Uppsala Master's Thesis in Computing Science
Examensarbete DV3 20 poäng
Datavetenskapligt program, 160 poäng
November 21, 2007

Extending a Real-Time Model-Checker to a Test-Case Generation Tool Using libCoverage

Fredrik Stenh

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Supervisor: Anders Hessel
Examiner: Paul Pettersson

Passed:

Abstract

UPPAAL is a model-checker developed by the Department of Information Technology (DoCS) at Uppsala University in Sweden together with Aalborg University in Denmark. UPPAAL can be used to model, simulate, and verify timed automata. It has been used in many case studies since the first release in 1995.

libCoverage is a library developed at Uppsala University which can be used to generate test-cases for real-time systems described as networks of timed automata. The test-cases are generated based upon a given coverage criteria, where the coverage criteria is specified by a parameterized observer automaton. The library is designed to extend model-checking tools, such as UPPAAL or SPIN.

The aim of this thesis is to extend UPPAAL 4.0 with libCoverage. For this purpose the grammar of the property file was extended to support libCoverage specific queries, and a modified reachability algorithm was presented which supports coverage exploration. We also extended each state with information about its current coverage, and implemented a wrapper which makes it possible for libCoverage to fetch information from UPPAAL about the system of timed automata.

In conclusion, we show that UPPAAL 4.0 can be extended with libCoverage to support test-case generation.

Acknowledgments

This is a Master's Thesis in Computing Science written at the Department of Information Technology at Uppsala University.

I would especially like to thank my supervisor, Anders Hessel, and my examiner, Paul Pettersson, for their help to accomplish this thesis. Thanks also to my family for support and encourage during this work. Special thanks to my sister for her help with proof reading and all pleasant lunches we had. Thanks to all other people that have helped me throughout my work.

I have for a long time looked forward to this moment when I at last finish my work with this thesis by writing this page. This page also finishes my studies in Computer Science at Uppsala University, thanks to all my friends that made these five years to a great time.

Uppsala, November 2007

Fredrik Stenh

Contents

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Goals and Objectives . . . 2
1.3 Restrictions . . . 2
1.3.1 Implementation . . . 2
1.3.2 Investigate and Document . . . 3
1.4 Structure of the Thesis . . . 3

2 Background **5**
2.1 Timed Automata . . . 5
2.2 Reachability Analysis . . . 7
2.3 Observers . . . 9
2.4 Uppaal . . . 11
2.4.1 Modeling Language . . . 13
2.4.2 Reachability Analysis . . . 13
2.4.3 Pipeline . . . 14
2.5 libCoverage . . . 15

3 Extending Uppaal With libCoverage **17**
3.1 libUTAP . . . 17
3.2 Extended State . . . 18
3.3 Coverage Exploration Algorithm . . . 18
3.4 Pipeline . . . 19
3.5 Wrapper . . . 20

4 Evaluation **23**
4.1 Observers . . . 23
4.1.1 Location Coverage Observer . . . 23
4.1.2 Edge Coverage Observer . . . 23
4.1.3 Definition Use Coverage Observer . . . 24
4.2 Train gate . . . 24
4.2.1 The Train . . . 25
4.2.2 The Gate Controller . . . 26
4.2.3 Results . . . 26
4.3 Audio-Control Protocol . . . 27
4.3.1 Results . . . 28
4.4 Wireless Application Protocol . . . 28
4.4.1 Results . . . 31

5 Conclusion and Discussion **33**

A LibCoverage API **35**

A.1 Wrapper 35

A.1.1 const std::set< std::vector<int> > &**getDef()** . 35

A.1.2 const std::set< std::vector<int> > &**getUse()** . 36

A.1.3 const std::set< std::vector<int> > &**getLoc()** . 36

A.1.4 const std::set< std::vector<int> > &**getEdge()** 36

A.1.5 const std::set< std::vector<int> > &**getSync()** 36

A.1.6 int **evalVar**(int varID) 37

A.1.7 void **activeProcs**(std::set<int> &activeProcs) . 37

A.1.8 void **clearCache()** 37

A.1.9 int **edgeFromParam**(std::pair<int,int> edge) . 37

A.1.10 int **locFromParam**(std::pair<int,int> loc) . . . 37

A.1.11 void **setProcs**(const std::set<int> &procs) . . . 38

A.2 AllCoverage 39

A.2.1 static AllCoverage* Instance() 39

A.2.2 void setWrapper(Wrapper *wrp) 39

A.2.3 void initialCoverage(Coverage &cov) 39

A.2.4 int evalAcceptSize(Coverage &cov) 39

A.2.5 int makeEntry(OBSERVER::obs_signature_t &os, std::set<int> procs) 39

A.2.6 bool newCoverage(Coverage &cov) 40

A.3 Coverage 41

A.3.1 int relation(const Coverage &cov) const 41

A.4 Observer 42

A.4.1 OBSERVER::obs_signature_t 42

A.4.2 OBSERVER::obs_param_t 42

A.4.3 OBSERVER::ParamType 42

1 Introduction

Who has not used a computer without notice any strange behavior sometime? It may be a faulty painting program, a word-processor with a bug, or an operating system that suddenly crashes. In these cases no personal or material damage occurs, except from the possibility of data loss. But what if a breaking system of a car does not react on the drivers command due to some software error of the control unit, or if the computer system controlling a nuclear power plant fails. This may lead to serious personal and material damage, or even a disaster if a core meltdown occurs.

Error-prone software is also costly for the manufacturer who is responsible for the product. To decrease the number of errors, testing is the most common method used to verify if the behavior of a system complies to its specification. Code walk-throughs, code inspections, and code reviews are all examples of other methods that can be used to improve the quality of a system/software.

There are two categorizes of testing, *wite box* testing and *black box* testing. In white box testing the internal structure of the software is known, and this knowledge is used to design test cases. One drawback with the way test cases are designed in white box testing is that if the implementation changes, the test cases probably have to be changed too. This is not the case in black box testing where there are no knowledge about implementation details used. Test cases are instead derived from the specification of the system.

Model-based testing is a black box testing method where test cases are derived from a model of the system. A tool which examines the model and then automatically generates test-cases according to a given *test purpose* is often used for model-based testing. A *test purpose* is a specification of what to be tested. An example of a test purpose can be "test a state change from state A to state B" in a model. If this test purpose is given to the test case generation tool, it will try to generate a test case that changes the state of a model from state A to state B. *Coverage criterion* is a type of test purpose that specifies a number of items, so-called *coverage items*, that must be covered or visited by a test case. For example, if a "full edge coverage" criterion is specified for a given automaton, a test case must take all edges in that automaton.

The CO✓ER tool, which is described later in this thesis, is an example of a test-case generation tool. It takes a model of timed automata and a test purpose or a coverage criteria (described as an observer), and generates test-cases.

1.1 Problem Statement

At Uppsala University a library named libCoverage has been developed which can be used to extend model-checking tools, such as UPPAAL or SPIN, with functionality to automatically generate test-cases for systems described as networks of automata. The library has been applied to extend UPPAAL [?] 3.3. The extended version of UPPAAL is named CO✓ER [?].

After almost three years of development UPPAAL has now reached version 4.0. It is thus desirable to apply libCoverage to the latest version to benefit from the improved performance. In this thesis, it will be shown how UPPAAL 4.0 can be modified to be used to generate abstract test-cases for real-time systems.

1.2 Goals and Objectives

The goals of this master thesis project are to

- Port the existing implementation of libCoverage from UPPAAL 3.3 to the current version, UPPAAL 4.0. Almost three years of development have been spent between versions 3.3 to 4.0.
- To investigate and document how the libCoverage library can be applied in other state-space exploration tools, e.g., other model-checking tools such as SPIN [?], or online testing tools such as TRON [?], or software testing tools such as gcov [?].
- Evaluate the performance of UPPAAL 4.0 extended with libCoverage on existing models to compare with the current implementation on UPPAAL 3.3.

1.3 Restrictions

We had to make some changes to the plan of goals due to some parts took more time than expected. Here we presents the restrictions we had to make.

1.3.1 Implementation

The first plan was to extend UPPAAL 4.0 with all functionality that the old CO✓ER had implemented (the extended version of UPPAAL 3.3). But very soon we realized that the task was overwhelming so we decided to develop a basic implementation instead. This version should support:

- local search algorithm (how the model is searched for coverage) [?],

- coverage queries in the query-file,
- parameters to the observer which references to model identifiers of the following types:
  - local variables of processes, where the process must not be member of a set, e.g. `proc(i)` or `foo(index)`,
  - global variables, and
  - processes.
- write the resulting test-case trace to standard out (`stdout`).

1.3.2 Investigate and Document

The intension was to investigate and document how libCoverage could be applied to other state-space exploration tools. But we had to restrict the task to only include Uppaal due to the lack of time.

1.4 Structure of the Thesis

The rest of this thesis is organized as follows: in the next section, we describe timed automata and reachability analysis. We also describe the tool Uppaal, observer automata, and the library libCoverage. In section 3 we present the changes done to extend Uppaal 4.0 with libCoverage. Section 4 presents the evaluation of the extended version of Uppaal 4.0. Section 5 concludes the thesis. Finally, as an appendix, we describe the API of libCoverage.

