algorithm is mainly dealing with them.
- We adapt the reachability analysis algorithms for our test generation purposes, design and implement a driver to handle timing aspects during test execution in Section 4.3.

We append interesting parts of our code to Appendix A, and the implementation status is described in Section 4.4. We describe only the most important classes, atributes and methods for test generation and execution, since documenting entire T-UPPAAL code would take more than 700 pages and this is not a project goal. We recommend first to read a description and get the idea and purpose of a class and only then refer to the source code for actual details.

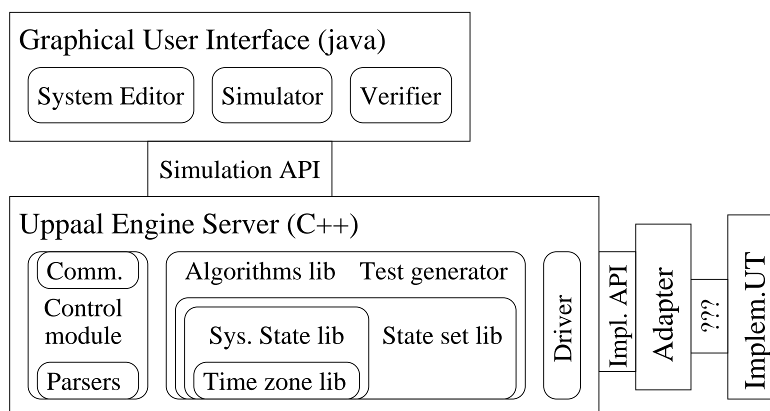



Figure 23: Test generator architecture.

## 4.1 Assumptions, Requirements and Desired Features

In this report we have mentioned numerous assumptions and requirements. In this section we repeat them in order to have a consistent overview of the desired testing tool features.

So far we mentioned these assumptions:

1. The implementation model must be strongly input enabled, i.e. it must accept an input offered by an environment. However we allow modeling of the environment that the input enableness requirement may be lowered as much as the environment model allows. One must be careful while modeling a system which is not strongly input enabled: any ignorance of the input offered by the environment leads to unpredictable behavior of the implementation.

2. The environment is fully permissive if the model of the environment is not provided.

3. The model of the environment and the model of implementation must be separable when given the closed UPPAAL timed automata network with the environment and the implementation models embedded into a single parallel composition.

4. Assumptions about a specification:

   - The specification must be strongly converging, i.e. it cannot contain 0-delay infinite loops which allow the implementation to ignore the input infinitely.
   - The specification is complete. We do not consider the possible rules for making the specification complete. This assumption still allows the future extensions which convert partially defined specifications into a complete ones.

However the automated test generation and execution on-the-fly is still impossible in practice if the reachability algorithms used do not comply with the following requirements:

1. The complexity of the algorithm must not exceed the time limits needed by testing on-the-fly, i.e. the computation of reachable symbolic states one step ahead must not last longer than $\mu/2$ time units specified in the test specification, otherwise the algorithm will not be able to choose a delay as small as one time unit.

2. The clock precision should be approximately $\pm 0.5$ time units because in UPPAAL clock constrains values on invariants and guards, or assignments use integer numbers.
3. The adapter to an implementation under test must not be a bottle neck, i.e. amount of synchronizations and transfered data must not degrade the performance of the testing.
4. To satisfy the first and the second requirements, the test generation and execution must be evaluated experimentally. The complexity of the test specification can be measured by the maximum size of reachable symbolic states set during one test primitive computation.

To make testing tool more usable we need even more desired features:

1. Visualization options:
   - *Current symbolic state set display* would be essential monitor of the actual testing procedure taking place.
   - Display a *trace* (history) of a test is essential while debugging real-time systems.
2. Verification properties for a test specification:
   - A feature for checking a test specification to be well defined (the environment and the implementation are separable) allows automatic migration from modeling real time systems to testing them.
   - verification for completeness.
   - verification for the implementation to be weakly or strongly input enabled.
3. More advanced testing extensions:
   - Extend testing for the partially defined specification where the implementation reacts on the input offered with specific rules. This feature requires at least converter which converts partially defined specification into a complete one. However the complete specification might appear too big for test execution on-the-fly, then the test generation and execution algorithm must be reconsidered.
   - Extend testing for weakly input enabled implementations. The design of the test generation and execution algorithm must be reconsidered.
4. Features for a test process control and GUI:
   - *Save, load and rerun* a test case is useful when the test generation is too slow and a preset version of test case is needed to run the test in full performance.
   - Implementation simulation module in UPPAAL (Figure 24): load any implementation model and test it against the specification in order to evaluate the testing procedure itself.

## 4.2 UPPAAL Design

UPPAAL architecture is split into three parts: system model representation (Section 4.2.1), symbolic state representation (Section 4.2.2) and algorithms. Such configuration is easier to maintain, extend and even document. We give more detailed insight on symbolic state representation in Section 4.2.3 which describes the time zone representation. Section 4.2.4 gives an overview how the algorithms are ordered.

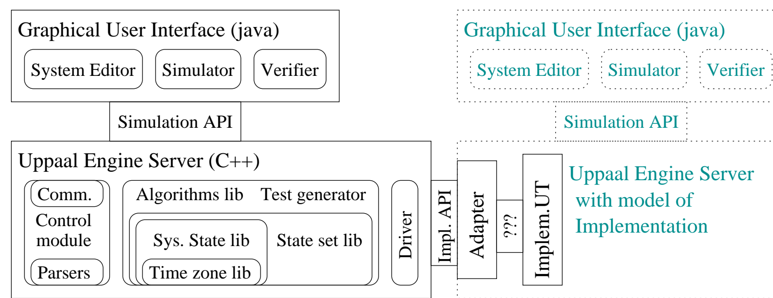



Figure 24: General implementation under test plug-in.

### 4.2.1 System Model Representation

Figure 25 shows the class diagram of entities responsible for the system model representation. Class *System* corresponds to a system definition section in UPPAAL system specification. Objects of a class *System* are responsible for holding the objects of class *Process*. *Process* entity correspond to an instantiated timed automaton template. If the UPPAAL specification instantiation section does not include instantiation statement then a template is instantiated with defaults when the template is mentioned in the system definition. The objects of class *Process* hold the structure of a timed automaton, i.e. the lists of locations and transitions. A timed automaton location is represented by *ControlState* class and *Transition* class corresponds to an edge between two locations (*ControlState* objects).

The system model classes are int a tree-like hierarchy without loops (the leaves of the hierarchy are always *ControlState* objects). All of these classes inherit two common interfaces: *Object* and *ModelElement*. The *Object* interface handles reference counting and decentralized memory deallocation. The *Object* also provides an interface for setting and getting the named properties. In this case a *named property* is the data object which can be accessed using property's name. The property name and the property type are subject to classes which implement the property interface. The *ModelElement* interface is used to implement and maintain independent algorithms that traverse the system model representation in a visitor design pattern. The *ModelElement* interface is mainly used for constructing and initializing the system model representation, but could also be useful for other computations, such as checking whether the IUT and environment models are separable (Definition 2.15).

Once the system model representation is built, it is kept constant, i.e. it does not change during reachability analysis computations. Normally UPPAAL maintains a singleton of the class *System* and uses it to compute symbolic state transitions on *GlobalState* (described in Section 4.2.2) class objects. Therefore these methods are most commonly used in reachability analysis when generating test on-the-fly:

**System::getProcess(index)** returns a reference to a *Process* which has a given index in a list of processes.

**Process::getTransitionsFrom(GlobalState)** returns a list of edges going from locations specified in *GlobalState* vector of locations (one location per one timed automaton in the system model).

**Transition::getSync(GlobalState)** returns a channel identifier if the edge contains a channel synchronization and 0 if there is no channel synchronization.

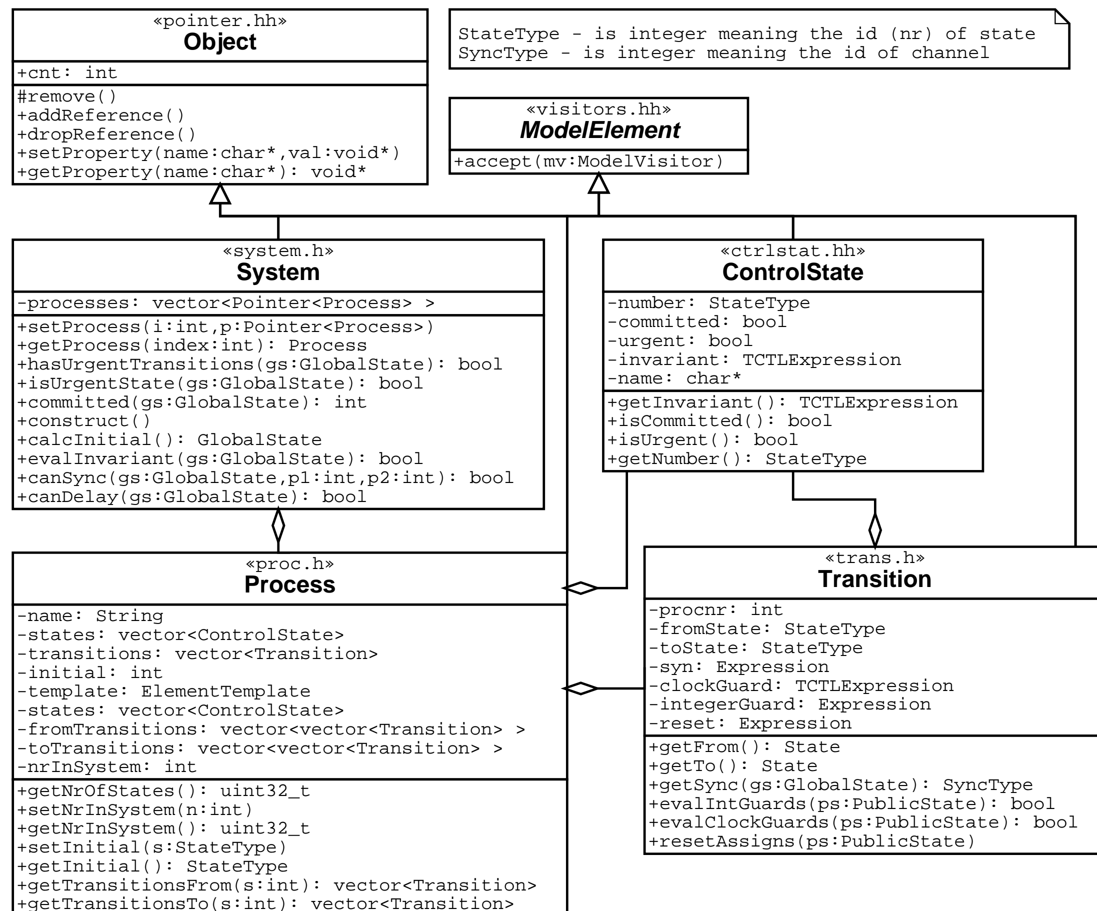



Figure 25: Class diagram of system model representation in UPPAAL code.

**Transition::evalIntGuards(PublicState)** method evaluates the integer guards on a given symbolic state, returns true if the guards are satisfied and false otherwise.

**Transition::evalClockGuards(PublicState)** method evaluates the clock guards on a given symbolic state, returns true if the guards are satisfied and false otherwise. This method alters the symbolic state by changing the zone according to the guards which is different than in integer guard evaluation: integer tables are not modified during integer guard evaluation as time zones are during clock guard evaluation.

**Transition::resetAssigns(PublicState)** method alters the given symbolic state according by the assignments specified on the transition.

**System::evalInvariant(GlobalState)** method alters the symbolic state time zone by evaluating the invariant conditions on locations in a given symbolic state.

Notice that most of the methods accept either *PublicState* or *GlobalState* (described in Section 4.2.2) which represent a symbolic system state and are separated from system model representation.

### 4.2.2 Symbolic System State Representation

Figure 26 shows the *GlobalState* class with context which represents the symbolic state in UPPAAL. Since there are a lot of sophisticated computations on large amounts of symbolic states *GlobalState* has an interface of an *Object* which decentralizes the deallocation of objects. A generic separated *PublicState* is used also for other symbolic state operations like action recording on *PublicState*. Such operations are irrelevant for our project. The last *StateInfo_t* interface is used to align symbolic states into traces which is relevant in reachability analysis when recreating the possible run traces out of set of recorded symbolic states. The *GlobalState* uses *Region*

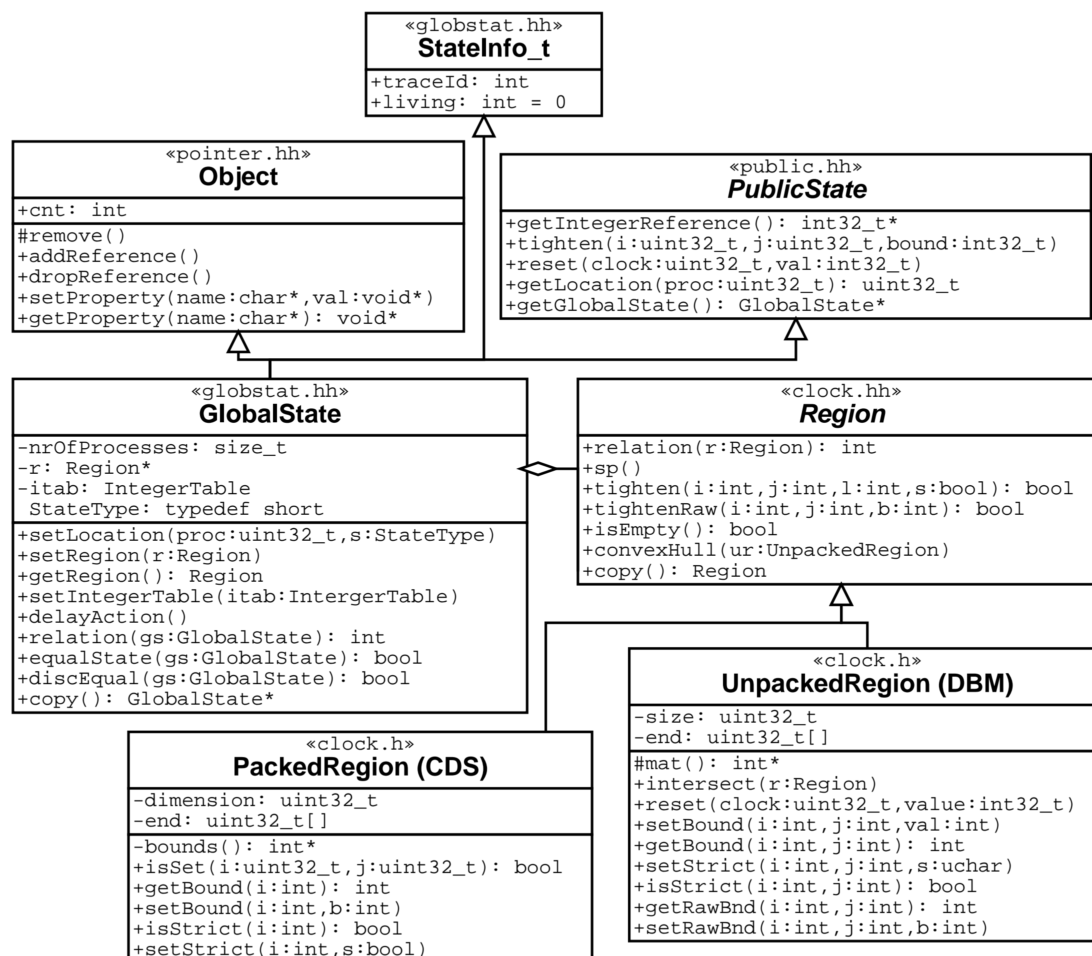



Figure 26: Class diagram of symbolic system state representation.

objects to maintain the time zone constraints as described in Section 4.2.3. The integer guards are saved in the *GlobalState::itab IntegerTable* which is an encapsulation of a dynamic integer array. The location vector is kept outside the *GlobalState* class and allocated by dedicated memory allocator to save and to reuse the memory space more effectively. The *StateType_t* stands for a location index in a timed automaton. While computing symbolic state successors the following *GlobalState* methods are used:

**setLocation(proc, s)** transfers the symbolic state to another location specified by index *s* in the timed automaton with index *proc*.

