BRICS

Basic Research in Computer Science

UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems

Johan Bengtsson
Kim G. Larsen
Fredrik Larsson
Paul Pettersson
Wang Yi

BRICS Report Series **RS-96-58**

ISSN 0909-0878 **December 1996**

See back inner page for a list of recent publications in the BRICS
Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through World Wide
Web and anonymous FTP:

`http://www.brics.dk/`
`ftp://ftp.brics.dk/`
This document in subdirectory `RS/96/58/`

UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems *

Johan Bengtsson[2] *Kim Larsen*[1]
Fredrik Larsson[2] *Paul Pettersson*[2] *Wang Yi*[**2]

[1] BRICS*** , Aalborg University, DENMARK
[2] Department of Computer Systems, Uppsala University, SWEDEN

Abstract. UPPAAL is a tool suite for automatic verification of safety and bounded liveness properties of real-time systems modeled as networks of timed automata. It includes: a *graphical interface* that supports graphical and textual representations of networks of timed automata, and automatic transformation from graphical representations to textual format, a *compiler* that transforms a certain class of linear hybrid systems to networks of timed automata, and a *model–checker* which is implemented based on constraint–solving techniques. UPPAAL also supports diagnostic model-checking providing diagnostic information in case verification of a particular real-time systems fails.
The current version of UPPAAL is available on the World Wide Web via the UPPAAL home page `http://www.docs.uu.se/docs/rtmv/uppaal`.

1 Introduction

UPPAAL is a new tool suite for automatic verification of safety and bounded liveness properties of networks of timed automata [13, 8, 6]. The tool was developed during the spring of 1995 as the result of intense research collaboration between BRICS at Aalborg University and Department of Computing Systems at Uppsala University. The two main design critea for UPPAAL has been *efficiency* and *ease of usage*.

The current version of UPPAAL, as well as its future extensions, is implemented in C++. Model–checking is often hampered by various state–explosion problems. In UPPAAL thes problems are dealt with by a combination of on–the–fly verification together with a new and coarser symbolic technique reducing the verification problem to that of solving simple linear constraint systems. The features and tools of UPPAAL includes:

* This work has been supported by the European Communieties (under CONCUR2 and REACT), NUTEK (Swedish Board for Technical Development) and TFR (Swedish Technical Research Council)

** This author would also like to thank the Chinese NSF and the Hong Kong Wang's Foundation for supporting a visit to the Institute of Software, Chinese Academy of Sciences, in 1995.

*** **B**asic **R**esearch in **C**omputer **S**cience, Centre of the Danish National Research Foundation.

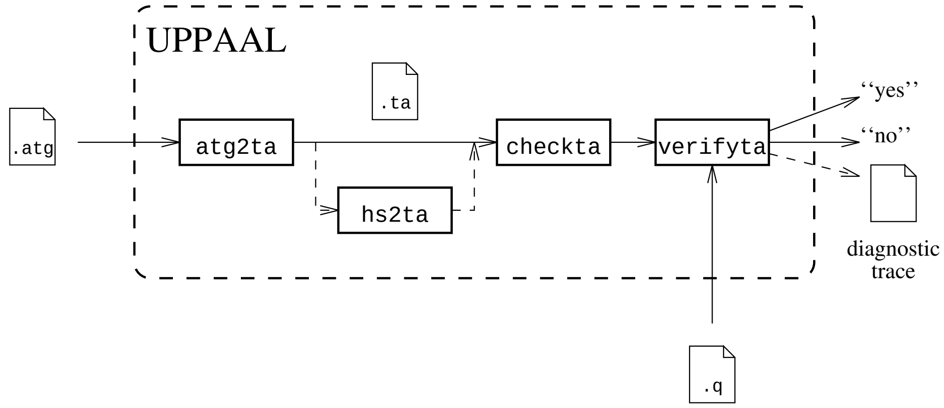

Fig. 1. Overview of UPPAAL

- A graphical interface based on Autograph.
- An automatic compilation of the graphical definition into a textual format.
- Analysis of certain types of hybrid automata by compilation into ordinary timed automata. In particular UPPAAL allows automata with varying and drifting time–speed of clocks.
- A number of simple, but in practice extremely useful syntactical checks are made before verification can commence.
- Generation of diagnostic traces in case verification of a particular real–time system fails.