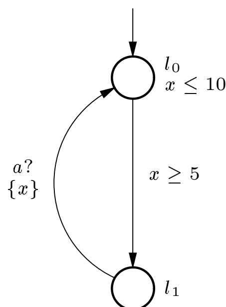


Figure 1: Example of a simple timed automaton.

2 Background

2.1 Timed Automata

It has been very common in the literature to describe real-time systems with timed automata since they were first presented by Alur and Dill in [**?**]. A timed automaton is a finite state machine extended with non-negative real-valued clocks. A clock measures the time passed since the last reset of the clock. When the system starts, all clocks are initialized to zero, and then propagated with the same rate.

Apart from clocks the automaton also consists of actions, locations, and edges, where actions are used by the automaton to interact with its environment. The action can be suffixed with *?* (an input action), or suffixed with *!* (an output action). Locations may have timing constraints called *invariants*. An automaton may stay in a location as long as the invariant is satisfied by the clocks values. Edges may be labeled with timing constraints called *guards*, actions, and a set of clocks to reset. An edge can be executed only when the clocks values satisfy the guard and the environment can interact with the action.

The timed automaton presented in Figure **??** have two locations, l_0 and l_1, where l_0 is the initial location with an invariant, $x \leq 10$, forcing the automaton to leave l_0 within ten time units. But the guard, $x \geq 5$, of the edge from l_0 to l_1 force it to wait at least five time units before the edge may be taken. So the automaton may only move to location l_1 in the interval between five to ten time units. Once in the location l_1 the automaton may stay for arbitrary long time and the edge back to l_0 can only be executed when the input $a?$ is available from the environment. The clock, x, is also being reset when the edge is taken.

We will now give a more formal definition of timed automata. Let A be

a set of actions, and let C be a set of clocks. Further let $G(C)$ be a set of guards of the form:

$$g ::= \quad x \sim n \quad | \quad x - y \sim n \quad | \quad g_1 \land g_2$$

where x, $y \in C$, $n \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. A timed automaton over (A, C) is then a quadruple $\langle L, l_0, I, E\rangle$ where

- L is a set of locations,
- $l_0 \in L$ is the initial location,
- $I : L \to G(C)$ assigns invariants to locations, and
- E is a set of edges

An edge between two locations is a quintuple $\langle l, g, a, r, l'\rangle \in E$ where

- $l \in L$ is the source location,
- $g \in G(C)$ is a guard,
- $a \in A$ is an action,
- $r \subseteq C$ is a set of clocks to reset, and
- $l' \in L$ is the target location

A state of a timed automaton is a tuple $\langle l, \sigma\rangle$ where $l \in L$ is a location and σ is a mapping where each clock $c \in C$ is assigned a non-negative real-value, $\mathbb{R}_+$. The initial state $\langle l_0, \sigma_0\rangle$ is a state where l_0 is the initial location of the automaton and σ_0 is the initial mapping where all clocks $c \in C$ are assigned zero.

An automaton can stay in a location as long as the invariant of the location is satisfied by doing a *delay* transition, $\langle l, \sigma\rangle \xrightarrow{d} \langle l, \sigma' + d\rangle$. In a delay transition all clocks are incremented with the amount of time units of the delay d, denoted $\sigma' + d$.

A *discrete* transition, $\langle l, \sigma\rangle \xrightarrow{a} \langle l', \sigma'\rangle$, is done when an edge is executed. To be able to do a discrete transition both the guard g of the edge and the invariant of the target location $I_{l'}$ must be satisfied by the new clock assignment σ'. Further, the automaton must also be able to perform the action a associated with the edge.

```
W = {⟨l_0, σ_0⟩}
P = ∅
while W ≠ ∅ do
    ⟨l, σ⟩ = W.selectState()
    W.removeState(⟨l, σ⟩)
    if testProperty(⟨l, σ⟩) then return true
    if σ ⊈ σ' for all ⟨l, σ'⟩ ∈ P then
        P.insertState(⟨l, σ⟩)
        for all ⟨l', σ'⟩ such that ⟨l, σ⟩ → ⟨l', σ'⟩ do
            if σ' ⊈ σ'' for all ⟨l', σ''⟩ ∈ W then
                W.insertState(⟨l', σ'⟩)
            endif
        done
    endif
done
return false
```

Figure 2: A standard reachability algorithm.

2.2 Reachability Analysis

Reachability analysis can be used to check properties of states of timed automata by traversing the state-space. It consists of two main tasks, computing the state-space of the automaton under consideration, and to check if a given property is satisfied by any states in the computed state-space. The computation of the state-space can be done before the search starts or done on-the-fly during the search. The latter have the benefit that no overhead is done, only the part of the state-space needed to prove the property will be generated.

Figure **??** presents a standard reachability algorithm doing on-the-fly computation of the state-space. It has two lists, a passed list (`P`) and a waiting list (`W`). The passed list holds already explored states while the waiting list contains states found to be reachable but not yet explored.

Initially the waiting list is populated with the initial state $\langle l_0, \sigma_0 \rangle$, and the passed list is empty. While the waiting list is not empty a state $\langle l, \sigma \rangle$ is selected from the waiting list. If the property is satisfied of the state $\langle l, \sigma \rangle$ no further search need to be done and the algorithm returns *true*.

The algorithm looks in the passed list for the state $\langle l, \sigma \rangle$. If the list contains the state it can be ignored because it has already been explored, otherwise the state is added to the passed list. Thereafter all successors to the state are found and added to the waiting list if they are not already in the list. Then the next iteration of the `while`-loop starts.

Figure **??** shows an example of a possible run of the reachability algorithm

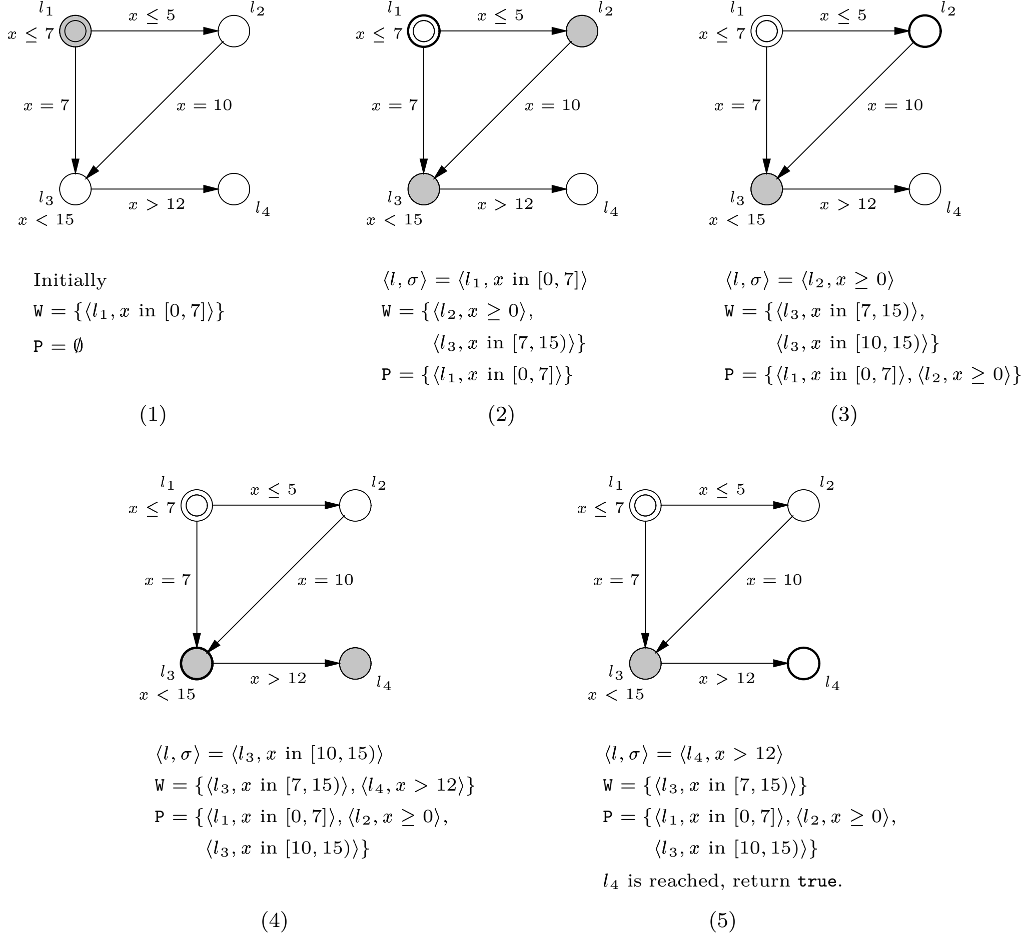


Figure 3: A possible run of the reachability algorithm when verifying the property "is it possible to reach location l_4". The initial location is marked with a double circle, the current location with a bold circle, and locations in the waiting list `W` are gray.

on a timed automaton. The automaton used in this example has one clock, x, and the property to verify is, "is it possible to reach location l_4":

(1) Initially the waiting list is populated with the initial location $\langle l_1, x$ in $[0,7]\rangle$ and the passed list is empty.

(2) The only state in the waiting list, the initial state, is selected as the current state. The property is tested on this state, but it does not hold since it is not location l_4, and the state is inserted into the passed list. Then all succeeding states, $\langle l_2, x \geq 0\rangle$ and $\langle l_3, x$ in $[7,15)\rangle$, are found and put in the waiting list.