**getLocation(proc)** returns the current location of the automaton with specified index *proc*.

**setRegion(r), getRegion()** sets and returns the symbolic time zone representation.

**setIntegerTable(itab), getIntegerReference** sets and returns the integer table array.

**relation(gs)** determines the inclusion relation to another symbolic state *gs*: *noInclusion*, *argIncludedIn*, *argIncludes* or *equal*.

**equalState(gs)** determines whether the location vectors of two symbolic states are equal.

**discEqual(gs)** determines whether both the location vectors and integer tables of two symbolic states are equal.

**tighten(i, j, bound)** applies (tightens) *bound* constraint on clocks with *i* and *j* indexes.

**reset(clock, value)** resets the clock with index *clock* to the *val* value.

**copy()** returns a new copy the symbolic state, usually used for further successor computation.

All operations on clocks are forwarded to operations in a *Region* class described in Section 4.2.3.

### 4.2.3 Time Zone Representation

As the class diagram from Figure 26 already depicts there are several implementations of time zone representation in UPPAAL:

**UnpackedRegion** implements the classical difference bound matrix (DBM) with fundamental operations described in Section 2.2.

**PackedRegion** implements the *compact data structure* (CDS) which stores the minimal amount of constraints while saving the memory space, e.g. only three bounds are stored for the constraint system shown in Figure 7 and the constraint indexes marked in bitmap vector. The memory space complexity is still quadratic as in DBM case, but it is believed that in practice it is much better.

Beside these time zone implementations UPPAAL is also capable of using another compact time zone representation structure called *clock difference diagram* instead of a *PackedRegion*. “Packed” implementations of a *Region* have advantage of lower memory consumption when used in huge passed-waiting symbolic state lists. In contrary, “unpacked” implementations require less processing efforts to compute the same operations. Therefore, the *PackedRegion* objects are exclusively used when storing the symbolic states and the implementation of operations on constrains are omitted, such as *up* operation in *sp* method. *UnpackedRegion* objects have full functionality implemented and used in symbolic state successor computations. We concentrate on the most important properties and methods used in our test generation:

**Region::relation(r)** determines what is a relation of this time zone to a given *r*: *none*, *subSet*, *superSet* or *equal*.

**sp()** computes a future operator on a time zone.

**tighten(i, j, l, s)** applies a constraint on *i* and *j* clocks with a limit *l*. The constraint has a strict form (less than) if *s* is true, otherwise the constraint is weak (less or equal).

**tightenRaw(i, j, b)** applies a constraint on *i* and *j* clocks where the strict bit and a limit is encoded in a boundary *b*.

**isEmpty()** method checks whether the constraint system represents an empty zone.

**convexHull(ur)** computes the smallest super set time zone which includes both this and *ur* zones. This method can be used to merge several symbolic states into one, thus reducing the amount of symbolic states. Such over approximation is not desired in test generation since we want to have a precise reachable symbolic state set.

**copy()** constructs and returns a copy of a zone for further successor computations.

**UnpackedRegion::size** stores the amount of clocks in DBM, therefore the matrix is of $size \times size$ size.

**UnpackRegion::end** stores the constraint values of DBM in a dynamic array. The array is accessed via protected method *mat()*.

**UnpackedRegion::intersect(r)** computes an intersection of this and *r* time zones. The result is kept in current (this) object.

**UnpackedRegion:: setBound, getBound, setStrict, isStrict** are simple setters and getters for the differences on clocks.

**UnpackRegion:: setRawBnd, getRawBnd** are a setter and a getter for clock value difference limits where the strict bit is encoded in the bound.

### 4.2.4 Pipeline Architecture

UPPAAL contains a number of algorithms implemented for various kinds of property checking, such as reachability checking, liveness checking, deadlock detection and other purposes like trace simulation. The list is still growing, e.g. we want to extend the algorithm library to handle test generation. Most of the algorithms share similar and sometimes the same operations on symbolic states and time zones. Therefore algorithms are put into separate components and components are divided into shared sub-components to ease code management and to avoid code duplication. The key for splitting the algorithms into their sub-parts is a pipeline paradigm, where an algorithm is encapsulated into a component which consists of a line (sometimes of several branching lines) of abstract smaller algorithms which are encapsulated in sub-components. Note that any algorithm can become a sub-algorithm for bigger algorithms because of this algorithm wrap into a component and sub-component.

Figure 27 shows the fundamental interfaces needed to implement the pipeline paradigm. The interfaces provide pure virtual methods to ensure correct polymorphic behavior of the implementations. The most abstract interfaces are:

**Sink** is a component which accepts data objects fetched by a general pointer to it. The method *tryPut(p)* returns *true* upon successful operation and *false* otherwise.

**Source** is a component which provides the data objects on request and puts the data into a memory location pointed by a general pointer (usually it is a pointer to an object pointer). The method *tryGet(p)* returns *true* upon success and *false* otherwise.

**Generator** is a component which provides the data objects without a request and puts the data into *Sink* selected by the method *setSink(s)*.

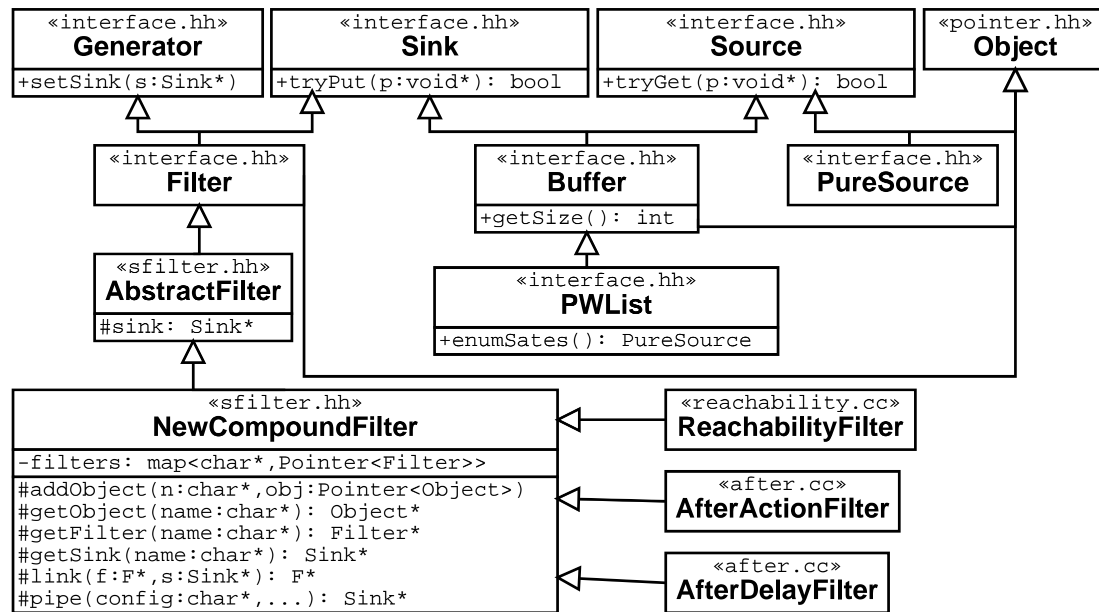



Figure 27: Class inheritance diagram of interfaces for algorithm pipelines.

Notice that the operation success and the type of the data fetched and received depend on the context of the component implementing them.

The next level of abstract interfaces consists of the following:

**Filter** is an interface for a transitional component which accepts data as a *Sink* and produces data as a *Generator* when given a chance to do that.

**Buffer** is an interface for intermediate data storage which accepts data as a *Sink* and provides when requested as a *Source*.

The interfaces mentioned also require to implement the *Object* interface which serves for reference counting and setting and getting various properties that component algorithms may need. Types and amounts of data processed through these interfaces are not restricted.

The *Buffer* interface is used for storing the passed and waiting lists of symbolic states and usually provides a *PureSource* interface to read the accumulated passed list.

The majority of UPPAAL algorithms consist of many pipe-lined filters and buffers, therefore they implement the *NewCompoundFilter* interface which helps to connect and manage the sub-components in an organized fashion:

**addObject(n, obj), getObject(name)** enables the implementing class to store and access its sub-components of type *Object*.

**getFilter(name), getSink(name)** provide access to objects by name already with a prepared interface of *Filter* or *Sink*.

**link(f, s)** links the flow of data from specified filter *f* to specified sink *s*.

**pipe(config, ...)** links the flow of data through specified components in a varying argument list according to a configuration scheme string *config*.

The *NewCompoundFilter* has other advantages such that it broadcasts property settings to all objects registered. Property broadcasting is convenient when using the same memory allocator and trace storage.

In the following sections we explain our symbolic state estimation algorithms which are based on a *ReachabilityFilter* idea in UPPAAL.

## 4.3 Specific Testing Extensions

Having discussed the overall UPPAAL architecture we describe what testing specific extensions we added. Section 4.3.1 describes the implementation of the *after action* algorithm (Algorithm 11). Section 4.3.2 describes the implementation of the *after delay* algorithm (Algorithm 13). We describe how we attach IUT and handle the real-time events in Section 4.3.4. And finally we show how the test generation and execution algorithm works in Section 4.3.5.

### 4.3.1 After Action Filter

The *after action* estimation algorithm is implemented as a compound component called *AfterActionFilter*. Figure 28 shows the containment scheme of sub-components inside the after action algorithm. Arrows show the data flow among the filters, i.e. an arrow means that the source component acts like a *Generator* (or sometimes as *Source*) and is passing the data to a destination component acting like a *Sink*. The label on an arrow indicates the type of data which is passed among the components.

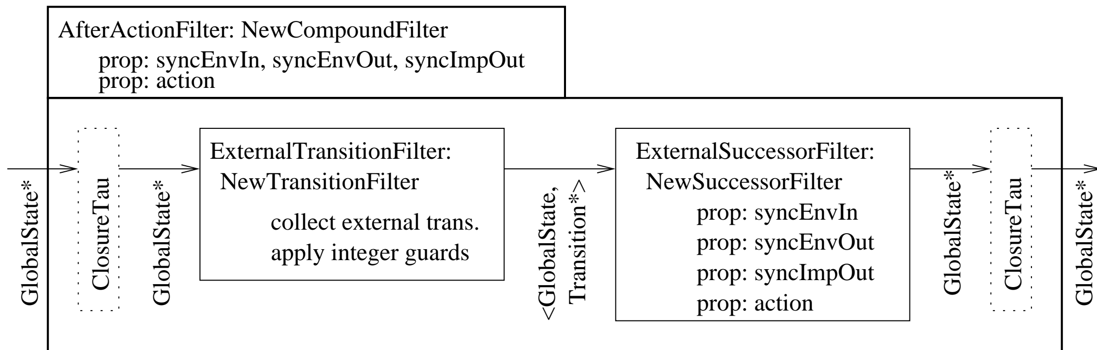



Figure 28: Data flow diagram for after action algorithm.

The header of the *AfterActionFilter* indicates that it is acting similarly to *NewCompoundFilter*, in this particular case the *AfterActionFilter* inherits the class *NewCompountFilter*. The main component header also says that the *AfterActionFilter* has several properties (*prop*): *syncEnvIn* - available environment input synchronization channels; *syncEnvOut* - available environment output synchronization channels; *syncImpOut* - available implementation output channels; and *action* - specifies the action (channel synchronization with data variable values) to be applied on symbolic states.

The *AfterActionFilter* accepts *GlobalState* objects and produces *GlobalState* objects meaning the reachable symbolic states after a specified action in the *action* property. The *AfterActionFilter* has some use conventions: 1) it must reserve an entire symbolic state set before any computations, 2) the owner of the filter may request what observable actions are available and can be triggered from the current set of states, 3) the owner must set an action for successor

symbolic states to be computed, 4) feed a *NULL* pointer and only then the successor symbolic states will be produced to its *Sink*. The component consists of two internal filters:

**ExternalTransitionFilter** acts like a native UPPAAL *NewTransitionFilter* except that it collects transitions containing only the observable channel synchronizations. We mention briefly in Figure 28 that it also considers the integer guards on selected edges. A further transition computation is done in the next filter passing a pair of *GlobalState* object and a list of potential edges (*Transition**) for computing observably synchronized transitions.

**ExternalSuccessorFilter** acts like a native UPPAAL *NewSuccessorFilter* except that it maintains a lists of possible observable channel synchronizations, collects all possible successors after any observable action and waits for a *NULL* object to pass the required successors to its *Sink*. The filter applies the clock guards, assignment expressions and the invariants defined on the destination locations during the successor computations.

The *ClosureTau* filters mentioned in the Figure 28 are in dotted boxes meaning that these components are not implemented, but for the *AfterActionFilter* to function correctly we require and assume that a symbolic state set fetched is closed under unobservable transitions. For simplicity purposes we omit the *ClosureTau* filter implementation and use the *AfterDelayFilter* with a zero delay instead, which is equivalent to $\tau$-closure although not optimally implemented because of redundant computations in *LimitedDelayFilter* and *StrictDelayFilter* components.

