Software Tools for Technology Transfer Manuscript-Nr.
(will be inserted by hand later)

UPPAAL in a Nutshell

Kim G. Larsen[1], *Paul Pettersson*[2], **and** *Wang Yi*[2]

Department of Computer Science and Mathematics, Aalborg University, Denmark. Email: `kgl@cs.auc.dk`
Department of Computer Systems, Uppsala University, Sweden. Email: `{paupet,yi}@docs.uu.se`

Abstract. This paper presents the overall structure, the design criteria, and the main features of the tool box Uppaal. It gives a detailed user guide which describes how to use the various tools of Uppaal version 2.02 to *construct* abstract models of a real-time system, to *simulate* its dynamical behavior, to *specify* and *verify* its safety and bounded liveness properties in terms of its model. In addition, the paper also provides a short review on case-studies where Uppaal is applied, as well as references to its theoretical foundation.

1 Introduction

Uppaal is a tool box for modeling, simulation and verification of real-time systems, based on constraint–solving and on-the-fly techniques, developed jointly by Uppsala University and Aalborg University. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and (or) shared variables [34, 26]. Typical application areas include real–time controllers and communication protocols in particular, those where timing aspects are critical. It is designed mainly to check invariant and reachability properties by exploring the state–space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints.

The two main design criteria for Uppaal have been *efficiency* and *ease of usage*. The restriction to reachability analysis has been crucial to the efficiency of the Uppaal model-checker. Another important key to efficiency is the application of *on-the-fly* searching technique combined with the *symbolic* technique that reduces verification problems to that of manipulating and solving simple constraints [34, 26]. To facilitate modeling and debugging, the Uppaal model–checker may automatically generate a *diagnostic trace* that explains why a property is (or is not) satisfied by a system description. The diagnostic traces generated by the model–checker may be graphically visualized using the simulator.

Since its first release in 1995, Uppaal has been applied in a number of case studies (see Section 6 for a summary). To meet requirements arising from the case studies, It has been extended with various features. The current version of Uppaal is implemented in C++, XForms and Motif. This paper is devoted to an informal presentation of Uppaal. We present the semantics model implemented in Uppaal, its various features, review and provide references on the theoretical foundation and applications to case–studies. We also provide a detailed user guide. The paper is organized as follows:

Section 2 describes the overall structure and the main components of Uppaal. Section 3 is an *informal* presentation of the syntax and semantics of the Uppaal model. Section 3 presents the logic and the kernel of the model-checking algorithm of the Uppaal model-checker. Section 5 serves as a user guide, describing in details how to use the various tools of Uppaal. Section 7 concludes the paper with a brief description on recent and possible future development of Uppaal.

2 Overview of UPPAAL

An overview of Uppaal is shown in Figure 1. In this section we briefly describe the main features of Uppaal.

Uppaal consists of three main parts: a description language, a simulator and a model-checker. The description language is a non-deterministic guarded command language with data types[1]. It serves as a modeling or design language to describe system behavior as networks of

[1] Currently, only `integer` and `clock` with restricted forms of operations are implemented.

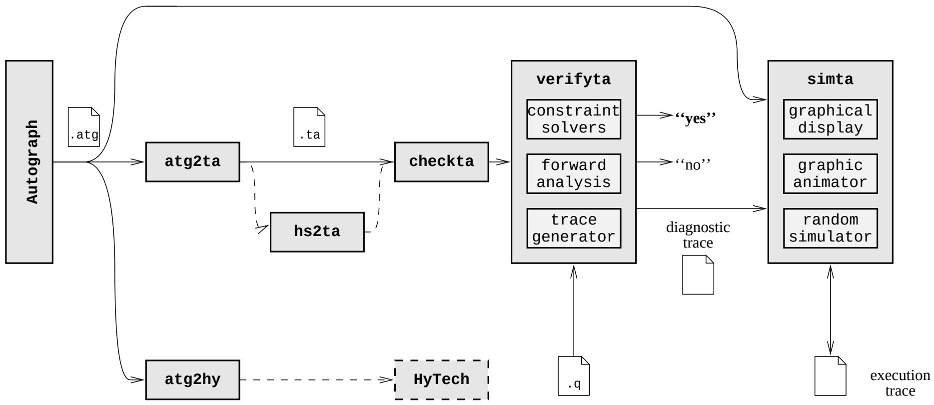

Fig. 1. Overview of UPPAAL

timed automata extended with data variables. The simulator and the model–checker are designed for interactive and automated analysis of system behavior by manipulating and solving constraints that represent the state–space of a system description. They have a common basis, i.e. constraint–solvers. The simulator enables examination of *possible* dynamic executions of a system during early modeling (or design) stages and thus provides an inexpensive mean of fault detection prior to verification by the model-checker which covers the *exhaustive* dynamic behavior of the system.

2.1 Modeling

To facilitate modeling, UPPAAL provides both graphical and textual formats for the description language. One can use either the textual format or the AUTOGRAPH-based graphical user interface [11] to define system descriptions, namely networks of timed automata. As an example, the textual representation of the graphical system description in Figure 3 is shown in Figure 2.

The textual format (i.e. `.ta`) provides a basic *programming language for timed automata*. In certain cases, the textual format can be more convenient (and faster) to work with than the graphical interface. The compiler `atg2ta` automatically transforms system description in the graphical `.atg`–format into the textual `.ta`–format, thus supporting the important principle WYSIWYV[2].

The UPPAAL description language also supports modeling of simple linear hybrid automata, that is, timed automata with clocks whose rates may vary in a certain interval [31]. This extension of timed automata is useful for modeling of hybrid systems where the behavior of the system variables can be described or approximated using lower and upper bounds on their rates. Using abstraction techniques, this class of linear hybrid system can be transformed into timed automata and thus be verified using the techniques available for timed automata, implemented in UPPAAL. UPPAAL allows linear hybrid automata where the rates of clocks are given by an interval. Philips Audio-Control Protocol of [10] is an example of such a linear hybrid systems.

[2] What You See Is What You Verify.

2.2 Analysis

Model–Checking. The model–checker is designed to check for invariant and reachability properties, in particular whether certain combinations of control-nodes and constraints on clocks and integer variables are reachable from an initial configuration. Other properties such as bounded liveness properties can be checked by reasoning about the system in the context of testing automata or simply decorating the system description with debugging information and then checking reachability properties. Model-checking is performed by the module `verifyta` which takes as input a network of automata in the textual-format (i.e. `.ta`) and a formula. In checking a property, a *diagnostic trace* can be automatically reported by `verifyta` [25], that explains why the property is satisfied or not. Such a trace may be considered as diagnostic information of a system error, useful during the subsequent debugging of the system.

Simulation. The simulator allows the user to examine in an interactive and graphical fashion the dynamic behavior of a system. In contrast to the model–checker which explores the whole reachable state–space of a system — examining all the behavior of the system, the simulator explores only a particular execution trace i.e. a sequence of states of the system. This will in early stages of modeling (or design) provide an inexpensive mean of fault detection. In comparison the model–checker is obviously more expensive as it amounts to an exhaustive simulation covering all behavior of the system. Another useful application of the simulator is to visualize a diagnostic trace generated by the model–checker; thus the user can in an interactive and graphical fashion examines the execution trace that may result in a system error.

```
//
// Global declaration section
//

clock  x, y;
int    n;
chan   a;

//
// Component description section
//

process A {
  state  A0 { y<=6 }, A1, A2, A3;
  init   A0;
  trans  A0 -> A1 {
           guard  y>=3;
           sync   a!;
           assign y:=0;
         },
         A1 -> A2 {
           guard  y>=4;
         },
         A2 -> A3 {
           guard  n==5;
         };
}

process B {
  state  B0 { x<=4 }, B1, B2, B3;
  commit B1;
  init   B0;
  trans  B0 -> B1 {
           guard  x>=2;
           sync   a?;
           assign n:=5,x:=0;
         },
         B1 -> B2 {
           assign n:=n+1;
         },
         B2 -> B3 {
         };
}

//
// System description section
//

system A, B;
```

Fig. 2. Textual description.

3 The Bolts of UPPAAL — Modelling

In this section, we present the basic ingredienses of the UPPAAL model based on small examples. For a precise semantical treatment we refer the reader to [5].

We assume that a typical real-time system is a network of non–deterministic sequential processes communicating with each other over channels. In UPPAAL we use finite–state automata extended with clock and data variables to describe processes and networks of such automata to describe real–time systems.

3.1 Syntax

The basis of the UPPAAL model is the notion of timed automata [3] developed by Alur and Dill as an extension of classical finite–state automata with clock variables. To provide a more expressive model and to ease the modeling task, we further extend timed automata with more general types of data variables such as boolean and integer variables. Our final goal is to develop a modeling (or design) language which is as close as possible to a high–level real–time programming language with various data types. Clearly, this will create problems for decidability of model–checking. However, we can always require that the value domains of the data variables should be finite in order to guarantee the termination of a verification procedure (as in Murφ [17]).

In the current implementation of UPPAAL a system description (or model) consists of a collection of timed automata extended with integer variables in addition to clock variables. Consider the UPPAAL model of Figure 3. The model consists of two components `A` and `B` with control nodes $\{A_0, A_1, A_2, A_3\}$ and $\{B_0, B_1, B_2, B_3\}$ respectively. In addition to these discrete control structures, the model uses two clocks x and y, one integer variable n and one channel a.

The edges of the automata are decorated with three types of labels: a *guard*, expressing a condition on the values of clocks and integer variables that must be satisfied in order for the edge to be taken; a synchronization *action* which is performed when the edge is taken and finally a number of *clock resets* and *assignments* to integer variables. All three types of labels are optional.

In addition, control nodes may be decorated with so–called *invariants*, which are conditions expressing constraints on the clock values in order for control to remain in a particular node.