In this paper we present the various features of UPPAAL, review and provide pointers to the theoretical foundation as well as applications to various case–studies.

2 An Overview of UPPAAL

UPPAAL consists of a suite of tools for verifying safety properties of real-time system. An overview of the system is shown in Figure 1. In this section we briefly describe the main features of UPPAAL.

2.1 Graphical Description of Networks of Timed Automata

It is possible to draw networks of timed automata using Autograph, given that certain syntactical rules are followed, e.g. the different automata in the network must be enclosed in boxes with the name of the process in the structural label, there must be a textual box describing the system configuration, i.e. declaration of clocks, channels and auxiliary integer variables. To be able

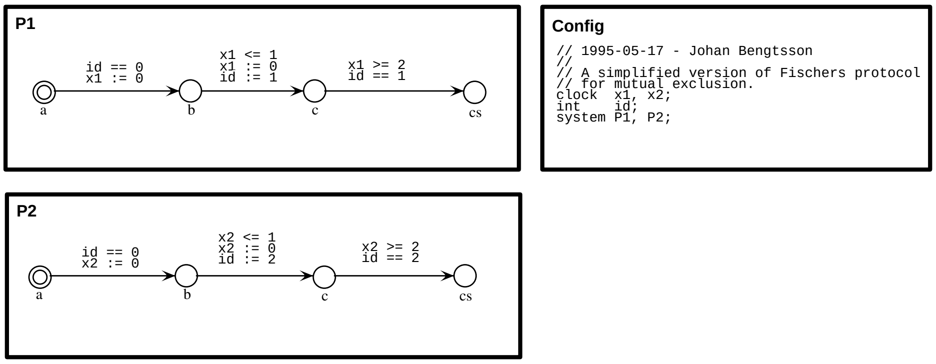

Fig. 2. Graphical Description of Fischers Mutual Exclusion Protocol

to import system descriptions, drawn with help of Autograph, into Uppaal the system must be saved in the Autograph `.atg`-format. In Figure 2 the Autograph version of Fischers Protocol [1, 10] is shown.

2.2 Textual Description of Networks of Timed Automata

In addition, Uppaal allows networks of timed automata to be described using a textual format (called `.ta`) providing a basic *programming language for timed automata*. In certain cases we found this textual format 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[4]. Figure 3 shows the resulting `.ta`–format for Fischers Protocol from Figure 2.

2.3 Linear Hybrid Systems

Under certain conditions, the model of timed automata may be generalized to allow clocks with rates varying between a lower and an upper bound, and to allow clock rates to change between different control-nodes (vertices) [9]. This extension of timed automata is useful for modelling of hybrid systems where the behaviour 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 speed of clocks is given by an interval. Hybrid automata of this form may be transformed into ordinary timed automata using the translator `hs2ta`. Philips Audio-Control Protocol of [3] is one such linear hybrid system and for its Autograph version is shown in Figure 5.

[4] What You See Is What You Verify.

```
//
// Declarations
//
clock x1, x2;
int   id;

//
// Processes
//
process P1 –
   state a, b, c, cs;
   init  a;
   trans a -¿ b –
           guard  id == 0;
           assign x1 := 0;
         ”,
         b -¿ c –
           guard  x1 ¡= 1;
           assign x1 := 0, id := 1;
         ”,
         c -¿ cs –
           guard  x1 ¿= 2, id == 1;
         ”;
”

process P2 –
   state cs,c,b,a;
   init  a;
   trans c -¿ cs –
           guard  x2 ¿= 2, id == 2;
         ”,
         b -¿ c –
           guard  x2 ¡= 1;
           assign x2 := 0, id := 2;
         ”,
         a -¿ b –
           guard  id == 0;
           assign x2 := 0;
         ”;
”

//
// System Configuration
//
system P1,P2;
```

Fig. 3. Textual Description of Fischers Mutual Exclusion Protocol

2.4 Syntactical Checks

Given a textual description of a timed automata in the `.ta`-format the program `checkta` performs a number of syntactical checks. In particular the use of clocks, auxiliary integer variables and channels must be in accordance with their declaration, e.g. attempted synchronization on an undeclared channel will be captured by `checkta`.