(3) The state $\langle l_2, x \geq 0 \rangle$ is selected from the waiting list and is also put in the passed list. The succeeding state, $\langle l_3, x \text{ in } [10, 15) \rangle$, is found and put in the waiting list.

(4) $\langle l_3, x \text{ in } [10, 15) \rangle$ is selected from the waiting list. The property does not hold for this state either, and it is inserted into the passed list. The succeeding state, $\langle l_4, x > 12 \rangle$, is found and put in the waiting list.

(5) Finally, the state $\langle l_4, x > 12 \rangle$ is selected as the current state from the waiting list. Now the property holds, location l_4 has been reached and the algorithm returns `true`.

Observe that the waiting list and passed list in the last run (5) is not affected when the property holds and the algorithm returns.

2.3 Observers

A *coverage criterion* is a specification of items, e.g. locations, edges, to be *covered* or *visited* by the timed automaton. An example of a coverage criterion is "full location coverage" which means that a trace (test case) should visit all location of a given timed automaton. An item to be covered or visited is called a *coverage item*.

A coverage item can be represented by an observer, which observes the execution of an timed automaton and "accepts" when the coverage item is covered by the trace.

An observer automaton consists of locations and edges where locations are labeled with a name and optional variables, and edges are labeled with predicates. There exists two special type of locations, the *initial location* marked with a black filled circle, and the *accepting location* marked with a double circle. An observer can only have one initial location but may have several accepting locations. The observer will only reach an accepting location when a coverage item is visited by the timed automaton. The observer location must take all possible edges each time the timed automaton makes a transition. If there is no possible edge for the location to take, it is removed from the execution of the observer. But the observer has always the possibility to stay in the initial location or in an accepting location.

The observer in Figure **??**(i) has two accepting locations, `acc(start)` and `acc(stop)`. Thus, the observer represent two coverage items, namely that location "start" and location "stop" should be covered. If the target location (abbreviated as *tl* in the figure) of the transition done by the timed automaton is the location named `start`, then the predicate `tl = start` will become true and the observer will reach the accepting location `acc(start)`. Later

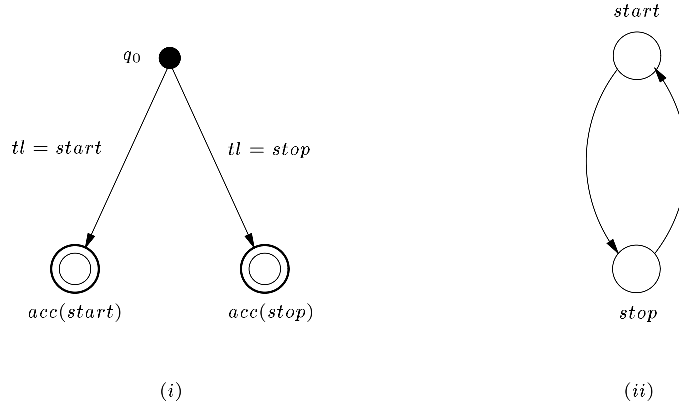


Figure 4: (i) Example of an observer with two accepting locations. (ii) A simple automaton with two locations.

in the execution of the timed automaton the target location of a transition may be `stop`, and then the predicate `tl = stop` of the other observer edge will become true. The observer will now also reach the accepting location `acc(stop)`.

Figure **??**(ii) shows a simple automaton with only two locations, `start` and `stop`. If the observer from Figure **??**(i) is superposed on this automaton, the observer will represent the coverage criterion "full location coverage" of the automaton. The "full location coverage" criterion implies that all locations of the automaton should be visited.

To give a formal definition, an observer automaton is a quadruple $\langle Q, q_0, Q_f, B \rangle$ where

- Q is a set of locations,
- $q_0 \in Q$ is the initial location,
- $Q_f \subset Q$ is a set of accepting locations, and
- B is a set of edges

An edge between two locations is a triple $\langle q, p, q' \rangle \in B$ where

- $q \in Q$ is the source location,
- p is a predicate over attributes of a timed automaton, e.g. edges, locations, variables, etc., and
- $q' \in Q$ is the target location

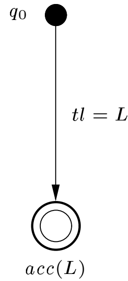


Figure 5: Example of a parameterized observer, representing the coverage criterion "full location coverage".

It is common that a coverage criterion not only consists of a single coverage item, but a large set of coverage items. For instance the coverage criterion "full location coverage" will have a coverage item for each location to be visited, and an observer for this criterion will have as many accepting locations as there are locations in the timed automaton. Instead of using an ordinary observer as the one showed in Figure **??**(i), we can use a parameterized observer. This is an observer where parameters can be used on locations and edges, and has the advantage that the value domains of the parameters do not have to be specified at the same time as the observer is designed. This makes the observer more general because the same observer can be used on more than one timed automaton without making any changes to the observer or to the timed automaton. The parameters can refer to edges, locations, variables, etc. in the timed automaton.

Figure **??** shows a parameterized observer for the coverage criterion "all location coverage". The variable `L` represents the target location of a transition, and thus will the predicate `tl = L` be evaluated to *true* for each transition.

If the parameterized observer in Figure **??** is superposed on the automaton in Figure **??**(ii), it will achieve the same result as the ordinary observer in Figure **??**(i) when it was superposed on the automaton.

2.4 UPPAAL

UPPAAL is a tool for modeling, simulation, and verification of timed automata. It is developed by Uppsala University in Sweden and Aalborg University in Denmark. It has been used in many case studies since the first release in 1995. UPPAAL has a graphical user interface which makes it easy for the user to create and design a system of timed automata and then switch to the simulator (see Figure **??**) to run the system interactively, or use the

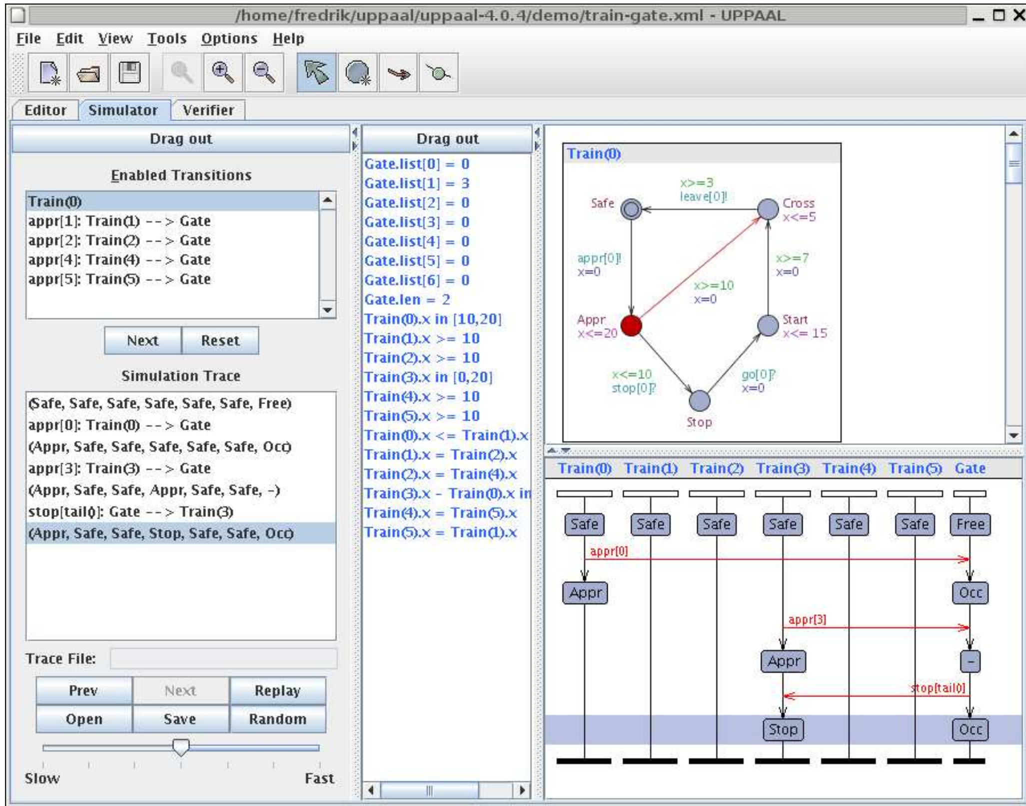


Figure 6: The simulator screen of the UPPAAL tool.

verifier to verify properties of the system. When verifying a property, UPPAAL is able to generate a diagnostic trace showing how to stimulate the system to reach a state that satisfy or contradict the property. It is then possible to load a generated trace into the simulator where it can be examined. UPPAAL can generate three different types of traces:

some trace - first path to a state will be returned.

shortest trace - returns the trace with least number of transitions to reach the state.

fastest trace - is the trace which let least time pass to reach the state.

UPPAAL consists of two applications, the GUI implemented in Java, and the verification engine written in C++. The GUI communicates with the verification engine using a socket which makes it possible to do the verification either local on the same computer as the GUI is running on, or on a powerful server in a network.

```
Queue = PWList = {⟨l_0, σ_0⟩}
while Queue ≠ ∅ do
    ⟨l, σ⟩ = Queue.selectState()
    Queue.removeState(⟨l, σ⟩)
    if testProperty(⟨l, σ⟩) then return true
    for all ⟨l', σ'⟩ such that ⟨l, σ⟩ → ⟨l', σ'⟩ do
        if σ' ⊈ σ'' for all ⟨l', σ''⟩ ∈ PWList then
            PWList.insertState(⟨l', σ'⟩)
            Queue.insertState(⟨l', σ'⟩)
        endif
    done
done
return false
```

Figure 7: The improved reachability algorithm used by UPPAAL.