3.1.1 Guards

Guards express conditions on the values of clocks and integer variables that must be satisfied in order for the edge to be taken. In Figure 3 the edge between A_0 and A_1 can only be taken, when the value of the clock y is greater than or equal to 3. Similarly the edge between A_2 and A_3 can only be taken when the value of the integer variable n equals 5. Formally, guards are conjunctions of timing and data constraints; a timing constraint is of the form: $x \sim n$ or $x - y \sim n$, where n is a natural number

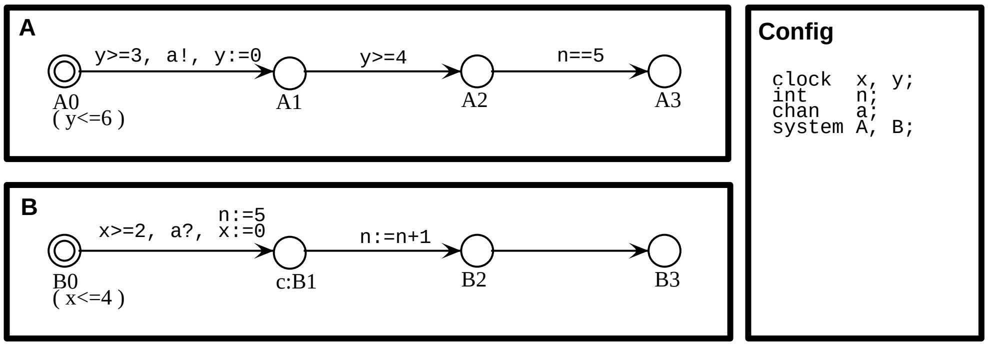

Fig. 3. An example UPPAAL model.

and $\sim\in\{\leq,\geq,=,>,<\}$; a data constraint is of a similar form $i\sim j$ or $i-j\sim k$ but with k being an arbitrary integer. The default guard of an edge is *true*.

3.1.2 Reset–Operations

When taking an edge clock and data variables may be subject to simple manipulations in terms of *resets* being assignments of the form $w := e$, where w is a clock or data variable and e is an expression. In the current version of UPPAAL reset–operations on clock variables must be of the simple form $x := n$, where n is a natural number, and reset–operations on integer variables should be in the form $i := c*i+c'$, where c, c' are integer constants (note that c, c' may be zero or negative). As examples reconsider Figure 3. Here the clock y is reset to 0 when the edge between A_0 and A_1 is taken. Similarly the integer variable n is incremented when the edge from B_1 to B_2 is taken.

3.1.3 Channels, Synchronization and Urgency

A UPPAAL model consists of a network of (extended) timed automata. Automata may communicate either via integer variables (which in UPPAAL are global) or using communication channels. As in CCS [30] communication on channels occur as two–process synchronizations. In Figure 3 the two processes may communicate via the channel a.
To denote the actions that processes can perform when synchronizing with each other we use the notation $a!$ and $a?$ (denoting the complementary actions of sending and receiving on channel a). Absence of synchronization action indicates an internal (non–synchronizing) edge similar to τ–transitions in CCS. In Figure 3, the edge between A_1 and A_2 is an example of an internal edge of the process A.
To prevent a network from delaying in a situation where two components are already able to synchronize, a channel may be declared as being *urgent*. For efficiency reasons edges labelled with synchronization actions on urgent channels may not have guards on clocks.

3.1.4 Committed Locations

To introduce the notion of committed locations in timed automata, consider the scenario shown in Figure 4:

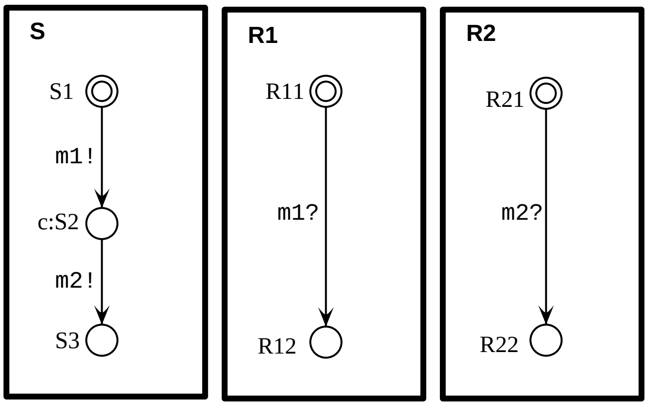

Fig. 4. Broadcasting Communication and Committed Locations.

A sender S is to broadcast a message m to two receivers R_1 and R_2. As this requires synchronization between *three* processes this can not directly be expressed in UPPAAL where synchronization, as in CCS, is between two processes based on complementarity of actions. However, as an initial attempt we may model the broadcast as a sequence of two two-process synchronizations, where first S synchronizes with R_1 on m_1 and then with R_2 on m_2. However, this is not an accurate modeling as the intended atomicity of the broadcast is not preserved (i.e. other processes may interfere during the 'broadcast' sequence). To ensure atomicity, we mark the intermediate location S_2 of the sender S as a so-called *committed location* (indicated by the c:-prefix). The atomicity of the action sequence $\mathsf{m}_1!\mathsf{m}_2!$ is now achieved by insisting that a committed location must be left immediately! This behavior is quite similar to what has been called “urgent transitions” [19, 15, 7] which insists that the next transition taken must be an action (and not a delay). The precise semantics of committed locations will be formalized in the transition rules for networks of timed automata with data variables in the following.

3.1.5 Invariants

To enforce progress in a system, control nodes may be decorated with so–called *invariants*, which express constraints on the clock values in order for control to remain in a particular node. The default of a location invariant is *true*. Thus, in Figure 3, control can only remain in A_0 as long as the value of y is no more than 6.

3.2 Semantics

Formally, states of a UPPAAL model are of the form $(\bar{l}, v)$, where $\bar{l}$ is a *control vector* indicating the current control node for each component of the network and v is an *assignment* given the current value for each clock and integer variable. A UPPAAL model determines the following two types of *transitions* between states:

Delay transitions As long as none of the invariants of the control nodes in the current state are violated time may progress without affecting the control node vector and with all clock values incremented with the elapsed duration of time. In Figure 3, from the initial state $((A_0, B_0), x = 0, y = 0, n = 0)$ time may elapse 3.5 time units leading to the state $((A_0, B_0), x = 3.5, y = 3.5, n = 0)$. However, time cannot elapse 5 time units as this would violate the invariant of B_0.

Action transitions If two complementary labelled edges of two different components are enabled in a state then they can synchronize. Thus in state $((A_0, B_0), x = 3.5, y = 3.5, n = 0)$ the two components can synchronize on $\mathtt{a}$ leading to the new state $((A_1, B_1), x = 0, y = 0, n = 5)$ (note that x, y, n have been appropriately updated). If a component has an internal edge enabled, the edge can be taken without any synchronization. Thus in state $((A_1, B_1), x = 0, y = 0, n = 5)$, the B–component can perform without synchronizing with A, leading to the state $((A_1, B_2), x = 0, y = 0, n = 6)$.

The above two types of transitions may be overruled by presence of urgent channels and committed locations in the following ways:

Urgent Channels In a state where two components may synchronize on an urgent channel no further delay is allowed. Thus, in Figure 3 if channel a is urgent[3], time may *not* elapse 3.5 units from the initial state $((A_0, B_0), x = 0, y = 0, n = 0)$ as synchronization on a is already possible in the state $((A_0, B_0), x = 3, y = 3, n = 0)$.

Committed Locations If in a state one of the components is in a control node labelled as being committed, no delay is allowed to occur and any action transition (synchronization or not) *must* involve the particular component (the component is so–to–speak committed to continue). In the state $((A_1, B_1), x = 0, y = 0, n = 5)$ B_1 is committed; thus without any delay the next transition must involve the B–component; i.e. the next state of the network is $((A_1, B_2), x = 0, y = 0, n = 6)$. Hence the two first transitions of B are guaranteed to be performed atomically.

4 The Nuts of UPPAAL — Specifying

The UPPAAL model-checker is designed to check for simple invariant and reachability properties. A number of other properties, including bounded reachability properties, may be checked by reasoning about the system in the context of testing automata. We give an informal presentation of the UPPAAL logic and an example use of testing automata in the next two sections. Hereafter, we give a short review of the model-checking technique used in UPPAAL and point out some recent developed and implemented space-saving improvements.

4.1 UPPAAL Specifications

In the current version, UPPAAL is able to check for reachability properties, in particular whether certain combinations of control-nodes and constraints on clock and data variables are reachable from an initial configuration. The properties that can be analyzed are of the forms:

$$\varphi ::= \forall\Box\beta \mid \exists\Diamond\beta \qquad\qquad \beta ::= a \mid \beta_1 \wedge \beta_2 \mid \neg\beta$$

Where a is an atomic formula being either an atomic clock (or data) constraint or a component location ($A_i\,\mathtt{at}\ l$). Atomic clock (data) constraints are either integer bounds on individual clock (data) variables (e.g. $1 \le x \le 5$) or integer bounds on differences of two clock (data) variables (e.g. $3 \le x - y \le 7$).

Intuitively, for $\forall\Box\beta$ to be satisfied all reachable states must satisfy β. Dually, for $\exists\Diamond\beta$ to be satisfied some reachable state must satisfy β. Formally let $\leadsto$ denote the transitive closure of the delay- and action-transition relations between states. Then the satisfaction relation $\models$ between states and formulas are defined as follows:

$$\langle\bar{l}, v\rangle \models \exists\Diamond\beta \iff \exists\langle\bar{l'}, v'\rangle.\langle\bar{l}, v\rangle \leadsto \langle\bar{l'}, v'\rangle \wedge \langle\bar{l'}, v'\rangle \models \beta$$
$$\langle\bar{l}, v\rangle \models \forall\Box\beta \iff \forall\langle\bar{l'}, v'\rangle.\langle\bar{l}, v\rangle \leadsto \langle\bar{l'}, v'\rangle \Rightarrow \langle\bar{l'}, v'\rangle \models \beta$$

Satisfaction with respect to a boolean combination β of atomic formulas is defined inductively on the structure

[3] Note, that strictly speaking, this would violate the syntactic restriction that the guard of edges labelled with urgent actions must be empty.

