UPPSALA UNIVERSITY

DoCS

Department of Computer Systems

UPPAAL

A Tool for Automatic Verification of Real-Time Systems

Johan Bengtsson and Fredrik Larsson

Master of Science Thesis
DoCS 96/67

January 15, 1996
ISSN 0283-0574

Abstract

As real-time systems often operate in safety-critical environments, it is extremely important to know that these systems are correct. A traditional way to guarantee the correctness of a real-time system is by simulation and testing. However, even if the system passes all imaginable tests, it does not guarantee that the system works correctly in all real life situations. This motivates the use of formal methods for modeling systems and verifying that they fulfill their specifications.

In this report we describe networks of timed automata as a formal model for real-time systems; we also describe how to extend this model to handle systems with drifting clocks. A simple logic for expressing safety properties of a system is presented, together with an algorithm for checking if a model satisfies a given property. We have implemented a tool, UPPAAL, for automatic verification, based on the algorithm and model mentioned above. To handle the infinite state-space UPPAAL uses constraint solving techniques to group states having the same properties. UPPAAL also adopts the on-the-fly verification techniques, to avoid constructing the parts of the state-space that are not reachable.

The performance of UPPAAL has been tested on various examples, including Fischers Protocol for mutual exclusion and Philips Audio Control Protocol. The performance of UPPAAL has also been compared to other verification tools, and the result shows that UPPAAL is not only faster, but also capable of dealing with larger systems.

Acknowledgments

We want to thank Wang Yi (DoCS, Uppsala), Kim Larsen (BRICS, Aalborg) and Paul Petersson (DoCS, Uppsala) for help and guidance. We also want to thank the Department of Computer Systems at Uppsala University for a inspiring environment, and the Swedish Board for Technical Development (NUTEK) for the funding.

Contents

Chapter 1

Introduction

The use of real-time systems becomes more and more common in our society nowadays. They are often embedded in safety-critical applications and it is therefore extremely important that they work correctly. One will also save money if errors can be detected as early as possible in the development process.

People who develop computer systems often use simulation techniques to "ensure" that the system works correctly. Simulation is the process when people interactively test how the system reacts in different situations on different inputs. But even if the system reacts satisfactory in these tests one will never know for sure that the system works correctly in all possible situations. This is why we need the verification phase in system development.

Verification can be thought of as a simulation of all possible inputs in all possible system states. It requires a formal way to model the system, a logic to express the properties and an algorithm that can verify if the model satisfies the properties expressed in the logic or not. For example if we have an automatic railway controller we want it to satisfy the property that not more than one train is allowed to cross the bridge simultaneously. This property is an example of a safety property of the system. Another thing one might want to assure is that a train will cross the bridge before a certain time. This is an example of a bounded liveness property.

If formal methods shall be useful it is important that they are powerful, easy to use and that there exist efficient programs doing the verification automatically. The user shall only be required to model the system, enter properties that he/she wants to check, press a button and after some time get information telling if, and maybe why, the properties were satisfied by the system model or not. However, one must remember that it is the model that is verified so; if the model does not correspond to the system the results are useless. Therefore modeling is also an important phase in system development. But in this report, we will mainly deal with automatic verification.

We have implemented a tool called UPPAAL that can be used to verify safety and bounded liveness properties of real-time systems modeled as networks of timed automata or linear hybrid automata[1].

The implementation of UPPAAL has been done with the following things in mind:

- It should be as fast as possible.

[1] A linear hybrid automaton is an extension of the timed automaton, where the clocks may proceed with different rates.

- It must require as little memory as possible because the systems that one wants to verify can be large and complex.
- The design must be modular so that it's easy to reuse parts of the implementation and to add new types and features.
- Of course, it must also be easy to use.

These requirements are sometimes a bit contradictory. The choice of algorithms and data structures often forces one to do trade-offs between time and memory usage. General operations that are easy to reuse are not as fast as they could be if they were optimized by use of more problem specific details. The tool is implemented in C++ using an object-oriented design. The implementation will be discussed in detail in a later chapter.

This report is organized in the following way:

Chapter 2 is about how to model real-time systems. It introduces timed automata as a formal model for real-time systems. There are also some examples that illustrate how the modeling works.

Chapter 3 is about the verification techniques implemented in UPPAAL. It describes the problem of reachability analysis and the verification algorithm based on constraint solving techniques.

Chapter 4 gives an overview of UPPAAL. It describes the functionality and structure of UPPAAL.

Chapter 5 describes in detail how UPPAAL is implemented. There will be proofs showing that used algorithms correspond to the semantic definitions of the operations mentioned in the algorithm description in chapter 3. It will describe the algorithms and data structures chosen and in some cases discuss alternative approaches and motivate why they weren't chosen.

Chapter 6 is about the performance of UPPAAL. It shows some examples that UPPAAL has been tested on and the time UPPAAL spent verifying them.

The last chapter of the report, chapter 7, contains concluding remarks.

Appendix A is the users guide to UPPAAL.

Chapter 2

Modeling Real-time Systems

We study real–time systems consisting of communicating processes with shared clocks. The systems are described by networks of timed automata [AD90] extended with auxiliary data variables and with a notion of parallel composition. Instead of interpreting parallel composition as logical conjunction (*i.e.* communication is only possible if all components in the system wants to communicate on the same channel) we use a CCS-like interpretation of parallel composition (proposed in [YPD94]), allowing one-to-one communication and interleaving.

2.1 Networks of Timed Automata

A timed automaton is a standard finite–state automaton extended with a finite collection of real valued clocks. The clocks are assumed to proceed at the same rate and their values may be compared with natural numbers or reset to 0. We have extended the notion of timed automata to include integer variables, *i.e.* integer valued variables that may be compared to natural numbers or assigned to any value of the form $ax + b$ where $a, b \in \mathbf{Z}$ and x is the variable being reassigned.

In our model we also allow clocks not only to be reset, but also to be set to any non–negative integer value.

Definition 1 *(Atomic Constraints) Let C be a set of real valued clocks and I a set of integer valued variables. An atomic clock constraint over C is a constraint of the form: $x \sim n$, for $x \in C$, $\sim \in \{\leq, \geq, =\}$ and $n \in \mathbf{N}$. An atomic integer constraint over I is a constraint of the form: $i \sim n$, for $i \in I$, $\sim \in \{\leq, \geq, =\}$ and $n \in \mathbf{Z}$.*

Let $\mathcal{C}_c(C)$ denote the set of all clock constraints over C, and let $\mathcal{C}_i(I)$ denote the set of all integer constraints over I. □

Definition 2 *(Guards) Let C be a set of real valued clocks and I a set of integer valued variables. A guard g over C and I is a formula generated by the following syntax: $g ::= c \mid g \wedge g$, where $c \in (\mathcal{C}_c(C) \cup \mathcal{C}_i(I))$* □

We let $\mathcal{B}(C, I)$ stand for the set of all guards over C and I.

Definition 3 *(Assignments) Let C be a set of real valued clocks and I a set of integer valued variables. A clock assign over C is a tuple $\langle v, c\rangle$, where $v \in C$ and $c \in \mathbf{N}$. An integer assign over I is a tuple $\langle v, c_1, c_2\rangle$ representing the assignment $v = c_1 \cdot v + c_2$, where $v \in I$ and $c_1, c_2 \in \mathbf{Z}$.* □

We will use $\mathcal{A}(C, I)$ to denote the power-set of all assignments over I and C.

Definition 4 *(Timed Automaton) A timed automaton A over a finite set of actions Act, clocks C and integer variables I is a tuple $\langle L, l_0, E\rangle$, where L is a finite set of nodes (control-nodes), l_0 is the initial node, and $E \subseteq L \times \mathcal{B}(C, I) \times Act \times \mathcal{A}(C, I) \times L$ corresponds to the set of edges. To denote, $\langle l, g, a, r, l'\rangle \in E$, we will write $l \xrightarrow{g,a,r} l'$.*

In order to study compositionality problems we introduce a parallel composition between timed automata. In order to get the kind of parallel composition we want, we have to introduce the notion of co–actions, that is done by defining a synchronization function $\mathcal{T}$.

Definition 5 *(Synchronization Function) Let $\mathcal{T} \subseteq Act \times Act$ be a function such that:*

$$\langle a_i, a_j\rangle \in \mathcal{T} \Rightarrow \langle a_j, a_i\rangle \in \mathcal{T} \text{ for all } a_i, a_j$$

□

Definition 6 *(Parallel Composition) Let A_1, A_2 be two timed automata. Then the parallel composition $(A_1|A_2)$ is a timed automaton $\langle L, l_0, E\rangle$, where $(l_1|l_2) \in L$ whenever $l_1 \in L_1$ and $l_2 \in L_2$, $l_0 = (l_{1,0}|l_{2,0})$. The edges E are defined as follows:*

- $(l_1|l_2) \xrightarrow{g,\tau,r} (l_1'|l_2')$ *if* $(l_1 \xrightarrow{g_1,a_1,r_1} l_1') \wedge (l_2 \xrightarrow{g_2,a_2,r_2} l_2') \wedge (g = g_1 \cup g_2) \wedge (\langle a_1, a_2\rangle \in \mathcal{T}) \wedge (r = r_1 \cup r_2)$
- $(l_1|l_2) \xrightarrow{g,a,r} (l_1'|l_2)$ *if* $l_1 \xrightarrow{g,a,r} l_1'$
- $(l_1|l_2) \xrightarrow{g,a,r} (l_1|l_2')$ *if* $l_2 \xrightarrow{g,a,r} l_2'$

Note that parallel composition is commutative and associative. □

A state of a timed automaton A is a pair $\langle l, u\rangle$ where l is a node of A and u is an assignment, mapping each clock in C to a value in $\mathbf{R}^+$, and each integer variable in I to a value in $\mathbf{Z}$. We will use $g(n)$ to denote that the assignment u satisfies the guard g. The initial state of A is $\langle l_0, u_0\rangle$, where u_0 is the assignment mapping all variables to 0.

An automaton may take two types of transitions, from state to state:

- Delay transition: $\langle l, u\rangle \overset{\epsilon(d)}{\rightsquigarrow} \langle l, u'\rangle$ following the rules given in definition 7
- Action transition: $\langle l, u\rangle \overset{g,a,r}{\rightsquigarrow} \langle l', u'\rangle$ following the rules given in definition 8

Definition 7 *(Delay transition) Let $\langle l, u\rangle$ and $\langle l', u'\rangle$ be two states of a timed automaton A, and let d be a positive real. Then*

$$\langle l, u\rangle \overset{\epsilon(d)}{\rightsquigarrow} \langle l', u'\rangle \text{iff} \begin{cases} l' = l \\ u'(x) = u(x) + d & \text{if } x \in C \\ u'(x) = u(x) & \text{if } x \in I \\ d \leq M(l, u) \end{cases}$$

Where $M(l,u)$ is the maximal delay of $\langle l,u\rangle$, defined as follows:

$$M(l,u) = \begin{cases} \sup\{t \mid g(u+t)\} & \text{if } \exists\, l' : l \xrightarrow{g,a,r} l' \\ \infty & \text{otherwise} \end{cases}$$

□

Intuitively this means that an automaton may not stay in a control–state long enough for the last outgoing edge from that state to close. This results in a maximal progress behavior of the automaton.

Definition 8 *(Action Transition) Let $\langle l,u\rangle$ and $\langle l',u'\rangle$ be two states of a timed automaton A. Then*

$$\langle l,u\rangle \overset{g,a,r}{\rightsquigarrow} \langle l',u'\rangle \text{ iff } (l \xrightarrow{g,a,r} l') \wedge g(u) \wedge \left(u'(x) = \begin{cases} c_0 & \text{if } x \in C \ \wedge \langle x, c_0\rangle \in r \\ c_1 u(x) + c_0 & \text{if } x \in I \ \wedge \langle x, c_1, c_0\rangle \in r \\ u(x) & \text{otherwise} \end{cases} \right)$$

□

2.2 Networks of Linear Hybrid Automata

The model of timed automata can be extended by allowing the clocks to proceed with different rates, and also allowing the rates to drift within a bounded interval. This model is often refered to as linear hybrid automata.

It has been shown in [AOY94] that for a restricted subclass of linear hybrid automata, each linear hybrid automaton can be simulated by a timed automaton. The restrictions nedded are:

- The clocks must never be stoped, *i.e.* it must never be possible for a clock to have rate 0.
- If we have an edge e between two nodes l and l', any clock which rate in l' differs from the rate in l must be reset on e.
- Any clock with negative rate must have a finite upper bound in each node, and any clock with positive rate must have a finite lower bound in each node.