2.4.1 Modeling Language

The UPPAAL modeling language extends timed automata with additional features. Some of these, which are used in the models used in the evaluation section, are integer variables, urgent locations, and committed locations. Integer variables and arrays of integers may be updated on edges, and predicates over integer variables can be used as guards on the edges of an automaton. An urgent location, marked with an `U` in the automaton, is a location that must be left without letting any time pass. A committed location must, just like an urgent location, also be left without letting any time pass, but the location must be left immediately as the next transition done by the system. A committed location is marked in the automaton with a `C`. A initial location of an automaton is marked with a double circle. We will use the same notation as UPPAAL in this thesis to represent timed automata.

2.4.2 Reachability Analysis

UPPAAL uses an improved version of the reachability algorithm where the two lists, *passed* and *waiting*, have been unified into a single list and a trivial queue structure only containing references to states in the list. By unifying the two lists one inclusion checking is eliminated which improves the performance.

The improved algorithm is presented in Figure **??** where the list and the queue are called `PWList` and `Queue` respectively. The algorithm works as follows: initially both the queue and the list are populated with the initial state $\langle l_0, \sigma_0 \rangle$. While the queue is not empty, a state $\langle l, \sigma \rangle$ is removed from the queue. If the property is satisfied by the state $\langle l, \sigma \rangle$ no further search need to be done and the algorithm returns *true*. Otherwise all successors to the

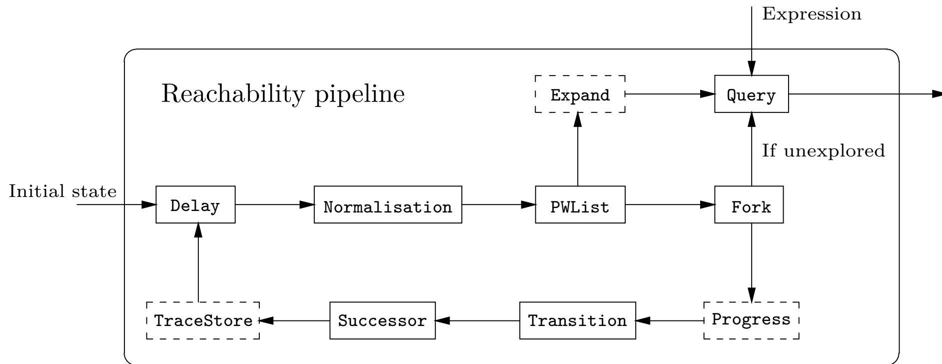


Figure 8: The reachability pipeline. Optional elements are dotted.

state $\langle l, \sigma \rangle$ is found, and each successor that is not already in the list is added to both the list and the queue. Then the next iteration of the `while`-loop is reentered.

2.4.3 Pipeline

The verification engine is designed like a pipeline inspired from computer graphics. The pipeline consists of *filters* and *buffers* connected together. A filter is a component that performs some computation of data received on its *put*-method. The result is then forwarded to the next component. A buffer on the other hand does not do any work on data, i.e. it is a purely passive component. It only stores data received with a *put*-method, and offers data with a *get*-method.

The pipeline in Figure **??** implements the reachability algorithm in Figure **??**. It consists of the following components:

- `Transition` and `Successor` - filters computing the edge successors.
- `Delay` and `Normalisation` - filters computing the delay successors.
- `PWList` - a buffer for the unified passed and waiting list.
- `Progress` (optional) - a filter for generating progress information, e.g. number of explored states and throughput.
- `TraceStore` (optional) - a filter for storing information needed when generating diagnostic traces.

- `Expand` (optional) - it may be necessary to explore some states using the `Expand` filter when multiple properties are verified and the previously generated reachable state space is reused.

- `Query` - a filter for evaluating the state property.

By adding or removing components it is easy to change the behavior of the verification engine. This can be done at runtime and removed components do not generate any overhead.

2.5 libCoverage

libCoverage is an API for coverage computation of timed automata with respect to a given coverage criteria. It was developed by Anders Hessel at the Department of Information Technology (DoCS) at Uppsala University in Sweden and is intended to be used to extend a model checking tool, such as SPIN or UPPAAL, with functionality for coverage exploration.

Instead of a predefined set of coverage criteria specified in the library, libCoverage uses observer automata to define coverage criteria. The observer is specified in a separate file, using an expressive observer language, which is then read by libCoverage. Figure **??** shows an observer which corresponds to the definition use coverage criteria. The observer is presented both in the observer language used by libCoverage, Figure **??**(i), and graphically, Figure **??**(ii).

The observer takes an variable from the automaton as an argument, which will be represented by `X`. Further there are three predefined macros used as guards on the edges, `def(X,E1)`, `def(X)` and `use(X,E2)`.

- `def(X,E1)` will evaluate to true if the variable `X` is defined in the automaton on an edge. The edge will be bound to the variable `E1`.

- `def(X)` will evaluate to true if the variable `X` is defined in the automaton. Thus will the negation, ¬`def(X)`, be true if `X` is not defined.

- `use(X,E2)` will evaluate to true if the variable `X` is used in the automaton on an edge. The edge will be bound to the variable `E2`.

At the location `defined(X,E1)` `X` is bound to the variable given as argument and `E1` is bound to the edge where `X` was defined. Further, at the location `du(E1,E2)` `E1` is bound to the edge where `X` was defined, and `E2` is bound to the edge where `X` was used.

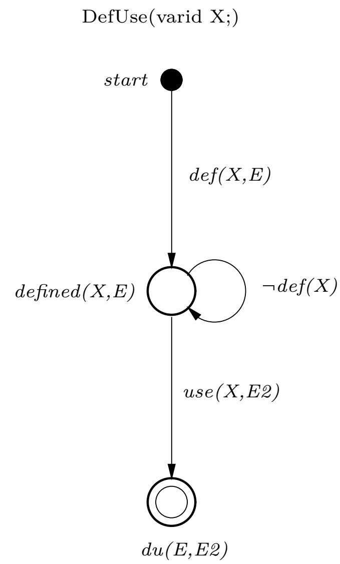
If we look at the code that is defining the observer, Figure **??**(ii), the first row specifies the name of the observer, `du`, and that the argument is

```
observer du (varid X;) {
    node defined (varid, edgeid);
    node du (edgeid, edgeid);

    rule start         to defined(X,E) with def(X,E);
    rule defined(X,E)  to defined(X,E) with no def(X);
    rule defined(X,E)  to du(E,E2)      with use(X,E2);

    accepting du;
}
```

(i)



(ii)

Figure 9: Example of a def-use observer presented both textual (i) and graphically (ii).

of the type `varid` which is a variable from the automaton. The second and third rows defines the two locations, `defined` and `du`, with the types of their variables. Then the following three rows defines the edges and their guards (specified after the `with`-keyword). Finally, the last row tells that the location `du` is an accepting location. For a complete description of the observer language used by libCoverage we refer the reader to [?].

3 Extending Uppaal With libCoverage

In this section we describe the changes we had to make to Uppaal to integrate libCoverage with it. We will use the name co✓er to denote Uppaal 4.0 extended with libCoverage if nothing else is mentioned.

3.1 libUTAP

To do a reachability analysis using Uppaal, we need to supply it with a model file and a query file. The model file contains the description of the system of timed automata, and the query file is where properties to be verified of the system are placed. To parse these files Uppaal uses the Uppaal Timed Automata Parser Library (libUTAP) [?].

To let co✓er know that we want to do a coverage exploration and which observer to use, we have to be able to express that in the property file. We must be able so specify

- that we want to do a coverage exploration,
- which observer to be used, i.e. the coverage criteria,
- parameters to the observer, and
- processes to restrict the observer to, i.e. the observer will only react on the restricted processes and ignore all other.

We can not use the standard notation used in the property file to express this type of queries. To change the notation we have to change the grammar used by libUTAP to parse the property file. We extend this grammar with the grammar presented in Figure **??**, where the model identifiers given as parameters to the observer may be of the following type:

`COVPROP`	::=	’`cover`’ `OBSERVER` [’`(`’ `OBSPARAMS` ’`)`’] [’`restrict`’ `COVPARAMS`]
`OBSERVER`	::=	the name of the observer-file excluding the .obs extension, e.g. if the observer file is named foo.obs we only write foo.
`OBSPARAMS`	::=	`ARGSET` | `ARGSET` ’,’ `OBSPARAMS`
`ARGSET`	::=	’{’ comma separated list of model identifiers ’}’
`COVPARAMS`	::=	’`(`’ comma separated list of processes ’`)`’

Figure 10: The grammar libUTAP was extended with in order to read co✓er queries.

```
Queue = PWList = {⟨(l_0, σ_0) ‖ Q⟩}
maxCoverage = 0
maxState = null
while Queue ≠ ∅ do
    ⟨(l, σ) ‖ Q⟩ = Queue.selectState()
    Queue.removeState(⟨(l, σ) ‖ Q⟩)
    if |Q ∩ Q_f| > maxCoverage then
        maxCoverage = |Q ∩ Q_f|
        maxState = ⟨(l, σ) ‖ Q⟩
    endif
    for all ⟨(l′, σ′) ‖ Q⟩ such that ⟨(l, σ) ‖ Q⟩ → ⟨(l′, σ′) ‖ Q′⟩ do
        if σ′ ⊈ σ″ or Q′ ⊈ Q″ for all ⟨(l′, σ″) ‖ Q″⟩ ∈ PWList then
            PWList.insertState(⟨(l′, σ′) ‖ Q′⟩)
            Queue.insertState(⟨(l′, σ′) ‖ Q′⟩)
        endif
    done
done
return maxState
```

Figure 11: Modified version of the reachability algorithm to support coverage exploration.