### 4.3.2 After Delay Filter

The *AfterDelayFilter* component is an implementation of the *after delay* algorithm. It is more complex than *AfterActionFilter* since we have to compute a closure of internal transitions which may occur in specified interval of time. The *AfterDelayFilter* has similar conventions: 1) it must get the entire symbolic state set, 2) the owner must set the delay property, 3) the owner must feed a *NULL* pointer and only then the actual computation starts, a new reachable symbolic state set is produced and directed to *Sink*. Figure 29 shows the internals of the *AfterDelayFilter*:

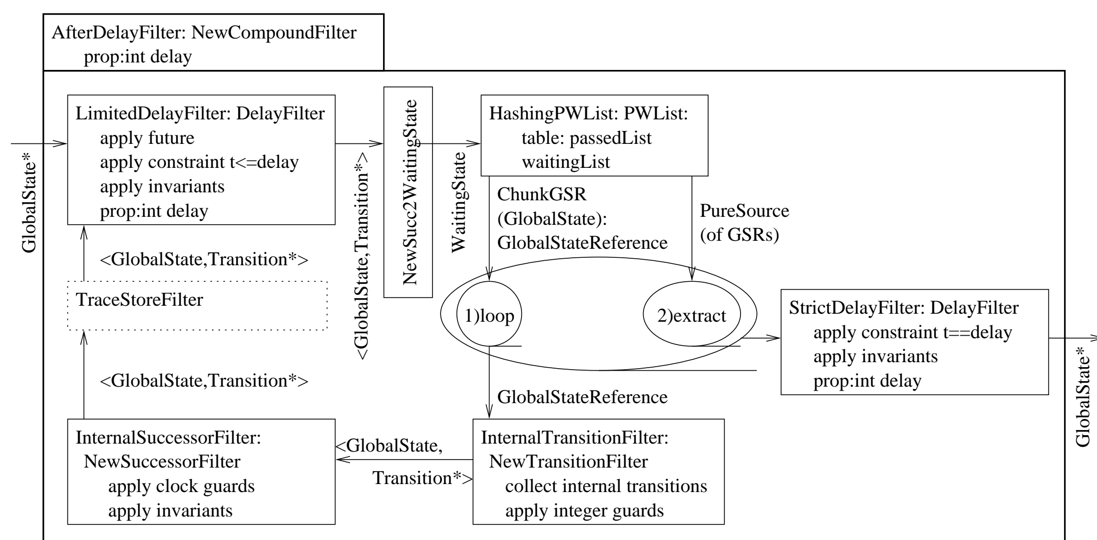



Figure 29: Data flow diagram for after delay algorithm.

**LimitedDelayFilter** is similar to native UPPAAL *DelayFilter* except it applies an additional constraint for the time zone future to restrict the symbolic state estimation to actual period of delay specified by the property *delay*, otherwise we may explore much bigger symbolic state space than is needed at a given moment.

**NewSuccessorToWaitingStateFilter** is a native UPPAAL component which prepares the symbolic state structure for storage in passed-waiting list.

**HashingPWList** is a native UPPAAL component implementing the passed-waiting list which uses hashing techniques for quick check of symbolic state presents in the passed list.

**InternalTransitionFilter** is similar to *NewTransitionFilter* except it filters out the transitions containing observable channel synchronizations.

**InternalSuccessorFilter** is a native UPPAAL *NewSuccessorFilter* for now.

**StrictDelayFilter** is similar to *DelayFilter* except it omits the future operator and applies the additional constraint instead. The constraint restricts that the additional clock $t$ must be equal to *delay* property. *StrictDelayFilter* also applies a reset $t := 0$, which prepares the symbolic state for the next round of test generation and execution algorithm. The reset ensures that the additional clock will never reach its maximum value and will not wrap-over. However, this can be considered as redundant and might be changed to compute the total length of the test (very few adjustments are needed).

Before the *NULL* pointer is fed to *AfterDelayFilter*, the symbolic states are processed in the *LimitedDelayFilter* and stored in the *HashingPWList*. When a *NULL* pointer is fed, the procedure marked by big ellipse (Figure 29) is started. This procedure consists of two loops (two smaller inner ellipses in Figure 29):

1. the first one fetches symbolic states from the *HashingPWList* waiting list and fetches it directly to the *InternalTransitionFilter*, thus eventually all symbolic states ends up in the passed list of *HashingPWList*;

2. the second one extracts the symbolic state enumerator from the *HashingPWList* passed list and feeds the symbolic states to the finalizing *StrictDelayFilter* component.

### 4.3.3 Buffered Filter

*BufferedFilter* is used to store the symbolic states temporary between the applications of *AfterDelayFilter* and *AfterActionFilter*. *BufferedFilter* buffers the symbolic states it is receiving into the list and outputs the contents when *NULL* pointer is received. The *BufferedFilter* component encapsulates the *std::list* from C++ Standard Template Library. We assume that the intermediate state sets are small enough (in respect to symbolic states in reachability algorithm) and we do not do much computations while keeping them in a filter. Having a *std::list* in a *Filter* interface proved to be convenient and transparent in a context of UPPAAL algorithms and structures.

### 4.3.4 Driver Implementation

We developed *TestDriver-TestAdapter* interface for UPPAAL in order to be able to communicate with an implementation under test. Figure 30 shows all test execution related classes. *TestDriver* has several purposes:

- Translates UPPAAL specific input and output data into transmittable shape and back. For demonstration purposes the transmittable data type is a character string which encodes the necessary information about input and output actions.
- Transmits and receives the information about input and output events to and from *TestAdapter*.
- Time-stamps the events. UPPAAL gets all the timing information about when the output actually happened and how much time elapsed after last event.
- Maintains an independent log file about the input and output events during test execution.

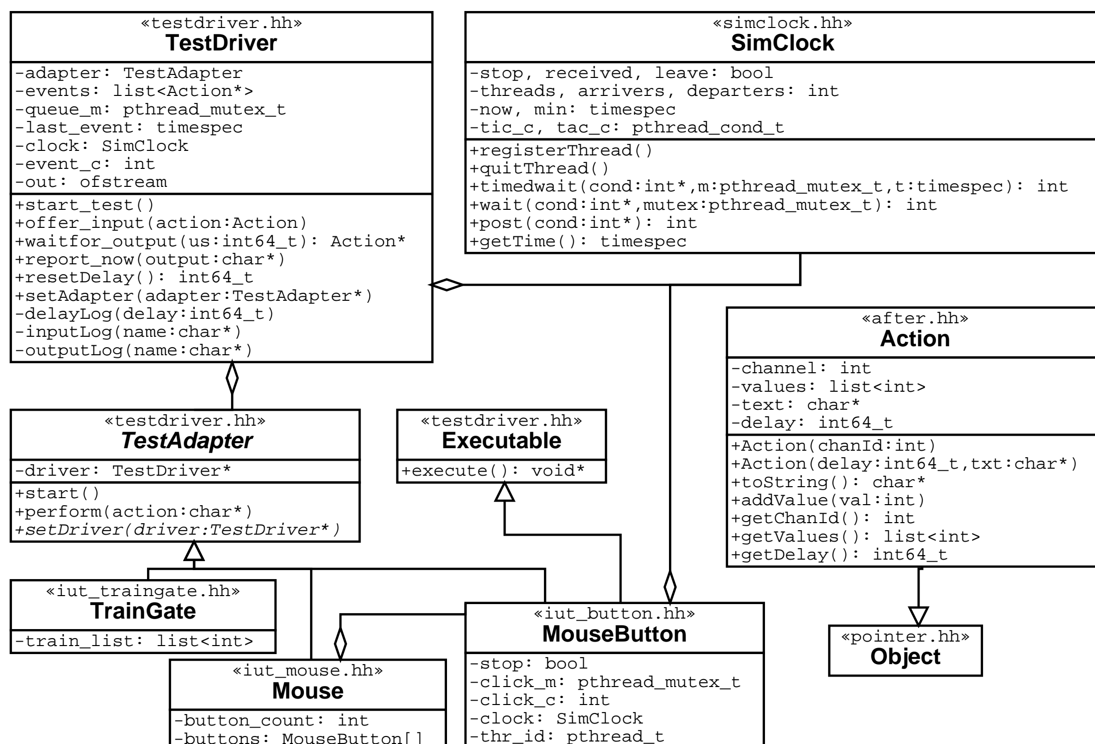



Figure 30: Class diagram of connection to implementation under test.

*TestAdapter* is responsible for the connection to a real implementation under test (IUT). Depending on the nature of the IUT *TestAdapter* translates the character strings into input actions, translates output actions into character strings and passes them to *TestDriver*. As Figure 30 shows, we have implemented three adapters and put the IUT simulating code into these classes: *TrainGate*, *MouseButton* and *Mouse*. Section 5 discusses the purpose and functionality details of our simulated implementations under test.

The following describes the main attributes and methods involved in communication between UPPAAL and implementation under test:

**setAdapter(adapter)** is a setter for adapter field which attaches a *TestAdapter* with IUT to the driver and calls the *TestAdapter::setDriver(this)* method.

**start_test()** initializes the internal driver structures and calls *TestAdapter::start()* to acknowledge the start of the test with a IUT. *TestDriver* just resets *last_event* field to a current time, meaning that the first and still the last test event (start) occurred at that moment.

**offer_input(action)** sends an input action to IUT via calling *TestAdapter::perform(action)*. Luckily the objects of type *Action* are capable of converting themselves to a character string which is written to a log file as well.

**waitfor_output(us)** checks the event queue *events*. If the queue is not empty the method takes out and returns the first event from the queue. If the queue is empty it blocks and waits for an output or the timeout in *us* microseconds, whichever occurs first. The *NULL* pointer is returned if no output was observed until timeout occurred.

**resetDelay()** is used by test generation algorithm to reset the *last_event* timer to current time. This has sense when test generation algorithm chooses several times to wait for output and the output is not produced. The method returns the amount of time passed from the last event occurrence.

**report_now(output)** method is called by *TestAdapter*. *TestDriver* constructs an *Action* object, puts it into *events* queue and wakes up the thread potentially waiting in the *waitfor_output(us)* method.

The driver-adapter connection allows an implementation to be separated from T-UPPAAL as much as possible. T-UPPAAL and an implementation are even not forced to share the same clock. *MouseButton* and *TestDriver* uses their own function calls to determine how much time has passed. Although the very first testing attempts revealed clock synchronization issues because of delays during computation time. We have developed a simulated clock library which aims at synchronizing T-UPPAAL and IUT clocks. The idea behind the simulated clock is that it does not allow the simulated time to flow during the computations and increments the simulated clock value only when all threads are waiting for it to do so. In order the simulated time to be transparent to the IUT and T-UPPAAL we replaced all the time flow related functions calls to *SimClock* method calls (see Figure 30):

**timedwait(cond, m, t)** replaces *pthread_cond_timedwait* which waits either for condition *cond* or timeout at *t* to occur. During the waiting time mutex *m* is released and regained when some event has occurred. The integer *cond* plays a role of a semaphore (counts the amount of resources) rather than a condition variable in *pthread_cond_timedwait*.

**wait(cond, m)** replaces *pthread_cond_wait* which waits for the condition *cond* indefinitely. During the waiting time the mutex *m* is released and regained when the condition is reached.

**post(cond)** replaces *pthread_cond_signal* which signals the thread waiting for condition. We also count how many times the condition was signaled in case there are several resource producers triggering the same condition or consumers waiting for the same condition.

**gettime()** replaces *gettimeofday*. The method returns the simulated clock value for given moment. The simulated clock value is set to zero initially.

**registerThread()** adds a new (current) thread. This method call increments the thread counter and makes sure that all interested threads arrive into simulated clock monitor to be able to increment the simulated clock value with respect to all interested threads.

**quitThread()** removes a thread. This method call decrements the thread counter and makes sure that all threads potentially waiting will not starve because one thread willing to leave.

Note that neither *sleep* nor *usleep* are used in our code, therefore the replacement for *sleep* is not implemented.

Figure 31 shows an example use of *SimClock* when waiting for resources in a simulated time. At first all interest threads have to register themselves in a *SimClock* object. The thread-

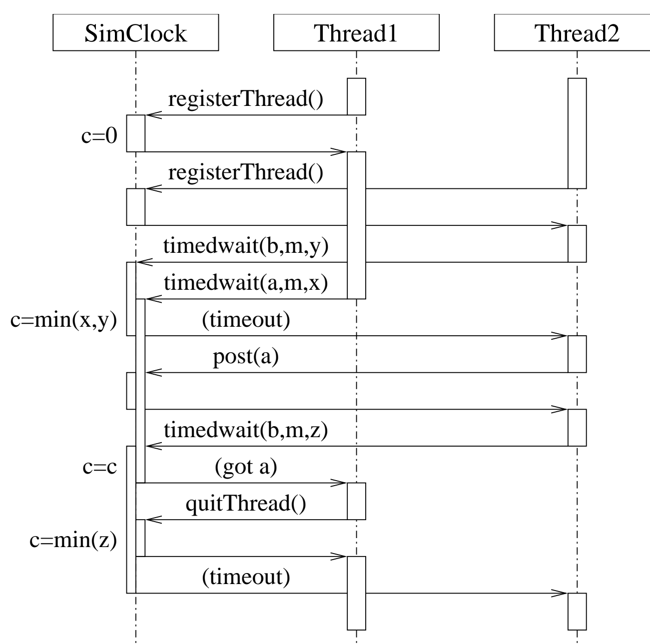



Figure 31: Message chart of thread synchronization and resource sharing through *SimClock*.

registration is done automatically when threads are accessing the simulated clock through *getGlobalClock()* function, since the intention is to have a singleton of *SimClock* in entire program. The first call to *getGlobalClock* creates a singleton object of *SimClock* and set the clock value *c* to zero. When all threads are registered, they can start waiting for conditions. *Thread2* owns a mutex *m* and is willing to wait for *b* until *y*. The clock releases the mutex *m* and adds the thread into a queue waiting for *tic_c* condition (see Figure 30). *Thread1* then acquires the mutex *m* and is willing to wait for *a* until *x*. The clock releases the mutex $m$. Now the last registered thread entered the clock monitor and is deciding to increment the time to value $y$ since it is less than $x$ ($c = min(x, y)$) and triggers the *tic_c* condition which transfers all threads to another queue waiting for *tac_c*. The last thread entering to *tac_c* queue triggers condition *tac_c* to allow all threads to re-acquire their mutex and test their conditions for leaving the clock monitor. Such double barrier of thread queues is needed to make sure that threads do not enter the second queue immediately after they left the monitor. The code between the queues contains the clock value incrementation which should be executed only by the last thread. The code for *SimClock* is in Appendix **??**

### 4.3.5 Test Generation and Execution Algorithm

The final implementation has two principal differences with respect to the one we described in Section 3.2: 1) we check for potential outputs produced before we choose either a delay or an

action and 2) we compute after delay closure before computing after action closure to ensure the $\tau$-closure. Figure 32 shows the message chart of T-UPPAAL component communication during the *MouseButton* testing. *UppAal* refers to all symbolic state estimation algorithms we use in