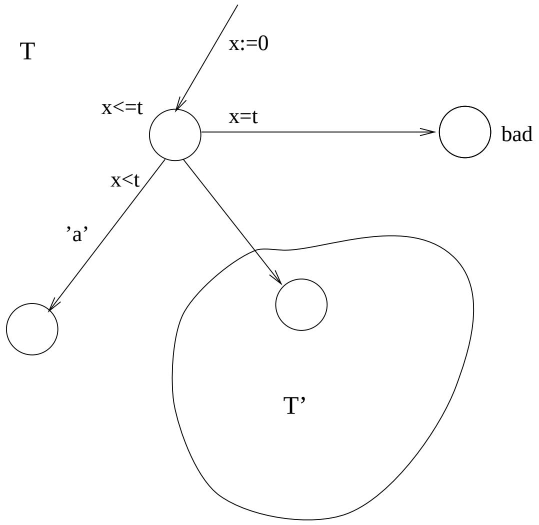

Fig. 5. Test Automata for $\phi\,\mathsf{Until}_{<t}\,a$

PASSED:= $\{\}$
WAITING:= $\{(\bar{l}_0, D_0)\}$
repeat
  begin
    get $(\bar{l}, D)$ from WAITING
    if $(\bar{l}, D) \models \beta$ **then return "YES"**
    else if $D \not\subseteq D'$ **for all** $(\bar{l}, D') \in$ PASSED **then**
      begin
        add $(\bar{l}, D)$ to PASSED
        SUCC:=$\{(\bar{l}_s, D_s) : (\bar{l}, D) \leadsto (l_s, D_s) \wedge D_s \neq \emptyset\}$
        for all $(\bar{l}_{s'}, D_{s'})$ in SUCC **do**
          put $(\bar{l}_{s'}, D_{s'})$ to WAITING
      end
  end
until WAITING=$\{\}$
return "NO"

Fig. 6. An Algorithm for Symbolic Reachability Analysis.

of β (behaving as usual with respect to the boolean connectives). Satisfaction with respect to an atomic formula is given by the following definitions:

$$\langle \bar{l}, v\rangle \models c \iff v \in c$$
$$\langle \bar{l}, v\rangle \models A_i \mathsf{at}\ l \iff l_i = l$$

4.2 Test Automata — Beyond Reachability

Our (simple and efficient) model-checking technique extends to the logic presented in [25], which also allows for bounded liveness properties to be specified. Currently, bounded liveness properties must be obtained by reachability analysis of the system in the context of testing (and time-sensitive) automata.

Consider the following real-time property

$$\psi = \phi\,\mathsf{Until}_{<t}\,a$$

stating that the (atomic) property a must hold before t time units and that ϕ must hold until then. Here we assume that a is of the form $A_i\mathsf{at}\ l$. Now to verify that a system S satisfies the formulae we extend it with the test automata T of Figure 5 as a component.

Here T' is assumed to be an already constructed test automata for the sub-property ϕ, and 'a' is a (urgent) probe action inserting into the component A_i at location l. Now, it may be shown that our original system S will satisfy $\phi\,\mathsf{Until}_{<t}\,a$ if and only if $S\,|\,T$ satisfies the invariance property $\forall\Box\neg(T\mathsf{at}\ bad)$.

We conjecture that all bounded liveness properties of the logic in [25] can be translated into reachability problems in this manner, and in a forthcoming version of UPPAAL we intend to provide automatic support for generation of test automata from logical formulas (as has been done in the tool SPIN, where "never-claims" are directly generated from Linear Temporal Logic properties). For an initial investigation we refer the reader to [20].

4.3 Model Checking

The model-checking procedure implemented in UPPAAL is based on an interpretation using a finite-state symbolic semantics of networks. More precisely, we interpret the logic with respect to symbolic states of the form $(\bar{l}, D)$, where D is a constraint system (i.e. a conjunction of atomic clock and data constraints) and $\bar{l}$ a control-vector. Thus, a symbolic state $(\bar{l}, D)$ represents all the states $(\bar{l}, v)$ where v satisfies the constraint D. Based on this notion of symbolic state, the heart of the UPPAAL model-checking procedure is the abstract reachability algorithm shown in Figure 6[4], which reduces the reachability problem to that of solving simple constraint systems. The algorithm is to check whether a timed automaton may reach a state satisfying a given state formula β.

We observe that several operations of the algorithm are critical for efficient implementations. First, the algorithm depends heavily on the test operations for checking the inclusion $D \subseteq D'$ (i.e. the inclusion between constraints D and D') and the emptiness of D_s in constructing the successor set SUCC of $(\bar{l}, D)$. Clearly, it is important to design efficient data structures and algorithms for the representation and manipulation of clock constraints. One such well-known data structure is that of Difference Bounded Matrices [4, 16, 33], DBM, which offers a canonical representation for constraint systems. It has been successfully employed by several real-time verification tools, e.g. UPPAAL [8] and KRONOS [15].

[4] The relation $\leadsto$ has been extended to symbolic states in the obvious fashion.

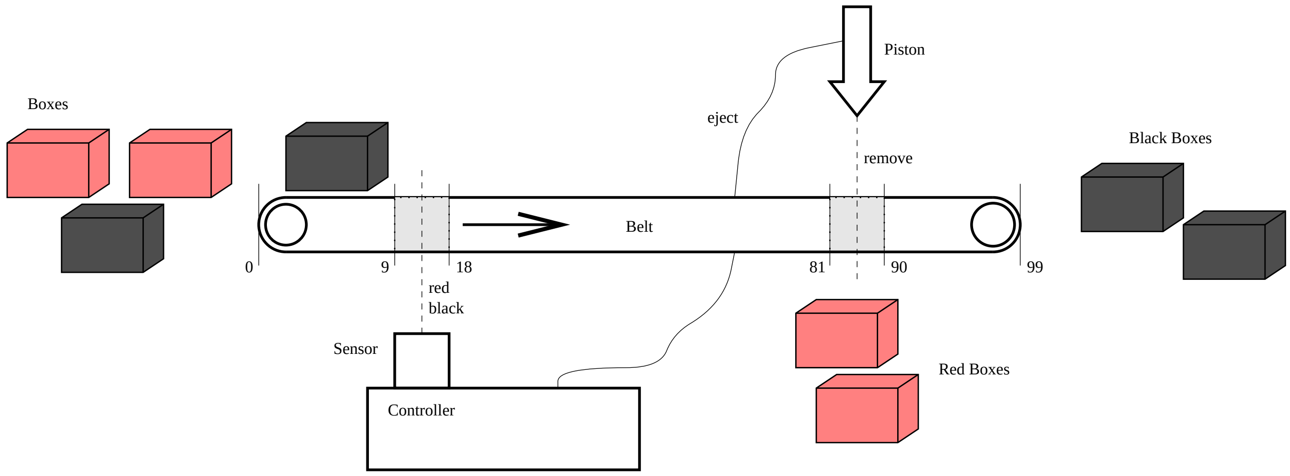

Fig. 7. The Box-Sorter Unit.

In the UPPAAL-implementation the reachability algorithm of Figure 6 is extended so that a *diagnostic trace* is automatically generated as a side-effect in case reachability is established. If the symbolic state-space is examined in a breadth-first manner (corresponding to organizing the waiting set as a queue) the trace is guaranteed to be the shortest possible.

4.4 Optimizations

A DBM representation is in fact a weighted directed graph where the vertices correspond to clocks (including a zero-clock) and the weights on the edges stand for the bounds on the differences between pairs of clocks [4, 16, 33]. As it gives an explicit bound for the difference between each pair of clocks, its space-usage is in the order of $\mathcal{O}(n^2)$ where n is the number of clocks. However, in practice it often turns out that most of these bounds are redundant. In [27], we have presented a new compact data structure for DBM, which provides *minimal* and *canonical* representations of clock constraints and also allows for efficient inclusion checks. The representation is obtained by a minimization of the weighted directed graph representing the constraint system, and our experimental results demonstrate truly significant space-savings as well as better time-performance.

In addition to the *local* reduction technique above, which is to minimize the space-usage of each individual symbolic state, we have developed a *global* reduction technique to reduce the total number of symbolic states to save in the global data structure, i.e. the passed list. It is completely orthogonal to the local technique and is based on static analysis of the control structure of the system. Again our experimental results demonstrate significant space-savings and improved time-performance.

Finally, it is possible to save the symbolic state-space generated during checking of a property and re-use it in the checking of other properties. In cases where several correctness properties have to be examined this leads to significant time-savings.

5 How to Use UPPAAL — a User Guide

This section describes how to model, simulate and verify real-time systems using UPPAAL. We will focus on the graphical interface and the graphical modeling language of UPPAAL. However, it should be noticed that the tool also has a text-based interface and a textual representation of the modeling language.

As a running example throughout this section we will use the box-sorting unit of [23].

Example 1 (The Box-Sorter Unit). *The box-sorter shown in Figure 7 is made in LEGO*[5]*. It sorts red and black boxes. The sorter is built around a belt that transports boxes in the unit, which consists of four components: a color-sensor, a piston, a controller, and an observer.*

A box starts at the leftmost extreme of the belt, represented by position 0. At some position from 9 to 18 its color is sensed by the color-sensor which is attached to the controller. The controller reacts if the color is red by sending an eject request to the piston, after a certain delay. The piston ejects within one time unit after the arrival of a request. When the piston is ejected it is guaranteed to remove the box if it is positioned in the interval 81 to 90. If the box is not removed (i.e. if the box is black), it proceeds to position 99, representing the rightmost extreme of the belt, where it falls off the belt.

The observer is not participating in the sorting of boxes. Its only task is to observe that no red boxes appears on the rightmost extreme of the belt. As the observer is not

[5] Information about LEGO can be found on World Wide Web at: `http://www.lego.com/`.

part of the sorting mechanism, we have not shown him in Figure 7 but he can be imagined to sit at the far right end.

For simplicity we regard the system as being correct if the observer sees no red boxes (i.e. only black boxes) at the rightmost extreme of the belt. □

5.1 XUPPAAL

XUppaal is to provide a user-friendly graphical interface to the tools in Uppaal. It offers support to the user by managing the working file names, by providing an easy way to give optional setting, and to execute the various tools of the tool-box. The application is implemented in C and the Forms Library for X[6].

In the following we give a short overview of the tool. The description continues in Section 5.4 which explains how to verify and generate diagnostic traces with XUppaal. A more detailed description of XUppaal can be found in [5].

5.1.1 The Files

During an XUppaal session the user works with three different kind of files: a system description, a requirement specification, and a trace file.