2.3 Examples

2.3.1 Fischers Protocol

The protocol was proposed by Fischer and described by Lamport [Lam87]. Its purpose is to guarantee mutual exclusion in a concurrent system consisting of several processes using a variable shared among the processes and properly timing the processes in changing the shared variable. Each of the processes is assumed to have a local clock. The idea behind the protocol is that the timing constraints on the local clocks are set so that only one process can change the global variable to its own process number, then read the global variable later and if the shared variable is still equal to its own number, enter the critical section.

We will study two versions of the protocol, one simplified version allowing only one process to enter the critical section and never exit it, and one more advanced version with cyclic processes.

We start with a simplified version of the original protocol. It permits only one process to enter its critical section, and that process will then stay in the critical section forever. Recovery from failure to enter the critical section is omitted. This version of the protocol has been thoroughly studied by researchers *e.g.* [AL93, Sha93, YPD94].

Assume a concurrent system with n processes $\mathtt{P}_1 \ldots \mathtt{P}_n$. We use a global clock $\mathtt{x}_i$ to model the local clock of process $\mathtt{P}_i$. A process $\mathtt{P}_i$ may be in either of four local states `A`,`B`,`C` or `Cs`. Initially all processes are in their `A`–state and the shared variable `id` is 0.

A process $\mathtt{P}_i$ that wants to enter the critical section can try to do so if $\mathtt{id} = 0$. Then it changes its local state from `A` to `B`. In state `B` it can only stay for less than T time–units before it proceeds to state `C`. When proceeding from state `B` to state `C` the process changes the value of `id` to it's own process number (*i.e.* $\mathtt{id} := \mathtt{i}$). In state `C` the process has to wait for, at least, T time–units. Then, before proceeding it must check that the shared variable `id` still has the value `i`. If it does, then $\mathtt{P}_i$ can proceed to `Cs`.

A graphical modeling of the simplified version of Fischers protocol, with two processes and $T = 1$, is shown in figure 2.1.

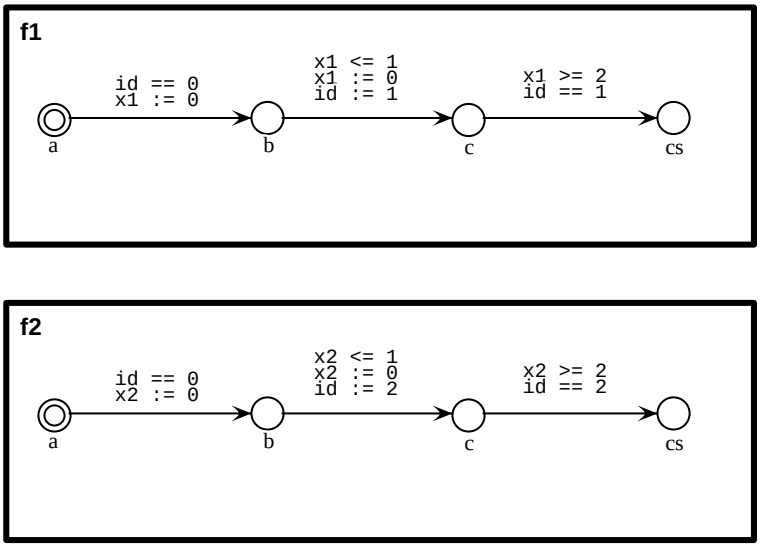

Figure 2.1: A simplified version of Fischers mutual exclusion protocol, with two processes.

Now we extend the protocol to cyclic processes. Each process may stay in the critical section for an unbounded amount of time each cycle, and recover from failure to enter the critical section.

As in the simple case we assume a concurrent system with n processes $\mathtt{P}_1 \ldots \mathtt{P}_n$, and we use n global clocks $\mathtt{x}_1 \ldots \mathtt{x}_n$ to model the local clocks of the processes.

Also as in the simple case, when a process wants to enter the critical section, it can try to get access only if a shared variable `id` is 0. If it is, the process can proceed to a new state, `B`. In the new state it can only stay for less than T time–units before it has to proceed to state `C`. When proceeding from state `B` to state `C` the process changes the value of `id` to it's own process–ID. In state `C` the process has to wait for, at least T time–units. If `id` still contains the process–ID of this process, it can enter the critical section, otherwise it must restart the entering procedure, from

the beginning. After leaving the critical section, the process just resets the value of `id` to 0 and start over.

A graphical model of a system with two processes is shown in figure 2.2.

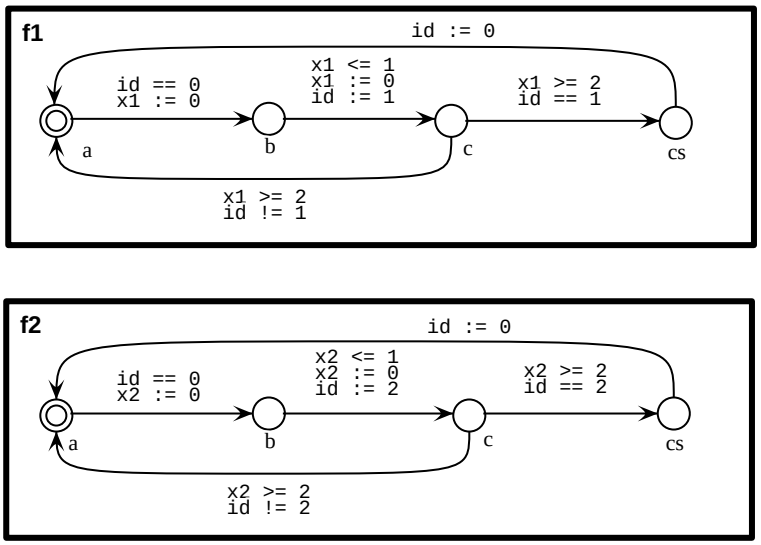

Figure 2.2: Fischers mutual exclusion protocol, with two cyclic processes.

2.3.2 Philips Audio Control Protocol

This is an example from a real–world application, and it was first described and verified by Bosscher *et. al.* [BPV93].

The example is a simplified version of an audio control protocol developed by Philips for the physical layer of an interface bus, that connects the various parts of stereo equipment. The protocol uses Manchester encoding, and it has to deal with uncertainty in the timing of the events, due to both hardware and software constraints.

Description of the Protocol

The protocol uses the well–known Manchester encoding of bit strings. In this encoding it is assumed that the time axis is divided into time–slots of equal length and each one is capable of transmitting one bit of data. In the message sent over the bus, a "1" is represented by an up going edge in the middle of a time–slot and a "0" is represented by a down going edge in the middle of a time–slot. If two bits with the same value are sent in a row, an additional edge is placed in between the two time–slots. An example of the encoding is shown in figure 2.3.

In addition to clock drift of up to 5% between the different parts of the system, the protocol has to face a number of other difficulties:

1. Although the receiver knows the length of the time–slots, it does not know when the first slot begins. This problem is resolved by requiring that the first bit sent is always a "1".

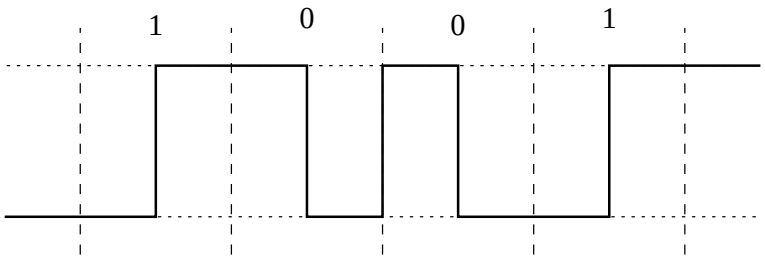

Figure 2.3: Manchester encoding of 1001

2. A receiver does not know the length of the message it is receiving.
3. In reality, the signal on the bus is not perfectly block–shaped. In particular, the time it takes for the bus to change from high to low make it impossible for the receiver to reliably detect a down going edge, and for this reason the receiver has to decode the message without seeing the down going edges. This is always possible, except that a message ending with "10" cannot be distinguished from the same message ending with "1". To solve this problem, we require that the messages always end with "00" or have an odd number of bits
4. Different senders may start sending at the same time; so bus collisions may occur.
5. The message delay in the bus can be significant.

For simplicity, all complications that arise from problems 4 and 5 are ignored. We assume a setting where one sender, and one receiver are communicating through a bus with negligible delay.

Modeling the Protocol

In modeling the protocol, we adopt the method described by Ho and Wong–Toi [HWT95]. The protocol is divided into four processes:

1. An input–process that nondeterministically generates valid messages, starting with a "1" and ending either after an odd number of bits, or with "00". To achieve this, we use an integer valued variable k as a "parity–counter", *i.e.* k is one if an odd number of bits have been generated, and 0 if an even number of bits have been generated.

   Each time the sender consumes a bit, the value of the next bit is chosen. At this point the process may also end the message. The input–process gives the sender–process opportunity to check the value of the next bit to be sent, using three different channels: `head_1` for checking if next bit is "1", `head_0` to check if next bit is "0", and `head_e` to check if the message has ended.

2. A sender–process that generates Manchester encoded signals by looking at the next bit, and determine the time for the next voltage change. It uses the channel `up` to signal an up going edge. A down going edge is not signaled at all, to model that the down going edges can't be detected.

3. A receiver–process that decodes the incoming message by measuring the times between the `up` signals. If no new `up` signal arrives in due time, we assume that the transmission has ended. We use an integer valued "length parity counter", m, as well as the four channels; `output_0`, `output_1`, `output_neq_0`, and `output_neq_1`, for error detection.

4. An output–process that checks if the bit received by the receiver is the expected bit, and signals the result using one of the four channels `output_0`, `output_1`, `output_neq_0`, or `output_neq_1`.

A graphical description of our model of the protocol can be seen in figure 2.4. The description is given as a linear hybrid–system, *i.e.* a system where the rate of clocks may vary, in a bounded interval. Those kind of systems can be modeled in the same way as timed automata, and they can be transformed into timed automata [AOY94].

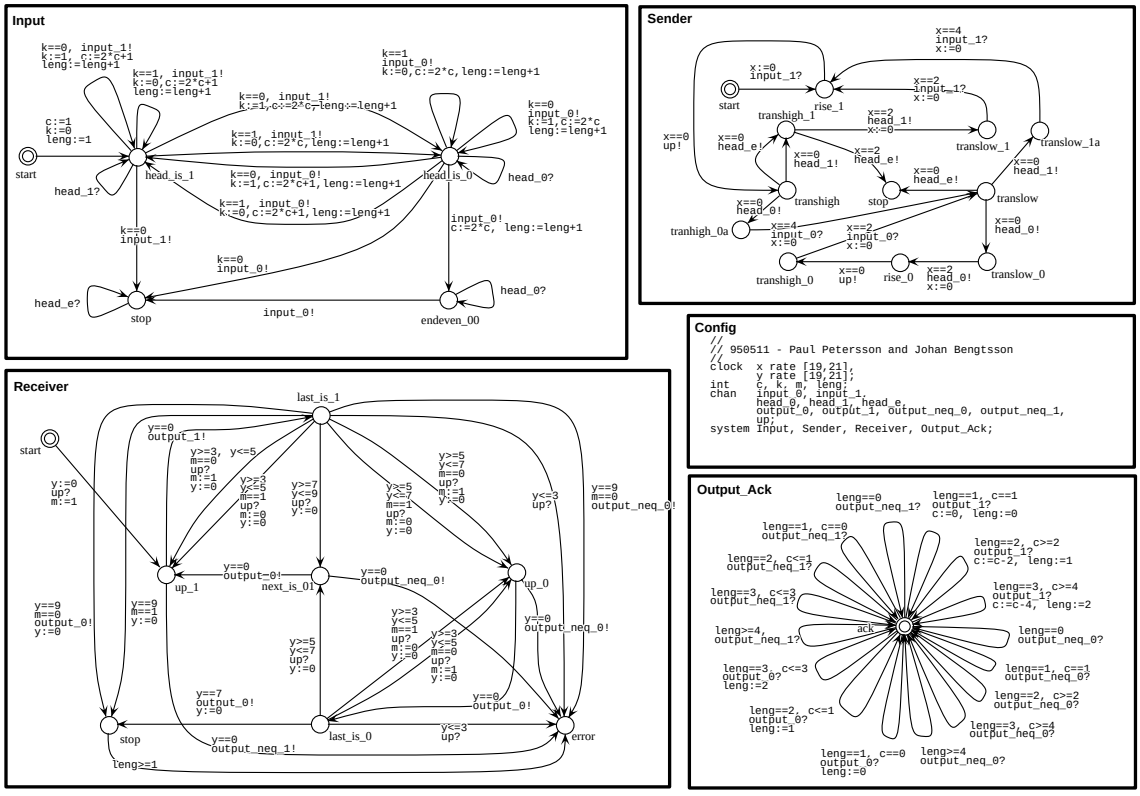

Figure 2.4: Audio Control Protocol

Chapter 3

Verifying Real-time Systems

As mentioned earlier, the increasing use of real–time systems makes it important to develop methods for verifying that the systems behave correctly. In particular, as many of the real–time systems today are embedded in safety–critical environments, where the consequences of a system failure could cause a disaster.