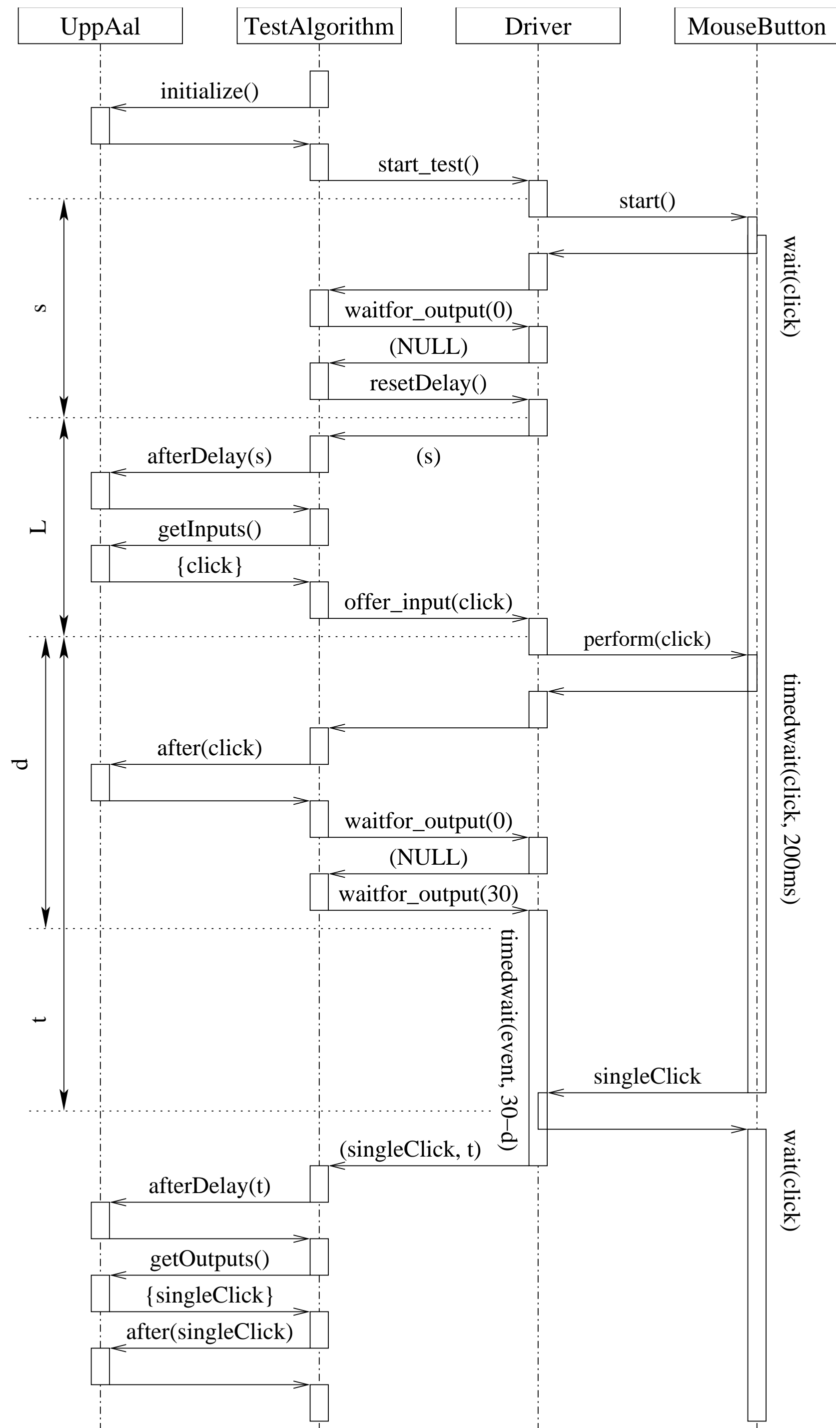



Figure 32: Test generation and execution message chart.

our test generation. Testing starts with the initialization of T-UPPAAL structures by reading a

test specification. *TestAlgorithm* executes the following actions in a row according to Figure 32:

1. Sends a message to the IUT to acknowledge the start of the test.
2. Checks for potential outputs by calling *waitfor_output(0)* and receives *NULL* meaning that no outputs have been observed so far.
3. Decides to offer an input:
   (a) Computes the after $s$ delay closure by calling, where $s$ is the time delay from the last event observed by the *Driver*.
   (b) Computes the set of possible inputs: $\{$*click*$\}$.
4. Sends the input *click* and computes the reachable symbolic state set after *click* synchronization.
5. Checks for potential outputs and observes nothing.
6. Decides to delay 30 time units ($30 \times 10ms = 300ms$):
   (a) Schedules the time out after $30 - d$ and waits for output. $d$ is a delay from the last event.
   (b) *Driver* receives the output *singleClick* after $t$ (from last event) and puts into a queue.
   (c) The *TestAlgorithm* thread wakes up and picks the output action from the queue.
7. Computes the after $t$ delay closure.
8. Computes the set of possible outputs $\{$*singleClick*$\}$.
9. Verifies that the actual output is in the set of possible outputs.
10. Computes the reachable state set after *singleClick*.
11. Continues the algorithm in a loop which starts by calling *waitfor_output(0)...* until the testing time runs out.

The testing trace in Figure 32 reveals the following issues:

1. The time delay marked by $L$ is lost, i.e. it is neither considered nor applied on the reachable symbolic state set. If the symbolic state set is quite large, the delay $L$ can be large enough to interfere with the test event run, e.g. *MouseButton* could be waiting for a second click and produce a *singleClick* earlier than *TestAlgorithm* expects. The cause of the problem lies in the test algorithm design, which "does before it thinks", i.e. it is already too late to compute another after delay closure when we have the set of possible inputs already computed.
2. The time delay marked by $d$ might be larger than *TestAlgorithm* has decided to wait. This can be fixed by forcing the minimum for choosing the delay which can be extracted from *Driver*, but then we put additional constraints which leave the potential IUT errors untested.

3. The computation time on *UppAal* axis depend on the system model, which is hard to benchmark and plan ahead. On the other hand *TestAlgorithm* and *Driver* code is almost independent from the system model and rely only on the amount of possible inputs and outputs, which has logarithmic complexity.

The first two issues do not appear when using the simulated clock, which means that the test generation and execution algorithm has the desired functionality although the real-time handling is poor. For better real-time handling we propose one-step ahead planning: the test primitive should consist of one delay and input ahead; if the output is received before the input is offered, the symbolic state set must be discarded and the output action applied on the old symbolic state set.

### 4.4 Implementation Status

There have been numerous T-UPPAAL code files created and modified in order to provide automated test generation and execution capabilities. Table 1 summarizes the modifications in a parser library: we added a new grammar for reading test specification files. The new grammar is based on UPPAAL timed automata model specification, so only few rules needed to be added. Note that we omit the files we did not modify. Table 2 shows what files we have changed to support our test extensions in UPPAAL data structures. Table 3 describes our contributed new files to maintain reachable symbolic states, communicate with implementation under test, simulate the clock and a few samples of IUT for testing our extensions.

| **File name** | **Lines** | **Modified** | **Purpose of modification** |
|---|---|---|---|
| utap.hh | 46 | 4 | Function header for parsing. |
| builder.hh | 214 | 15 | IUT interface structures. |
| libparser.hh | 53 | 1 | Test spec. grammar identifier. |
| system.hh | 225 | 32 | IUT interface structures. |
| system.cc | 824 | 26 | Methods for IUT interface structures. |
| typechecker.hh | 378 | 14 | IUT interface typecheck structures. |
| typechecker.cc | 2379 | 98 | IUT interface typecheck and build. |
| lexer.ll | 204 | 5 | Keywords, microseconds handling. |
| parser.yy | 2038 | 97 | Grammar rules added. |
| pretty.cc | 824 | 38 | Parser testing. |
| **Total:** | 9052 | 330 | Test specification parser. |

Table 1: Modified parser files: test specification grammar added.

## 5 Experiments

In this section we describe how we make a system specification and how it looks like. We also use the specification for testing the test generation and execution algorithm.

Section 5.1 describes the first example of a sample system of a mouse button, which is simple to understand and test the basic functionality of the T-UPPAAL tool. The models of environment and implementation each has one automaton which communicate through observable actions. We make this example more complex by adding two or four buttons working in parallel where each of them has a separate clock.

| File name | Lines | Modified | Purpose of modification |
|---|---|---|---|
| chan.h | 141 | 54 | Channel I/O status information interface. |
| chan.cc | 80 | 13 | Channel I/O status information handling. |
| clock.h | 583 | 1 | Id of additional clock. |
| clock.cc | 1432 | 1 | Variable for id of additional clock. |
| integer.h | 285 | 10 | Variable name resolution to id. |
| integer.cc | 454 | 2 | Variable name index. |
| common_builder.cc | 458 | 45 | IUT interface transfer from parser to UPPAAL. |
| pw.hh | 32 | 2 | Interface of function creating *HashingPWList*. |
| pw.cc | 1604 | 5 | Implementation of function creating *HashingPWList*. |
| verifyta.cc | 2235 | 292 | Initialization and test algorithm implementation. |
| **Total:** | 31501 | 425 | IUT interface support and test algorithm. |

Table 2: Modified UPPAAL engine files.

| File name | Lines | Purpose |
|---|---|---|
| after.hh | 104 | State estimation interface. |
| after.cc | 1496 | State estimation implementation. |
| testdriver.hh | 157 | Test driver interface. |
| testdriver.cc | 210 | Test driver implementation. |
| simclock.hh | 98 | Simulated clock interface. |
| simclock.cc | 291 | Simulated clock implementation. |
| iut_button.hh | 45 | Button IUT interface. |
| iut_button.cc | 139 | Button IUT implementation. |
| iut_mouse.hh | 28 | Mouse IUT interface. |
| iut_mouse.cc | 44 | Mouse IUT implementation. |
| iut_traingate.hh | 28 | Train gate IUT interface. |
| iut_traingate.cc | 42 | Train gate IUT implementation. |
| **Total:** | 2682 | State estimation and communication with IUT. |

Table 3: New files created for test generation and execution.

For the second example we will make a more complex specification with many observable and unobservable actions. The implementation model has two automata and the environment model is active, i.e. has a clock.

## 5.1 Single Mouse Button

In this section we consider a simple implementation of a controller which decides whether it observed a single click or two clicks which form a double click. This combination is common using GUI. An operating system acts differently on a single or double-click. A single click selects an object on a screen and a double click executes a program related to the object (Figure 33).

A user presses a mouse button and each time the controller must interpret whether it was a single click or a double-click. In our case the mouse controller is an implementation under test and a user and an operating system forms the environment. To test whether the controller works correct we start from making a model using UppAal modeling tool. The mouse model

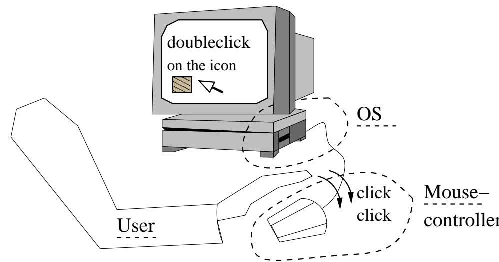
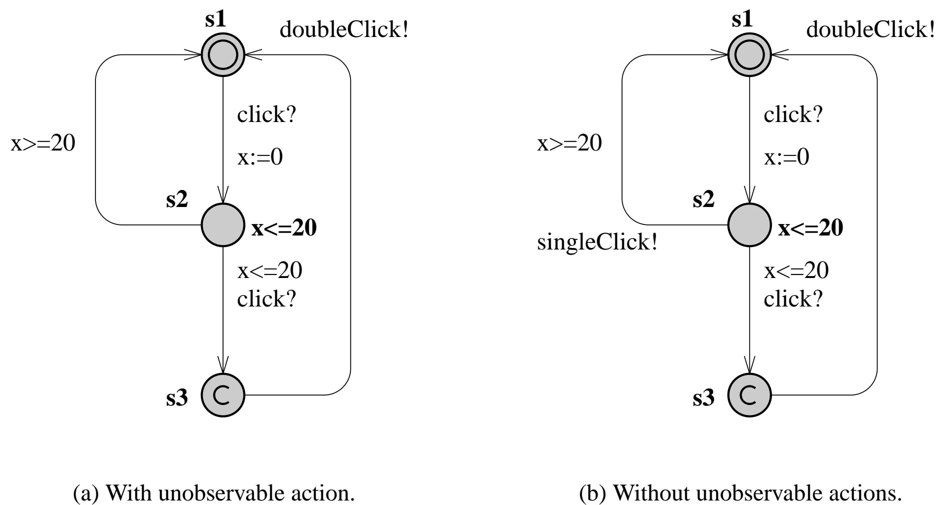



Figure 33: A general picture of a user and a mouse interaction.

contains an automaton or several automata which model behavior of the IUT and the general environment. There can be several ways to model the system. System models will be described in the following subsections.

### 5.1.1 Model of the Implementation for the Mouse-button system

The model of the IUT shows behavior of a controller in Figure 34. The first model has three states, one clock variable and two channels (Figure 34(a)).




(a) With unobservable action.

(b) Without unobservable actions.

Figure 34: Mouse double-click IUT model.

The IUT accepts clicks from the general environment and detects a double-click if the IUT received two clicks before 20 time units has passed. The automaton starts in the initial state in the location s1 and moves to the second location after the first click. The automaton moves into the third location s3 if the second click occurs before 20 time units have elapsed and produces a double-click on the next transition. The automaton moves silently to the initial location if 20 time units elapse and no click has occurred.

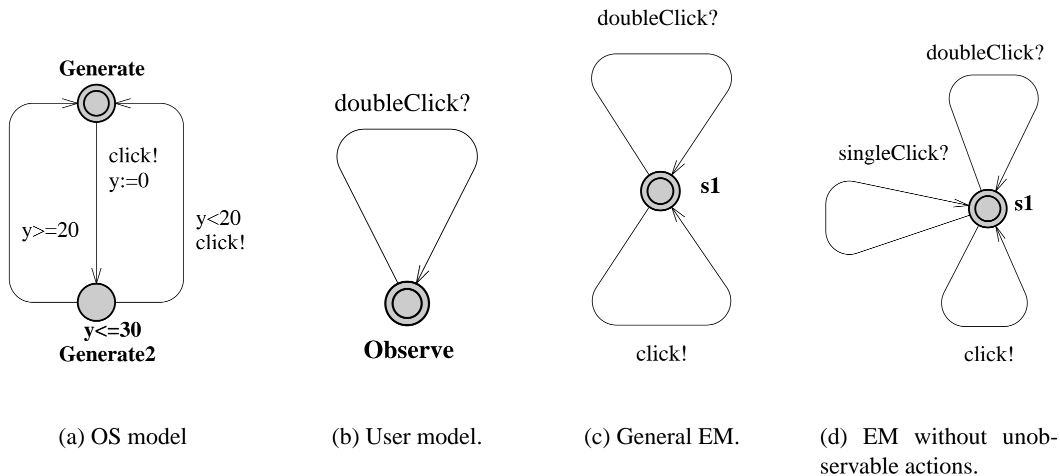
There is no silent action in the second model (Figure 34(b)) where we added an additional channel *singleClick!*. We used this model during debugging the test generation algorithm and wanted all actions to be observable. The implementation reveals time shift between the model clock and the IUT clock because of computational time. The *singleClick* helps to synchronize

clocks between an implementation and the UppAal engine. The as-synchronization occurs and becomes uncontrollable when we run the implementation and UppAal on the same machine with other processes running. The time shift is not experienced when time delays are longer (in order of seconds) and computation time is comparably short 3-4 milliseconds. A virtual clock is used to ignore computational time for short delays.