- local variables of processes that are not in a set, e.g. `proc(i)`,
- global variables, and
- processes

3.2 Extended State

To keep track of the coverage of each state, we have to extend the representation of a state with extra information. This is done by adding a new private variable to the state object in UPPAAL. This new variable is a reference to an object of the libCoverage called `Coverage` containing the information about the coverage of the state. This information will be updated each time a new successor is found. We will write $\langle (l, \sigma) \parallel Q \rangle$ to denote the extended state where Q is the information about the coverage of the state.

3.3 Coverage Exploration Algorithm

The reachability algorithm, explained in section **??**, must be changed to support coverage exploration instead of verifying properties. Figure **??** shows the new version where the call to the `testProperty()` method, which was used to test if a given property was satisfied of a state, has been removed. Further, two new variables are introduced, `maxState` and `maxCoverage`. Variable

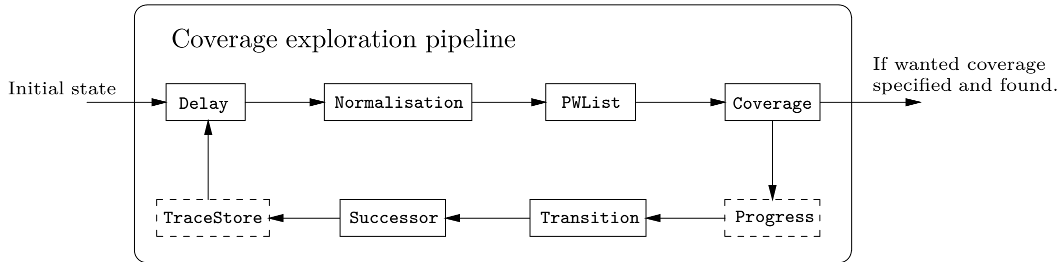


Figure 12: Pipeline for coverage exploration. Optional elements are dotted.

`maxState` is a copy of the state which has the largest coverage found so far in the system, and `maxCoverage` is the value of the coverage of the state in `maxState`.

The algorithm works as follows: first all variables are initialized, `Queue` and `PWList` are set to the initial state. Then a state $\langle (l, \sigma) \parallel Q \rangle$ is removed from the queue. The coverage of the selected state is calculated, denoted $|Q \cap Q_f|$, and compared to the coverage of the maximal state found so far. The state $\langle l, \sigma \parallel Q \rangle$ will be saved as a maximal state if it has a larger coverage than the current maximal state. Further, all succeeding states are computed and if they are not subsets of any state in the `PWList`, with respect to the symbolic part or the coverage, they are added to both the `PWList` and the `Queue` to be explored later.

The above described work flow will be repeated until the `Queue` is empty, and then the state with the maximal coverage is returned.

3.4 Pipeline

Because of the changes done to the reachability algorithm presented above we have to conform the pipeline.

This is done by replacing the `Expand` and `Query` filters with a new filter called `Coverage`, see Figure ??. This new filter keeps track of the state with the largest coverage found so far, called `maxState` in Figure ??.

The inclusion checking done by the `PWList` on each succeeding state must also be changed. In the present version a succeeding state with a new coverage may be discarded if the symbolic part of the state is equal or a subset to a state already in the `PWList`. This is simply corrected by extending the inclusion checking to also take the coverage into consideration: if the symbolic part of the state or the coverage of the state is not a subset of a state already in the `PWList` then the state is a new state and added to the `PWList`.

It is possible to do both reachability analysis and coverage exploration

in CO✓ER. Which pipeline to build is decided at run time and depends on which type of query there is in the property file. The switching between different pipelines are totally transparent for the user.

3.5 Wrapper

In order to have libCoverage working, it must be possible for libCoverage to fetch information about the system of timed automata. How to retrieve this information is specific for each program that uses libCoverage. This is solved by libCoverage by using a model checker specific wrapper. Each program must implement a wrapper which inherit from libCoverage's Wrapper class and implement the following methods:

getDef() - returns all variables defined on the transition the system currently taking.

getUse() - returns all variables used on the transition the system currently taking.

getLoc() - returns all locations of the system of the timed automata.

getEdge() - returns all active edges, i.e. edges the system currently are taking.

getSync() - returns all synchronization channels of the active edges.

evalVar(varID) - returns the value of the variable identified by `varID`.

activeProcs(activeProcs) - the variable `activeProcs` is populated with identifier for each active processes.

edgeFromParam(edge) - returns the system's identifier of the edge `edge`.

locFromParam(loc) - returns the system's identifier of the location `loc`.

setPocs(procs) - defines which processes the observer will react on, defined by the `restrict` directive in the query file.

printVar() - prints auxiliary information about a variable to a specified stream.

We have omitted the types of the methods and variables above for readability, please see Appendix **??** for complete information.

Figure **??** presents the layout of the system architecture of CO✓ER. Here it is illustrated that libCoverage uses the wrapper to fetch information from

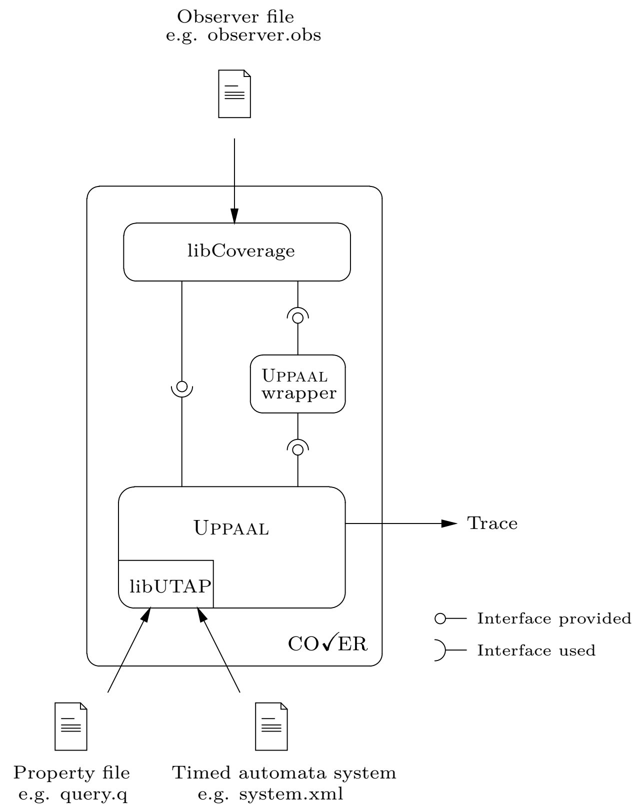


Figure 13: System architecture of co✓er.

Uppaal, while Uppaal communicates directly with libCoverage. It is also illustrated that Uppaal uses libUTAP to parse both the query and the system file, while libCoverage parses the observer file.

4 Evaluation

To evaluate the performance and verify the correctness of the program when using more complex systems, we executed a number of test runs where the results was compared with results from CO✓ER based on UPPAAL 3.3.

For the test runs we use three existing systems described in the literature. The train gate [?], models a train cross with four trains, an audio control protocol used by Philips in HIFI components [?], and a model of a WAP protocol [?].

4.1 Observers

In the test run we use three parameterized observers, a location coverage observer, an edge coverage observer, and a definition use coverage observer. Here we give a description of the observers:

4.1.1 Location Coverage Observer

Figure ??(i) shows the location coverage observer. It takes an argument `P`, which is a set of automata. The observer has two locations, an initial location `start`, and an accepting location `locN(L)`. There is also an edge from the initial location to the accepting location, with the assignment `L=loc(P)`. The operation `loc(P)` returns the location of the active automaton if the active automaton is a member of the set `P`.

The observer can only take the edge from the initial location to the accepting location if a location is assigned to the variable `L`. When the observer reaches the accepting location `locN(L)` the variable `L` is bound to a location of a automaton in the set `P`.

4.1.2 Edge Coverage Observer

The edge coverage observer is presented in Figure ??(ii). The observer is similar to the location coverage observer with a initial location, an accepting location, an edge from the initial location to the accepting location, and takes a set of automata as an argument. But instead of collecting a location it collects an edge.

The operation `edge(P)`, in the assignment on the edge from the initial location to the accepting location, returns an edge of the active automaton if the automaton is a member of the set `P`. When the observer is in the accepting location `edgeN(E)`, `E` is bound to an edge that belongs to a automata in the set `P`.