2.5 Model–Checking

In the current version UPPAAL is able to check for reachability properties, in particular whether certain combinations of control-nodes and constraints on clocks and integer variables are reachable from an initial configuration. The desired mutual exclusion property of Fischers protocol (Figure 2 and Figure 3) falls into this class. Bounded liveness properties can be obtained by reasoning about the system in the context of testing automata. The model-checking is performed by the module **`verifyta`** which takes as input a network of timed automata in the `.ta`-format and a formula. **`verifyta`** can also be used interactively. In case verification of a particular real-time system fails (which happens more often than not), a *diagnostic trace* is automatically reported by **`verifyta`** [7]. Such a trace may be considered as diagnostic information of the error, useful during the subsequent debugging of the system. This principle could be called WYDVYAE[5].

3 The UPPAAL Model

In this section, we present the syntax and semantics of the model used in UPPAAL to model real–time systems. The emphasis will be put on the precise semantics of the model. For convenience, we shall use a slightly different syntax compared with UPPAAL's user interface.

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

Alur and Dill developed the theory of timed automata [2], as an extension of classical finite–state automata with clock variables. To have a more expressive model and to ease the modelling 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 modelling (or design) language which is as close as possible to a high–level real–time programming language. Clearly this will create problems for decidability. 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. The current implementation of UPPAAL allows integer variables in addition to clock variables.

In a finite–state automaton, a transition takes the form $l \xrightarrow{\alpha} l'$ meaning that the process modelled by the automaton will perform an α–transition in state l and reach state l' in doing so. Note that there is no condition on the transition. Alur and Dill [2] extend the untimed transition to the timed version: $l \xrightarrow{g,a,\phi} l'$ where g is a simple linear constraint over the clock variables and ϕ is a set of clocks to be reset to zero. Intuitively, $l \xrightarrow{g,a,\phi} l'$ means that a process in control node l may perform the α-transition instantaneously when g is true of the current clock values and then reach control node l' with the clocks in ϕ being reset. The constraint g is called a *guard*. In UPPAAL, we allow a more general form of guard that can also be a constraint over data variables, and extend the reset–operation on clocks in timed automata to data variables.

Now assume a finite set of clock variables C ranged over by x, y, z etc and a finite set of data variables V ranged over by i, j, k etc.

Guard over Clock and Data Variables We use $G(C, V)$ to stand for the set of formulas ranged over by g, generated by the following syntax: $g ::= \ a \mid g \wedge g$,

[5] What You Don't Verify You Are Explained.

where a is a constraint in the form: $x \sim n$ or $i \sim n$ for $x \in C$, $i \in V$, $\sim \in \{\leq, \geq, =\}$ and n being a natural number. We shall call $G(C, V)$ *guards*. Note that a guard can be divided into two parts: a conjunction of constraints g_c's in the form $x \sim n$ over clock variables and a conjunction of constraints g_v's in the form $i \sim n$ over data variables. We shall use $\mathtt{tt}$ to stand for a guard like $x \geq 0$ which is always true, for a clock variable x as clocks can only have non-negative values. In UPPAAL's representation of automata, the guard $\mathtt{tt}$ is often omitted.

Reset–Operations To manipulate clock and data variables, we use reset–set in the form: $\overline{w} := \overline{e}$ which is a set of assignment–operations in the form $w := e$ where w is a clock or data variable and e is an expression. We use R to denote the set of all possible reset–operations.

The current version of UPPAAL distinguishes clock variables and data variables: a reset–operation on a clock variable should be in the form $x := n$ where n is a natural number and a reset–operation on an integer variable should be in the form: $i := c * i + c'$ where c, c' are integer constants. Note that c, c' can be negative.