### 5.1.2 Models of the Environment for the Mouse button

We distinguish two parts of the environment - a user which performs clicks and an operating system, which observes a double click. The model of an OS is in Figures 35(a) and the user model in Figure 35(b). In the OS model there is a restriction that the second click should be produced at least within 30 time units. This invariant allows to proceed testing without long delays between clicks and still gives time for a single click and for a double click to be produced.

We use a more general model where behavior of the environment is combined in one model. The general environment model in Figure 35(c) has one state and two channel variables. It does not have any restriction about when a click is generated. We assume that at some point in time it will generate a click as well as a second click which would produce a double-click at the IUT. This gives freedom for a tester to simulate the environment, generate clicks at any time and observe output immediately when it is produced.




(a) OS model (b) User model. (c) General EM. (d) EM without unobservable actions.

Figure 35: Models of the environment.

The last model in Figure 35(d) has an additional input channel in case if only a single click has been produced. The channel synchronizes with controller from Figure 34(b) and in such a way the model contains only observable actions which are useful during debugging as it was explained in the previous section.

Later a model description is used for a test specification. We need to be sure that the models are correct otherwise the specification will be erroneous and produce wrong test runs. We verified whether the system model is deadlock free.

### 5.1.3 Sample Test Specification of the Mouse-button

We make a test specification from the system description file in .ta format which is used in UPPAAL. We add additional information such as a timeout of testing, testing precision units,

which stands for amount of microseconds used for one model time unit, testing inputs and outputs. A test specification file has a .tta file extension, which has the following additions to a .ta file:

```
input click();
output doubleClick();
precision 10000; // time in microseconds (0.01sec)
timeout 1000; // amount of precision units (1000x0.01sec=10sec)
```

We propose several potential test runs that can be used to test the correctness of the given mouse double-click detector.

| **Runs** | **Run1** | **Run2** | **Run3** | **Run4** |
|---|---|---|---|---|
| Events | click!<br>delay(1)<br>click!<br>doubleClick? | click!<br>delay(19)<br>click!<br>doubleClick? | click!<br>delay(21)<br>click! | click!<br>delay(10)<br>doubleClick? |
| Result | Pass | Pass | Fail | Fail |

Table 4: Potential test runs

The first two test runs pass a test because a double-click is produced by the IUT (Figure 34) after two clicks which occurred within a time period of 20 time units. The Run3 and Run4 fail the test. In the third run the IUT produced no output singleClick? and the second click occurred instead. In the fourth run the double click was produced after the first click which the model requires two clicks before a double-click.

Infinite number of test cases can be generated from the specification because there can be infinite number of delays. Therefore an event selection selection strategy becomes a very important issue and we do it by random choices.

### 5.1.4 Implementation of the mouse button

We display a part of a sample mouse implementation code in Figure 36. The whole file iut_button.cc code is displayed in the Appendix **??**.

A message sequence chart in Figure 37 shows 40 out of 1000 time units interval of a test sequence. A time line of a global clock is displayed on the column Time. Delays are on the left from the time line and on the right are displayed values of a global clock at a certain state of testing. We assume that the initial state is left as soon as possible and there is no delay before the first action *click*. An output occurs immediately, i.e. time is not allow to pass if an output is available.

From the message chart diagram we can write a test sequence $click? \cdot 20 \cdot click? \cdot 0 \cdot doubleClick! \cdot 0 \cdot click? \cdot 20 \cdot singleClick! \cdot 0 \cdot click?$. This test sequence has delays which are equal to maximum values of the invariant constrain in the controllers state *S2* and on guards of the transitions leading from that state.

We might want to use the sequence again for testing another IUT. Unfortunately there is a small probability that the environment automaton repeats this sequence again because the environment automaton can generate inputs at undefined moments. A solution of the problem is writing another environment automaton which would simulate that particulate sequence as proposed in [14].

```
void* MouseButton::execute()
{
    struct timeval now;
    struct timespec timeout;
    pthread_mutex_lock(&click_m);
    sem_post(&started_s);// acknowledge the start
    while (!stop) {
        pthread_cond_wait(&click_c, &click_m);// wait for click
        if (stop) break; // it was a signal to stop
        else sem_post(&click_s); // acknowledge that we got the click                10
        gettimeofday(&now, &tz);
        now += 195000;
        timeout.tv_sec = now.tv_sec;
        timeout.tv_nsec = now.tv_usec*1000;
        int res = pthread_cond_timedwait(&click_c, &click_m, &timeout);

        if (res == ETIMEDOUT) {
            driver->report_now("singleClick()");
        } else {
            driver->report_now("doubleClick()");                                   20
            sem_post(&click_s);
        }
    }
    pthread_mutex_unlock(&click_m);
    return NULL;
}
```

Figure 36: A mouse button implementation.

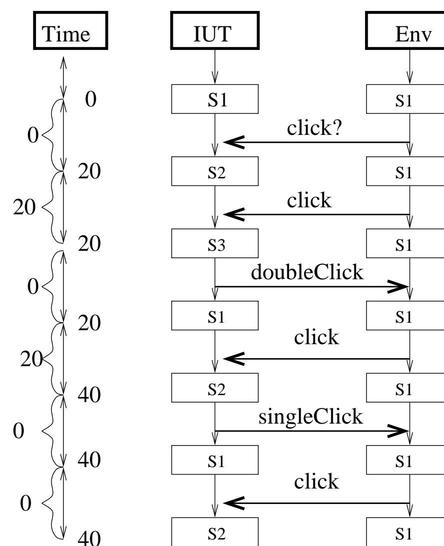



Figure 37: Message sequence chart of the mouse with one button system.

This automaton is a test purpose automaton made for simulating one test sequence displayed in Figure 38 and it replaces the environment model automaton. Channel names with a mark "?" means outputs from IUT and "!" means inputs into IUT. An IUT passes the test run if it ends in

the terminal state *Pass*. The implementation fails a test run if it outputs a wrong output, i.e. the automaton deadlocks in the state *Fail*.

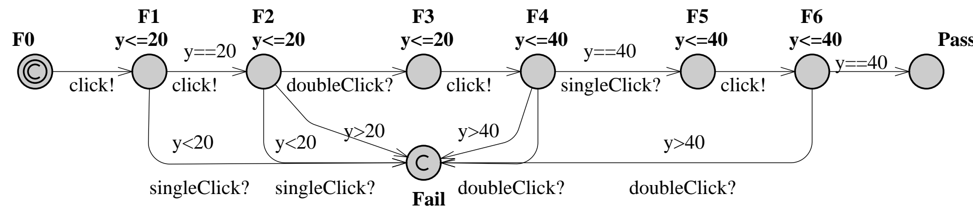



Figure 38: A test purpose automata without clock resets for a test sequence $click? \cdot 20 \cdot click? \cdot 0 \cdot doubleClick! \cdot click? \cdot 20 \cdot singleClick! \cdot 0 \cdot click?$.

We use a test purpose automaton which has a separate clock. The clock simulates global time and is never reset, and can track time passing during test execution in Figure 38.

Another possibility is to have resets on every transition as in Figure 38. In such a way we do not need to remember what the global time is at a certain state or transition. It is easier to keep track of delays by reseting a clock but resets may take some time and test execution takes longer.

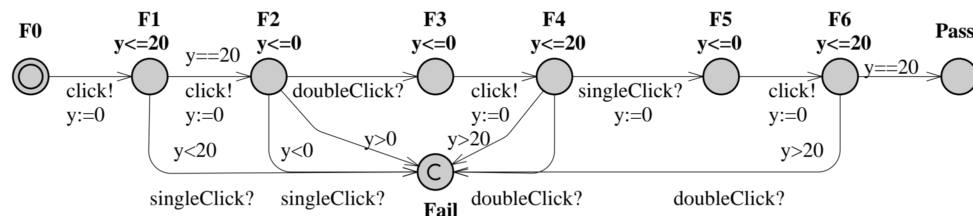



Figure 39: A test purpose automata with clock resets for a test sequence $click? \cdot 20 \cdot click? \cdot 0 \cdot doubleClick! \cdot click? \cdot 20 \cdot singleClick! \cdot 0 \cdot click?$.

In a next step we make a specification with the test automaton instead of the environment automaton. During test execution we do not know in which location of a test automata testing ends therefore we need a practical solution to identify a test verdict. We make a *Fail* location committed to identify failure right away if the system deadlocks in that state. A test is passes if a timeout expires and no failures are detected, i.e. the system is deadlocked in the *Pass* location or a test is inconclusive if the system is deadlocked in a non terminal state.

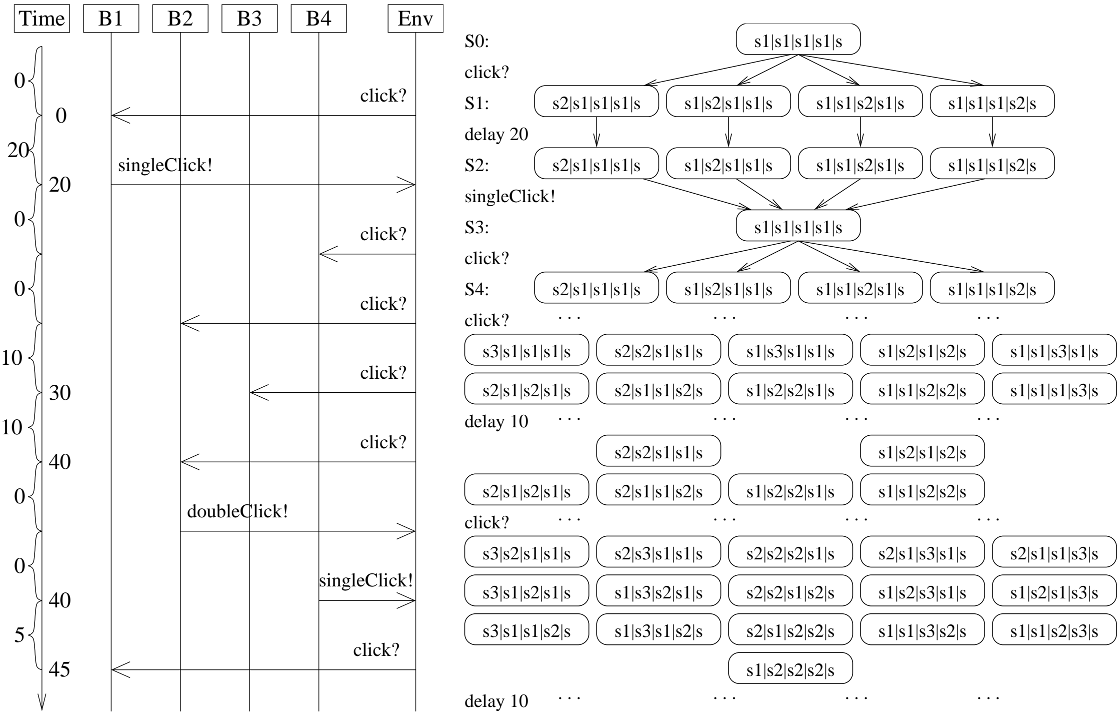
### 5.1.5 Multi-Button Mouse

The single button experiment tests the basic functionality of T-UPPAAL where the tool uses only one clock and one symbolic state in computations, therefore more elaborate tests are needed. For experiment to be exhaustive and still simple to understand and implement we made a mouse with several independent buttons, which does not have much sense in a real world but generates reachable symbolic states with several clocks. In addition, we add non-determinism when choosing which button is receiving inputs and sending outputs, i.e. the environment is not aware of which button is pressed when it sends a click, and which button is sending a *singleClick* or

*doubleClick* notification. In this case, the IUT network consists of several mouse-button automata and the environment network is the same as in Section 5.1.2. The simulation code for IUT requires an additional class *Mouse* which encapsulates several objects of class *MouseButton* (see Figure 30). When a driver is calling the method *Mouse::perform(action)*, the mouse chooses randomly a button and passes the message to it. The output is produced by the objects of class *MouseButton* themselves like in previous example. The mouse button properties (like delay between clicks) are kept the same as in the single button example.

Figure 40(a) shows a test message sequence diagram for a mouse controller with four buttons. Delays are displayed on the left and values of the global clock on the right of the time line. As we can see the environment sends *click* and one of the randomly chosen button accepts it. If two clicks arrive within 20 time units into the same button then it produces an output *doubleClick* and if the button gets just one click within 20 time units then a *singleClick* is produced.

Figure 40(b) shows the symbolic state estimation (without zones) as we expected it to be growing. But in practice it is even worse: the symbolic state set $S3$ contains four symbolic




(a) Message sequence chart. (b) Expected symbolic state set estimation.

Figure 40: Testing of 4-button mouse.

states instead of one, just because not all clocks in the system were reseted, i.e. all symbolic states contain the same location vector $\langle s1, s1, s1, s1, s\rangle$ (as shown $S3$ in Figure 40(b)), but the zones are different because the first zone had a reset on clock $B1.x$, the second zone had a reset on clock $B2.x$ and so on. The situation goes even worse as the time elapses and more clicks are produced (only *click* resets the clock). The delay and the output of *singleClick* and *doubleClick* reduce this "clock reset" uncertainty (e.g. going from symbolic state set $S5$ to $S6$), but not so dramatically as it grows after a *click*. We may attempt to reset all clocks by sending series of *click* inputs in a zero time, but this will not help since we have no control over which button

receives a *click* and there always is a positive probability that some clock was never reseted and this is reflected on the symbolic state set.