It has been pointed out in [Hal93, YPD94], that the practical goal of verification of real–time systems, is to verify simple safety properties *i.e.* properties of the type "can we guarantee that a bad thing won't occur ?" or "are we sure that eventually a good thing will occur ?". Properties usually formalized in temporal logic as $\forall\Box\neg$*bad–thing* and $\exists\Diamond$*good–thing* For traditional finite–state systems this kind of properties can be verified by checking all possible reachable states of a system. Unfortunately the systems considered here are infinite–state because of the real–valued clocks.

3.1 Reachability Analysis by Constraint Solving

Adopting the methodology developed in [YPD94], we start with describing a simple reachability problem for timed automata. A generalized version of the problem will be given later.

Definition 9 *(Simple Reachability) Let $\langle l_0, u_0\rangle$ and $\langle l_f, u_f\rangle$ be states of a timed automaton A. Then $\langle l_f, u_f\rangle$ is reachable from $\langle l_0, u_0\rangle$ iff there exists a trace $\langle l_0, u_0\rangle \overset{Q_1}{\rightsquigarrow} \cdots \overset{Q_n}{\rightsquigarrow} \langle l_f, u_f\rangle$ that leads to the final state.* □

Note that the simple reachability relation is the transitive and reflexive closure of the transition relation.

To solve the problem with infinte state–space we use constraint systems to symoblically represent sets of assignments that we call time regions.

Definition 10 *(General Reachability) Let l_0, l_f be nodes of a timed automaton A, and let U_0, U_f be sets of assignments. We say that $\langle l_f, U_f\rangle$ is reachable from $\langle l_0, U_0\rangle$ iff there exists $u_0 \in U_0$ and $u_f \in U_f$ such that $\langle l_f, u_f\rangle$ is reachable from $\langle l_0, u_0\rangle$.* □

We will now describe an algorithm for solving the general reachability problem, using constraint-solving techniques.

Let A be the timed automaton to be analyzed. We assume that the clocks of A can be ordered as a vector $\langle c_1, c_2, \ldots, c_n \rangle$. Then the clock part of an assignment can be seen as a point in the n–dimensional space $\mathbf{R}_+^n$. We will use linear constraint systems to describe regions of such points, and solve the general reachability problem by manipulating such linear constraint systems.

A Class of Simple Linear Constraint Systems

The class of constraints we will study will always take the form: $l_i \leq x_i \leq u_i$ or $l_{ij} \leq x_i - x_j \leq u_{ij}$. We will use the term *linear constraint system* to denote a set of constraints of that form. A *solution* to a linear constraint system r is an assignment that maps each variable to a value that satisfies the constraints.

The solution set of a constraint system r (*i.e.* the set of all solutions to r.) can be seen as a convex polytope in the n–dimensional space.

From now on we will simply call a constraint system r a *region*, refering to its solution set. We will use $r = \emptyset$ to denote that r is not satisfiable,(*i.e.* its solution set is empty), $r \subseteq r'$ to denote that r implies r' and $r \wedge r'$ to denote the intersection of the solution sets of r and r'.

To represent a constraint system, we introduce a particular clock x_0 which always has value 0. Then we may assume that all constraints in the system have the form $l_{ij} \leq x_i - x_j \leq u_{ij}$. Constraint systems on this form can easily be represented by matrices.

3.2 Operations on Constraint Systems

The subject of this section is to describe an algorithm that decides if a symbolic state $\langle l_f, U_f \rangle$ is reachable from a symbolic state $\langle l_0, U_0 \rangle$.

The algorithm will manipulate r by use of constraint solving. The guards and resets on the transitions and delays in the state are the things that affect the constraints and motivate the operations introduced. The direction of the reachability analysis, forward or backward, will need different operations on constraint systems.

Definition 11 *(the delay operations)*

1. *The weakest precondition of a time region r is defined as*

$$wp(r) = \{x \mid \exists d \geq 0 \ : \ x + d \in r\}$$

2. *The strongest postcondition of a time region $r$is defined as*

$$sp(r) = \{x \mid \exists d \geq 0 \ : \ x - d \in r\}$$

□

This operations are needed because the automaton may delay in a state, perform delay transitions, before an action transition is performed. If the reachability analysis isforward we start from $\langle l_0, U_0 \rangle$ and search forward for $\langle l_f, U_f \rangle$. In this case the strongest postcondition is used. If we do backward reachability analysis we search backward from $\langle l_f, U_f \rangle$ to $\langle l_0, U_0 \rangle$ and handle delays with weakest precondition.

Definition 12 *(guard conjunction) If r is a time region and g is a set of guards the time region after the guard conjunction is the intersection of the solution sets: $g \cap r$* □

This operation is performed when determining which transitions are enabled or not. If r' becomes inconsistent we have $r' = \emptyset$.

Definition 13 *(operations treating resets) Let r be a time region and k the clock to be reset.*

1. *$reset(r, k)$ is defined as*

$$\{v \mid \exists w \in r \ : \ v_k = 0 \wedge \forall i \neq k \ : \ v_i = w_i$$

2. *$free(r, k)$ is defined as*

$$\{v \mid \exists w \in r \ : \ v_k \in \mathbf{R} \wedge \forall i \neq k \ : \ v_i = w_i$$

3. *The Boolean operation $check(r, k)$ is defined as*

$$\{v \mid v \in r \wedge v_k = 0\} \neq \emptyset$$

□

The reset operation is used in forward reachability analysis to handle the resets that may be performed when the system perform a transition. In backward reachability analysis the free and check operations are used to handle the resets on the transitions. It is straight forward to extend the above definition to handle sets of resets and resets involving other values than 0.

The implementation of the algorithm and operations are much easier if the time regions are of a special form called closed.

Definition 14 *Let r be a time region of the form $x_i - x_j \leq u_{i,j}, x_0 = 0$. r is closed if it satisfies*

$$\forall d_{i,j} \leq u_{i,j} \exists x_i \exists x_j \ : \ x_i - x_j = d_{i,j}$$

□

Time regions are closed under the operations defined above.

3.3 Algorithms

We're now in a position to present an algorithm performing reachability analysis using constraint solving. The algorithm checks if a state in a timed automaton is reachable from the initial state or not.

When searching the state space we need two buffers that we can call wait and passed respectively. The wait buffer holds the states not yet explored and the passed buffer holds the states explored so far. We will treat the forward and backward reachability analysis algorithms in separate subsections.

3.3.1 Forward Reachability Analysis

If we do forward reachability analysis we initially store $\langle l_0, U_0 \rangle$ in the wait buffer. We then repeat the following:

Algorithm 1 *1. Pick a state $\langle l_i, U_i \rangle$ from the wait buffer.*

2. *Check if $l_i = l_f \wedge U_i \subseteq U_f$. If that is the case, return the answer* yes.
3. *If $l_i = l_j \wedge U_i \subseteq U_j$, for some $\langle l_j, U_j \rangle$ in the passed buffer, drop $\langle l_i, U_i \rangle$ and go to step 1. Otherwise save $\langle l_i, U_i \rangle$ in the passed buffer. If $U_j \subset U_i$ we can replace the state $\langle l_j, U_j \rangle$ with $\langle l_i, U_i \rangle$. (To save space)*
4. *Find all l_k that are reachable from l_i in one step regardless of the assignments, taking only actions into account. Let g_k be the set of guards on the performed transition and a_k the set of resets.*
5. *Now set $U_k = reset(sp(U_i) \cap g_k, a_k)$. If $U_k \neq \emptyset$, store $\langle l_k, U_k \rangle$ in the wait buffer.*
6. *If the wait buffer is not empty go to step 1, otherwise return the answer* no.

3.3.2 Backward Reachability Analysis

To simplify the algorithm description below we define an operation *inv_reset* which entirely is composed of operations defined above.

Definition 15 *Let r be a time region and a a set of resets. We define*

$$inv\_reset(r, a) = \begin{cases} free(r, a) & check(r, a) \\ \emptyset & otherwise \end{cases}$$

□

If we do backward reachability analysis we initially store $\langle l_f, U_f \rangle$ in the wait buffer. We then repeat the following:

Algorithm 2 *1. Pick a state $\langle l_i, U_i \rangle$ from the wait buffer.*

2. *Check if $l_i = l_0 \wedge U_0 \subseteq U_i$. If that is the case, return the answer* yes.
3. *If $l_i = l_j \wedge U_i \subseteq U_j$, for some $\langle l_j, U_j \rangle$ in the passed buffer, drop $\langle l_i, U_i \rangle$ and go to step 1. Otherwise save $\langle l_i, U_i \rangle$ in the passed buffer. If $U_j \subset U_i$ we can replace the state $\langle l_j, U_j \rangle$ with $\langle l_i, U_i \rangle$.*
4. *Find all l_k that leads to l_i in one step regardless of the assignments, taking only actions into account. Let g_k be the set of guards on the performed transition and a_k the set of resets.*

5. *Now set* $U_k = inv\_reset(wp(U_i) \cap g_k, a_k)$. *If* $U_k \neq \emptyset$ *store* $\langle l_k, U_k \rangle$ *in the wait buffer.*

6. *If the wait buffer is not empty go to step 1, otherwise return the answer* no.

Everything treated in this section will be discussed in greater detail later, when the implementation is described.

Chapter 4

UPPAAL

4.1 Overview

UPPAAL is a set of tools for automatic verification of safety and bounded liveness properties of real–time systems, modeled as networks of timed automata or simple linear hybrid systems[1]. The toolkit is implemented in C and C++, and it runs on several different types of UNIX–systems, including SunOS, Linux and HP–UX.

The functions of the toolkit are implemented in terms of modules. To simplify the connection of modules there are some shell-scripts, *e.g.* `atg2hs2ta` (connecting `atg2ta` and `hs2ta`). There is also a shell-script working as an integrated environment for verification.

The basic structure of the toolkit is shown in figure 4.1.

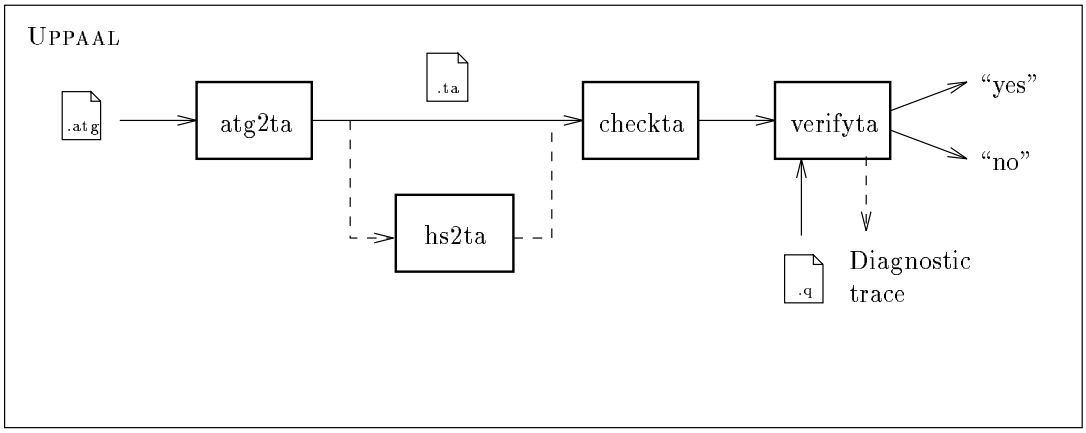

Figure 4.1: An overview of UPPAAL

[1] UPPAAL is capable of transforming some simple classes of linear hybrid systems into timed automata.

4.2 Functionality

4.2.1 `atg2ta`

The `atg2ta` program is responsible for transforming graphical descriptions in Autograph to the textual format. In drawing the graphical description the following rules should be followed:

- The different processes of a system must be enclosed in boxes, with the name of the process as a structural label.
- All states in each process must have a name associated. The name can be given in any visible label assigned to the state. The state names of a process are local, and can be reused in other processes.
- All transitions must be between states in the same process. Transitions between processes are not allowed.
- Conditions on the transitions can be given in any visible label for that transition.
- There must be a box describing the system configuration. It is a box with the word "config" as structural label, and declarations of the variables and system configuration, in the behavioral label. The declarations and system configuration should be given on the form described in the textual description format.
- A box containing no states, and without "config" as structural label, is considered as a comment, and is ignored by the translation program.