Channel, Urgent Channel and Syncronization We assume that processes synchronize with each other via channels. Let A be a set of channel names and out of A, there is a subset U of urgent channels on which processes should synchronize that whenever possible. We use $\mathcal{A} = \{\alpha? | \alpha \in A\} \cup \{\alpha! | \alpha \in A\}$ to denote the set of actions that processes can perform to synchronize with each other. We use name(a) to denote the channel name of a, defined by name$(\alpha?) =$ name$(\alpha!) = \alpha$.

Automata with clock and data variables Now we present an extended version of timed automata with data variables and reset–operations.

DEFINITION 1. *An automaton A over actions $\mathcal{A}$, clock variables C and data variables V is a tuple $\langle N, l_0, E \rangle$ where N is a finite set of nodes (control-nodes), l_0 is the initial node, and $E \subseteq N \times G(C, V) \times \mathcal{A} \times 2^R \times N$ corresponds to the set of edges. To model urgency, we require that the guard of an edge with an urgent action should always be $\mathtt{tt}$, i.e. if name$(a) \in U$ and $\langle l, g, a, r, l' \rangle \in E$ then $g \equiv \mathtt{tt}$. In the case, $\langle l, g, a, r, l' \rangle \in E$ we shall write, $l \xrightarrow{g,a,r} l'$ which represents a transition from the node l to the node l' with guard g (also called the enabling condition of the edge), action a to be performed and a set of reset–operations r to update the variables.* □

Concurrency and Synchronization To model networks of processes, we introduce a CCS–like parallel composition operator for automata. Assume that $A_1 ... A_n$ are automata with clocks and data variables. We use $\overline{A}$ to denote their parallel composition. The intuitive meaning of $\overline{A}$ is similar to the CCS parallel composition of $A_1 ... A_n$ with *all* actions being restricted, that is,

$$(A_1 | ... | A_n) \backslash A$$

Thus only synchronization between the components A_i is possible. We shall call $\overline{A}$ a network of automata. We simply view $\overline{A}$ as a vector and use A_i to denote its ith component.

3.2 Semantics

Informally, a process modelled by an automaton starts at node l_0 with all its clocks initialized to 0. The values of the clocks increase synchronously with time at node l. At any time, the process can change node by following an edge $l \xrightarrow{g,a,r} l'$ provided the current values of the clocks satisfy the enabling condition g. With this transition, the variables are updated by r.

Variable Assignment Now, we introduce the notion of a *variable assignment*. A variable assignment is a mapping which maps clock variables C to the non-negative reals and data variables V to integers. For a variable assignment v and a delay d, $v \oplus d$ denotes the variable assignement such that $(v \oplus d)(x) = v(x) + d$ for any clock variable x and $(v \oplus d)(i) = v(i)$ for any integer variable i. This definition of $\oplus$ reflects that all clocks operate with the same speed and that data variables are time–insensitive. For a reset-operation r (a set of assignment–operations), we use $r(v)$ to denote the variable assignment v' with $v'(w) = \mathrm{val}(e, v)$ whenever $w := e \in r$ and $v'(w') = v(w')$ otherwise, where $\mathrm{val}(e, v)$ denotes the value of e in v. Given a guard $g \in G(C, V)$ and a variable assignment v, $g(v)$ is a boolean value describing whether g is satisfied by v or not.

Control Vector and Configuration A *control vector* $\overline{l}$ of a network $\overline{A}$ is a vector of nodes where l_i is a node of A_i. We shall write $\overline{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\overline{l}$ is replaced by l_i'.

A *state* of a network $\overline{A}$ is a configuration $\langle \overline{l}, v \rangle$ where $\overline{l}$ is a control vector of $\overline{A}$ and v is a variable assignment. The initial state of $\overline{A}$ is $\langle \overline{l}_0, v_0 \rangle$ where $\overline{l}_0$ is the initial control vector whose elements are the initial nodes of A_i's and v_0 is the initial variables assignment that maps all variables to 0.