Figure 41 shows how uncertain we are about clock value when we have the IUT network of two equal timed automata with single location and single edge with action synchronization and a reset on the local clock $x$. The clock $t$ is our additional global clock for tracking the time. The initial clock value is $0$. We apply a bounded future operator to estimate symbolic states after a delay. Then we offer an input which resets one of the clocks, we do not know which, therefore we remember both as potential. In such a way from a set with single symbolic state we move the set of two symbolic states (the first has the first automatons clock reset, and on second - the second automatons clock reset). After another delay and action round we add one more symbolic state in the same fashion and so on. For the two clock reset problem we have

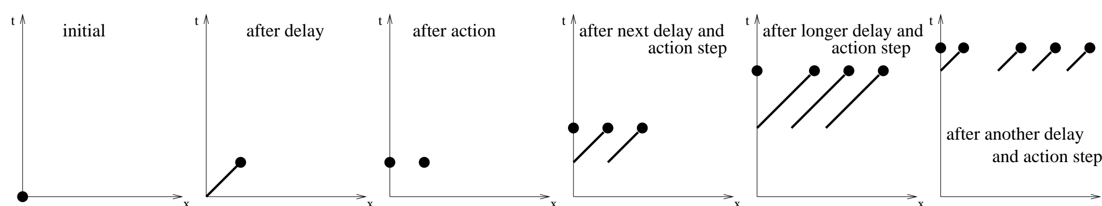



Figure 41: Uncertainty in clock $x$ value growing on a single location automaton with one observable action.

one additional symbolic state at first and the each following step adds two additional symbolic states. For the problem with three potential clock resets we would add two additional symbolic states after the first step and after second step we have 9 symbolic states in a set. However it is difficult to generalize the formula now and we leave it for future.

### 5.1.6 Experiment Results of Mouse-click systems

We tried 2-button and 4-button mice and experiments showed that the test algorithm works with many clock as well as with one, however the reachable symbolic state set grows exponentially because of uncertainty in a button selection. As you may already noticed, T-UPPAAL has to compute all possible combinations of each button receiving a click, thus one *click* multiplies the symbolic state set by four in 4-button mouse experiment. It was observed during experiments that uncertainty decreases when a *singleClick* or a *doubleClick* is received.

As time elapses, the symbolic state set grows and shrinks (but still grows more than shrinks) and eventually the “simulated time” timeout is reached which means the test passes without faults found. On one hand we reached our goal in testing the symbolic state set estimation algorithm with thousands of states. On the other hand we know that simple models can trigger exponential amount of reachable symbolic states. This state explosion remains an open issue.

## 5.2 Train Gate Controller

We take a more complex example of an implementation under test to evaluate the suitability of T-UPPAAL to more practical problems, for example, bridge crossing. A more complex specification in terms of number of models, channels, variable sharing, etc. may reveal different problems than the previous one during debug of the test generation and execution algorithm.

We take a demo system from the UPPAAL examples which is a model of a train gate controller. For more details about this example see [13].

A railroad bridge often has only one or two railroads and is a bottle neck for trains which approach from different directions and from many railroads. Malfunctioning of the gate controller system may cause unplanned train delays or even a collision and catastrophe. In a real life the gate system assures that only one train can cross the bridge at a time (Figure 42). The gate sends a signal *Stop* to an approaching train if another train is crossing the bridge and signals *Go* when the bridge is empty and available for crossing. Trains inform the gate when they are approaching (*Approaching*) and leaving (*Leaving*) the bridge .

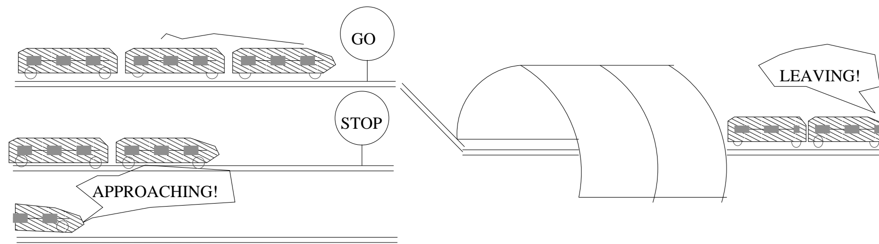



Figure 42: A general picture of a train bridge system.

The system models show behavior of four trains on different railroads and one gate for a bridge with one railroad. Our goal is to test behavior of the gate system. In the demo system the implementation model of the gate system consists of two automata: a gate controller and a queue (Figure 43). The environment model consists of four train automata.

The environment and the implementation models synchronize through observable actions *appr, leave, go, stop* and they share a variable *e* for passing information about train sequence in the queue (Figure 43). The implementation automata - gate and queue - synchronize internally through actions *add, rem, empty, nonempty*. We will write actions or channel names in a small letter and model locations names in a capital letter.

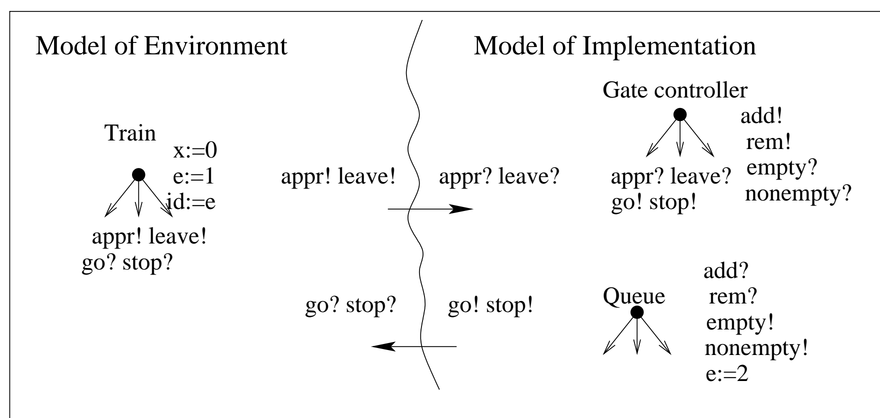



Figure 43: The train demo system environment and implementation models in a single timed automata network model.

We had to adapt the train-gate demo example to avoid shared variables between the environment and the implementation because the variable *e* was assigned in both the queue and the train model.

We could separate the environment and the implementation by separating *e* variables into

two, where one gets a new value during assignment in the queue model and another is assigned in the train model. Variable value is passed as a parameter during synchronization on the *appr* channel. This case was mentioned in Section 2.9 but it is not suitable for our implementation of the test generation algorithm which at the present can not handle parameter passing through channels.

We need to use an alternative way to model the data exchange. We introduce unique channel names for every shared variable value and in such a way the variable is not used in the environment model. By doing that we follow the second suggestion for data variable synchronization written in (Section 2.9).

The new model setting is shown in Figure 44. For the interface between the environment and the implementation we introduce a unique channel name for every train. Now instead of four we have sixteen observable channels and a separate automata for every train (Figure 44). We use a shared variable *e* to pass information about a train identifier inside the gate system, i.e. between the controller and the queue.

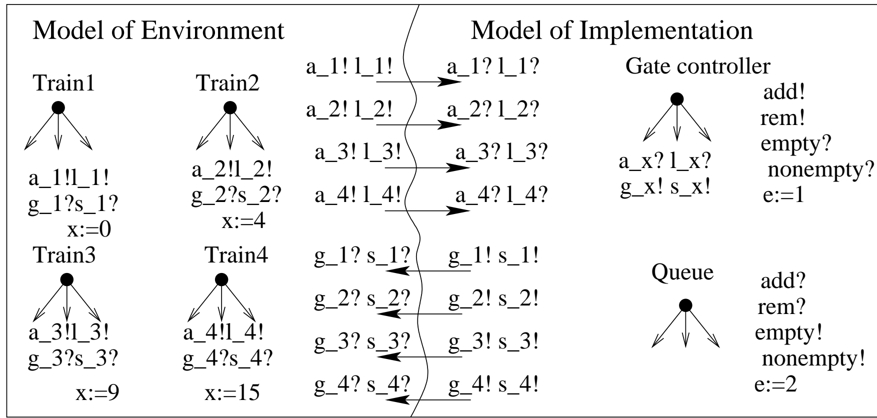



Figure 44: The train bridge system environment and implementation models in a single network model.

In the following sections we describe the train-gate system environment and implementation models used in the demo example and adapted for our test specification.

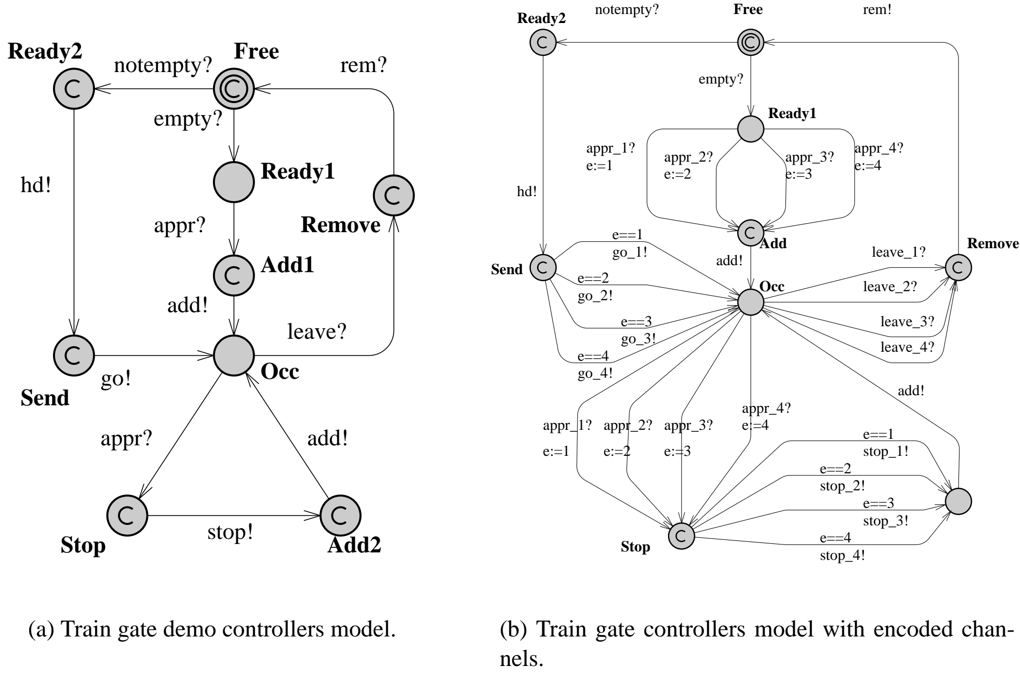
### 5.2.1 Model of the IUT in the Train-gate system

As mentioned above our IUT is a gate system model consisting of a controllers and a queue automata. It is responsible for letting the trains go if the bridge is available or stopping them if another train is crossing the bridge.

We had to modify the controller automata to make it suitable to use for a specification. Modifications concerns channel names, number of transitions and a train identification variable *e*. The demo example controller is in Figure 45(a) and the modified controller in Figure 45(b)). We use the same queue automata as in the demo example (Figure 46).

Only the controller model synchronizes with the environment, i.e. has observable actions: input action *appr* (for approaching), *leave* and output actions *stop* and *go* (Figure 45(a)).

We can see in Figure 45(a) and 45(b) that both controller automata have five states. The initial state *Free* of the controller model indicates that the bridge is available for crossing. If the list with waiting trains is empty then the controller moves to the state *Ready1*. When a train




(a) Train gate demo controllers model.

(b) Train gate controllers model with encoded channels.

Figure 45: Train gate IUT models.

approaches the gate, the controller gets an *appr* action from the train and immediately sends an order *add* to the queue model to add the train identifier into a list. The controller signals *rem* to the queue only after the train has left the bridge and has sent a *leave* signal to the controller.

If an approaching train is not the first in the queue then the controller is in the state *Occ*. The controller signals *stop* to the train and *add* to the queue for adding the trains identifier into the list. The train is allowed to *go* when the controller is in the *Send* state. Before that the controller sends *hd* action to the queue for updating the index of the train.

The number of transitions is different but behavior of controllers is equivalent. In our controller model every channel is used for synchronizing only with a particular train on a separate transition in Figure 45(b). The channel identify the train model to which the controller synchronizes.

The queue is considered as entirely internal part of the IUT Figure 46. It assist to the gate controller model by implementing a FIFO list of train identifiers. The gate controller and the queue model synchronize through internal actions *add*, *rem* (for adding or removing from a list), *empty*, *nonempty* and *hd* (for checking the list and updating the train index variable *e*).

We use a shared variable between the controller and the queue model for passing a train identifier number. This variable is used in transition guards in the controller to enable a synchronization only with a certain train, which makes it different from the mouse button example.

### 5.2.2 Models of the Environment in the Train-gate System

The environment of the system consists of several trains which want to cross the bridge but first they need to inform the gate when they approach or leave. They also have to obey the orders from the gate to stop or to go. Trains have limited time for crossing the bridge and a certain delays before they can start crossing the bridge.

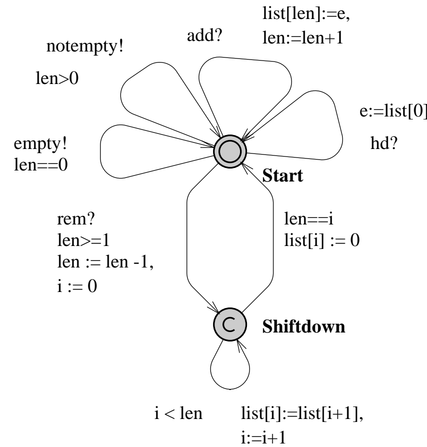



Figure 46: Queue model.

Instead of one train model Figure 47(a) we made four - each for every train, see Figure 47(b). Every train model has five states, one local clock variable and four channel variables which identify a particular train, ex.: *Appr_1* identify *Train1* and *Appr_3* identify *Train3*. The shared train identifier variable *e* is not used anymore in our model and in such a way we avoid data value passing during synchronizations between the environment and the IUT.

A train sends a signal *appr* to the gate and waits in the state *Appr* up to 20 time units. The train model moves to the state *Stop* within 10 time units if it gets an order to stop from the gate model. It is important that the train gets stop before 10 time units if the crossing is not free. After 10 time units the train can go from the state *Appr* to the *Cross* state. It can stay there up to 5 time units or leave the bridge at or after 3 time units by sending a signal *leave* to the gate.