The *system description* file contains the system description. It is assumed to be given in the textual format (called `.ta`) or on the graphical format (called `.atg`) used by Autograph and the simulator. Section 5.2 provides a guide for definition of system descriptions using Autograph. Once the file have been created its syntax can be checked by invoking `Syntax Check` in the `Run` menu of XUppaal.

The *requirement specification* file holds a set of formulae in a textual representation (called `.q`). The file can be created using a simple editor invoked in the `Run` (item `Req. Spec. Editor`) menu of XUppaal or using an external text editor.

The *trace* file is used to store information about diagnostic traces generated by the verifier. A trace is a sequence of (symbolic) states and transitions, that represents an (symbolic) execution of the system. It is often useful for discovering why a property is (or is not) satisfied by a system description. The trace file can be input to the simulator which is able to display traces graphically (see Section 5.3).

[6] Information about the Forms Library for X is available on World Wide Web: `http://www.uwm.edu/~zhao` and on anonymous ftp: `ftp://bloch.phys.uwm.edu/pub/xforms`.

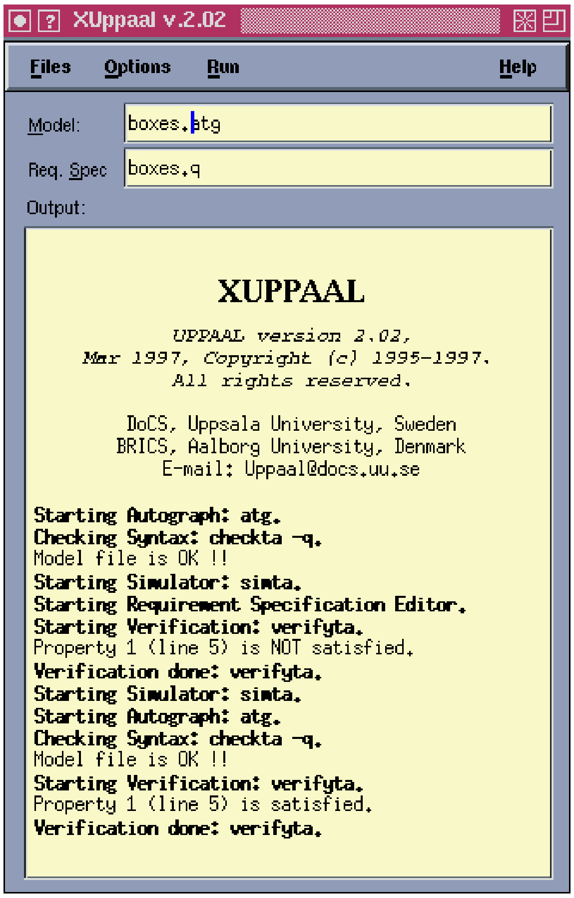

Fig. 8. The Graphical Interface of XUppaal.

5.1.2 Getting Started

XUppaal is activated from the command line with the command `xuppaal`. On startup, XUppaal shows the user with its main window as shown in Figure 8. The main window consists of four parts: the menu bar, the two input fields that displays the name of the currently specified system description file (labeled `Model`) and requirement specification file (labeled `Req. Spec.`), and the output browser (labeled `Output`).

5.1.3 The Menu Bar

The menu bar has four sub-menus: `Files`, `Options`, `Run` and `Help`. Each sub-menu is invoked by clicking the menu label or using snap-keys. The snap-keys are given below enclosed in parenthesis (where `C` denotes the `Ctrl`-button).

The `File` menu (`C-x`) contains entries for selecting the files to work with, and for exiting XUppaal. The `Help` menu (`C-h`) provides online help on the six topics: `General` (`C-g`), `Files` (`C-f`), `Options` (`C-o`), `Run` (`C-r`), `Problems` (`C-p`) and `Version` (`C-v`). When an entry is selected, the help text is displayed in the output browser. The output

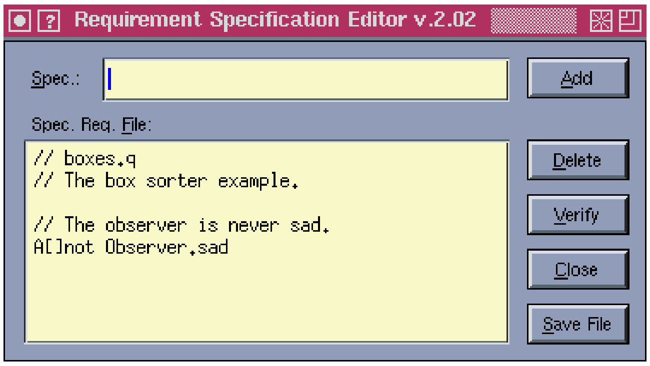

Fig. 9. The Requirement Specification Editor.

generated by selecting the menu item `Version` is shown on the first seven lines of text in the browser in Figure 8.

We proceed by giving a more detailed description of the two sub-menus: `Options` and `Run`.

The Options Menu `(C-o)` provides a list of choices that mainly affect the verification session. Each entry in the menu can be toggled on or off (which is visualized in the menu by a filled or empty check-box to the left of each menu item):

`Auto Check Syntax (C-a)`: Automatically perform syntax check before simulation and verification sessions.
`Diagnostic Info (C-i)`: Generate diagnostic traces in textual format and present the result in the output browser.
`Diagnostic File (C-f)`: Produce diagnostic traces on the specified trace file.
`Breadth-First (C-b)`: Explore the state-space of the system by breadth-first search[7].
`Depth-First (C-d)`: Explore the state-space of the system by depth-first search.
`Local Reduction (C-l)`: Use compact data-structures to represent constraints (instead of DBM, see Section 4.4).
`Global Reduction (C-s)`: Perform Control Structure Analysis (see Section 4.4).
`Re-use State-Space (C-r)`: Re-use the generated portion of the state-space when verifying *several* reachability properties (see Section 4.4).

The Run Menu `(C-c)` contains commands that are associated with the various tool programs of UPPAAL. The outputs produced by the programs are always displayed in the output browser.

`Autograph (C-a)`: Start the graphical editor Autograph.
`Syntax Check (C-c)`: Perform syntactical check on the textual format of the system description.
`Simulation (C-s)`: Start the simulator with the specified system description file.
`Req. Spec. Editor (C-e)`: Open the requirement specification editor shown in Figure 9.
`Verification (C-v)`: Check if the system description satisfies the requirement specification by model-checking.
`Show Model (C-f)`: Display the system description in textual representation[7] .
`Show Req. Spec. (C-r)`: Show the contents of the requirement specification file.
`Clear Output (C-o)`: Clear the output browser (i.e. erase the contents of the browser).

[7] This option generates a shortest diagnostic trace when used in combination with one of the two trace options above.

Example 2 (Specifying the File Names). *As a preliminary step when working with the box-sorter example we define the file names for XUPPAAL to work with. We do this by typing in the system description file name "*`boxes.atg`*" in the field labeled* `System` *and the requirement specification file name "*`boxes.q`*" in the field labeled* `Req. Spec.` *The trace file is specified by selecting* `Set Trace File` *from the* `Files` *menu. We use the file name* `boxes.tr`*. The resulting XUPPAAL input fields are shown in Figure 8.* □

5.2 How to Model

UPPAAL allows for systems descriptions to be defined textually or by drawing using Autograph. In this section we describe how to define a system using Autograph. For a description of the textual representation of systems used in UPPAAL we refer the reader to [5].

5.2.1 Systems Descriptions in Autograph

Autograph is a graphical tool for drawing automata-based system descriptions. It is very general though it was developed for the FC2TOOLS set [11]. This section will explain the subset of features of Autograph, that are needed to define UPPAAL system descriptions.

To define network of timed automata in Autograph it is necessary to define a mapping from the elements of timed automata to graphical objects in Autograph. Here we summarize the mapping of the components that a system description should consist of.

A location is denoted by a vertex labeled with its name (required) and its invariant (which is optional). By default, the invariant is *true*.
A transition is denoted by an edge connecting two vertices. The transition may be labeled with a guard which is *true* by default, a synchronization action (which is τ by default), and a list of assignments.

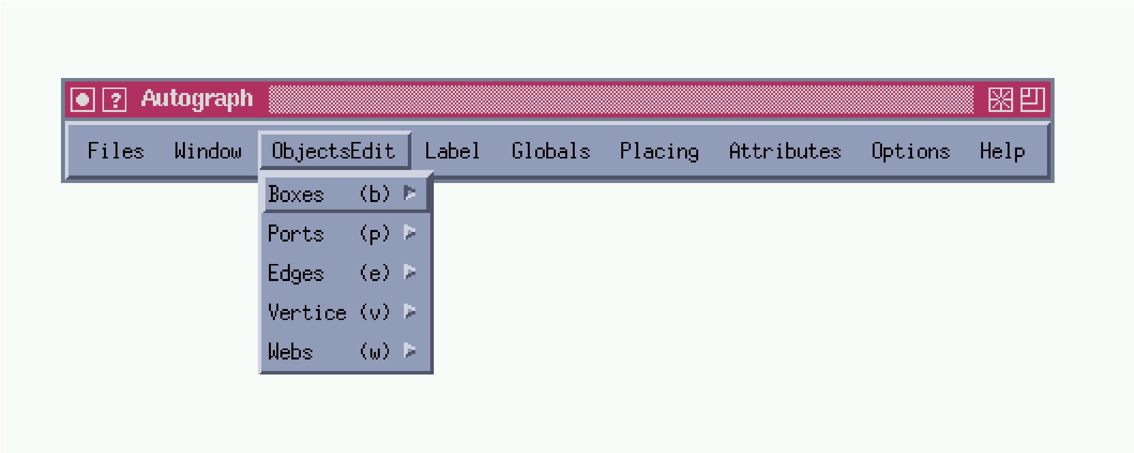

Fig. 10. The menu bar with the `ObjectEdit` menu selected.

A timed automaton is denoted by a box containing the locations and transitions of the automaton with the initial location being denoted by an *initial vertex*. The box should be labeled with the name of the automaton.

A declaration for the variables, clocks and channels, both normal and urgent, used in the network of timed automata are placed in a box which must be labelled with `Config`.

A network of timed automata is represented by a set of set of automata boxes and the `Config` box.

If a graph is created according to these definitions and to the small set of rules that will be introduced in the remainder of this section, it can be used directly in the UPPAAL toolkit.