Maximal Delay To model progress properties, we need a notion of maximal delay. Let $\langle l, v \rangle$ be a configuration of an automaton A. Note that A in location l may have a number of outgoing transitions with guards. The process modelled by A in state $\langle l, v \rangle$ may have to wait for the guards to become true, which enables the transitions. However, we do not want the process to stay forever in the same control–node, i.e. l; in other words, some discrete transition must be taken within a certain time bound. We require that the bound should be the maximal delay before all the guards are completely closed, that is, they will never become true again. This is formalized as follows:

Definition 2. *(Maximal Delay for Automata)*

$$MD(l, v) = \max\{d \mid l \xrightarrow{g,a,r} l' \text{ and } g(v \oplus d)\} \qquad \square$$

Note that $\max\{\} = 0$. This will be the case when all the guards for outgoing transitions in l have already been closed in state $\langle l, v\rangle$ or in other words, the process reaches a time–stop process, which means that A is physically unrealizable. Now we extend the notion of maximal delay to networks of automata, which insures that synchronization on urgent channels happens immediately.

Definition 3. *(Maximal Delay for Networks of Automata)*

$$MD(\bar{l}, v) = \begin{cases} 0 & \textit{if } \exists \alpha \in U, l_i, l_j \in \bar{l} : l_i \xrightarrow{\alpha?, r_i} \ \& \ l_j \xrightarrow{\alpha!, r_j} \\ \min\{MD(l, v) \mid l \in \bar{l}\} & \textit{otherwise} \end{cases}$$

□

Transition Rules The semantics of a network of automata $\overline{A}$ is given in terms of a transition system with the set of states being the set of configurations and the transition relation defined as follows:

Definition 4. *(Transition Rules for Networks of Automata)*

- $\langle \bar{l}, v\rangle \leadsto \langle \bar{l}[l'_i/l_i, l'_j/l_j], (r_i \cup r_j)(v)\rangle$ *if there exist* $l_i, l_j \in \bar{l}, g_i, g_j, \alpha, r_i$ *and* r_j *such that* $l_i \xrightarrow{g_i, \alpha!, r_i} l'_i$, $l_j \xrightarrow{g_j, \alpha?, r_j} l'_j$, $g_i(v)$ *and* $g_j(v)$.
- $\langle \bar{l}, v\rangle \leadsto \langle \bar{l}, v \oplus d\rangle$ *if* $d \leq MD(\bar{l}, v)$ □

4 The UPPAAL Model–Checker

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.

Logic The properties that can be analysed 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 (c) or a component location ($A_i \texttt{at}\ l$). Atomic clock (data) constraints are either integer bounds on individual clock (data) variables (e.g. $1 \leq x \leq 5$) or integer bounds on differences of two clock (data) variables (e.g. $3 \leq x - y \leq 7$).

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 network configurations. Then the satisfaction relation $\models$ between network configurations 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 of β (behaving as usual with respect to the boolean connectives). Satisfaction with respect to an atomic formula is given by the following definitions:

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

Our (simple and efficient) model–checking technique extends to the logic presented in [7], which also allows for bounded liveness properties to be specified. Currently, bounded liveness properties are obtained by reachability analysis of the system in the context of testing (and time–sensitive) automata. We conjecture that all bounded liveness properties of the logic in [7] can be translated into reachability problems in this manner.

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 network configurations of the form $[\bar{l}, D]$, where D a constraint system (i.e. a conjunction of atomic clock and data constraints) and $\bar{l}$ a control–vector. Some of the rules defining this symbolic interpretation is given in Table 1.

$$\frac{D \subseteq c}{\vdash [\bar{l}, D] : c} \qquad \frac{l_i = l}{\vdash [\bar{l}, D] : A_i \mathsf{at}\ l} \qquad \frac{\vdash [\bar{l}, D] : \beta}{\vdash [\bar{l}, D] : \exists\Diamond\beta}$$

$$\frac{\vdash \left[\bar{l}[m_i/l_i, m_j/l_j], (r_i \cup r_j)(D \wedge g_i \wedge g_j)\right] : \exists\Diamond\beta}{\vdash [\bar{l}, D] : \exists\Diamond\beta} \quad \begin{bmatrix} l_i \xrightarrow{g_i, \alpha?, r_i} m_i \\ l_j \xrightarrow{g_j, \alpha!, r_j} m_j \end{bmatrix}$$

$$\frac{\vdash [\bar{l}, D^{\uparrow}] : \exists\Diamond\beta}{\vdash [\bar{l}, D] : \exists\Diamond\beta}$$