During the conversion `atg2ta` does only minimal syntax checks, to allow both timed automata and hybrid systems to be described using basically the same graphical syntax.

4.2.2 `hs2ta`

This program is capable of transforming simple linear hybrid systems into networks of timed automata. The systems handled belong to a subclass of $\mathrm{LHS}_{\neq 0}$, where the rates of the hybrid variables are independent of the state of the system, a class of system sometimes refered to as multirate automata. The transformation is performed using an algorithm described by Sifakis *et. al.* [AOY94].

4.2.3 `checkta`

The `checkta` program is a program for checking the syntax of a description given in the textual format. The description to check must be a timed automaton, if it is a hybrid system, it must first be transformed using `hs2ta`.

The errors caught by checkta are, except for structural errors that violates the grammar[2]:

[2] A context free grammar over the textual description format is shown in table 5.1

- That all datatypes used are supported. (Today, the only supported datatypes are clocks and integers.)
- That all variables and channels are declared.
- That the processes have unique names.
- That all states in the processes are declared, and that all processes have an initial state.
- That both source and destination of the transitions are states, and that the states are declared in the same process as the transition. Note that state declarations are local to the process, so the same state name can be reused in other processes.
- That the guard and assign labels on the transitions are well formed.
- That only declared processes are used in the system.

4.2.4 `verifyta`

The program **`verifyta`** is the verification engine of the UPPAAL toolkit. It takes a description and a property, and gives "yes" or "no" as an answer. As an option, it is also capable of showing an example trace that violates, or confirms the property.

The verification is performed in a backwards manner, *i.e.* the verifier starts in a symbolic state of the system, that satisfies the property (or its negation), and tries to find its way back to the initial state. The choice of states to start the verification in, is determined by the structure of the property. If the property is of the type $\exists\Diamond P$ then the symbolic states satisfying the property are chosen as start states, and if we find a way back to the initial state the property is satisfied. If we, on the other hand, have a property of the type $\forall\Box P$ we start in the symbolic states satisfying the negation of the property, and if we find a way back to the initial state, we know that the property is violated.

Chapter 5

Implementation

5.1 User Interface

5.1.1 Graphical interface

The graphical interface is implemented based on Autograph[1].

Autograph is capable of saving the drawn automata, in three different formats; `fc2`–format, `atg`–format, and `postscript`–format. Since the `fc2`–format is to limited for us to use, and the `postscript`–format is far to complex, we use the `atg`–format.

Given a file on `atg`–format, a part of the tool extracts all vital information, and translates it into our own textual description language. In the translation procedure, we do not pay attention to syntactical errors. This is to simplify the translation part, and to allow further extensions to the graphical "language".

An example of a system described in the graphical format is shown in figure 5.1

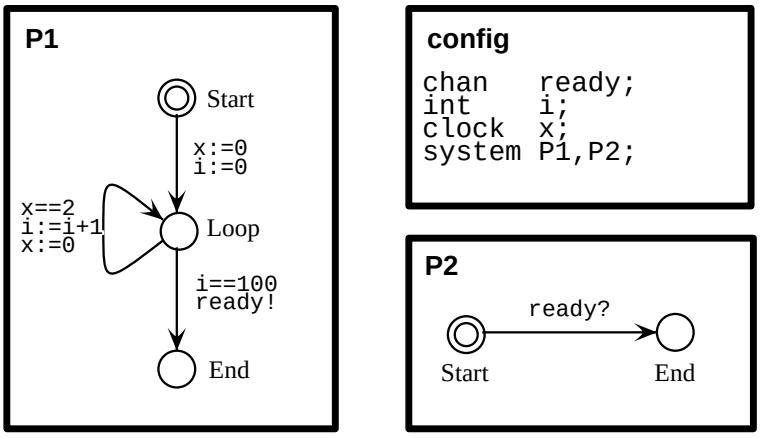

Figure 5.1: An example of a system described in the graphical format.

[1] Autograph is a tool for drawing automata, developed at CMA, France.

Ita	→	*VarList ProcList Globals*
VarList	→	ϵ | *Channel VarList* | *Var VarList*
ProcList	→	*Proc* | *Proc ProcList*
Globals	→	`system` *IdList* `;` *OpHide* | *OpHide* `system` *IdList* `;`
Channel	→	`chan` *IdList* `;`
Var	→	*Type IdList* `;`
Proc	→	`process` *Id* `{` *ProcBody* `}`
IdList	→	*Id* | *Id* `,` *IdList*
OpHide	→	ϵ |`hide` *IdList* `;`
ProcBody	→	*StateDecls TransDecls*
StateDecls	→	`state` *IdList* `;` `init` *Id* `;` | `state` *IdList* `;` `init` *Id* `;` `final` *Id* `;`
Transdecls	→	`trans` *TransList* `;`
TransList	→	*Trans* | *Trans* `,` *TransList*
Trans	→	*Id* `->` *Id* `{` *OpGuard OpSync OpAssign* `}`
OpGuard	→	ϵ | `guard` *GuardList* `;`
OpSync	→	ϵ | *Id* `!` | *Id* `?`
OpAssign	→	ϵ | `assign` *AssignList* `;`
GuardList	→	*Guard* | *Guard* `,` *GuardList*
AssignList	→	*Assign* | *Assign* `,` *AssignList*
Type	→	`clock` | `int` | ...
Guard	→	*ClockGuard* | *IntGuard* | ...
Assign	→	*ClockAssign* | *IntAssign* | ...
ClockGuard	→	*Id RelOp Nat*
IntGuard	→	*Id RelOp Int*
ClockAssign	→	*Id* `:=` *Nat*
IntAssign	→	*Id* `:=` *IntExpr*
IntExpr	→	*Int* `*` *Id* + *Nat* | *Int* `*` *Id* - *Nat* | *Id* + *Nat* | *Id* - *Nat* | *Id* | *Int*
RelOp	→	`<=` | `>=` | `==`
Id	→	*Alpha* | *Id AlphaNum*
Nat	→	*Num* | *Num Nat*
Int	→	*Nat* | `-` *Nat*
Alpha	→	`A` | ... | `Z` | `a` | ... | `z`
Num	→	`0` | ... | `9`
AlphaNum	→	*Alpha* | *Num* | `_`

Table 5.1: Context free grammar for the textual description language.

5.1.2 Textual Description Language

The language

The textual description language is designed to be a simple textual representation of networks of timed automata. It also serves as a simple programming language for Timed Automata.

The language is defined by the context free grammar given in table 5.1.

5.1.3 Property Description Language

The Language

The logic used to describe properties, in the tool, is a simple logic, to specify properties such as "Can some state be reached from the initial state ?" or "Can we be sure that some state is not reachable from the initial state ?".

The syntax of the property description language is defined by the context free grammar given in table 5.2

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

Table 5.2: Context free grammar describing the property description language

Parsing of the Logic

The properties are parsed using a parser generated using Flex and Bison, the wildcards are expanded, and an expression tree is built.

Before we can deliver the properties to the verifier they have to be transformed to disjunctive normal form. That is done automatically in a very naïve way, following the transformation–rules in table 5.3. Atomic propositions are either propositions telling that one process is in a specified state (*e.g.* $P_1.S_1$), or simple constraints on the variables or clocks (*e.g.* $c_1 \leq 5$).

After the transformation, we will have an expression, either of the form $\exists\Diamond P$, or of the form $\neg\exists\Diamond P$, where P is a disjunct of conjuncted atomic properties (*i.e.* P has the form $(a_{1,1} \wedge \ldots \wedge a_{1,n}) \vee \ldots \vee (a_{m,1} \wedge \ldots \wedge a_{m,q})$. $a_{i,j}$ are atomic propositions.).

Verifying Properties

Given a property in the form $A_1 \vee \cdots \vee A_n$, each A_i is used to build a symbolic final state for the system. Those states are then delivered, one by one, to the verifier, until a reachable one is found, or until they all have been tested.

If one of the final states built are reachable we can stop our verification process, since we now have an example (in case of a $\exists\Diamond P$ property) or a counter–example (in case of a $\neg\exists\Diamond P$ property).

$Nf(P)$	$=$	P	if P is atomic
$Nf(\neg P)$	$=$	$\neg P$	if P is atomic
$Nf(\neg(\neg P))$	$=$	$Nf(P)$	
$Nf(\neg(P_1 \wedge P_2))$	$=$	$Nf(\neg P_1) \vee Nf(\neg P_2)$	
$Nf(\neg(P_1 \vee P_2))$	$=$	$Nf(\neg P_1 \wedge \neg P_2)$	
$Nf(P_1 \vee P_2)$	$=$	$Nf(P_1) \vee Nf(P_2)$	
$Nf((P_1 \vee P_2) \wedge (P_3 \vee P_4))$	$=$	$Nf(P_1 \wedge P_3) \vee Nf(P_1 \wedge P_4) \vee Nf(P_2 \wedge P_3) \vee Nf(P_2 \wedge P_4)$	
$Nf(P_1 \wedge (P_2 \vee P_3))$	$=$	$Nf(P_1 \wedge P_2) \vee Nf(P_1 \wedge P_3)$	
$Nf((P_1 \vee P_2) \wedge P_3)$	$=$	$Nf(P_1 \wedge P_3) \vee Nf(P_2 \wedge P_3)$	
$Nf(P_1 \wedge P_2)$	$=$	$Nf(Nf(P_1) \wedge Nf(P_2))$	If none of the rules above are applicable
$Nf(\exists\Diamond P)$	$=$	$\exists\Diamond Nf(P)$	
$Nf(\forall\Box P)$	$=$	$\neg\exists\Diamond Nf(\neg P)$	

Table 5.3: Transformation–rules for Transforming Expressions to Disjunctive Normal Form

5.2 The Constraint Solvers

5.2.1 The Clock Constraint Solver

Representation

A region is a constraint system describing all values that are allowed for the clocks in the system. To represent regions, we only need two types of constraints: $l_i \leq x_i \leq u_i$ and $l_{i,j} \leq x_i - x_j \leq u_{i,j}$. x_i and x_j are clocks, $l_i, u_i, l_{i,j}$ and $u_{i,j}$ are integer constants. No other form of clock constraints can occur. The first thing we notice is that we don't need $l_{i,j}$. They can be removed since $l_{i,j} = -u_{j,i}$. If the system has n clocks the indices i and j will range from 1 to n. We can represent the constraints in a $n \times n$-matrix containing the upper bounds and a n-dimensional vector containing the lower bounds. The matrix element (i, j) contains $u_{i,j}$, the upper bound for $x_i - x_j$ when $i \neq j$, and the ith element of the vector contains l_i, the lower bound for x_i. We use the diagonal elements (i, i) to store u_i, the upper bound for x_i.

However, there's a more uniform representation which makes algorithm design and implementation much simpler. Instead of using a $n \times n$-matrix we will use a $(n+1) \times (n+1)$-matrix. The indices i and j now ranges from 0 to n and the extra elements $(i, 0) = u_{i,0}$ now contains the upper bounds for $x_i - 0 = x_i$. By analogy we let the elements $(0, j) = u_{0,j}$ contain the upper bounds for $0 - x_j = -x_j$. Now we don't need the n-dimensional vector any longer. The reason is that $u_{0,i} = -l_i$ Now we're able to write all clock constraint in a uniform way, namely on the form $x_i - x_j \leq u_{i,j}$ where i and j ranges from 0 to n and $i \neq j$, keeping in mind that $x_0 = 0$ and the system still has n clocks x_1 through x_n.

Before we can close the discussion of representation there's one thing that remains to be solved. How do we represent the fact that there is no restriction for a given clock or clock difference. This must be done by reserving a special value to put in the matrix. In order to choose an appropriate value we have to find what values are allowed for $u_{i,j}$. Clock values are always non-negative so $u_{0,j} \leq 0$ and $u_{i,0} \geq 0$. The other bounds $u_{i,j}, i = 1, ..., n, j = 1, ..., n, i \neq j$ have no such restrictions, they can range from $-\infty$ to $+\infty$. We choose a large integer, much larger than the maximum upper bound to be used, and let it represent infinity. We denote this integer INF and treat it like ∞ so that $INF \pm n = INF$ for any integer n. We will use $-INF$ as $-\infty$.

Note that the matrix representation contains some redundancy because we don't use the matrix

diagonal. According to our interpretation these elements shall contain the upper bounds for $x_i - x_i = 0$ which clearly is zero; it even holds for our definition of x_0. Despite of that, we store them to make the algorithms a bit easier and faster. If not storing them, we will save some memory but lose some time due to extra tests needed when reading and writing matrix elements. These elements will also be used to indicate some properties about the constraint systems. The problem now is to find algorithms that given matrices of the type described above return matrices of the same type corresponding to constraint systems describing the clock assignments after the clock constraint operations in chapter 3.