5.2.2 Getting Started

Autograph can be activated from the `Run` menu of XUPPAAL. On startup the program displays a menu bar which is depicted in Figure 10 with the `ObjectsEdit` menu selected. All menus mentioned in the remainder of this section are parts of this menu bar.

Note that all items inside a menu bar, can also be selected by a snap-key given below in parenthesis, and a selection will be active until another selection is made.

To get started it is necessary to open a window to draw in. This is done by selecting `Create` from the `Window` menu. It is possible to resize and reposition the window.

5.2.3 Creating Locations

Locations of a timed automata are created by choosing `Create` (v) from the `Vertice` sub-menu in the `ObjectsEdit` menu. To instantiate a location just find the wanted position and click the left mouse button to place it.

If the vertex is supposed to model an initial location, select `Initial (i)` from the `Vertice` sub-menu of the `ObjectsEdit` menu and click at the chosen vertex.

To create the name and the invariant of the location, a label must be attached to the vertex. The actual creation of labels will be described in the next section. There is a set of rules concerning location labels that must be followed to avoid errors when using the drawing in UPPAAL:

- Every location must have a name associated.
- All names are specified by the regular expression [a-zA-Z][a-zA-Z0-9_]*. If the location is committed, its name must be prefixed with C: or c:.
- Location names are local to a process and can be reused in other processes.
- An invariant is a conjunction of inequalities. It is written as a list of inequalities separated by commas and surrounded by a pair of parentheses, e.g. `( clock1<=42, clock2<=117 )`.
- The invariant can be put either on the same line as the location name or on a line for itself.

5.2.4 Creating Labels

In Autograph there are four kinds of labels: Struct, Behav, Logic and Hook. The different kinds of labels may have different meanings depending on the kind of component they are attached to.

In UPPAAL the notion of different labels has been avoided as much as possible. Therefore, the default label (automatically selected by Autograph) is always used for components with only one label. To label a component with its default label select `Create/Edit Default (a)` from the `Label` menu, then click on the component to label. This opens an editor window for typing in text for the label.

The only component type with more than one label is the Config box described in Section 5.2.8.

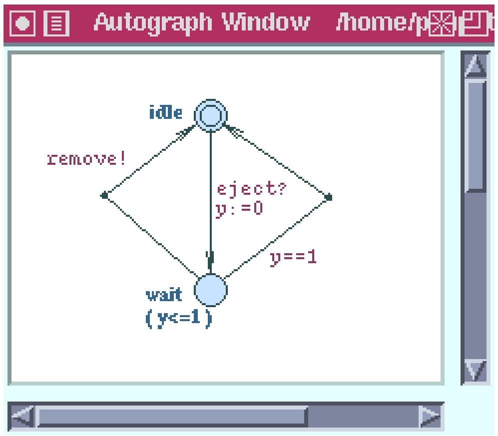

Fig. 11. Locations and transitions of the piston.

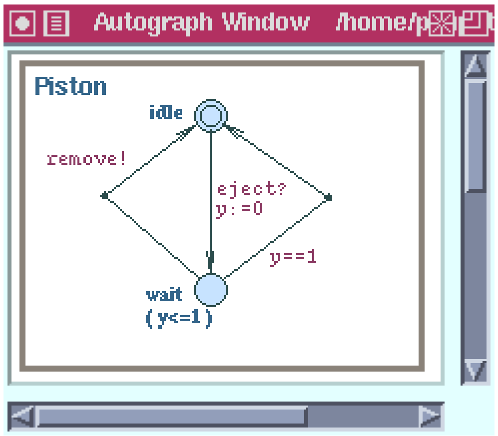

Fig. 12. An automaton defining the piston of the box-sorter unit.

Labels can be re-edited by selecting `Reedit` (`R`) from the `Label` menu. After the label has been clicked an editor for editing (the four kinds of) labels will appear.

5.2.5 Creating Edges

The next step in defining a network of automata is to connect locations with transitions, denoted by edges. Edges can only be drawn between two vertices belonging to the same automaton. The start and end vertices can be the same.

To create an edge, first select `Create` (`e`) from the `Edges` sub-menu under `ObjectsEdit` and then select the start and end vertex. One can create curved edges. The simplest way is to drag the mouse from the start of the edge to a point and then continue to the end of the edge.

The optionals that can belong to a transition are put in a label attached to the edge. An edge can have more than one set of labels; whereas all other components can *only* have one.

As for location labels, there is a set of rules for transition labels:

- If a transition is synchronizing, its label should have a line containing the name of the channel it is synchronizing on.
- Guards are written as lists of comma-separated constraints.
- Assignments are written as lists of comma-separated assignments on the form: `<name> := <expression>`, where expressions follow the syntax defined in Section 3.

Example 3. To start modeling the box-sorter unit we draw the locations and the transitions of the piston automaton. The piston waits for the controller to send an `eject`*-signal. It then ejects within one second, possibly resulting the removal of a box from the belt.*

A model of the piston is shown in Figure 11. The two locations idle *and* wait *are used to model the two operational modes of the piston. Initially the piston is idle. The piston will enter location* wait *when the signal* `eject` *is received. In location* wait *the piston is ready to remove a box (by synchronizing on* `remove`*) for 1 time unit after which the piston will return to the location* idle*. The clock variable* `y` *is used to model the timing behavior of the piston.* □

5.2.6 Creating Automata

An automaton is represented by a box containing its locations and transitions. Boxes are created by choosing `Create` (`b`) from the `Boxes` sub-menu in the `ObjectsEdit` menu. The box is drawn by picking the position of its upper left corner and dragging the mouse until the box has the wanted size (i.e. the position of the lower right corner).

Every automaton must be labeled with a name that is written according to the regular expression [a-zA-Z][a-zA-Z0-9_]*.

Example 4. We now finish the piston automaton by adding a bounding box around its locations and transitions. We name the automaton Piston *by adding a label to the box. The resulting automaton is shown in Figure 12.*

To create an automaton, one can also start with drawing the box and then the vertices, edges etc. □

5.2.7 Creating a Network of Automata

A network of automata is denoted by a collection of automata boxes representing the component automata and a distinct box, labeled with Config, for declarations.

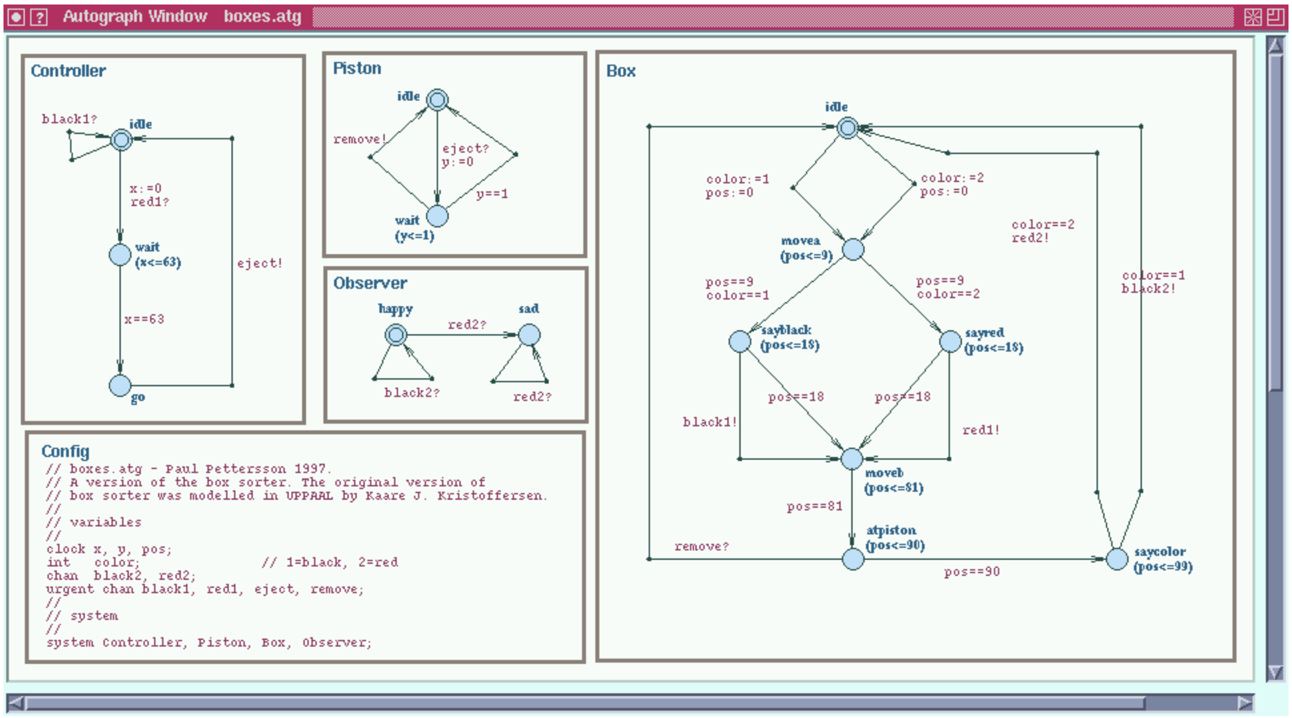

Fig. 13. The complete system description of the box sorter unit.

5.2.8 Creating Declarations

Declarations of objects in the network, i.e. variables, clocks and channels are placed in the Config box. In addition, comments for documentation can be put in this box. Declarations should be created according to the following rules.

- Declarations are written as lists of comma-separated object names preceded by the type of the entities declared and ended with a semi-colon.
- The type of a variable declaration is `int`.
- The type of a clock declaration is `clock`.
- The type of a declaration of normal channels is `chan` and for urgent channels it is `urgent chan`.
- There can be zero or more declarations of each type.
- Names of objects are specified by the regular expression `[a-zA-Z][a-zA-Z0-9_]*`.
- Comments are prepended with `//`. Except for this, there are no rules concerning the syntax of comments.

In addition to the object declarations the Config box also holds the definition of which automata the system consists of. This definition is written using the same syntax as above with the type of the declaration being `system`. Only the automata mentioned in the `system` definition are considered when the system is verified or simulated. There must be exactly one `system` definition.

There are also some rules concerning the Config box and other boxes with comments:

- There must be exactly one Config box.
- The Config box cannot contain any components. It is only allowed to have a label attached.
- Other boxes than the components and Config box are ignored by UPPAAL.