Table 1. Symbolic Interpretation of Reachability Logic

To read the rules of Table 1 some notation needs to be explained. For D a constraint system and r a set of variables (to be reset) $r(D)$ denotes the set of variable assignments $\{r(v) \mid v \in D\}$. Now $D^{\uparrow}$ denotes the following set of variable assignments

$$D^{\uparrow} = \{w \mid \exists v \in D \exists d \leq \mathrm{MD}(\bar{l}, v). w = v \oplus d\}$$

An important observation is that, whenever D is a constraint system (i.e. a conjunction of atomic clock and data constraints), then so are both $r(D)$ and $D^{\uparrow}$.

Moreover, due to Richard Bellman representing constraint systems as weighted directed graphs (with clock and data variables as nodes), these operations as well as testing for inclusion between constraint systems may be effectively implemented in $O(n^2)$ and $O(n^3)$ using shortest path algorithms [11, 12, 6].

Now, by applying the proof rules of Table 1 in a goal directed manner we obtain an algorithm (see also [13]) for deciding whether a given symbolic network configuration $[\bar{l}, D]$ satisfies a property $\exists\Diamond\beta$. To ensure termination (and efficiency), we maintain a (past–) list $\mathcal{L}$ of the symbolic network configurations encountered. If, during the goal directed application of the proof rules of Table 1 a symbolic network configuration $[\bar{l}, D']$ is generated which is already "covered" by a configuration $[\bar{l}, D]$ in $\mathcal{L}$ (i.e. $D' \subseteq D$) then the the goal directed search fails at $[\bar{l}, D']$ and backtracking is needed. If $[\bar{l}, D']$ "covers" some configuration $[\bar{l}, D]$ in $\mathcal{L}$ (i.e. $D \subseteq D'$) then $[\bar{l}, D']$ replaces $[\bar{l}, D]$ in $\mathcal{L}$.

5 Applications and Performance

Uppaal has been used to verify various benchmark examples and applications including: several versions of Fischer's protocol, Philips Audio-Control Protocol, the Train Gate Controller, the Manufacturing Plant, the Steam Generator, the Mine-Pump Controller and the Water Tank.

In [8] an experiment was performed using four existing real-time verification tools: Uppaal, HyTech (Cornell), Kronos (Grenoble) and Epsilon (Aalborg). In the experiment it was verified that the so-called Fischer's mutual exclusion protocol [10, 1], shown in Figure 2, satisfies the mutual exclusion property $\forall\Box\neg((P_1\ \texttt{at}\ cs) \wedge (P_2\ \texttt{at}\ cs))$. With all the tools installed on the same machine[6] the standard Unix command `time` was used to measure execution time. The resulting time-performance diagram, shown in Figure 4, indicate that Uppaal performs time- and space-wise favorably compared to the other tools in the experiment.

In [7], in this volume, the Philips Audio-Control Protocol [3, 4] was verified using Uppaal. A version of the protocol is shown in Figure 5. In the verification of this protocol, we found the diagnostic model-checking feature of Uppaal useful for detecting and correcting several errors in the description of the protocol. Uppaal verifies that the received bit stream is guaranteed to be identical to the sent bit stream in 3.8 seconds[7].

6 Conclusion and Future Work

In this paper we have presented the main features of Uppaal together with a review of and pointers to its theoretical foundation and application on case–studies.

[6] The tools were installed on a Sparc Station 10 running SunOS 4.1.3 with 64MB of primary memory and 64 MB of swap memory.

[7] Uppaal version 0.95 was installed on a Sparc Station 10 running SunOS 4.1.3, with 64 MB of primary memory and 64 MB of swap memory.