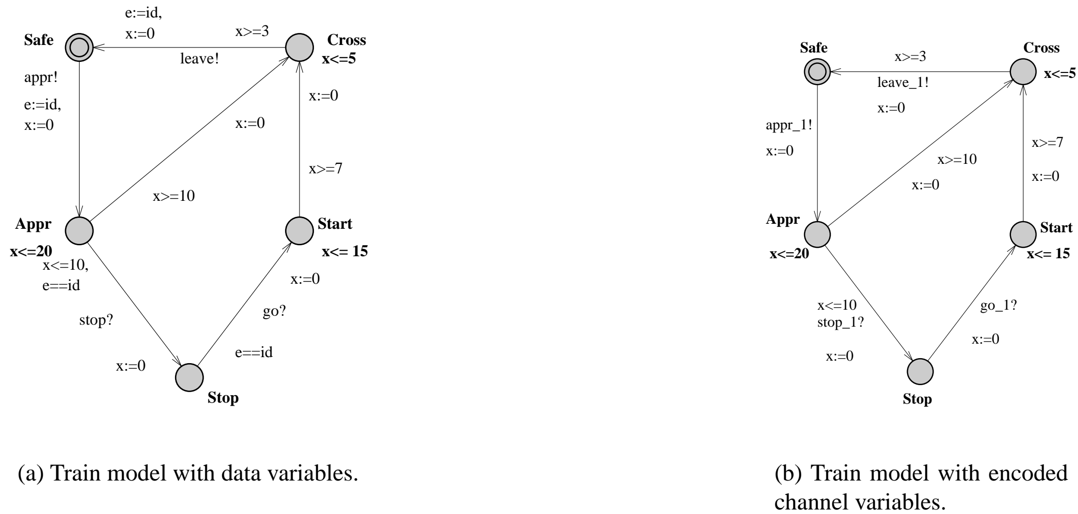
The train stays in the *Stop* state until the gate sends a permit to *go* and the train moves to the *Start* state. From this state the train moves to the *Cross* state after 7 time units but not later than 15 time units elapses.

The train gate environment is specific because it can not offer any input defined in the implementation interface at any time. A train can send a signal *leave* only after it has sent a signal *appr*.

As we can see in Figures 47 the environment model allows delays within a certain interval at different states or has clock guards on transitions. The IUT model outputs appropriate actions immediately after synchronization with the train or the queue model. Therefore it does not need clock variables and uses committed locations instead. Such behavior suites requirements and possibilities in a realistic train-gate system. It takes some time for a train to start moving from a place and that time is different from train to train because of a train length and weight. The gate controllers have to respond as fast as possible and small delays during mechanical railroad switches is not considered.

### 5.2.3 Test Specification of the Train-Gate

The test specification includes environment and IUT models. As we could see from train-gate models this specification is more complex that the mouse button specification because the IUT has the controller and the queue model which communicate internally. However IUT is simpler because it does not have a clock and timing is not an important issue.




(a) Train model with data variables.

(b) Train model with encoded channel variables.

Figure 47: The train models.

We add IUT interface part to the system model description (input and output actions), test timeout and precision of time units information to the system description file and use it as a system specification file for test generation. We name all observable actions for every train.

```
input appr_1(), appr_2(), appr_3(), appr_4(),
      leave_1(), leave_2(), leave_3(),leave_4();
output go_1(), go_2(), go_3(), go_4(),
       stop_1(), stop_2(), stop_3(), stop_4();
precision 1000000; // time in microseconds (1sec)
timeout 1200; // amount of precision units (1200x1sec=20min)
```

Even though the demo and our models differ slightly we wanted to keep them equivalent in the sense that they satisfy the same properties. We verified the following properties:

```
A[] not( deadlock ) // We verified if the model is deadlock-free

Train1.Appr --> Train1.Cross. // a train eventually crosses the
                                        bridge if the train has approached.

A[] not( ( Train1.Cross and ( Train2.Cross or Train3.Cross or Train4.Cross ) )
      or ( Train2.Cross and ( Train3.Cross or Train4.Cross ) )
      or ( Train3.Cross and Train4.Cross ) ) // no collisions in the
                                            critical section - the state Cross
```

We can consider the specification as partially defined with strong input enabling. There is a state *Ready1* where input *leave* would not be accepted and then we need to add a transition for channel *leave* going from and to the location *Ready1*. However we consider the specification as completely defined with weekly input enabled states, because the environment behavior assures that the same train can not send a *leave* action without sending *appr* first.

### 5.2.4 Implementation of the Train-Gate

In this section we demonstrate the implementation of the train-gate system. This implementation was used for debugging the algorithm.

```
void TrainGate::perform(const char* action)
{
    if (strncmp(action, "appr_", 5) == 0)    {
        char stopmsg[] = "stop_x()";
        stopmsg[5] = action[5];
        if (train_list.empty()) train_list.push_back(action[5]);
        else {
            driver−>report_now(stopmsg);
            train_list.push_back(action[5]);
        }                                                              10
    } else if (strncmp(action, "leave_", 6) == 0) {
        assert(!train_list.empty() && train_list.front()==action[6]);
        train_list.pop_front();
        if (!train_list.empty()){
            char gomsg[] = "go_x()";
            gomsg[3]=(char)train_list.front();
            driver−>report_now(gomsg);
        }
    } else cout << "Train: unacceptable action, ignoring.." << endl;
}                                                                      20
```

Figure 48: Train gate controllers implementation.

The implementation code for the member function *perform* is displayed in Figure 48. Action names are implemented in a string format, i.e. *appr_1* = "appr_1". When the implementation gets an action then it looks for the last symbol *x* in the actions name. The symbol indicates a train which sent the action. The implementation uses the symbol in combination with the action for reply to the same train.

When a train sends an action *appr_* then a train list is checked. The *train_list* variable is of a vector type variable therefore appropriate functions for a vector type variable are used. If the train list is empty the action is pushed into the list otherwise a stop message is sent to the train and only then the *appr_* action is pushed into the list. If *leave_* action is received then a *appr_* message with appropriate ending symbol is popped from the list. If the list is still not empty then the train whose action *appr_* is the first in the list gets an action *go_*.

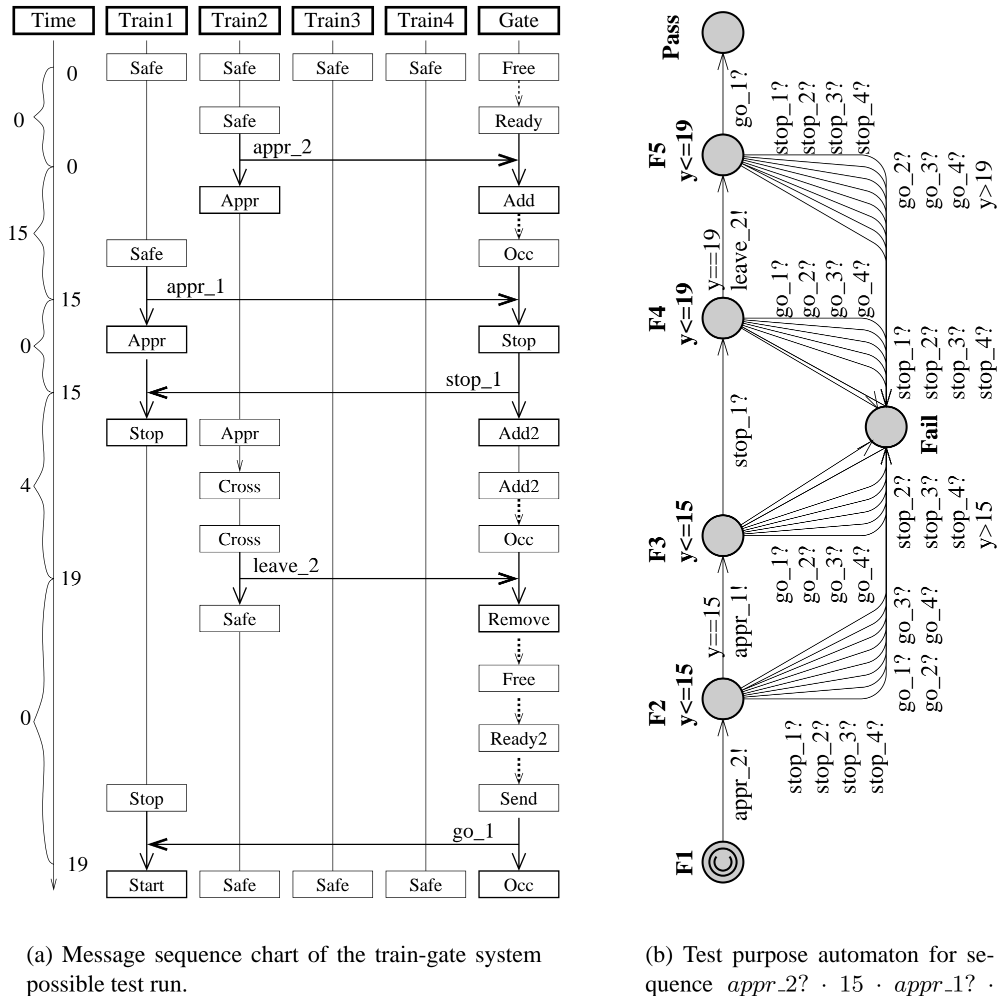
Possible communication between the IUT and environment is displayed in the message sequence chart in Figure 49(a). Only observable actions are displayed in the chart and in this part of a test run only 2 trains out of four communicate with the gate. The states that observable actions leads to are displayed in a bold-lined frame. Unobservable actions are displayed in a dotted line. They change the state of the gate system as a result of internal communication between the controller and the queue. Trains change the state during transitions without synchronization.

The Time column in the message sequence chart shows a time line of a global clock. Delays are displayed on the left from the time line. Clock values at certain states are diplayed on the left from the time line.

We can write a test sequence looking at the message sequence diagram and it is $appr_2? \cdot 15 \cdot appr_1? \cdot 0 \cdot stop_1! \cdot 4 \cdot leave_1? \cdot 0 \cdot go_2!$ , where 15 and 4 means delays between actions.

As we did in the mouse button example we make a test purpose automaton as proposed in [14] and displayed in Figure 49(b). An IUT passes the test if it ends in the terminal state *Pass*. The implementation fails a test if it outputs a wrong output, i.e. the automaton deadlocks in the

*Fail* state.




(a) Message sequence chart of the train-gate system possible test run.

(b) Test purpose automaton for sequence $appr_2? \cdot 15 \cdot appr_1? \cdot stop_1! \cdot 4 \cdot leave_1? \cdot go_2!$.

Figure 49: Train-gate testing events and a test purpose extracted.

In train test automaton we have to think and model all faulty implementation outputs. It becomes more complicated the more if an implementation has many different outputs. Non acceptable delays should also be modeled in the test automaton as it is in the states *F3, F5*. From this test purpose we can also think of a test run which should produce a failure, ex.: $appr_2?\ stop_2\ appr_1?$. Such a test run reveals whether the algorithm reveals non-conforming IUT.

### 5.2.5 Train-gate Experiment Results

As we mentioned before the train-gate system has a specific environment, i.e. one which has clocks and is restricted to offer certain inputs only at specified moments in time. We experienced problems in test generation and execution when dealing with invariants and "greater than" guards:

**minimum delay:** when the environment is in a location where all outgoing transitions are restricted by "greater than" guards T-UPPAAL must delay at least until one of the guards is satisfied otherwise the set of possible inputs is empty. Therefore a test generation and execution algorithm must compute a *minimum delay* based on the environment model before trying to offer any inputs. Unfortunately we did not foresee such situation and T-UPPAAL throws away all the symbolic states which do not have observable input to offer. A quick fix for this could be to postpone the input offer and choose another delay action. The fix requires to backup the symbolic states set before putting it into destructive *AfterActionFilter*.

**maximum delay:** when the environment is in a location with invariants specified, T-UPPAAL must not delay longer than the invariants allow, otherwise we break a constraint for the environment behavior and the state estimation engine will delete possibly the last legal symbolic state. Therefore a test generation and execution algorithm must compute a *maximum delay* allowed, which is the maximum of the minimums of differences of an invariant constraint boundary and clock's lower boundary, i.e. if $N_E$ is the network of the environment timed automata, $Z$ is a set of symbolic states, $[z_{i,j}]$ is a matrix representing zone $z$ and $Inv(l_k)_i$ is an invariant constraint boundary for clock $i$ on timed automaton $k$ location $l_k$, then:

$$\delta_{max} = max\Big\{min\{Inv(l_k)_i - z_{i,0} \mid k \in N_E, \forall i\} \mid \langle \bar{l}, z\rangle \in Z,\ \bar{l} = \langle l_k \rangle\Big\}$$

The minimum and maximum delay computation may seem complex and expensive but it should be easy to integrate into a *AfterDelayFilter* redesigning it to maintain additional properties called *minDelay* and *maxDelay*. For example the following formula can be used to extract the maximum delay while computing a bounded future operator with additional clock $t$:

$$\delta_{max} = (((z \wedge I)_{t:=0})^{\uparrow} \wedge I)_{t,0}$$

which means that we intersect zone $z$ with the invariant system $I$ on a given symbolic state locations, reset the special clock $t$ (which is already included in our earlier computations), apply future operator, intersect with the invariant system $I$ and take the $t, 0$ constraint which is the upper boundary for clock $t$. The calculations must be performed maintaining the canonical form of the matrix. The later formula does not distinguish invariants on the environment from invariants on the IUT, i.e. it will also consider when the IUT is supposed to produce the next potential output, which does not interfere with the test generation and execution algorithm.

## 5.3 Performance Issues

We want to measure the UPPAAL code performance and for that purpose we use a program execution profiling tool GNU profiler *gprof*. We get a flat profile and a call graph of the program which was run on a test specification file. A part of the flat profile files after running the algorithm on the mouse system with one, two and four buttons are displayed in Tables 5, 6, 7.

The table has such a field:

- *time*- the percentage of the total execution time the program spent in this function. These should all add up to 100.
- *cumulative s*- the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

| Time | Cummul sec. | Self sec. | Calls | Self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 71.90 | 226.88 | 226.88 | 15e+04 | 1.52 | 1.52 | PWListBase:: Enum::tryGet() |
| 24.31 | 303.57 | 76.69 | 7e+04 | 1.03 | 1.03 | PWListBase:: clear() |
| 1.19 | 307.33 | 3.76 | | | | internal_mcount |
| 0.14 | 307.77 | 0.44 | 15e+04 | 0.00 | 0.00 | SimClock::timedwait () |
| 0.10 | 308.07 | 0.30 | | | | ostream::operator¡¡ () |
| 0.09 | 308.36 | 0.29 | 7.4e+04 | 0.00 | 0.01 | InternalTransitionFilter:: tryPut() |
| 0.09 | 308.63 | 0.27 | 8.7e+04 | 0.00 | 0.01 | ExternalSuccessorFilter:: tryPut() |
| 0.09 | 308.90 | 0.27 | 1 | 270.00 | 3e+05 | handleTestGeneration() |
| 0.06 | 309.09 | 0.19 | 15e+05 | 0.00 | 2.05 | AfterDelayFilter:: tryPut() |
| 0.06 | 309.28 | 0.19 | 7e+04 | 0.00 | 0.00 | ExternalTransitionFilter:: tryPut() |

Table 5: Flat profile of testing execution for a one-button mouse

| Time | Cummul sec. | Self sec. | Calls | Self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 69.80 | 161.96 | 161.96 | 3e+04 | 5.58 | 5.58 | PWListBase::Enum:: tryGet() |
| 22.08 | 213.19 | 51.23 | 4e+04 | 13.33 | 13.34 | PWListBase:: clear() |
| 1.70 | 217.13 | 3.94 | | | | internal_mcount |
| 0.37 | 217.99 | 0.86 | 3.8e+04 | 0.02 | 0.02 | UnpackedRegion:: addClose() |
| 0.27 | 218.62 | 0.63 | 3e+04 | 0.02 | 0.07 | ExternalSuccessorFilter:: tryPut() |
| 0.27 | 219.25 | 0.63 | | | | _mcount |
| 0.19 | 219.68 | 0.43 | 29e+04 | 0.00 | 0.00 | UnpackedRegion:: isEmpty() |
| 0.16 | 220.06 | 0.38 | 9e+04 | 0.00 | 0.00 | UnpackedRegion:: unpackTo() |
| 0.16 | 220.42 | 0.36 | 68e+04 | 0.00 | 0.00 | Constant:: evaluate() |
| 0.16 | 220.78 | 0.36 | 0.8e+04 | 0.05 | 0.05 | SimClock::timedwait() |
| 0.13 | 222.73 | 0.31 | 1 | 310.00 | 2.2e+05 | handleTestGeneration(void) |

Table 6: Flat profile of testing execution for a two-button mouse.