Example 5. The complete system description of the box-sorter unit is shown in Figure 13. It consists of four automata: Piston, Controller, Box, *and* Observer. *In this example we describe the two automata* Box *and* Controller. *The* Piston *automaton was described in Example 4 and the* Observer *will be described in Example 6.*

The Box *automaton models the behavior of a box. It uses the integer variable* `color` *to represent the color of the box (where 1 is black and 2 is red) and the clock variable* `pos` *to represent its position on the belt. In the initial location* idle *is the box not yet placed on the belt (even if* $pos \geq 0$*). On the outgoing transitions from* idle *to* movea *the box is put on the belt and assigned a color. The two locations* sayblack *and* sayred *model the behavior of a box when it passes the color-sensor. It offers the signals* `black1!` *or* `red1!` *to the sensor until position 18 is reached, where it enters location* moveb. *The location* atpiston *models the situation when the box passes the piston, therefore the box offers synchronization on the channel* `remove` *to model the possibility of getting removed. If not removed, the box proceeds to location* saycolor *where it returns to its initial location by synchronizing on channel* `black2` *or* `red2`*, which are synchronized with the observer.*

The Controller *automaton models the controller and its integrated color-sensor. It stays in its initial location* idle

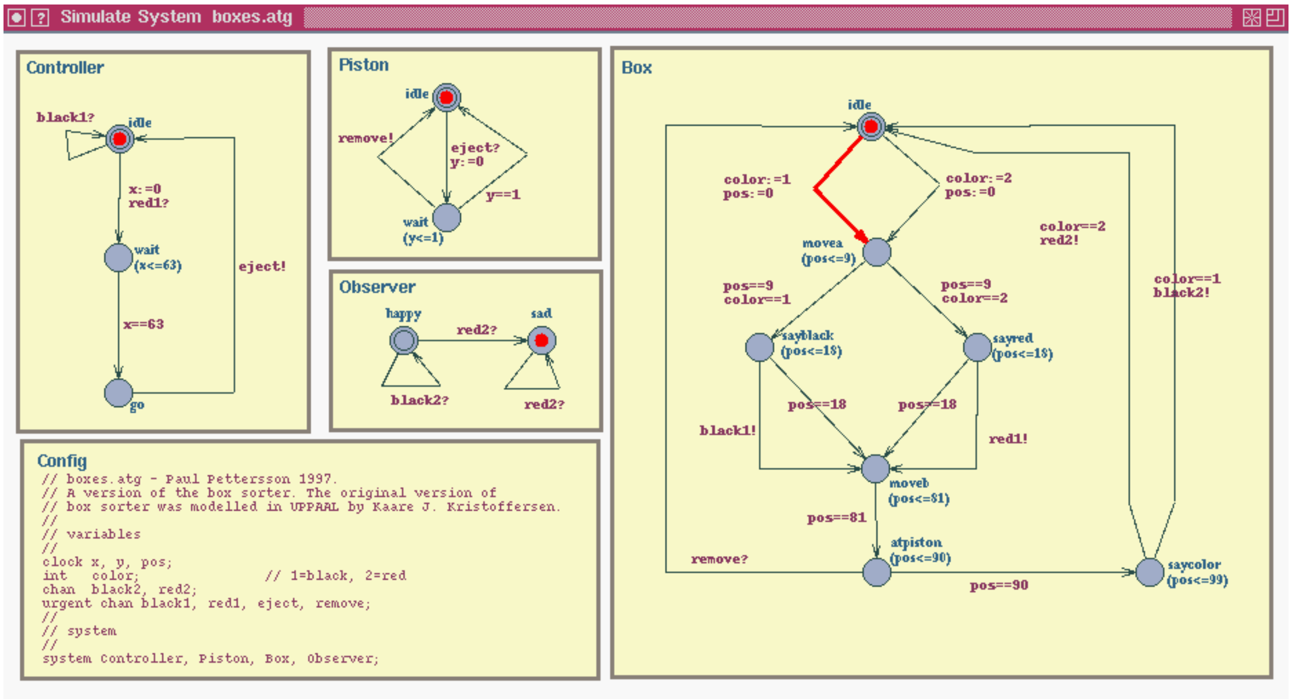

Fig. 14. A System window.

as long as only black boxes appear, modeled by a `black1`*-synchronizing self-loop. When a red box appears, it delays for 63 time units in location* wait *and there after requests a reject from the piston by synchronization on the* `eject`*-channel as soon as possible.*

To model that the color-sensor and the piston react immediately on input, the four channels `black1`, `red1`, `eject`, *and* `remove` *are declared as urgent.* □

5.2.9 Saving the System Description File

To save a file, select either `Save atg` (`s`) or `Save atg as` (`S`) from the `Files` menu and click on the drawing to be saved. This leads to the opening of a file selector window, where the path and name of the file can be entered. When ready press the `OK` button.

The saved `.atg` files can then be used in UPPAAL.

5.3 How to Simulate

The symbolic simulator enables the user to simulate and debug the dynamical executions of a network of timed automata given its statical structure i.e. a system description. This section provides a guide for the interface of the simulator and show how to use it.

5.3.1 Getting Started

The simulator can be activated from the `Run` menu in XUPPAAL. The system description file (an `.atg`-file) specified in the `Model` field of XUPPAAL will be loaded into the simulator on startup.

The graphical interface to the simulator consists of two windows: the System window to show the system description to simulate, and the Main window to control the simulator.

The interface uses the standard X notions for mouse and keyboard control meaning that e.g. double-clicking the mouse and using tab-key to switch between groups in windows can be used.

5.3.2 The System Window

The System window holds the Autograph drawing of the system being simulated. Figure 14 shows an example of a System window containing the definition of the box sorter unit.

During a simulation, the current location vector and one of the currently available transitions are depicted in the window. For the location vector this is accomplished by marking the current location of each of the automata, e.g. in the figure, Box is in location idle. A possible transition is depicted as a highlighted arrow going from the source location of the transition to its destination. If the transition is a synchronization, an arrow will be shown

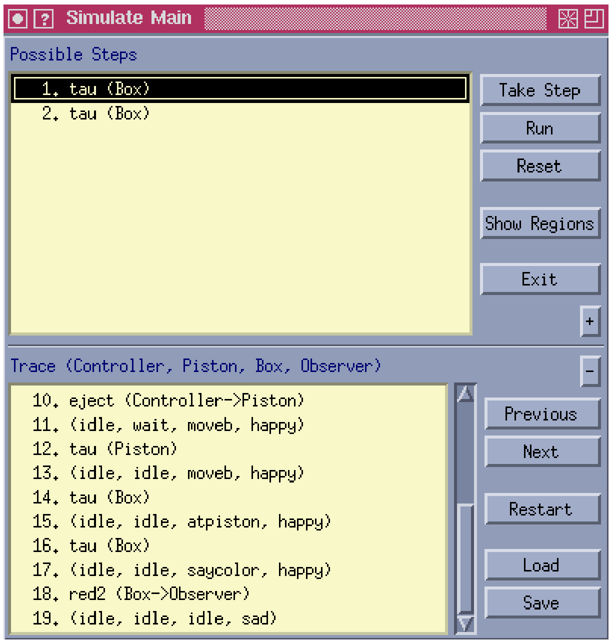

Fig. 15. The Main window.

for each of the two synchronizing transitions, otherwise only one. In the figure a transition from idle to movea in Box is shown.

It is not possible to manipulate the contents of the System window directly as its only purpose is to display the structure of the current system and to show how simulations progress. All changes to the System window are performed indirectly through the Main window described below.

5.3.3 The Main Window

The simulator is controlled from the Main window which is split up into two parts: one containing the basic simulation control, and the other containing the more advanced control mechanisms concerned with traces. Figure 15 shows the Main window with the trace so far and the possible step for the system depicted in Figure 14.

The upper part of the window has six buttons and a field holding a list of steps possible in the current state of the system. The lower part also has six buttons and a field containing the trace of the current simulation. When the simulator is activated only the upper half of this window is opened.

The upper six buttons are used for controlling the most basic functionality of the simulator. The semantics of the buttons is as follows:

`Take Step`: Take the step selected in the list of possible steps. A step is selected if it is highlighted in the list.

`Run`: Open the Run window which is used for controlling the automatic mode, where the simulator itself randomly selects transitions.

`Reset`: Set the automata in the system to their initial locations and clears the trace.

`Show Regions`: Open the Regions window showing the regions valid for the current configuration of the simulated system.

`Exit`: Exit the simulator.

+ (plus): Open the lower half of the Simulate Main window, which displays the trace generated so far.

In the list of possible steps, the elements are written in one of the forms listed below. The first is used if the transition of the step is a synchronization between two processes and the second is used if the transition is non-synchronizing. It is possible to take a step directly from the list by double-clicking it.

– `<channel> (<sender> -> <receiver>)`
– `tau (<process>)`

A trace is a list whose elements alternate between location vectors and transitions. Location vectors are simply written as a list of location names where the names are ordered the same way as the list of automata given in the header of the lower half of the Simulate Main window. The order of the automata in the Simulate Main window shown in Figure 15 is: Controller, Piston, Box and Observer.

Transitions are written in the same form as the possible steps. Selecting an element in the trace changes the System window so that instead of reflecting the current state of the system it will show the selected transition or location.

The six buttons in the lower part of the Main window are all used for controlling the trace functionality. Their semantics is given in the following:

- (minus): Close this half of the window.

`Previous`: Highlight the element immediately preceding the current selection in the trace. If no element is selected, nothing happens.

`Next`: Highlight the element immediately following the current selection in the trace. If no element is selected, nothing happens.

`Restart`: Restart the simulation from the current selection in the trace.

`Load`: Open a file selector window for loading a trace.

`Save`: Open a file selector window for saving the current trace.

5.3.4 The Run Window

Instead of creating traces manually, by taking one step at a time, it is possible to let the simulator proceed automatically by randomly selecting enabled steps and show

Fig. 16. The Run window.

the progress of the simulation in the System window. This can be done from the Run window as shown in Figure 16.