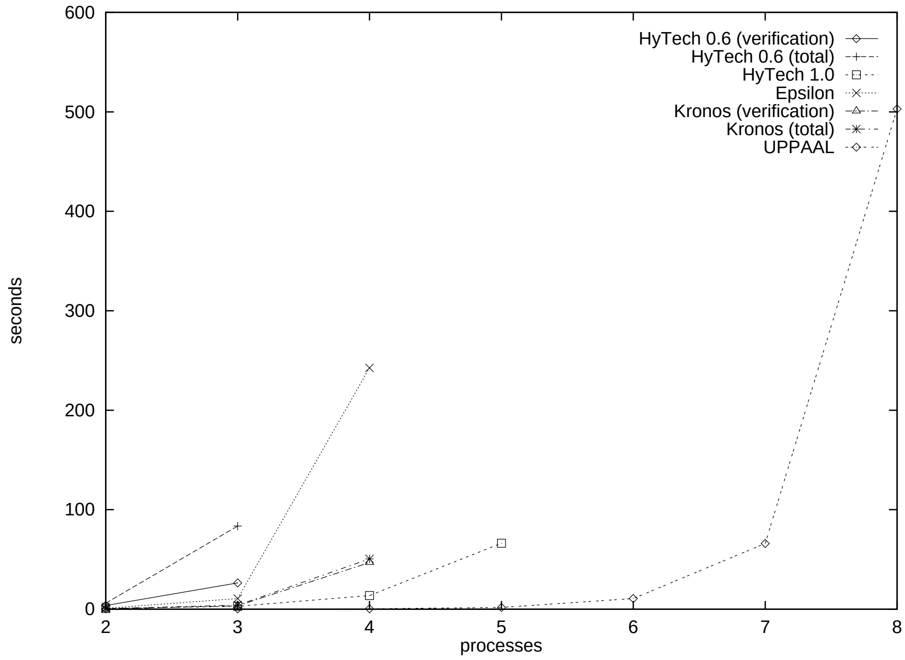

Fig. 4. Execution Times for Fischer's Protocol.

Future versions of UPPAAL will extend the current model–checker to the safety and bounded liveness logic of [7]. Also future versions of UPPAAL will integrate the newly developed compositional model–checking technique of [6], which, judged from experimental results using a CAML prototype implementation [5], seems to be a powerful technique in the on–going fight against explosion problems.

References

1. Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. *Lecture Notes in Computer Science*, 600, 1993.
2. R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of ICALP'90*, volume 443, 1990.
3. D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of FTRTFT'94*, volume 863 of *Lecture Notes in Computer Science*, 1993.
4. Pei-Hsin Ho and Howard Wong-Toi. Automated Analysis of an Audio Control Protocol. In *Proc. of CAV'95*, volume 939 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.
5. F. Laroussinie and K.G. Larsen. Compositional Model Checking of Real Time Systems. In *Proc. of CONCUR'95*, Lecture Notes in Computer Science. Springer Verlag, 1995.