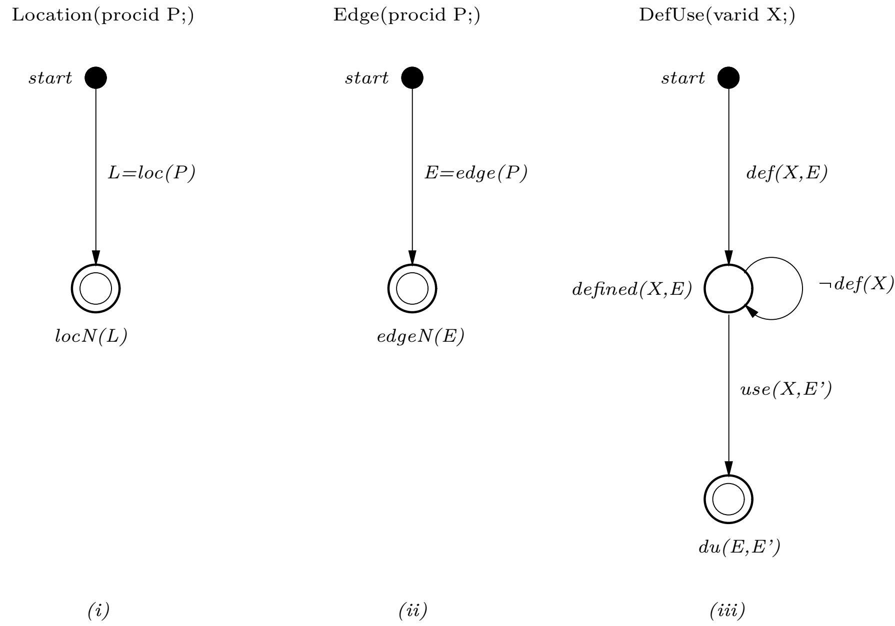


Figure 14: Observers used in evaluation

4.1.3 Definition Use Coverage Observer

The definition use coverage observer, Figure ??(iii), collects all du-pairs in the automaton it is superposed on.

The observer takes a set of variables, `X`, as an argument. When a variable of the set `X` is defined on some edge, `def(X,E)`, in the automaton that the observer is superposed on, the observer is moved from the initial location to the the location named `defined(X,E)`. Here `X` is bound to the defined variable, and `E` is bound to the edge where the variable was defined.

The observer can stay in the `defined` location as long as the variable is not redefined, `¬def(X)`. When the same variable is used on some edge, `use(X,E')`, will the observer reach the accepting location `du(E,E')` where `E'` is bound the edge where the variable was used.

4.2 Train gate

The train gate model [?] consists of a train gate controlling the access to a bridge with capacity for only one train at a time, and four trains trying to cross the bridge. The idea of the system is that the gate controller should guarantee that the number of trains crossing the bridge is always limited to one, i.e., mutually exclusive access to the bridge. Naturally, the gate

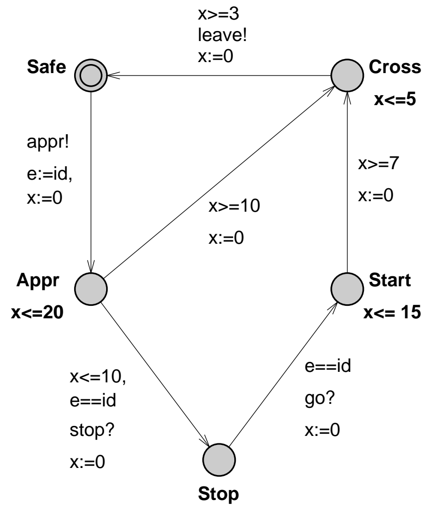


Figure 15: The train automaton.

controller should not satisfy this property in a trivial way, e.g., by blocking all access to the bridge.

The model consists of six timed automata: four trains, one gate controller, and one queue used by the gate controller. Trains arrive to the gate at random time points and in random order. Arriving train signals to the gate controller to check if the gate is free. If so, the train will cross, otherwise it will receive a stop signal and stop before the bridge. It is then put in the queue, and waits until it receives a signal from the gate controller, indicating that it is his turn to cross the bridge. The train also signals to the gate controller when it has crossed the bridge.

4.2.1 The Train

The train automaton, shown in Figure **??**, models the behavior of a train in the system. It has two output channels, `appr!` and `leave!` used to signal that it is approaching the bridge and leaving the bridge, respectively. Further, it has two input channels `stop?` and `go?`, received when the train must stop and when it can start again.

When the train is approaching the bridge it signals `appr!` to the gate controller. If the train receives a `stop!` signal within 10 time units it will stop and wait for a `go?` signal to start again and then within 7 to 15 time units start to cross the bridge. If no stop signal was received it will start crossing the gate. It may take up to 10 time units for the train to reach

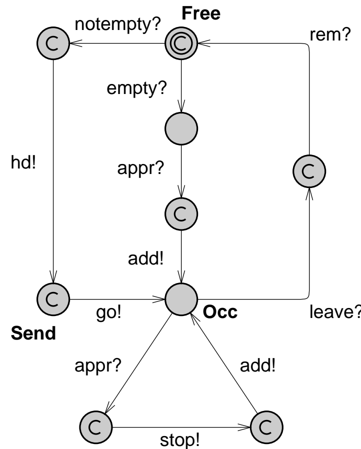


Figure 16: The gate controller automaton.

the bridge, and three to five time units to cross the bridge. When the train leaves the bridge it will send a `leave!` signal to the gate controller.

4.2.2 The Gate Controller

The gate controller, presented in Figure **??** ensures that the bridge is free before a train starts to cross. The initial state `Free` indicates that the bridge is not occupied and a train can cross. In this state it is possible for the gate controller to receive two inputs from the queue: `empty?` indicating that the queue does not contain any trains and `notempty?` if there is a train in the queue. If the queue is empty when a train is approaching it can immediately cross the bridge and the gate controller is moved to the `Occ` state representing it is occupied. If the queue was not empty it will instead pick the first train in the queue and send a `go!` signal to the corresponding train.

4.2.3 Results

We have applied the version of CO✓ER developed in this thesis to generate test cases for the train gate model described above. We have generated test cases with three different characteristics: *some trace*, the *shortest trace*, and the *fastest trace*. Two different search methods was also used: *depth first search* and *breadth first search*.

Table **??** and Table **??** presents the result when using depth first search and breadth first search respectively. In the first three lines of the tables we show the time and space needed to generate the test cases, as well as the

Example	Some trace			Shortest trace			Fastest trace		
	time	**mem**	**len**	**time**	**mem**	**len**	**time**	**mem**	**len**
Train loc	0.48	55.4	22	22.3	73.9	14	7.71	55.4	22
Train edge	0.37	55.4	23	14.0	74.1	15	4.89	55.4	23
Train du	0.57	55.4	22	25.6	75.1	14	8.86	55.4	22
Philips loc	1.16	55.6	53	38.1	109	29	43.7	122	29
Philips edge	1.98	55.6	478	665	768	76	876	889	84
Philips du	3.36	55.6	767	972	902	104	950	951	111
WAP loc	74.9	388	1448						
WAP edge	73.0	389	1449		N/A			N/A	
WAP du	263	398	1514						

Table 1: Depth first search. Time (in seconds), space (in Mb), and length (in number of transitions) performance of CO✓ER.

length of the generated trace measured in number of transitions.

A interesting observation is that when we use breadth first search all three different type of traces found the shortest trace. An inspection of the traces shows that, as expected, the edge covering test case also covers all the locations.

4.3 Audio-Control Protocol

The audio control protocol studied here is developed by Philips and is referenced already 1994 in [?]. The protocol is used to connect up to 10 audio components like cd-players, amplifier etc. Connected devices are able to share control-information, e.g. a receiver can inform connected devices about which button was pressed. The information is sent by using the Manchester encoding over a single wire.

The protocol supports collision detection, e.g. if more than one device tries to send a message at any time unit a collision will occur, the first device to notice the collision will back off and wait sending the message until there is no other sending over the wire.

The system modeling the protocol is a network of seven timed automata: two senders, a wire, a receiver, two message generators and an output checker.

The idea of the system is to test if a message can be correctly transmitted from the sender to the receiver even if a collision occurred during the transmission. The test is done by having the two message generators generating messages for the two senders. One of the message generators informs the output checker of the generated message. Both senders then send messages through the wire to the receiver. The receiver informs the output

Example	Some trace			Shortest trace			Fastest trace		
	time	**mem**	**len**	**time**	**mem**	**len**	**time**	**mem**	**len**
Train loc	0.68	55.4	14	0.70	55.4	14	0.67	55.4	14
Train edge	0.59	55.4	15	0.57	55.4	15	0.58	55.4	15
Train du	0.71	55.4	14	0.71	55.4	14	0.71	55.4	14
Philips loc	2.54	55.6	29	2.75	55.6	29	3.19	55.6	29
Philips edge	14.6	67.6	76	25.9	72.6	76	27.0	82.5	76
Philips du	14.9	59.6	104	18.3	61.3	104	24.1	70.3	110
WAP loc	64.8	399	180	65.2	405	180	73.5	456	180
WAP edge	93.2	564	207	92.8	574	207	110	660	207
WAP du	384	595	205	387	606	205	464	700	205

Table 2: Breadth first search. Time (in seconds), space (in Mb), and length (in number of transitions) performance of CO✓ER.

checker about the received message so it can be compared with the generated message. Further description of the system can be found in [?].