- *self s* - the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.
- *calls* - the total number of times the function was called.
- *self ms/call* - the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.
- *total ms/call* - the average number of milliseconds spent in this function and its descendants per call. This field in also used in the call graph analysis (Figure 8).
- *name* - this is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds and calls fields are sorted.

As we can see in Table 5 most of the time takes functions for removing states from a store with a *PWLISTBASE::clear()* function and enumeration nodes in a hash table. The handleTest-Generation procedure takes a small amount of execution time but total ms/call is very big because a children function *AfterDelayFilter::tryPut()* contains above mentioned functions. A

| Time | Cummul sec. | Self sec. | Calls | Self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 43.61 | 532.6 | 532.62 | | | | internal_mcount |
| 13.28 | 694.87 | 162.25 | 1.8e+09 | 0.00 | 0.00 | PWListBase::clear() |
| 6.95 | 899.21 | 84.83 | 1.8e+09 | 0.00 | 0.00 | GlobalState::relation () |
| 6.75 | 981.64 | 82.43 | 1.4e+06 | 0.06 | 0.46 | HashingPWList:: tryPut() |
| 9.79 | 814.38 | 119.51 | 1.8e+09 | 0.00 | 0.00 | UnpackedRegion::relation () |
| 6.95 | 899.21 | 84.83 | 1.8e+09 | 0.00 | 0.00 | GlobalState::relation () |
| 3.94 | 1082.28 | 48.14 | 1.8e+09 | 0.00 | 0.00 | IntegerTable::operator== () |
| 3.84 | 1129.15 | 46.87 | 1.8e+09 | 0.00 | 0.00 | PWListBase::WaitingNode:: getNode() |
| 3.51 | 1172.03 | 42.88 | | | | _mcount |
| 1.80 | 1193.99 | 21.96 | 1.8e+09 | 0.00 | 0.00 | PWListBase::Node:: isWaiting() |
| 0.00 | 1221.02 | 0.04 | 1 | 40.00 | 6.4e+05 | handleTestGeneration(void) |

Table 7: Flat profile of testing execution for a four-button mouse.

call-graph of the handleTestGeneration procedure is displayed in Table 8 and it shows performance of every function and its children in terms of a time used and number of calls into a certain function.

A similar flat-profile is obtained from test execution on a two-button mouse system in Table 6. Situation changes when we have a system with much bigger state grow as in the mouse system with four buttons. The flat profile of this system is in Table 7. In such a system the most time takes comparison of states (*GlobalState::relation*). We can say that hashing tables are more useful when working with systems with many states. Hashing functions gives an obvious time overhead when working with smaller systems.

| Time | Self | Children | Called | Name |
|---|---|---|---|---|
| | 0.04 | 645.42 | 1/1 | main [1] |
| 54.8 | 0.04 | 645.42 | 1 | handleTestGeneration(void) [3] |
| | 1.50 | 321.53 | 509/1017 | BufferredFilter::tryPut(void *) [49] |
| | 0.99 | 213.51 | 338/1017 | AfterDelayFilter::tryPut() [5] |
| | 0.50 | 107.39 | 170/1017 | AfterActionFilter::tryPut() [82] |
| | 0.00 | 0.00 | 1/1 | createAfterDelayFilter() [110] |
| | 0.00 | 0.00 | 1/1 | createAfterActionFilter() [112] |
| | 0.00 | 0.00 | 553/723 | Pointer¡Action¿::operator=(Action *) [109] |
| | 0.00 | 0.00 | 94/634 | Pointer¡Action¿:: Pointer(void) [2583] |
| | 0.00 | 0.00 | 170/173 | AfterActionFilter::setProperty() [118] |
| | 0.00 | 0.00 | 338/341 | AfterDelayFilter::setProperty() [172] |

Table 8: Call graph. Granularity: each sample hit covers 4 byte(s) for 0.01 sec.

A number in the brackets shows an index of a function. The column *Called* shows two numbers: the first informs how many times a parent function called the child function and how many times other functions called that child function.

*BufferedFilter::tryPut* and *AfterDelayFilter::tryPut()* functions takes most of the time compare to other functions but it is a result of performance of their children function. Such a use of time is unavoidable, but it may be optimized using optimization options of a compiler.

Another way for measuring performance of the program is how many time it takes to calculate a set of states for different implementations with different set of states. In Table 9 we tried to visualize dependency of a time consumed during an event and a set of states calculated.

| IUT | States | Events | User time Sec |
|---|---|---|---|
| mouse1 | 1 | Input | 0.01-0.02 |
| | | Output | 0.089-0.012 |
| | | Delay | 0.088-0.089 |
| mouse2 | 1-8 | Input | 0.015-0.045 |
| | | Output | 0.024-0.042 |
| | | Delay | 0.085-0.091 |
| mouse4 | 1-100 | Input | 0.01 |
| | | Output | 0.01 |
| | | Delay | 0.01 |
| mouse4 | 100-1000 | Input | 0.01-0.05 |
| | | Output | 0.01-0.05 |
| | | Delay | 0.01-0.26 |
| mouse4 | 1000-10000 | Input | 0.07-0.51 |
| | | Output | 0.07-0.53 |
| | | Delay | 0.12-12.6 |
| mouse4 | 10000-103344 | Input | 0.59-2.44 |
| | | Output | 0.62-1.07 |
| | | Delay | 10.4-120 |

Table 9: Performance measuring states/sec.

As we can see from the Table 9 calculation of a state set after delays consumes most of the time. Statistically the mouse click system which can be in bigger state set requires more time for calculating a new state set than if it is in a fewer state set. Calculating a state set in a four button mouse the user time deviates from 0 to 120 seconds or more because an expected state set changes from 1 to 103344 or more as a result of an exponential grow of states showed in Figure 40(b). However the performance measures do not scale as we expected, in contrary the symbolic state set size cannot be used in foreseeing how much time it will take to compute a new symbolic state set. We think that this is because of uncertainty in time zones, i.e. the symbolic state set with the same location vectors but different time zones takes more computation time than the symbolic state set of the same size but with different location vectors, because of hash table lookups are based on location vectors and are useless when we lookup for symbolic states where most of them have the same location vector.

# 6 Epilogue

Our aim of one year project was to introduce and implement an on-the-fly test generation and execution module for UPPAAL tool. We needed to get familiar with basic issues related with testing subject. We got acquainted with testing theory for non-timed systems and explained how and what requires that conformance testing could be applied in testing real time systems.

We continued with testing concepts for real time systems. The notion of timed automata and its semantics were introduced as well as test specification concepts for a UPPAAL automata

network. We formulated *on-the-fly* test generation and execution algorithm and proposed ideas for designing and implementing the algorithm in theUPPAAL tool.

Algorithm implementation details were visualized and explained through class diagrams and algorithm behavior through message sequence charts. We made several sessions of experiments with different implementations. Experiment models were used for testing the algorithm and for gathering the algorithm performance data. We measured performance of the program using the profiling tool *gprof* and explained the results.

We conclude this project accomplishments in Section 6.1. During the project work we found a number of new ideas which were not implemented due to project time constraints and they are summarized in Section 6.2.

## 6.1 Conclusions

The testing theory based on timed automata is not developed yet. The correctness criteria for testing may vary widely depending on what is the actual aim of the testing. The phrase "to ensure that the implementation behaves according the specification" can be interpreted in many ways and we chose just one of them: the implementation is forbidden to do the unspecified actions and everything that is allowed must be allowed by the specification.

Further investigations revealed the possibilities for detailed testing based on the specific environment for the implementation which intrigues even more to investigate the possibilities of testing based on timed automata theory. The application of timed automata theory in testing is not expected to help in thorough testing with 100% error free guarantee, but rather automate the on-the-fly testing based on the documents prepared in product design phases, which releases many human hours, saves the testing memory and paper space and still smart and faithful in revealing the implementation errors.

Implementation of the test algorithm required a thorough analysis of the existing UPPAAL source code. We tested T-UPPAAL with three sample implementations under test: 1) single button for basic functionality, 2) multi-button mouse for many states and 3) train-gate for composite networks with complex environment. The testing and performance measuring of T-UPPAAL revealed the following issues:

**Time synchronization** between the IUT and the environment. The clocks are strictly internal parts of the IUT and the environment, which makes it difficult to ensure that their values match. This problem is common in distributed systems and in our case the solution depends on how state estimation algorithms are quick: the state estimation is quick enough if it computes symbolic states in half of the precision time, which assures that the test algorithm is capable of reacting as soon as possible without the implementation noticing a delay. As we see further, the performance depends on some system model properties.

**Environment constraints** are not properly handled in the test algorithm. The problem was noticed very late in the project, therefore we could not implement a fix, but the solution is described as computing the minimum and the maximum delay.

**Exponential growth** of symbolic states in non deterministic system models. The problem has a combinatorial nature and is common in systems making the non deterministic choices, but in addition we experienced the uncertainty in clock values which is successfully handled in model verification. The problem is that our test algorithm is interested in concrete clock values at concrete point in time and the model verification can handle many clock values by putting them into abstract time zone whose interpretation is not clear for a concrete

point in time. The later means that not entire potential of symbolic state estimation is used in our algorithm and some radical changes may be needed, such as changing the semantics of symbolic state set.

**Performance** of the test algorithm is as good as the UPPAAL algorithms perform. When the symbolic state set is small (up to the order of ten) the hash table operations become the main expense of computation. In contrary when the symbolic state set is large, the main expense is the symbolic state comparison computation in passed and waiting lists. The symbolic state estimation algorithms themselves are not that expensive as the temporary storage. A simple solution could be to provide different versions of T-UPPAAL: one optimized for small symbolic state sets and another for large and growing symbolic state sets.

From the project steering point of view we learned that it is difficult to plan project activities ahead when unexpected issues pop up during testing, which suggests that the field of the project is still uncertain and under research.

The overall look of the T-UPPAAL is not fixed yet too, still many optional features must be reviewed, evaluated and prioritized since there are still many directions to go further. In the next section we outline the next steps for the future work.

## 6.2 Future Work

We have implemented the first test generation and execution algorithm prototype. Therefore the natural next steps would be:

1. Fix the environment constraint issues found.
2. Optimize the passed and waiting list usage.
3. Solve the symbolic state explosion problem or prove that it cannot be solved in a framework of testing.
4. Minimize the computation delays which are not included into symbolic state estimation. This might require to change test generation and execution algorithm strategy, e.g. to generate one delay and one input action ahead of execution.
5. Provide an automatic way of obtaining the test purpose of test run in a form of timed automata. This feature may be useful for T-UPPAAL test run repetition when assisting in a system debugging.
6. Provide an automatic calculation of an executed test coverage which shows how good executed tests are.
7. Develop a graphical user interface to T-UPPAAL.

Further branches of the project may procede with the development of universal interface to implementation under test involving shared library loading, operating system sockets etc., development of the test event visualization followed by the list of desired features mentioned in Section 4.1.

There is also a room for proving the correctness of the test generation and execution algorithm. So far we described three similar versions of this algorithm (Section 1.7, Section 3.1 and Section 3.2) which increase the confidence and might provide a way of proving the correctness.

The success of this project would open a great oportunity to work on concrete test strategies involving the test primitive selection and optimization of test generation coverage of the test specification. Yet another use of T-UPPAAL would be easy modelling of the real-time tests (especially non-deterministic ones) while just specifying different models of the environment.

## References

[1] Brian Nielsen. Specification and Test of Real-Time Systems. Ph.D. Thesis defended $10^{th}$ November 2000. ISSN 1399-8145.

[2] Jan Tretmans. Testing Techniques. Lecture notes. Formal methods & Tools Group, Faculty of Computer science, University of Twente, The Netherlands, 2001.

[3] Joost-Pieter Katoen. Concepts, algorithms and tools for Model Checking. Lecture notes of the course “Mechanised validation of paralel systems”. Course number 10359 1998/1999.

[4] Martha Gray, Alan Goldfine, Lynne Rosenthal, Lisa Carnahan. Conformance Testing. National Institute of Standards and Technology. Information Technology Laboratory. USA http://www.oasis-open.org/cover/conform20000112.html

[5] Jan Tretmans. Specification Based Testing with Formal Methods: From Theory via Tools to Applications. Formal Methods and Tools group. University of Twente Enschede. The Netherlands. http://www.cs.auc.dk/˜bnielsen/tomas/material/tretmans250401.pdf

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

[7] Franck Cassez et al. Modeling and verification of parallel processes: 4th summer school, MOVEP 2000 Nantes, France, June 19 - 23, 2000: revised tutorial lectures. LNCS 2067. Berlin : Springer, 2001. UPPAAL website: http://www.docs.uu.se/docs/rtmv/uppaal/

[8] Paul Pettersson. Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice. A dissertation in Computer Systems for the degree of Doctor of Philosophy. Publicly examined in room X, Uppsala University, 19 February 1999. Technical Report DoCS 99/101. ISSN 0283-0574.

[9] David L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In International Workshop on Auto-matic Verification Methods for Finite State Systems, Grenoble, France, June 1989. LNCS 407.

[10] Stavros Tripakis. Fault Diagnosis for Timed Automata. In FTRTFT, 2002

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

[12] H.Brinksma, K.G. Larsen, G.J.Tretmans, M.J.Plastmeijer, B.Nielsen. Systematic Testing of Realtime Embedded Software Systems - STRESS. Proposal for the research project 2001-2005.

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

[14] Anders Hessel, Kim G.Larsen, Brian Nielsen, Paul Pettersson, Arne Skou. Time-optimal Real-Time Test Case Generation using UPPAAL. Submitted to International Conference on Software Engineering and Formal Methods, Australia, 2003.

# A Source Code

[Confidential].