Conjuncting Guards

One operation that we'll need quite frequently is to conjunct a clock guard and determine if the constraint system still is consistent, that is if the constraints contain no contradictions. A contradiction arise when a lower bound of a clock or clock difference is greater than the corresponding upper bound. A consistent constraint system must satisfy $u_{i,j} + u_{j,i} \geq 0$ for all i and j. This condition can be intuitively understood if one observe that the quantity $u_{i,j} + u_{j,i}$ is the length of the allowed interval for $x_i - x_j$ which is greater than or equal to zero in a consistent constraint system. The guards allowed in the system are of the forms $x_i \geq l_i$ and $x_i \leq u_i$

or written in the form developed in the previous subsection $-x_i \leq -l_i = u_{0,i}, x_i \leq u_i = u_{i,0}$. There is a simple way to conjunct such a guard. Assume that we want to conjunct a guard of the form $l \leq x_i \leq u$ and let $u'_{i,0}$ and $u'_{0,i}$ be the current value of $u_{i,0}$ and $u_{0,i}$ respectively. Let $min(x, y)$ denote the minimum of the two integers x and y. The idea is to set $u_{i,0} = min(u'_{i,0}, u)$ and $u_{0,i} = min(u'_{0,i}, -l)$. This will of course yield a correct constraint system but it can cause some problems as we'll see in the examples below. We will use this method anyway because a guard can be conjuncted in constant time. After the conjunction the constraint system can be transformed to a representation that doesn't cause us any problems. The next example will show what we mean when we talk about different representations for a given constraint system.

In the following examples we will not use the representation of constraints that is used in the implementation. Instead we will use a more familiar representation which is easier to read.

Example 1 *Assume that we want to conjunct the guard* $0 \leq x_2 \leq 2$ *to the constraint system*

$$0 \leq x_1 \leq 4, 1 \leq x_2 \leq 3, x_1 - x_2 \leq 0, x_2 - x_1 \leq 0$$

After the conjunction the constraint system has changed to

$$0 \leq x_1 \leq 4, 1 \leq x_2 \leq 2, x_1 - x_2 \leq 0, x_2 - x_1 \leq 0 \tag{5.1}$$

This representation does not have the lowest possible upper bounds but it is correct. The last two constraints in 5.1 are simplified to $x_1 = x_2$ *and the constraint system can be written as*

$$1 \leq x_1 \leq 2, 1 \leq x_2 \leq 2, x_1 - x_2 \leq 0, x_2 - x_1 \leq 0 \tag{5.2}$$

The constraint systems 5.1 and 5.2 are in fact equivalent but the representation in 5.2 is much more useful and does not cause the problems associated with the representation in 5.1.

If we add an inconsistent guard we will definitely get an inconsistent constraint system but that can happen even if the conjuncted guard is consistent. In order to apply the syntactic definition of inconsistency we must have the constraint system represented as in 5.2 and not as in 5.1.

Let r_1 be the constraint system in the example above after the guard conjunction. We will call 5.2 the closed representation of r_1 and 5.1 an open representation of r_1. A closed constraint system is a representation where all upper bounds are as low as possible still containing the same set of solutions. A solution is a vector that satisfies the constraint system, in this case a clock assignment.

Example 2 *Let us now conjunct the guard $x_1 = 3$ to r_1. If we use the representation in 5.1 we get*

$$3 \leq x_1 \leq 3, 1 \leq x_2 \leq 2, x_1 - x_2 \leq 0, x_2 - x_1 \leq 0 \tag{5.3}$$

The representation in 5.2 yields

$$3 \leq x_1 \leq 2, 1 \leq x_2 \leq 2, x_1 - x_2 \leq 0, x_2 - x_1 \leq -1 \tag{5.4}$$

One immediately discovers the inconsistency in 5.4 but it's much more difficult to detect the inconsistency in 5.3.

Another problem arises when we look at relationships between regions.

Example 3 *Let r_1 be as before and define r_2 to be the region*

$$0 \leq x_1 \leq 5, 1 \leq x_2 \leq 2, x_1 - x_2 \leq 0, x_2 - x_1 \leq 0$$

If we apply the relation algorithm to check implication between constraint systems, described later, on r_2 and 5.1 we will find that the solution set of 5.1 is included in r_2. If we close r_2 and apply the algorithm on 5.1 and r_2 we see that the solution set of r_2 is included in 5.1. If we finally apply the algorithm on 5.2 and the closed representation of r_2 we will get the correct answer, r_1 and r_2 are equivalent.

Definition 16 *Two constraint systems are equivalent if they have the same set of solutions.* □

The property that a constraint system is closed is very important for our further work. We postpone the treatment of how to check for inconsistency of a region for a while and first discuss how to find the closed representation of a constraint system. We also save the discussion of the relation algorithm until later.

Closing the Constraint System

As mentioned in the previous subsection the concept of closed constraint systems is very important. We therefore need an efficient algorithm to close a region. As noticed we also talked about "an open representation" and "the closed representation" of a constraint system. This indicates that there are many open representations for a given constraint system but only one closed. In fact, there are infinitely many open representations because all upper bounds $u_{i,0}$ and some upper bounds $u_{i,j}$ have the property that if one adds an arbitrary positive integer, the constraint system remains consistent and we get a new open representation of the same region.

Proposition 1 *The closed representation of a constraint system is unique.*

Proof To prove that the closed representation is unique we assume that we have two different closed representations, with upper bounds $u_{i,j}$ and $u'_{i,j}$, of a given region. If the representations are different there must exist at least one i and j for which $u_{i,j} \neq u'_{i,j}$. In a closed representation however each upper bound shall be as low as possible so only the minimum of these two upper bounds belongs to the closed representation, the other belongs to an open. To avoid the contradiction that one representation can be both open and closed we must conclude that our assumption made above was wrong and hence the closed representation is unique. □

It is now possible to give an alternative definition of what we mean when we say that two constraint systems are equivalent.

Definition 17 *Two constraint systems are equivalent if they have the same closed representation, that is if all corresponding upper bounds are equal.* □

When finding the lowest possible upper bound in a constraint system many candidates of upper bounds must be examined. For example in a system with only three clocks, finding the correct value of $u_{1,0}$ would require us to compare it to each of the candidates $u_{1,2}+u_{2,0}$, $u_{1,3}+u_{3,0}$, $u_{1,2}+u_{2,3}+u_{3,0}$ and $u_{1,3}+u_{3,2}+u_{2,0}$. These bounds may in turn depend on other bounds not yet computed which require us to determine in which order to compute the bounds if such a partial ordering exists. There is no efficient way to do that.

The solution is to model the constraint system as a weighted graph as described in [YL93]. We have one node for each clock, including x_0, and an edge from node i to node j labeled by $u_{i,j}$. That's the main reason for why we chose the representation as a matrix with upper bounds. It is in fact a way to represent a weighted graph. Another way of representing the region or a weighted graph is as a linked list. However our regions often consist of too many constraints to use the list representation. The different candidates of upper bounds $u_{i,j}$ now correspond to sums of weights along paths between node i and j. The upper bounds in the closed representation are the shortest paths, the paths with the lowest sum of weights. If we store all shortest paths in the matrix we get a matrix corresponding to the closed representation of the original region.

Theorem 1 *The upper bounds $u_{i,j}$ corresponding to the shortest paths from node i to node j are the upper bounds of the closed representation of the original constraint system.*

Before we prove this theorem we need a lemma.

Lemma 1 *In a constraint system with upper bounds $u_{i,j}$ obtained from the shortest path algorithm a clock assignment $\vec{x}$ with components $x_i = u_{i,0}$ and $x_i = u_{0,i}$ are solutions.*

Proof Let $x_i = u_{i,0}$. Of course $u_{i,0} \leq u_{i,0}$ and $-u_{i,0} \leq u_{0,i}$. What remains to prove is that $x_i - x_j = u_{i,0} - u_{j,0} \leq u_{i,j}$. This follows directly because the upper bounds are the shortest paths and hence satisfies

$$u_{i,0} \leq u_{i,j} + u_{j,0} \Leftrightarrow u_{i,0} - u_{j,0} \leq u_{i,j}$$

The second part of the proof, to show that $x_i = -u_{0,i}$ also is a solution to the constraint system, is similar and is not shown here. □

We are now ready to give the proof for the theorem.

Proof To prove that the matrix containing all shortest paths represents the closed original constraint system we must prove two things: We must prove that the obtained constraint system is equivalent to the original one and that the obtained upper bounds are as low as possible.

The first thing is obvious because we only examine the paths in the graph, or different combinations of constraints, to discover the one with smallest sum of weights. We therefor don't lose any solutions not present in the original constraint system, or add any extra solutions not present there. Hence the set of solutions is preserved.

To prove that the shortest path really corresponds to the upper bound of the closed representation, the constraint system has the tightest possible bounds, is more difficult: Assume that we have a clock assignment $\vec{x}$ satisfying $x_i - x_j = u_{i,j}$ without any special restrictions on i and j except that they are less than or equal to n. If this assignment is a solution we know that the constraint system is closed, because if we lower $u_{i,j}$ we will lose that solution.

We can assume that the constraint system is consistent and we select a x_i so that $x_i \leq u_{i,0}, -x_i \leq u_{0,i}$. If $x_j = x_i - u_{i,j}$ satisfies the constraint system we are ready. To prove that $x_j \leq u_{j,0}$ we observe that

$$x_j = x_i - u_{i,j} \leq u_{i,0} - u_{i,j} \leq u_{i,j} + u_{j,0} - u_{i,j} = u_{j,0}$$

The inequality $u_{i,0} \leq u_{i,j} + u_{j,0}$ is a direct consequence of the shortest path algorithm. It simply says that since we have all shortest paths we know that the path direct from node i to node 0 must be shorter than the path from node i to node 0 via node j.

To prove that $-x_j \leq u_{0,j}$ requires a little trick: The proof idea is to choose a x_i that satisfies the constraint system and then show that x_j also does so. Let us choose $x_i = u_{i,0}$ which according to the lemma satisfies the constraint system. We can now easily show that

$$-x_j = u_{i,j} - x_i \leq u_{i,0} + u_{0,j} - x_i = u_{i,0} + u_{0,j} - u_{i,0} = u_{0,j}$$

Note that the inequality $u_{i,j} \leq u_{i,0} + u_{0,j}$ is a direct consequence of the shortest path algorithm. This completes the proof that the shortest path algorithm gives a closed constraint system. □

Corollary 1 *When a constraint system is modeled as a graph which is processed by the shortest-path algorithm, consistency is preserved.*

There's an efficient way of finding all shortest paths for a weighted graph represented as a matrix. We will use Floyd's algorithm [Flo62] [2]. The matrix first contains weights telling us the lengths of the paths from node i to node j without visiting any other node. We now compare these paths from node i to node j with the paths we get if we are allowed to visit node 0 when moving from node i to node j and if we found any shorter path we store that one instead. In that way the matrix element (i, j) always contains the best path from node i to node j encountered so far. We repeat the procedure but now we're allowed to move from node i to j via node 1 and so on. We continue in this way, visiting a node with a higher index each time, until the procedure has been repeated for all nodes and the algorithm terminates with a matrix containing all shortest paths and hence a closed representation of the constraint system. We show some pseudo-code below.

Algorithm 3

- *For every node k do*
  - *For every node i do*
    - *For every node j do*

[2] Dijkstra's algorithm will be more inefficient because the graphs are often very dense.

- *Set* $u_{i,j} = min(u_{i,j}, u_{i,k} + u_{k,j})$

If the system has n clocks we have $n+1$ nodes in the graph and therefore $(n+1)^2$ paths yielding an algorithm with time complexity $O(n^3)$. But there is still one remaining problem to solve.

There's one requirement that the graph must satisfy when using Floyd's algorithm, or any shortest-path algorithm, to obtain a correct answer. The graph must not contain cycles with a negative sum of weights. If it does, the concept of "shortest path" will no longer have any meaningful interpretation since one can cycle around making a path arbitrarily short. The algorithm will still terminate even if the graph contains such cycles because it's totally deterministic but the result will be useless. How do we know that our weighted graph, modeling the constraint system, satisfies this property?

In order to answer this question we look at the sum of weights along the shortest cycle from a node i back to itself because if that cycle has a non-negative sum of weights all other cycles have that too. This cycle is easy to find after the shortest-path algorithm has been applied. Now when the matrix contains the shortest paths the shortest cycle from node i and back to itself must be one from node i, to another node j and directly back to i again because if there is a shorter one going via node k from i to node j then the path from node i to node j, stored in the matrix, would not be the shortest which leads to a contradiction. The sum of weights along these cycles are given by $u_{i,j} + u_{j,i}$ so the shortest such cycle has a sum of weights which is

$$min_{i,j}(u_{i,j} + u_{j,i}) = min_i(u_{i,i})$$

We have no cycles with a negative sum of weights if this quantity is greater than or equal to zero and that's precisely the condition that must hold for consistent constraint systems. According to the corollary above, consistency is preserved, So we can look in the matrix containing the shortest paths and draw conclusions about the original constraint system. This shows us that a graph, modeling a consistent constraint system contains no negative cycles and hence satisfies the condition required. It doesn't matter if the algorithm gives an incorrect answer for inconsistent constraint systems.