The window shows two text input fields, `Number Of Steps` and `Speed`, plus three buttons. The field `Number Of Steps` decides how many steps the simulator should take when generating the random trace and the `Speed` field decides how fast the screen should be updated during the run, with 1 being the highest speed. The meaning of the three buttons is as follow:

`Close`: Stop the run if it is currently executing, and close the Run window.
`Stop`: Stop the run. If the simulator is not running, nothing happens.
`Go`: Start or restart the execution of a run.

5.3.5 The Regions Window

In doing a simulation the user can watch the changes of the regions in the Regions window. Figure 17 shows the Regions window corresponding to the system state and transition selection of Figures 14 and 15.

The window is split up into four parts each holding one particular kind of region. The regions are represented as sets of equations and inequalities on the variables, clocks and differences between clocks in the system. Inequalities are written as intervals that the clock or clock difference must be in (see the figure).

The upper left quarter of the Regions window shows the *state entry region*. It is the region valid immediately after the last step, i.e. the step that took the system to the current state. The lower left quarter holds the *transition exit region*. For any of the possible steps that can take the system out of the current state, the transition exit region is the region where the system is able to take that particular step.

The *next state entry* region will then be the same as the *transition exit region* of the possible step which is taken next, except that it has been updated by the assignments of the transition.

The constraints of the latest step taken can put additional constraints on the regions for previous steps. This is not taken into account in the calculations of the regions in the left half of the window, but it is considered in the right where the adjusted counterparts of the two above mentioned types of regions are displayed.

5.4 How to Verify

The verification of a system description w.r.t. its requirement specification is conducted entirely from the XUppaal window. It is often the case that the verification will *not* succeed immediately even if the system has been carefully validated in the simulator. More likely, a number of problems, usually in the system description, will have to be resolved before the verification finally succeeds.

In this section we describe how XUppaal supports the work by allowing diagnostic traces, generated by the verification procedure, to be loaded and graphically displayed in the simulator. We begin by describing how to verify in Uppaal, using XUppaal.

5.4.1 Verification

To verify with XUppaal is very simple. When the file names have been specified, the next step is to select the optional settings that affect the verification method, and then it is ready for the start of a verification session. The menu of verification options has been described in Section 5.1.3.

The verifier (called `verifyta`) is activated by clicking `Verification` in the `Run` menu. XUppaal then first transforms the options, selected in the `Options` menu, to a list of flags to be accepted by the verifier. It then proceeds by compiling the graphical system description to its textual format. Finally, it spawns a child process running the verifier with the required parameters, i.e.: the list of flags, the system description file, the requirement specification and (optionally) the specified trace file.

All output[8] produced in the child process is displayed in the output browser of XUppaal. In particular, answers to whether each property of the requirement specification is satisfied or not are displayed. Each answer is numbered in order of appearance in the requirement specification file and with line numbers in parenthesis (see Figure 8). If an answer line is double-clicked in the output browser of XUppaal, the requirement specification editor (if open) will display the corresponding line of the requirement specification file in its output browser (labeled `Req. Spec. File`). This is useful when working with large specifications.

[8] The output which normally appears on standard output and standard error (i.e. stdout and stderr).

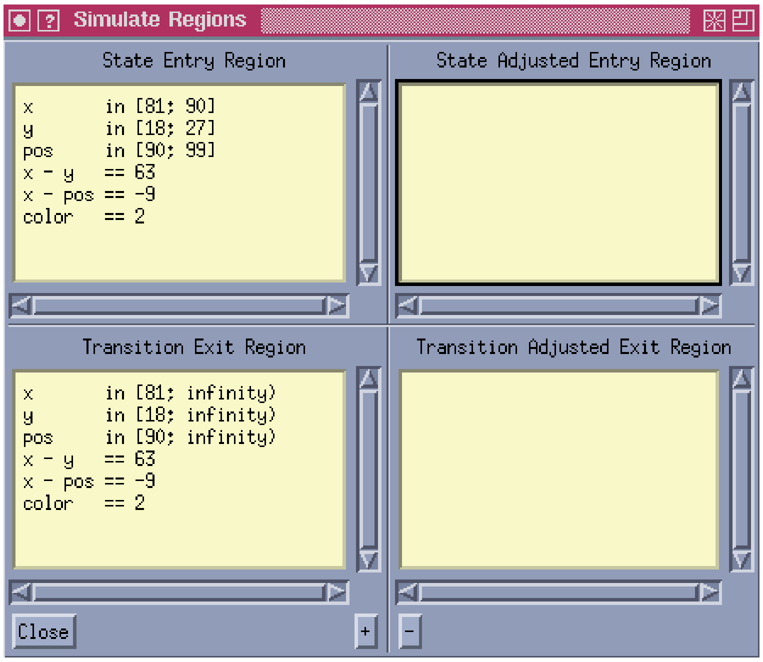

Fig. 17. The Regions window.

Example 6 (Verifying the Box-Sorter). *Now we show how to verify the correctness of the box-sorter unit. Since the sorter is considered correct if the observer sees no red boxes, we formalize the correctness in terms of reachable locations of the observer. We begin by describing the automaton* Observer *which models the observer.*

The Observer *automaton, which can be found in Figure 13, has two locations:* happy *and* sad*. It is initially in location* happy *and stays there as long as it sees only black boxes at the rightmost extreme of the belt. If a red box appears, it enters the location* sad *where it stays forever.*

To prove the box-sorter correct, it suffices to show that it is impossible for the observer to reach location sad*, i.e.* $\forall\Box\neg$ Observer.sad*. We phrase the property in the* UPPAAL *syntax,* `A[]not Observer.sad`*, and put the resulting formula in the requirement specification file, named* `boxes.q`*. This can be done using the requirement specification editor which is activated in the* `Run` *menu of* XUPPAAL.

Finally, we start the actual verification (`Verification` *in the* `Run` *menu). The verifier outputs the text lines 13 and 14 in the output browser of Figure 8, telling that the property is not satisfied.* □

5.4.2 Visualizing Diagnostic Traces

The simulator generates traces and allows for traces to be displayed, reexamined, replayed and reset from any intermediate point. Traces may also be reused using the save and load facilities. In particular, diagnostic traces generated by the verifier may be loaded for examination.

To use this facility, the first step is to specify the trace file name (see Section 5.1.1), and then instruct the verifier to generate traces on file. This is done by selecting item `Diagnostic File` in the `Options` menu of XUPPAAL. As a result, all succeeding verifications will output traces in the specified file until the `Diagnostic File` option is toggled off.

It should be noticed that it is possible to generate a *shortest* diagnostic trace, simply by instructing the verifier to explore the state space by breath-first search.

When the verification is finished, the trace file is loaded in the simulator by clicking the Load button in the Main window of the simulator (see Section 5.3.3).

Example 7 (Correcting the Box-Sorter). *To find out why the requirement specification is not satisfied by the system description we generate a diagnostic trace. As an exercise, we first produce a textual representation of a shortest diagnostic trace. To do so, we first select the options* `Diagnostic Info` *and* `Breadth-First` *in the* `Options` *menu and then initiate a new verification. The resulting trace is shown in Figure 18*[9]*.*

We can also try to debug graphically using the simulator. Recall that we have specified the trace file name in Example 2. A trace will be produced in that file if we select the option `Diagnostic File` *and then verify the system again. Next, we load the trace into the simulator using button* Load *in the main window of the simulator.*

[9] Normally, the trace would have been displayed in the output browser of XUPPAAL.

```
(Controller.idle Piston.idle Box.idle Observer.happy )
{ x=0 y=0 pos=0 color=0 }

(Controller.idle Piston.idle Box.movea Observer.happy )
{ x=0 y=0 pos=0 color=2 }

delay(9)

(Controller.idle Piston.idle Box.movea Observer.happy )
{x=9 y=9 pos=9 color=2 }

(Controller.idle Piston.idle Box.sayred Observer.happy )
{ x=9 y=9 pos=9 color=2 }

 Urgent sync: red1

(Controller.wait Piston.idle Box.moveb Observer.happy )
{ x=0 y=9 pos=9 color=2 }

delay(63)

(Controller.wait Piston.idle Box.moveb Observer.happy )
{x=63 y=72 pos=72 color=2 }

(Controller.go Piston.idle Box.moveb Observer.happy )
{ x=63 y=72 pos=72 color=2 }

 Urgent sync: eject

(Controller.idle Piston.wait Box.moveb Observer.happy )
{ x=63 y=0 pos=72 color=2 }

delay(1)

(Controller.idle Piston.wait Box.moveb Observer.happy )
{x=64 y=1 pos=73 color=2 }

(Controller.idle Piston.idle Box.moveb Observer.happy )
{ x=64 y=1 pos=73 color=2 }

delay(8)

(Controller.idle Piston.idle Box.moveb Observer.happy )
{x=72 y=9 pos=81 color=2 }

(Controller.idle Piston.idle Box.atpiston Observer.happy )
{ x=72 y=9 pos=81 color=2 }

delay(9)

(Controller.idle Piston.idle Box.atpiston Observer.happy )
{x=81 y=18 pos=90 color=2 }

(Controller.idle Piston.idle Box.saycolor Observer.happy )
{ x=81 y=18 pos=90 color=2 }

Sync: red2

(Controller.idle Piston.idle Box.idle Observer.sad )
{ x=81 y=18 pos=90 color=2 }
```

Fig. 18. A diagnostic trace.

The windows of the simulator now look exactly like in Figure 14, Figure 15 and Figure 17.

It can be deduced that there is an error in the Controller *automaton. After the controller has sensed a red box, it will miss the box if it waits too short time before ejecting the piston. If it waits another 9 time units before the* `eject` *signal is sent, the piston will hit the box.*

The error can be corrected simply by modifying the time bound from 63 to 72 in the invariant and the guard on the edge outgoing from location wait *in the* Controller*. It can be verified that the modified system is correct.* □

6 Case Studies

UPPAAL has been applied in a number of (industrial) case-studies. In this section we briefly review some of them.

Audio/Video Protocol: This is an audio control protocol highly dependent on real-time. The protocol is developed by Bang & Olufsen, to transmit messages between audio/video components over a single bus, and further studied in [1]. Though it was known to be faulty, the error was not found using conventional testing methods. Using UPPAAL, an error-trace is automatically produced, which reviled the error. Furthermore, a correction is suggested and automatically proved using UPPAAL.