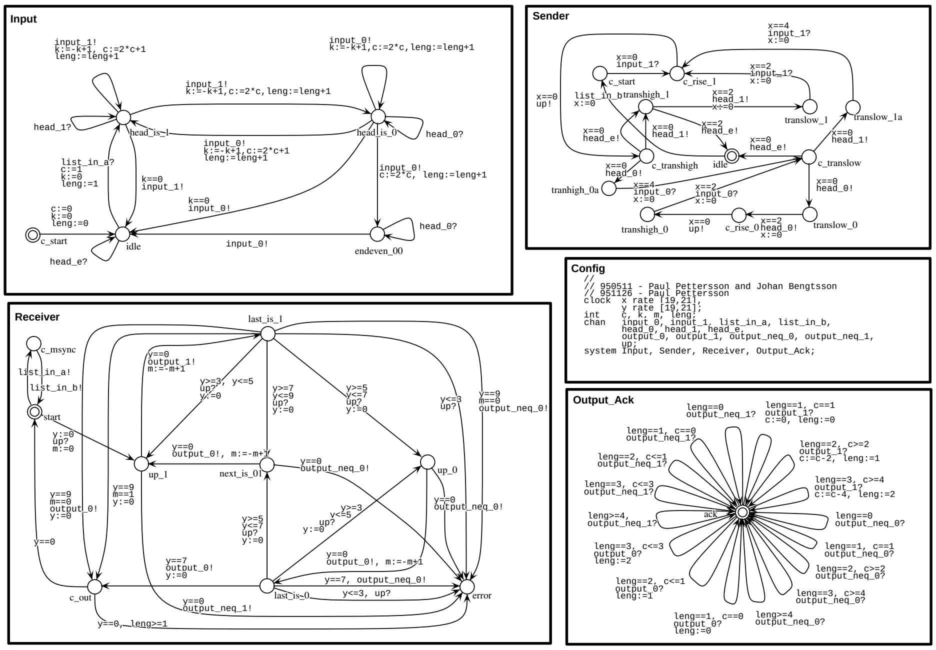

Fig. 5. Philips Audio-Control Protocol.

6. K.G. Larsen, P. Pettersson, and W. Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. To appear in *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, December 1995.
7. Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of the 4*th *DIMACS Workshop on Verification and Control of Hybrid Systems*, Lecture Notes in Computer Science, October 1995.
8. Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, 1995.
9. A. Olivero, J. Sifakis, and S. Yovine. Using Abstractions for the Verification of Linear Hybrids Systems. In *Proc. of CAV'94*, volume 818 of *Lecture Notes in Computer Science*, 1994.
10. N. Shankar. Verification of Real-Time Systems Using PVS. In *Proc. of CAV'93.*, volume 697, 1993.
11. C.E. Leiserson T.H. Cormen and R.L. Rives. *Introduction to ALGORITHMS*. MIT Press, McGraw-Hil, 1990.
12. Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real–time transition systems. In *Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 210–224, 1993.
13. Wang Yi, Paul Pettersson, and Mats Daniels. Autfomatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the* 7th International Conference on Formal Description Techniques, 1994.

Recent Publications in the BRICS Report Series

RS-96-58 **Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi.** UPPAAL — ***a Tool Suite for Automatic Verification of Real–Time Systems*****. December 1996. 12 pp. Appears in Alur, Henzinger and Sontag, editors, , '96 Proceedings, LNCS 1066, 1996, pages 232–243.**

RS-96-57 **Kim G. Larsen, Paul Pettersson, and Wang Yi.** ***Diagnostic Model-Checking for Real-Time Systems*****. December 1996. 12 pp. Appears in Alur, Henzinger and Sontag, editors, , '96 Proceedings, LNCS 1066, 1996, pages 575–586.**

RS-96-56 **Zine-El-Abidine Benaissa, Pierre Lescanne, and Kristoffer H. Rose.** ***Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution*****. December 1996. 35 pp. Appears in Kuchen and Swierstra, editors,** ***8th International Symposium on Programming Languages, Implementations, Logics, and Programs*****, PLILP '96 Proceedings, LNCS 1140, 1996, pages 393–407.**

RS-96-55 **Kåre J. Kristoffersen, François Laroussinie, Kim G. Larsen, Paul Pettersson, and Wang Yi.** ***A Compositional Proof of a Real-Time Mutual Exclusion Protocol*****. December 1996. 14 pp. To appear in Dauchet and Bidoit, editors,** ***Theory and Practice of Software Development. 7th International Joint Conference CAAP/FASE*****, TAPSOFT '97 Proceedings, LNCS, 1997.**

RS-96-54 **Igor Walukiewicz.** ***Pushdown Processes: Games and Model Checking*****. December 1996. 31 pp. Appears in Alur and Henzinger, editors,** ***8th International Conference on Computer-Aided Verification*****, CAV '96 Proceedings, LNCS 1102, 1996, pages 62–74.**

RS-96-53 **Peter D. Mosses.** ***Theory and Practice of Action Semantics*****. December 1996. 26 pp. Appears in Penczek and Szalas, editors,** ***Mathematical Foundations of Computer Science: 21st International Symposium*****, MFCS '96 Proceedings, LNCS 1113, 1996, pages 37–61.**