We have almost derived an algorithm to determine if a given constraint system is consistent or not and we explain it in detail, together with some improvements in the next subsection.

Checking for Consistency

This subsection will be a very short one since most of the work has already been done in the previous subsection. However the algorithm is placed in a separate subsection for readability. In an earlier subsection we came to the conclusion that a necessary condition for a closed constraint system to be consistent is $u_{i,j} + u_{j,i} \geq 0$. This property shall hold for all i and j. This condition is also sufficient because no other kind of contradictions can occur. This condition only holds for closed constraint systems. This yields an algorithm to check consistency for constraint systems with time complexity $O(n\ 2)$ where n is the number of clocks in the system. However, it is enough to look at $min_{i,j}(u_{i,j} + u_{j,i}) = min_i(u_{i,i})$ and see if it is greater than or equal to zero or not. This yields an algorithm with complexity $O(n)$.

However, we can do much better in this case because we only get inconsistencies after guard conjunctions. This implies that if the constraint system is inconsistent there exists at least one k such that $u_{0,k} + u_{k,0} < 0$. After the application of the closing algorithm we know that $u_{0,0} = min_k(u_{0,k} + u_{k,0})$ Hence, it's enough to examine $u_{0,0}$ in order to determine consistency. We now have an algorithm with time complexity $O(1)$.

To evaluate a clock guard with respect to a given global state, we do the following steps:

1. Conjunct the guard to the region in the global state.
2. Close the region.
3. Check if the resulting region is consistent. Return true if the constraint system is consistent, false otherwise.

The above is a description of a general evaluation procedure for evaluating clock guards and it is entirely composed of primitive operations, implemented and described above. It's not a precise description of the implementation used in the tool. When evaluating the guards belonging to a transition all of them must be satisfied. It's therefore a waste of time to do the last two steps after each conjunction. Even if we can save some guard conjunctions if we discover an inconsistency we must remember that the time of a guard conjunction is almost negligible compared to a close operation. Instead these two steps are done only once after all guard conjunctions. Here we clearly see a trade-off between generality and speed. Since this evaluation operation is done frequently during a verification, and the general evaluation procedure is easy to implement guided by the steps above, we determined that time was more important than generality in this case.

The Weakest Precondition

The weakest precondition deals with what happens when time flows. Thus, we define the weakest precondition of a region r as the region containing all clock assignments which when time flows after some finite delay will lead to a clock assignment in r. We denote that region r' so that $r' = wp(r)$. This can be expressed mathematically as if $\vec{z} \in r'$ and $\vec{x} \in r$ then $\vec{x} = \vec{z} + \vec{d}$ where $\vec{z}$ and $\vec{x}$ are clock assignments with components z_i and x_i respectively. The vector $\vec{d}$, with components d_i, is the delay required for the clock assignment $\vec{z}$ to lead to clock assignment $\vec{x}$. Since all clock rates are the same we have $d_i = d$ and r' consists of exactly those z_i for which $z_i + d = x_i$ for some finite delay d, $d = 0$ is allowed. How do we find the constraint system describing r'?

Assume that r' is a constraint system of the form $x_i - x_j \leq u'_{i,j}$ and r is on the form $x_i - x_j \leq u_{i,j}$. Intuitively we want to find as low lower bounds for the clocks as possible [3]. This can be obtained by setting $u'_{0,i} = 0$ and then close r' to get the correct lower bounds because no lower bound can be less than zero. Before we do that however we must ensure that the clocks don't drift away. We must remember that all clock rates are the same.

Example 4 *If we have an assignment $x_1 = 2, x_2 = 4$ then all previous assignments $\vec{z} \in r'$ leading to that assignment must have $z_1 - z_2 = -2$. The other constraints describing r' is $0 \leq z_1 \leq 2, 2 \leq z_2 \leq 4$.*

Note that if the previous value of $u_{1,2}$ is less than -2 we must keep that one instead so what we actually say is that the maximum difference between z_i and z_j is the smallest of $u_{i,j}$ and the difference between the maximum value of x_i and the minimum value of x_j, $u'_{i,j} = min(u_{i,j}, u_{i,0} + u_{0,j})$. However, if r is closed we already have the correct values for $u'_{i,j}$, $u'_{i,j} = u_{i,j}$.

Algorithm 4 *Let r be a closed constraint system of the form $x_i - x_j \leq u_{i,j}$ and $r' = wp(r)$ of the form $x_i - x_j \leq u'_{i,j}$. An open representation of r' can be obtained in the following way:*

[3] Because we relax the lower bounds the weakest precondition operation is sometimes called the down operation

- *Set* $u'_{i,0} = u_{i,0}$ *for all* i.
- *Set* $u'_{0,j} = u_{0,j}$ *for all* j.
- *Set* $u'_{i,j} = u_{i,j}$ *for all other pairs of* i *and* j.

Theorem 2 *The constraint system obtained with the algorithm above is the weakest precondition of the original constraint system.*

Proof In order to prove the above theorem we need to prove two things. We must prove that r' is big enough so that all $\vec{x} \in r$ are reachable if some finite delay d is added to some $\vec{z} \in r'$ [4]. We must also prove that r' isn't too big so that each $\vec{z} \in r'$ after some finite delay d leads to a clock assignment $\vec{x} \in r$.

The proof that r' is big enough is trivial: We can write all clock assignments $\vec{z}$ that after some delay $\vec{d}$ will lead to a clock assignment $\vec{x}$ as $\vec{x} - \vec{d}$. For all d we have

$$x_i - x_j = (x_i - d) - (x_j - d) = z_i - z_j \leq u'_{i,j} = u_{i,j}$$

Because $d \geq 0$ we must also have

$$-z_i = d - x_i \leq u'_{0,i} \Rightarrow -x_i \leq u'_{0,i}$$

We also know that $x_i \leq u_{i,0} = u'_{i,0}$.

Because z_i is a clock value and hence is non-negative we observe that $d \leq min_i(x_i)$. These observations are the complete proof that r' is big enough.

To prove that r' isn't too big is a little bit harder. We start with a clock assignment $\vec{z} \in r'$ and try to find a $\vec{d}$ such that $\vec{z} + \vec{d} = \vec{x} \in r$. If we choose

$$d = min_i(u'_{i,0} - z_i)$$

we'll get a valid d. Let's assume that this minimum occurs when $i = 1$. This is no restriction because we can always renumber the clocks in that way. Because $z_1 \leq u'_{1,0}$ we have $d \geq 0$. We also see that $d \leq min_i(x_i)$ because $z_i \geq 0$ and $min_i(x_i) = min_i(z_i) + d$. We also note that

$$z_i - z)_j = (z_i + d) - (z_j + d) = x_i - x_j \leq u'_{i,j} = u_{i,j}$$

We also get

$$x_i = z_i + d = z_i + u'_{1,0} - z_1 \leq z_i + u'_{i,0} - z_i = u'_{i,0} = u_{i,0}$$

In order to prove that $-x_i \leq u_{0,i}$ we use the following trick: We rewrite x_i as $x_i - x_1 + x_1$. According to our choice of d this can be written as

$$x_i = (x_i - d) - (x_1 - d) + x_1 = z_i - z_1 + u'_{1,0}$$

If we rely on the assumption that r is closed we can show that

$$-x_i = z_1 - z_i - u'_{1,0} \leq u'_{1,0} + u'_{0,i} - u'_{1,0} = u'_{0,i}$$

This completes the proof that our proposed algorithm is correct. □

[4] Note that no delay is added to x_0 so in the proves that follows we must treat all cases where $i = 0$ or $j = 0$ separately

Another way of finding the weakest precondition of a region is to view it as a linear programming problem, often abbreviated lp-problem [5]. We want to find the lowest possible values for the lower bounds $-u'_{0,i}$ and these are the lowest values that the clocks x_i can have, we can find them if we minimize $\sum_i x_i$ subject to the constraints $x_i \leq u'_{i,0}, x_i - x_j \leq u'_{i,j}$. Since the clocks are non-negative the sum will assume its minimum when every x_i is as small as it can be. Geometrically one can visualize the region as a convex polytope in $\mathbf{R}^{+n}$ and the theory for linear programming states that the optimal solution always is a corner point of the polytope. We can only accept integer solutions because $u'_{0,i}$ must be an integer. We know that the corner points of this polytope have integer coordinates because $u'_{i,0}$ and $u'_{i,j}$ are all integers so we don't need branch-and-bound or cutting-plane methods.

One of the most common methods for solving such problems is the simplex method. The time performance of this method is in average case proportional to $\rho m^2 n$ where n is the number of variables, m is the number of constraints and ρ is the matrix density. The matrix used most conveniently with the simplex method has a structure different from the matrix representation used so far. Since there are n^2 constraints, we need to introduce n^2 slack variables [6]. The simplex method require an initial solution to start with and if some $u'_{i,j}$ is negative, which is very likely to be the case, one must solve an extra lp-problem to get that initial solution. This problem is solved by use of artificial variables and there will be at most $\frac{n(n-1)}{2}$ of these in a consistent constraint system. This fact holds because $u_{i,j} + u_{j,i} \geq 0$ and if $u_{i,j} < 0$ we know that $u_{j,i} > 0$. The time complexity is $O(n^6)$ because $m = n^2$ and there is merely n^2 variables. It will take some extra time to build the new matrix as well because the representation required by the simplex method would make it difficult to for example close a constraint system in an efficient way. Adapting the simplex method to our representation would slow it down even more.

The matrix is also too large and sparse to be represented as a two-dimensional array so we must represent it as linked lists which makes matrix access slower but will save much memory in this case. The matrix has n^2+1 rows; one for the coefficients of the objective function and one for each constraint. It will have one column for each variable and one containing the upper bounds, so it has at least n^2+n+1 columns and when artificial variables are used it can have up to $n^2+n+1+\frac{n(n-1)}{2}$. Since there are only one or two variables in each constraint, most of the matrix contains zeros. To see how sparse it is, let us calculate the maximum matrix density ρ_{max}. This density only occurs for matrices corresponding to lp-problems with no need for artificial variables, that is when $u'_{i,0} \geq 0$ and $u'_{i,j} \geq 0$ for all i and j. This is very unlikely for our constraints systems and the density is often much lower. If the above case would be true we already know the optimal solution, $x_i = 0$ for all i. The objective function has n coefficients, n constraints involves one variable, n^2-n constraints involves two variables and there are n^2 upper bounds and an extra element for holding the optimal value for the objective function, so there are

$$n + 1 + n + 2(n^2 - n) + n^2 = 3n^2 + 1$$

non-zero elements in the matrix [7]. The total number of elements for this matrix is $(n^2+1)(n^2+n+1)$. This yields

$$\rho_{max} = \frac{3n^2+1}{(n^2+1)(n^2+n+1)} \approx \frac{3}{n^2+n}$$

All these things make this approach worser than the algorithm first discussed.

One thing that makes this problem theoretically interesting is that most of the matrix will be unchanged for different constraint systems. Probably this can be used to calculate something in

[5] This discussion requires previous knowledge of linear programming in general and the simplex method in particular.

[6] if some constraints are equalities we don't need to introduce slack variables for them, but for simplicity we introduce slack variables for all inequalities even if some of them will be zero in the optimal solution.

[7] In fact, some of the upper bounds can be zero so this is a little overestimation

advance and maybe decrease the complexity $O(n^6)$. Anyway, we didn't use this approach in the implementation; we didn't delve any deeper into it. We still need closed constraint systems in other contexts; this more complicated approach will not beat the first algorithm neither with respect to time nor memory.

To summarize the results in this subsection we have developed an algorithm to compute the constraint system corresponding to the weakest precondition of a given region. If there are n clocks and the region is closed, the algorithm has complexity $O(n)$ since we only need to set the matrix elements $(0, i)$ to zero. If the region is open the algorithm will have complexity $O(n^2)$ because we also have to change the elements (i, j) to $(i, 0) + (0, j)$ if this sum is smaller. The reason why this can be eliminated in the first case is that this change is exactly what is done in the first pass of the shortest-path algorithm treated earlier. Both algorithms yields an open representation of the region representing the weakest precondition and in order to get the correct lower bounds one must close it. In the verification algorithm all global states except one are closed so in the implementation we explicitly close that state and use the linear algorithm. That leads to slightly better performance, especially for systems with many clocks.