4.3.1 Results

In the test run all observers was superposed on the sender automaton, Figure ?? of the audio-control protocol model. The global variable `Volt` in the model was given as argument to the definition use coverage observer. Both the location coverage observer and the edge coverage observer received the two sender automata as argument.

The three rows denoted *Philips loc*, *Philips edge* and *Philips du* in Table ?? and Table ?? present the result of the test run. We find that the edge coverage observer and def-use coverage observer consume a lot more time than the location coverage observers superposed on the sender automaton when using depth first search.

Further we notice, when using breadth first search, that the length of the *fastest trace* of the definition use coverage test case is longer than the *some trace* and the *shortest trace*. This is the only time of the test run (when using breadth search first) where the length between the three different trace characteristics differs.

4.4 Wireless Application Protocol

The Wireless Application Protocol (WAP)[1] is an open international standard describing an architecture using wireless communication to access the Inter-

[1]For a specification of the Wireless Application Protocol visit the web page `http://www.openmobilealliance.org/tech/affiliates/wap/wapindex.html`.

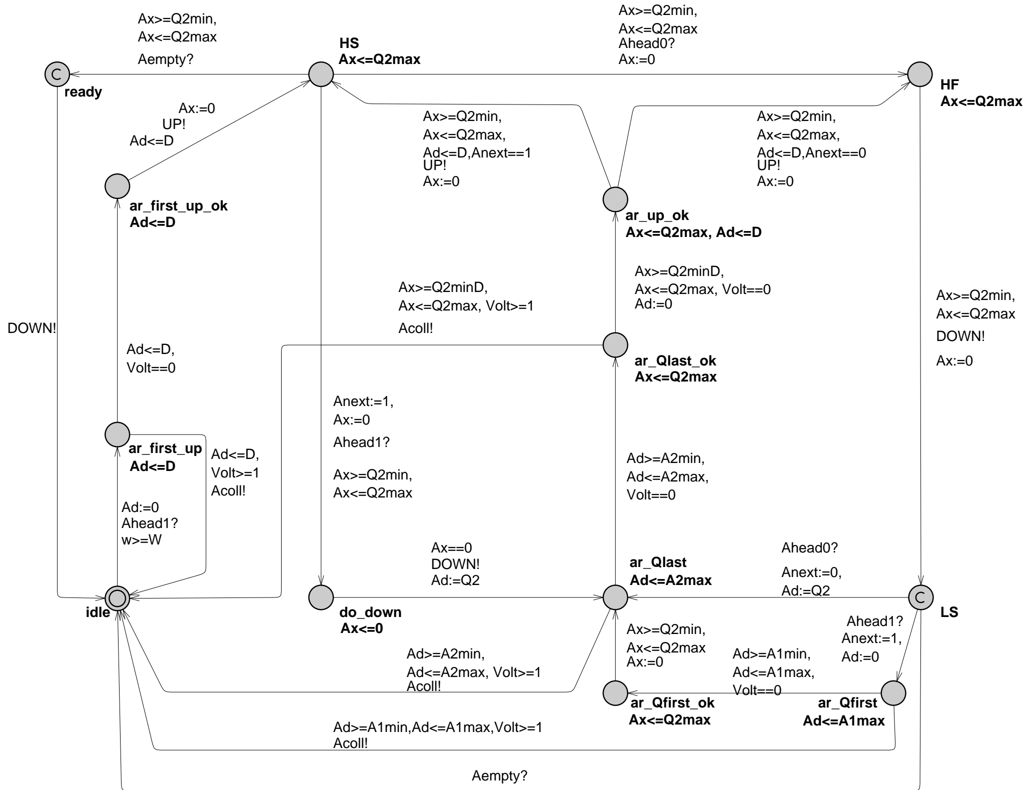


Figure 17: The sender automaton of the audio-control protocol model.

net from mobile devices, such as mobile phones and PDAs. The standard describes both a protocol and a content format, Wireless Markup Language (WML) which is the WAP analogy to HyperText Markup Language (HTML) used by HTTP.

A protocol gateway is used to convert contents between the WML format and the HTML format. It also serves as a proxy translating requests from the WAP protocol stack to the WWW protocols, i.e. HTTP and TCP/IP.

In the test we use a model of a WAP gateway. The model is described by Hessel and Pettersson in [?]. It models a WAP gateway developed by Ericsson in Sweden. This model is a session oriented version of the WAP protocol. Figure ?? shows the WAP protocol stack which consists of the following layers: Wireless Session Protocol (WSP), Wireless Transaction Protocol (WTP), Wireless Datagram Protocol (WDP), and a bearer layer such as GSM, CDMA or UDP. The WSP manages all the sessions in the protocol. A session is multiple transactions from a single terminal that can be

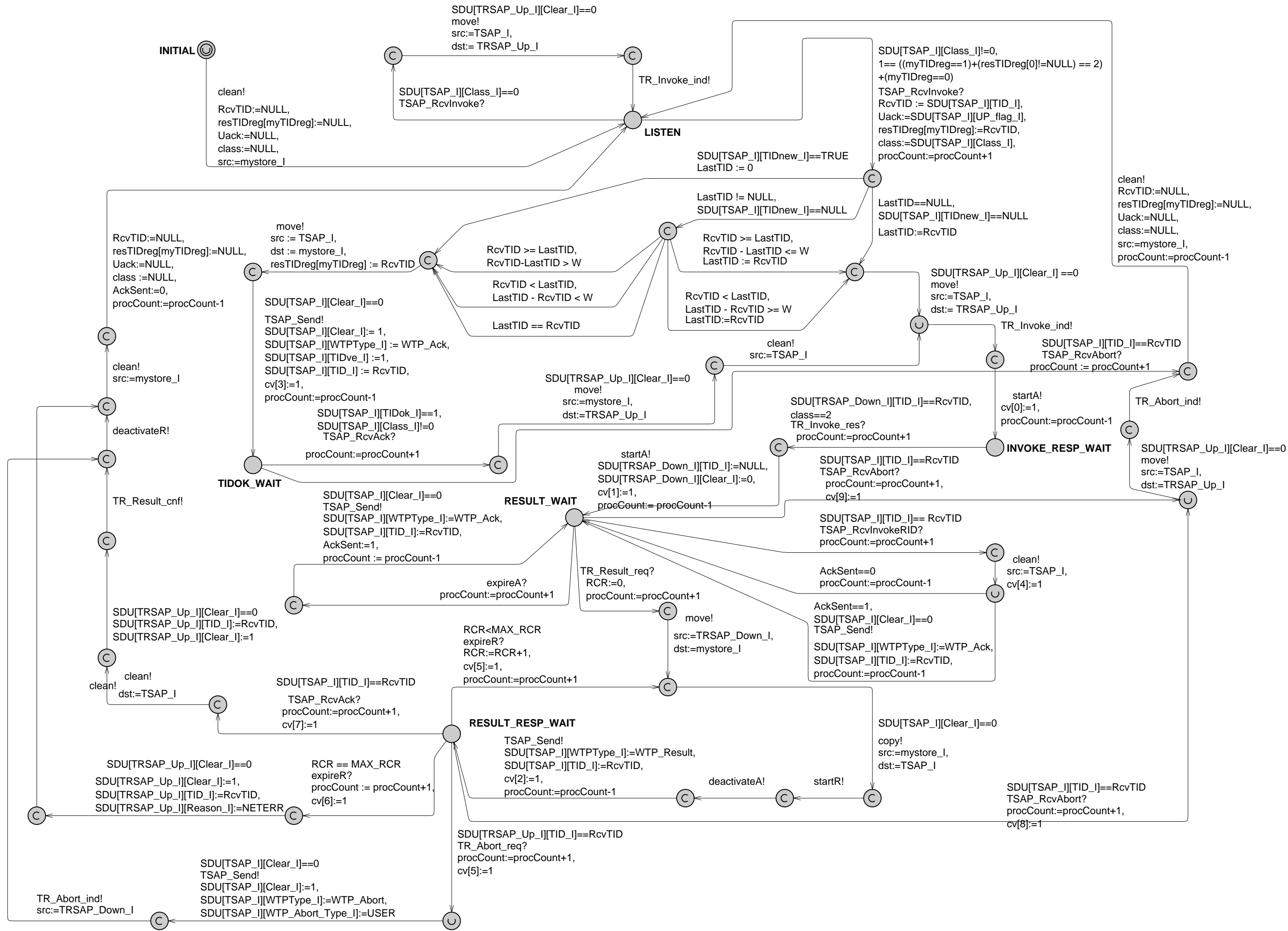


Figure 18: The transaction layer of the WAP protocol.

Session Layer	Wireless Session Protocol (WSP)
Transaction Layer	Wireless Transaction Protocol (WTP)
Transport Layer	Wireless Datagram Protocol (WDP)
Network Layer	Bearer layer, e.g. GSM, CDMA

Figure 19: The WAP protocol stack.

tied together into a session, e.g. when a terminal is logged on to a server. The WTP layer handles the sending and re-sending of transactions. The datagram layer and the bearer layer belong to the lower level of the stack and handle the transmission of data packages over some connection and is omitted in the model of the gateway.

The model we use in the test can be divided into two parts, the WAP gateway and a test environment. The test environment consists of two automata, a web server, and a terminal. The gateway model consists of seven automata modeling the WTP layer, the WSP layer, and globally shared data and timers.