Bounded Retransmission Protocol: The protocol is proposed and studied at *COST 247, International Workshop on Applied Formal Methods in System Design*. It is based on the alternating bit protocol over a lossy communication channel, but allows for a bounded number of retransmissions. In [13, 14], it is reported that a number of properties of the protocol is automatically checked with UPPAAL. In particular, it is shown that the correctness of the protocol is dependent on correctly chosen time-out values.

Collision Avoidance Protocol: The protocol in [20] is implemented on top of an Ethernet-like medium such as the CSMA/CD protocol. It is to ensure an upper bound on the communication delay between the network nodes. It was designed and proved correct using UPPAAL. The two main properties established show that the protocol is collision-free, and it does ensure an upper bound on the user-to-user communication delay (assuming a perfect medium).

Gear-Box Controller: In this industrial case-study, UPPAAL is applied to the design and analysis of a prototype gear-box controller for vehicles by Mecel AB[10] [28]. The gear-box controller is a component in the real-time distributed system that controls a modern car. The gear-requests from the driver are delivered via the man/machine interface over a communication network to the gear-box controller. The controller im-

[10] Mecel AB is a Swedish company developing control systems for vehicle industries.

plements the actual gear change by actuating the lower level components of the system (such as the clutch, the engine and the gear-box).

In the design of the controller, the symbolic simulator of UPPAAL is applied to validate the system behavior. The correctness of the gear-box controller design is established by automatic proofs of 46 properties derived from requirements specified by Mecel AB.

Philips Audio Protocol: The protocol is developed and implemented by Philips to exchange control information between components in audio equipment using Manchester encoding. The correctness of the encoding relies on timing delays between signals. It is firstly studied and manually verified in [10]. In [25] the protocol is modeled and verified using UPPAAL.

Philips Audio Protocol with Bus Collision: This is an extended variant of Philips audio control protocol with bus collision detection. It is significantly larger than the version above since several new components (and variables) are introduced, and existing components are modified to deal with bus collisions. Its correctness is originally proved by hand in [18], and by model-checking for the first time using UPPAAL in [6].

Mutual Exclusion Protocol: The so-called Fischers protocol has been studied previously in many experiments, e.g. [2, 32]. The protocol is to ensure mutual exclusion among several processes competing for a critical section using timing constraints and a shared variable. Different versions of the protocol have been verified using UPPAAL, c.f. [34, 24, 22].

TDMA Protocol Start-Up Mechanism: In [29], a formal verification of the start-up algorithm of a TDMA (Time Division Multiple Access) protocol is reported. It was proved using UPPAAL that an ensemble of three communicating stations becomes synchronized and operational within a bounded time from an arbitrary initial state, given a clock-drift corresponding to $\pm 10^{-3}$. Furthermore, an upper time-bound for the start-up to complete was derived.

7 Conclusions

In this paper, we have presented the overall structure, the design criteria, and the main features of UPPAAL. It is intended to be an informal but reasonably "complete" description of the syntax and semantics of the UPPAAL modelling and specification languages as well as the kernel of its model-checking algorithm. It contains neither definitions nor theorems, but mainly examples. The intended readers are engineers working with development and analysis of embedded systems in particular, real-time protocols and controllers. A user guide is given, which describes in details how to use the various tools of UPPAAL to *construct* abstract models of a system, to *simulate* its dynamical behavior, to *specify* and *verify* its safety and bounded liveness properties in terms of its model. In addition, we also provide a short review on case-studies where UPPAAL is applied, as well as references to its theoretical foundation.

However the document is by no means complete as UPPAAL is still under development. In fact, during the time when this paper is written, a new version of UPPAAL has been implemented, which is currently under internal test and evaluation. Future versions of UPPAAL will extend the current description language with data types such as arrays, records, lists etc. as well as various primitive operations on data types. In the near future, a new graphical interface will be available, which provides graphical supports for not only system descriptions at process level as the current version does but also hierarchical design and modularization. But that is all for now.

Acknowledgements. The authors wish to thank the people involved in the development of UPPAAL, including: Johan Bengtsson, Palle Christensen, Jesper Jensen, Per Jensen, Kåre Kristoffersen, Fredrik Larsson, and Thomas Sørensen.

References

1. *Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using* UPPAAL, 1997.
2. Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, 1993.
3. R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.
4. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
5. Johan Bengtsson, Palle Christensen, Per Jensen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Thomas Sørensen, and Wang Yi. UPPAAL*: a Tool Suite for Validation and Verification of Real-Time Systems.* `http://www.docs.uu.se/rtmv/uppaal/-uppaal-guide.ps.gz`, 1996.
6. Johan Bengtsson, David Griffioen, Kåre Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.
7. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer–Verlag, October 1995.
8. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the 2nd Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture

Notes in Computer Science, pages 431–434. Springer–Verlag, Mars 1996.

9. Johan Bengtsson and Fredrik Larsson. UPPAAL a Tool for Automatic Verification of Real-time Systems. Master's thesis, Uppsala University, 1996.
10. D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, 1994.
11. Amar Bouali, Annie Ressouche, Valérie Roy, and Robert de Simone. The FC2TOOLS set. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of 8*th *Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 441–445. Springer–Verlag, 1996.
12. Palle Christensen and Thomas Mark Sørensen. void. Master's thesis, Aalborg University, 1997.
13. P.R. D'Argenio, J.-P., Katoen, T. Ruys, and J. Tretmans. Modeling and Verifying a Bounded Retransmission Protocol. In *Proc. of COST 247, International Workshop on Applied Formal Methods in System Design*, 1996. Also available as Technical Report CTIT 96-22, University of Twente, July 1996.
14. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proc. of the 3*rd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science, pages 416–431. Springer–Verlag, April 1997.
15. C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 66–75, December 1995.
16. David Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 197–212. Springer–Verlag, 1989.
17. David Dill. The Murφ Verification System. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of 8*th *Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 390–393. Springer–Verlag, 1996.
18. W.O.D. Griffioen. Analysis of an Audio Control Protocol with Bus Collision. Master's thesis, University of Amsterdam, Programming Research Group, 1994.
19. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: The Next Generation. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 56–65, December 1995.
20. H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *Proc. of 2*nd *International Workshop on the SPIN Verification System*, pages 1–20, August 1996.
21. Jesper Gravgaard Jensen and Per Stoffer Jensen. Design and Implementation of NewPaal — a modelling language for real-time systems. Master's thesis, Aalborg University, 1997.
22. Kåre J. Kristoffersen, Francois Larroussinie, Kim G. Larsen, Paul Pettersson, and Wang Yi. A compositional proof of a real-time mutual exclusion protocol. In *Proc. of the 7*th *International Joint Conference on the Theory and Practice of Software Development*, April 1997.
23. Kim G. Larsen and Hans Hüttel. UPPAAL — An Automatic Tool for Verification of Real Time and Hybrid Systems. Seminar slides from *Livslang Uddannelse 96*, 1996. Email: `{kgl,hans}@cs.auc.dk`.
24. Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87, December 1995.
25. Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 575–586. Springer–Verlag, October 1995.
26. Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88, August 1995.
27. Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *In Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24, December 1997.
28. Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller: an Industrial Case Study using UPPAAL. Technical Report ASTEC 09/97, Dept. of Computer Systems, Uppsala University, August 1997. Available at `http://www.docs.uu.se/docs/rtmv/papers/-lpw-astec97.ps.gz`.
29. Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. Accepted for presentation at the *Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1997.
30. R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, 1989.
31. A. Olivero, J. Sifakis, and S. Yovine. Using Abstractions for the Verification of Linear Hybrids Systems. In *Proc. of 7*th *Int. Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, 1994.
32. N. Shankar. Verification of Real-Time Systems Using PVS. In *Proc. of 5*th *Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer–Verlag, 1993.
33. Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real–time transition systems. In *Proc. of 5*th *Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 210–224, 1993.
34. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7*th *International Conference on Formal Description Techniques*, 1994.

Appendix A: BNF for ta-format

```
Ita          →  VarList ProcList Globals
VarList      →  ε | Channel VarList | Var VarList
ProcList     →  Proc | Proc ProcList
Globals      →  system IdList ;
Channel      →  urgent chan IdList ;
                | chan IdList ;
Var          →  Type IdList ;
Proc         →  process Id { ProcBody }
IdList       →  Id | Id , IdList
ProcBody     →  StateDecls TransDecls
StateDecls   →  state IdList ; commit IdList ; init Id ;
                | state IdList ; init Id ;
Transdecls   →  trans TransList ;
TransList    →  Trans | Trans , TransList
Trans        →  Id SInv -> Id { OpG OpS OpA }
                | -> Id { OpG OpS OpA }
SInv         →  ε | { InvList }
InvList      →  Inv | Inv , InvList
Inv          →  Id <= Nat | Id < Nat
OpG          →  ε | guard GuardList ;
OpS          →  ε | Id! | Id?
OpA          →  ε | assign AssignList ;
GuardList    →  Guard | Guard , GuardList
AssignList   →  Assign | Assign , AssignList
Type         →  clock | int
Assign       →  ClockAssign | IntAssign
Guard        →  Id RelOp Nat | Id RelOp Id Op Nat
ClockAssign  →  Id := Nat
IntAssign    →  Id := IntExpr
IntExpr      →  Int * Id Op Nat | Id Op Nat | Id | Int
RelOp        →  < | <= | >= | > | ==
Op           →  + | -
Id           →  Alpha | Id AlphaNum
Nat          →  Num | Num Nat
Int          →  Nat | -Nat
Alpha        →  A | ... | Z | a | ... | z
Num          →  0 | ... | 9
AlphaNum     →  Alpha | Num | _
```

Appendix B: BNF for q-format

```
Prop         →  E<> StateProp | A[] StateProp
StateProp    →  AtomicProp | ( StateProp )
                | not StateProp
                | StateProp or StateProp
                | StateProp and StateProp
                | StateProp imply StateProp
AtomicProp   →  Id.Id | Id RelOp Nat
                | Id RelOp Id Op Nat
RelOp        →  < | <= | >= | > | ==
Op           →  + | -
Id           →  Alpha | Id AlphaNum
Nat          →  Num | Num Nat
Alpha        →  A | ... | Z | a | ... | z
Num          →  0 | ... | 9
AlphaNum     →  Alpha | Num | _
```