Treating Resets in Backward Reachability Analysis

When the system performs a transition, there's a possibility that a value v_k is assigned to a clock x_k. This can affect all clocks in the system due to the limits of clock differences. This section will examine how a constraint system is changed when such an assignment is processed in backward reachability analysis.

First of all we must check if the given reset could lead to the constraint system we have at the moment, that is if there exists any assignments with $x_k = v_k$ is in that region. We can do that by conjuncting the guard $x_k = v_k$ to that region and see if it still is consistent, if it is, we know that the reset could have been performed earlier. We will call this operation the check operation even though it can be composed of already well-known operations [8].

If the reset was possible our next step is to determine all possible values that the clocks could have had before the assignment was performed. We will call this operation the free operation because we free x_k and leave it unrestricted [9].

Algorithm 5 *Let r be a closed constraint system of the form $x_i - x_j \leq u_{i,j}$ and x_k the clock that shall be freed. Let r' be a constraint system of the form $x_i - x_j \leq u'_{i,j}$ denoting the constraint system that results when x_k is freed. An open representation of r' can be obtained in the following way:*

- *Set $u'_{k,j} = \infty$ for all j.*
- *Set $u'_{i,k} = u_{i,0}$ for all i.*
- *Set $u'_{i,j} = u_{i,j}$ for all other pairs of i and j.*

Theorem 3 *The algorithm above gives a constraint system containing exactly the assignments that are present in the current region after the assign is freed.*

[8] When $v_k = 0$ this operation is sometimes called the borderline operation because it can be interpreted as a projection of a region onto the coordinate axes $\vec{e_j}, j \neq k$.

[9] This is in some sense also a weakest precondition operation because it gives us all values that x_k could have had before one performed the reset

Proof Let r be a constraint system of the form $x_i - x_j \leq u_{i,j}$, and r' be of the form $z_i - z_j \leq u'_{i,j}$. $\vec{x}$ be a clock assignment before the assign $x_k = v_k$ is processed and $\vec{z}$ is a clock assignment with z_k unrestricted, that is when x_k is freed. We will first prove that r is big enough.

We only change the value of x_k so we must have

$$z_i = x_i, u'_{i,j} = u_{i,j}, i \neq k, j \neq k$$

If z_k is free we must have

$$z_i - z_k \leq z_i = x_i \leq u_{i,0} = u'_{i,k}, z_k - z_i \leq z_k \leq \infty = u'_{k,i}$$

This completes the proof that r' is big enough.

To prove that r' isn't too big we need to show that one always can find a z_k such that z_i satisfies it's constraints for all values of $u'_{i,k}$ and $u'_{k,i}$. We set $x_i = z_i, i \neq k$. If $z_i - z_k \leq u'_{i,k} = u_{i,0}$ any value for $z_k > 0$ will guaranty that $x_i = z_i$ satisfies the constraints in r. If $z_k - z_i \leq u'_{k,i} = \infty$ choose $z_k \leq u'_{k,i} + z_i \geq 0$. This shows that every $\vec{z} \in r'$ can reach an $\vec{x} \in r$. □

If we save the consistency check and do it when all resets on the transition has been conjuncted as guards the check assign is an $O(1)$ operation. We don't detect an impossible assignment as early as we can but we avoid that the check operation becomes an $O(n^3)$ operation. We also save the closing of the constraint system after the free operation until all resets on a transition has been freed. This means that the free operation is of complexity $O(n)$ because there are about $2n$ upper bounds to change. If the next operation to be performed on the clocks doesn't require a closed constraint system one can omit the closing after the free operation.

The Relation Algorithm

When searching the state space of the system we need to determine if the solution sets of two constraint systems overlap or if one of them is contained in the other. A region r is included in another region r' if all assignments in r also are found in r'. r and r' overlap if there is a solution in r not found in r' and vice versa.

If both r and r' are closed there is an easy way to determine if they overlap or if one region is included in the other. This is the relation algorithm that we used in the implementation.

Algorithm 6 *Let r and r' be two closed constraint systems of the form $x_i - x_j \leq u_{i,j}$ and $x_i - x_j \leq u'_{i,j}$ respectively. To determine the relation between r and r' we can proceed in the following way:*

- *If $u_{i,j} \leq u'_{i,j}$ for all i and j then r is included in r'.*
- *If $u'_{i,j} \leq u_{i,j}$ for all i and j then r' is included in r.*
- *If neither of the above conditions holds then r and r' overlap.*

Theorem 4 *If both constraint systems are closed the above algorithm will correctly determine if they overlap or if one is included in the other.*

Proof Let $\vec{x} \in r$ and $\vec{z} \in r'$. Let us first assume that $u_{i,j} \leq u'_{i,j}$ for all i and j. Since $x_i - x_j \leq u_{i,j} \leq u' i, j$ we see that $x \in r'$.

The proof of the overlapping test can be done as follows: Assume that $u_{i,j} \leq u'_{i,j}$ for some i and j and that $u_{k,l} > u'_{k,l}$ for some k and l. If $\vec{x}$ satisfies $x_k - x_l = u_{k,l}$ and $\vec{z}$ satisfies $z_i - z_j = u'_{i,j}$ we have $\vec{x} \notin r'$ and $\vec{z} \notin r$ and thus there exist solutions in one constraint system not present in the other. This completes the proof of the relation algorithm. □

The relation algorithm examines all bounds in the matrix and hence has complexity $O(n^2)$. where n is the number of clocks. One can speed it up if one uses the fact that if some bounds have a relation saying that the regions overlap, the other bounds need not be examined. Experiments shows that such a test shall be placed so it is done once for each row or column of the matrix and not once for each comparison because this will slow the whole algorithm down more than if we omitted the optimization.

The Strongest Postcondition

We need an operation called strongest postcondition that gives a constraint system describing all clock assignments that the clock assignments in a given region will lead to, after some finite delay d [10].

Let r be a region and r' the region corresponding to the strongest postcondition of r, $r' = sp(r)$. We want to find as high values as possible for $u'_{i,0}$. This can of course be done with linear programming by maximizing $\sum_i x_i$ subject to the other constraints but the disadvantages are the same as for the weakest precondition. There's a much better way which is quite analogous to how we determined the weakest precondition.

If r isn't closed we must use the old values of $u_{i,0}$ to see if they can be used to improve any bounds; we set $u'_{i,j} = min(u_{i,j}, u_{i,0} + u_{0,j})$. This ensures that the clocks don't drift away, remember that they have the same rate. If r is closed, this step can be omitted.

Algorithm 7 *Let r be a constraint system of the form $x_i - x_j \leq u_{i,j}$. Let $r' = sp(r)$ be a constraint system of the form $x_i - x_j \leq u'_{i,j}$. If r is closed an open representation of r' can be obtained in the following way:*

- *Set $u'_{0,i} = u_{0,i}$ for all i.*
- *Set $u'_{i,0} = \infty$ for all i.*
- *Set $u'_{i,j} = u_{i,j}$ for all other pairs of i and j.*

Theorem 5 *The algorithm described above will give a constraint system representing the strongest postcondition of a given region.*

Proof Let r be $x_i - x_j \leq u_{i,j}$ and r' be $z_i - z_j \leq u'_{i,j}$. Note that no delay is added to x_0 so bounds where $i = 0$ or $j = 0$ must be treated separately. We need to prove that r' is big enough and not too big. The proof will be very similar to that of the weakest precondition.

[10] This operation is sometimes called the up operation because the upper bounds are relaxed.

The proof that r' is big enough is simple: We can write all clock assignments $\vec{x}$ that after some delay $\vec{d}$ will lead to a clock assignment $\vec{z}$ as $\vec{z} - \vec{d}$. For all d we have

$$x_i - x_j = (x_i + d) - (x_j + d) = z_i - z_j \leq u'_{i,j} = u_{i,j}$$

Because $d \geq 0$ we must also have

$$z_i = x_i + d \leq u'_{i,0} \Rightarrow x_i \leq u'_{i,0}$$

We also know that $-x_i \leq u_{0,i} = u'_{0,i}$. The above is what is needed to show that r' is big enough.

To prove that r' isn't too big requires a little bit more. We start with a clock assignment $\vec{z} \in r'$ and try to find a $\vec{d}$ such that $\vec{z} - \vec{d} = \vec{x} \in r$. If we choose

$$d = min_i(u'_{0,i} + z_i)$$

we'll get a valid d. Let's assume that this minimum occurs when $i = 1$. This is no restriction because we can always renumber the clocks in that way. Because $-z_1 \leq u'_{0,1}$ we have $d \geq 0$. We also note that

$$z_i - z_j = (z_i - d) - (z_j - d) = x_i - x_j \leq u'_{i,j} = u_{i,j}$$

a.e also get

$$-x_i = d - z_i = z_1 + u'_{0,1} - z_i \leq z_i + u'_{0,i} - z_i = u'_{0,i} = u_{0,i}$$

In order to prove that $x_i \leq u_{i,0}$ we use the following trick: We rewrite x_i as $x_i - x_1 + x_1$. According to our choice of d this can be written as

$$x_i = (x_i + d) - (x_1 + d) + x_1 = z_i - z_1 - u'_{0,1}$$

If we rely on the assumption that r is closed we can show that

$$x_i = z_i - z_1 - u'_{0,1} \leq u'_{i,0} + u'_{0,1} - u'_{0,1} = u'_{i,0}$$

This completes the proof that our proposed algorithm is correct. □

The algorithm has complexity $O(n^2)$ for an open constraint system and $O(n)$ for a closed one where n is the number of clocks in the system. In an implementation most constraint systems may be closed; one usually gain in performance if one close the few open regions and use the linear algorithm. Although remember that if a closed representation is required the complexity will increase to $O(n^3)$.

The Reset Operation

We want an operation that performs a reset $x_k = v_k$. This operation will be called the reset operation [11].

The reset operation gives a constraint system containing the clock assignments that arise when a clock x_k is set to a value v_k in a given region.

Before we can set $u_{k,0}$ and $u_{0,k}$ to v_k we must check if they can be used to improve any bounds. If the region is closed we can omit this step, otherwise we set $u'_{k,j} = min(u_{k,j}, u_{k,0} + u_{0,j}), u'_{j,k} = min(u_{j,k}, u_{j,0} + u_{0,k})$

[11] The name of the operation comes from the fact that when $v_k = 0$ one usually says that a clock is reseted.

Algorithm 8 *Let r be a constraint system of the form $x_i - x_j \leq u_{i,j}$ and $x_k = v_k$ the reset to perform. Let r' be the constraint system that results after the reset and of the form $x_i - x_j \leq u'_{i,j}$. If r is closed, we can obtain an open representation of r' in the following way:*

- *Set $u'_{k,0} = u'_{0,k} = v_k$.*
- *Set $u'_{i,k} = u_{i,0} - v_k$ for all i.*
- *Set $u'_{k,j} = u_{0,j} + v_k$ for all j.*
- *Set $u'_{i,j} = u_{i,j}$ for all other pairs of i and j.*

Theorem 6 *The algorithm above gives the constraint system that results when a clock x_k in a given region is set to a value v_k.*

Proof Let r be the constraint system before the reset operation and of the form $x_i - x_j \leq u_{i,j}$. Let r' be the constraint system after the reset operation and of the form $z_i - z_j \leq u'_{i,j}$. As usual we need to prove that r' is big enough and not too big.

To prove that r' is big enough is quite straight forward: We want to prove that all $\vec{x} \in r$ with $x_k = v_k$ also belongs to r'. This is obtained by setting

$$z_k = x_k = v_k, z_i = x_i, i \neq k$$

When $i \neq k$ and $j \neq k$ we have

$$x_i - x_j = z_i - z_j \leq u'_{i,j} = u_{i,j}$$

Of course even $x_k = v_k$ is a point in r'.

$$x_i - x_k = x_i - v_k = z_i - z_k \leq u_{i,0} - v_k = u'_{i,k}$$

$$x_k - x_j = v_k - x_j = z_k - z_j \leq u_{0,j} + v_k = u'_{k,j}$$

This shows that r' is big enough.

In order to prove that r' isn't too big we must prove that all $\vec{z} \in r'$ will reach some $\vec{x} \in r$, that means there must exist a x_k, which can be set to v_k, so that all x_i satisfies the constraints in r. We first note that $x_i = z_i, i \neq k$ because we only change the value of x_k. Every x_k can be set to v_k and since r is consistent there must exist such x_k. □

Note that this operation is not a normal guard conjunction. It shall always be possible to set a clock to a given value. We therefor change $u_{k,0}$ and $u_{0,k}$ without comparing them to the old values. This implies that we must treat $u_{i,k}$ and $u_{k,j}$ in the same way. The resulting constraint system is not necessarily closed because the change of $u_{i,k}$ and $u_{k,j}$ may affect $u_{i,0}$ which in turn may improve $u_{i,j}$ and so on.