The test environment is used to support the gateway with stimuli, and the idea is to let the terminal model a mobile phone that non-deterministically requests a page from the web server by using the gateway. If the page exists on the server it will be sent back to the terminal through the gateway.

Due to the complexity of the model we only show the automaton of the WAP transaction layer (WTP) which is used to apply the test criteria in the test run, Figure **??** shows the automaton. The interested reader should read [**?**] for a complete description of the model.

4.4.1 Results

The observers were superposed on the WTP automaton in Figure **??** when running the test on the WAP model. The local variable `RcvTID` was given as an argument to the definition use coverage observer.

The last three rows of the Table **??** and Table **??** shows the result of the test run. Test trace generation for the *shortest trace* and *fastest trace* failed when we use the depth first search method because we ran out of memory. 4 GB of memory was not enough!

The length of the *some trace* when using depth first search is really long compared to the trace given by the breadth first search. It is about 700% longer in the worst case.

5 Conclusion and Discussion

In this thesis we showed that it is possible to extend UPPAAL 4.0 with libCoverage to automatically generate test-cases for real-time systems. The extended version of UPPAAL is called CO✓ER.

To realize the extended version of UPPAAL we had to make a number of changes to the program. We first extended the grammar of libUTAP which makes it possible to write CO✓ER specific queries in the property file. This let us specify which observer to use, parameters to the observer, and restrict the observer to a given set of processes. We also added information about the coverage to each state. Further, we described a modified version of the reachability algorithm, presented in [?], which support coverage exploration instead of verifying properties of a timed automata system.

The parameters given to the observer, which are specified in the property file, must be a model identifier. In the version of CO✓ER developed in this thesis we only support three different types of model identifiers:

- local variables of a processes that is not in a set, e.g. `proc(i)`,
- global variables, and
- processes

It would be desirable to also support identifiers like synchronization channels, processes in a set, and template identifiers.

Finally, to make the CO✓ER tool more user friendly it would be nice to have it integrated into the GUI. Where the user can design an observer in a click-n-drag fashion and write CO✓ER queries in the verifier screen.

Appendix

A LibCoverage API

In the following sections we presentes the API of libCoverage and classes needed to use libCoverage.

A.1 Wrapper

Each program, i.e. Uppaal, SPIN etc., that use libCoverage must implement a wrapper class that is special for that program. This wrapper class must inherit libCoverage's Wrapper class and implement the methods listed below:

```
const std::set< std::vector<int> > &getDef()

const std::set< std::vector<int> > &getUse()

const std::set< std::vector<int> > &getLoc()

const std::set< std::vector<int> > &getEdge()

const std::set< std::vector<int> > &getSync()

int evalVar(int)

void activeProcs(std::set<int> &activeProcs)

void clearCache()

int edgeFromParam(std::pair<int,int>)

int locFromParam(std::pair<int,int>)

void setProcs(const std::set<int> &p)

void printVar(std::ostream &o, int varIdx, ParamType type)
```

libCoverage is using this wrapper to fetch information about the system. The following sections will describe each method.

A.1.1 const std::set< std::vector<int> > &getDef()

This function returns a set with vectors identifying each variable that are defined on the active transition the system currently taking. The vector must be defined with size 3 (DefLastPos). Each position in the vector has a specific meaning described below.

v[DefEdgePos] integer identifying the active transition the system is currently taking.
v[DefProcPos] integer identifying the process taking the transition.
v[DefVarPos] integer identifying the defined variable.

A.1.2 const std::set< std::vector<int> > &getUse()

This function returns a set with vectors identifying each variable that are used on the active transition the system currently taking. The vector must be defined with size 3 (UseLastPos). Each position in the vector has a specific meaning described below.

v[UseEdgePos] integer identifying the active transition the system is currently taking.
v[UseProcPos] integer identifying the process taking the transition.
v[UseVarPos] integer identifying the defined variable.

A.1.3 const std::set< std::vector<int> > &getLoc()

This function returns a set with vectors identifying each location in the system. The vector must be defined with size 2 (LocLastPos). Each position in the vector has a specific meaning described below.

v[LocLocPos] integer identifying the location
v[LocProcPos] integer identifying the process the location belongs to.

A.1.4 const std::set< std::vector<int> > &getEdge()

This function returns a set with vectors identifying each active edge in the system. The vector must be defined with size 2 (EdgeLastPos). Each position in the vector has a specific meaning described below.

v[EdgeEdgePos] integer identifying the edge.
v[EdgeProcPos] integer identifying the process the edge belongs to.

A.1.5 const std::set< std::vector<int> > &getSync()

This function returns a set with vectors identifying all synchronization channels on the active edge in the system. The vector must be defined with size 2 (SyncLastPos). Each position in the vector has a specific meaning described below.

v[EdgeEdgePos]	integer identifying the synchronization channel.
v[EdgeProcPos]	integer identifying the process of the active edge.

A.1.6 int **evalVar**(int varID)

This function returns the value of the variable varID in the system.

Parameters	varID - Identifier of the variable to extract the value of.
Returns	Returns the value of the variable identified by varID.

A.1.7 void **activeProcs**(std::set<int> &activeProcs)

Identifier for each active process is inserted into the set activeProcs.

Parameters	activeProcs - populated with identifiers of active processes.

A.1.8 void **clearCache()**

Clears the cache built up by the first call to the wrapper after each transistion.

A.1.9 int **edgeFromParam**(std::pair<int,int> edge)

Converts from an internal representation of an edge used in libCoverage to a identifier used by the system.

Parameters	edge - First element is the process number, second element is the edge number.
Returns	A system specific identifier for the edge.

A.1.10 int **locFromParam**(std::pair<int,int> loc)

Converts from an internal representation of a location used in libCoverage to a identifier used by the system.

Parameters	loc - First element is the process number, second element is the location number.
Returns	A system specific identifier for the location.

A.1.11 `void` **setProcs**`(const std::set<int> &procs)`

The active flag of all edges for each process in the set `procs` will be set. The wrapper will then use this information to know which processes and edges to react on.

Parameters `procs` - a set with process identifiers.

A.2 AllCoverage

This class is used for calculating the coverage of states. It is implemented as a singleton. To receive an instance of this class the method Instance() must be called, see section ??

A.2.1 static AllCoverage* Instance()

Because this class is a singleton it is not possible to call the constructor. To get an instance of this class this method must be used.

Returns	An instance of this class.

A.2.2 void setWrapper(Wrapper *wrp)

Sets the wrapper libCoverage will use to get information from the system, see section ??.

Parameters	wrp - wrapper to be used for communication with the system.

A.2.3 void initialCoverage(Coverage &cov)

Calculates the initial coverage of a state which is written to the argument cov.

Parameters	cov - will be populated with the initial coverage of a state.

A.2.4 int evalAcceptSize(Coverage &cov)

Computes the number of accepting states in the coverage cov, i.e. the amount of coverage.

Parameters	cov - coverage object to count the number of accepting states.
Returns	the number of accepting states in the coverage objectcov.

A.2.5 int makeEntry(OBSERVER::obs_signature_t &os, std::set<int> procs)

Adds an observer to libCoverage. See section ?? for a description of OBSEVER::obs_signature_t.

Parameters	os - an observer signature describing it's name and parameters.
	procs - a set of process identifiers to restrict the observer to.
Returns	Returns an identifier for the observer.

A.2.6 `bool newCoverage(Coverage &cov)`

Will exercise the observers in order to look for new coverage in the current state.

Parameters	`cov` - coverage object to update.
Returns	`true` if a new coverage was found otherwise `false`.

A.3 Coverage

This class stores the information about the coverage of a state.

A.3.1 `int relation(const Coverage &cov) const`

Checks the relation between `this` coverage object and `cov`.

Parameters	`cov` - coverage to compare relation with.
Returns	`Coverage::equal` if `this` and `cov` are equal. `Coverage::argIncludedIn` if `cov` is the smaller set. `Coverage::argIncludes` if `cov` is the biggest set.

A.4 Observer

A.4.1 `OBSERVER::obs_signature_t`

Structure for describing an observer.

```
struct obs_signature_t {
   string name;
   std::list<obs_param_t> params;
   obs_signature_t(const char *n) : name(n) {}
   ~obs_signature_t() { }
};
```

`name`	the name of the observer.
`params`	parameters to the observer.
`obs_signature_t(const char *n)`	Constructor

A.4.2 `OBSERVER::obs_param_t`

Structure for describing a parameter to an observer.

```
struct obs_param_t {
   ParamType obstype;
   set< pair<int, int> > altList;
   obs_param_t(ParamType pt, set< pair<int, int> > altS)
       : obstype(pt), altList(altS) {}
   ~obs_param_t() {}
};
```

`obstype`	type of parameter (see **??**).
`altList`	set of parameters (of the same type). The data in each pair depends of the parameter type.
`obs_param_t(...)`	Constructor

The following table describes the meaning of the values in each pair of the set `altList`.

`obstype`	**first element**	**second element**
`ProcId_t`	-1	identifier of the process
`VarId_t`	-1	identifier of the global variable
`VarId_t`	identifier of the process the variable belongs to	identifier of the local variable

A.4.3 `OBSERVER::ParamType`

Enumeration of possible parameter types to the observer.

ProcId_t	process identifier.
VarId_t	variable identifier.