If n is the number of clocks the complexity of the reset operation is $O(n)$ for a closed constraint system and $O(n^2)$ for an open. This means that one can gain in performance if one use the linear version if many regions are closed and close the open ones before.

Adding Strict Constraints

So far, we have not been able to express the fact that a clock is strictly less or greater than a constant that is we have no support for strict constraints. This can be implemented by adding an $(n+1) \times (n+1)$-matrix with boolean flags indicating if an upper bound is strict or not. When conjuncting guards one must remember that a strict upper bound is smaller than a non-strict one with the same value. Also when estimating new upper bounds by adding other upper bounds, like what is done when a constraint system is being closed, one must remember that if one of the added upper bounds is strict the resulting upper bound is strict too.

It's in some sense a philosophical question if one can distinguish when a clock is exactly 2 or very close to 2. One can always model a system, by increasing the resolution, so that verification results obtain the preferred accuracy without these distinctions being made. These are the main reasons why strict clock constraints were not implemented. However it may become necessary to implement them in order to avoid some non-sound behaviour such as

$$\neg x \geq v \Leftrightarrow x \leq v$$

which is a wrong conclusion.

5.2.2 The Integer Constraint Solver

As mentioned earlier, the system also supports integers. They can be thought of as clocks with rate zero and therefor don't change with time. The only way to alter the value of an integer is to change it by use of resets. The outline of this section will resemble the outline of that describing the clock constraint solver. However, the operations are much easier because we have no constraints on the differences between integers. Also, because integers don't change automatically when time flows we can omit the weakest precondition and strongest postcondition operations.

Representation

An integer assignment consists of intervals showing the allowed values for each integer. An integer that is unbound can have values in the interval $[-\infty, \infty]$. If there are n integers we use two n-dimensional vectors $\vec{u}$ and $\vec{l}$ for storing the upper bound u_i and the lower bound l_i for each integer i respectively. We choose an integer, large enough, and treat it as ∞ in the way described for the clocks. In this constraint solver we also need to multiply and divide with one of the operands possibly being ∞. This representation contains no redundancy and is easy to use when designing and implementing algorithms.

Conjuncting a Guard

Conjuncting a guard to an integer constraint system is very easy. There are no constraints on integer differences and the concept of opened and closed representations are not needed. Assume that we want to conjunct a guard of the form $n_i \geq \hat{l}_i, n_i \leq \hat{u}_i$ and the allowed interval for n_i is $[l_i, u_i]$. Let the new interval for n_i be $[l'_i, u'_i]$. We now use the same method as when we conjuncted clock guards so we set

$$l'_i = max(l_i, \hat{l}_i), u'_i = min(u_i, \hat{u}_i)$$

Other integers will not be affected so the guard conjunction can be done in constant time. The only inconsistency that can arise is then $l_i' > u_i'$ and that consistency can be detected in constant time when the guard is conjuncted.

The evaluation of an integer guard with respect to a given assignment is an $O(1)$ operation made in the following two steps:

1. Conjunct the integer guard to the integer assignment in the global state.
2. Return true if no inconsistency has arisen, else return false.

Treating Resets in Backward Analysis

An integer reset can have the form $n_i = v_1 n_i + v_2$ and is therefor more general than a clock reset. If we know that n_i after the assign is in the interval $[l_i, u_i]$ we want to calculate the interval for n_i before the reset was made. Let this interval be $[l_i', u_i']$. It is easy to check by simple algebraic manipulations that

$$l_i' = \frac{l_i - v_2}{v_1}, u_i' = \frac{u_i - v_2}{v_1}$$

This is true if $v_1 > 0$; if $v_1 < 0$ then l_i' and u_i' must be exchanged. If $v_1 = 0$ the reset takes the form $n_i = v_2$ and must be treated separately.

However, we must have integer values for u_i' and l_i' so above expressions are correct if $v_1 \mid (u_i - v_2)$ and $v_1 \mid (l_i - v_2)$. If that is not the case then l_i' shall be the nearest integer above the rational number above and u_i' the nearest integer below.

It is possible that such an integer interval does not exist. For example, no integers n_i satisfies

$$2 \leq 5n_i + 1 \leq 5$$

because we must choose $l_i' = 1$ and $u_i' = 0$.

In analogy with the treatment of clock resets we need a check and a free operation. The check operation simply computes values for l_i' and u_i' and if the interval is consistent, $l_i' \leq u_i'$, the reset is possible. The free operation is used to determine the largest possible interval for the integer before the reset was performed. But that interval was determined before when the check operation was performed so in this case we don't need any free operation.

When $v_1 = 0$ the operations must be designed in a different way. Because the reset now looks like a clock reset we expect that the operations will work in the same way as in the clock constraint solver. The check operation will conjunct a guard of the form $n_i = v_2$ to the constraint system and see if it still is consistent. If it is, we know that the reset leads to a consistent constraint system and therefor is possible. The free operation now must set $l_i' = -\infty$ and $u_i' = \infty$ because the value assigned to n_i is independent of the value n_i had before. This means that we know nothing about l_i' or u_i' and must assume as large interval as possible.

Both operations have complexity $O(1)$ and are thus faster than their counterparts in the clock constraint solver. This is true for all operations in the integer constraint solver because an integer's value is independent of the values for the other integers.

The Relation Algorithm

When we search the state space we must be able to examine the relationship between two integer constraint systems. We need to determine if the solution set of one of them is included in the other or if they are disjoint or overlap. We will use essentially the same method as for the clock constraint solver.

Algorithm 9 *Let r and r' be two integer constraint systems of the form*

$$l_i \leq n_i \leq u_i, l'_i \leq n'_i \leq u'_i$$

respectively. The relation between r and r' can be obtained in the following way:

- *If $l_i > l'_i$ and $u_i < u'_i$ for all i then r is included in r'.*
- *If $l_i < l'_i$ and $u_i > u'_i$ for all i then r' is included in r.*
- *If neither of the above conditions holds then r and r' overlap or are disjoint.*

Theorem 7 *The above method correctly determine the relationship between two integer constraint systems.*

Proof The proof is essentially the same as that for the relation algorithm in the clock constraint solver.

If $l_i < l'_i$ and $u_i < u'_i$ for all i we know that all n_i are included in r'. The same argument again, with some straight forward modifications, can be used to get the condition that must hold when r' is included in r.

In all other cases it's easy to show that there exists a $n'_i \notin r$ and a $n_i \notin r'$. □

The relation algorithm for integers are faster than it's corresponding algorithm for clocks. It's complexity is $O(n)$ where n is the number of integers in the system. A test to stop earlier if the constraint systems are found to be overlapping would slow the algorithm down because the test would be done for every comparison. By doing so, we will not be able to stop the relation algorithm as early as we can but we will save time which was the aim of the optimization.

Performing Integer Resets

When doing forward reachability analysis we need an operation that performs the resets. If the reset has the form $n_i = v_1 n_i + v_2$ and n_i is in the interval $[l_i, u_i]$ the new interval $[l'_i, u'_i]$ is simply given by

$$l'_i = v_1 l_i + v_2, u'_i = v_1 u_i + v_2$$

If $v_1 < 0$ we exchange l'_i and u'_i. Maybe this operation also shall be called the reset operation in analogy with the counterpart in the clock constraint solver [12].

[12] Integers can be thought of as clocks with rate zero.

Adding Strict Constraints

Strict integer constraints can easily be transformed to non-strict integer constraints.

$$l_i < n_i < u_i \Leftrightarrow l_i + 1 \leq n_i \leq u_i - 1$$

Therefor we can obtain the same functionality without wasting memory on flags that keep track of if bounds are strict or not.

5.3 The Verifier

This short subsection will discuss how the system and the state space is represented and the preprocessing that must be done before the system can be verified using backward and forward reachability analysis. The following subsections will treat subjects briefly mentioned here in more detail.

5.3.1 Representation

The system is represented as an array of processes. We don't need to use a list representation because the number of processes are static and hence cannot vary dynamically. The process has a list of transitions and the states are represented as integers. When doing backward reachability analysis, that list is used to build an array of lists. All transitions leading to a given state s are stored in a list which is stored in array position s. If we want to do forward reachability analysis we instead store all transitions from a state s in a list stored in array position s. This array representation will speed up the search for possible transitions from or to a given state. We will call this array the transition array.

As mentioned earlier, a global state consists of a control state and variable assignments. The control state is simply an array of integers where the integer at position i tells the current state for process i. The initial state of the system is a global state whose control state is made up of the initial state for each process. The clock assignment in that state is $x_i = 0$ and the integers are unrestricted. The verification algorithm checks if a given global state is reachable from that initial state. When the user wants to verify a property, that property is broken down into a list of global states and the reachability analysis is done for each of them. The result of each analysis are then used to determine if the system satisfied the property or not.Although, before we can do the analysis we must determine if there are certain conditions on the variables that must be added. Such conditions can be given by the user and in that case they are conjuncted as ordinary guards. However, because the system must perform a transition if it's possible it can also be the case that the structure of the system implies other conditions which must be added here. That case is more complicated and will be treated in a separate subsection. If we do forward analysis it's not a good idea to break the property down to a list of global states and check if they're reachable. We need a more efficient way to test if the property is satisfied.

The global states not yet analyzed are stored on the wait list. The wait list is implemented as a stack so the state space are searched depth-first. This list may be implemented as a queue if we want breath-first search instead. However this may possibly affect some of the optimizations which are done when the state space is searched.

Global states that are analyzed are stored on a passed list. This list is needed to avoid loops

and pruning the state space making the search more efficient. The passed list will be treated in a separate subsection. We are now ready to discuss the verification algorithm.

5.3.2 The Verification Algorithm

The algorithm returns true if the given global state is reachable from the system's initial state and false otherwise. The algorithm is as follows:

Algorithm 10 *1. Repeat the following steps if the wait list is not empty:*

2. *Pick a global state from the wait list. Denote that state s.*
3. *If we do backward analysis we check if s is the initial state and the variables satisfies the initial conditions. If we do forward analysis we replace this test by a test to determine if the property is satisfied in state s. If the appropriate test is true, we return true.*
4. *Check the passed list to determine if there are any global state s' whose control state matches that of s. If no such s' exists, we add s to the passed list.*
5. *If we find such a s' we check the relation between s and s' to determine what to do next.*
   (a) *If the constraint system of variable of s is included in the constraint system on variables of s' we leave s and do not analyze it any more. All the assignments of s were explored when s' was analyzed.*
   (b) *If the constraint system on variables in s' are included in the constraint system on variables in s, we delete s' from the passed list and store s instead. This is something that save space.*
   (c) *If the constraint system on variables of s and s' overlap or are disjoint, we leave s' on the passed list and add s to the list as well.*
6. *If any of case 5b or case 5c is true, we know that there exists variable assignments in s not yet analyzed. Therefor, we now put all global states that are reachable from s in one step, on the wait list.*
7. *If the wait list becomes empty we know that the initial state, or final state, is not reachable and we return false, Otherwise we repeat everything again. This algorithm will terminate with an empty wait list or with a result indicating that the state in question is reachable.*

Before we close this subsection we need to explain some steps more thoroughly. To determine if s is the initial state, we first check if the control state of s is equal to the initial control state. If it is, we also check if the clock assignment $x_i = 0$ is included in the weakest precondition of the region in s. We must check the weakest precondition of the region because we are allowed to delay in the initial state before a transition is performed. If the last test also is true we know that we have reached the initial state because the other variables are unrestricted as stated earlier. This test has been speed up in the implementation by first test if $x_i = 0$ is included in the region of s and only compute the weakest precondition of the region if that test fails. This is in many cases an optimization because the test if the point $x_i = 0$ is in the region is much faster than the weakest precondition operation which in this case must be followed by a closing operation. For all systems that do not delay in the initial state the test will be satisfied the first time.

In the next subsection we will show how to find all global states that are reachable in one step from a given global state.

5.3.3 Finding Possible Transitions from a Global State

Let us call the current global state of the system gs. To find all possible transitions from gs we look in the transition array.The positions to look in are given by the control state of gs. We then do a linear search through all transitions in the lists stored there.

If a transition has an action, we search through the rest of the processes not yet examined to see if we can find a transition with a matching action. We do not search through all processes for a matching action because then we would find all transitions with actions twice, once for each process. If we find a matching actions we do as described below to see if the transitions with the action are both possible. If we find a transition with no action we do the same steps, described next.

We first take a copy of gs, let's call it gs', which can be modified without destroying gs. Our next step is to update the control state of gs' so that the current state of the process that wants to perform a transition is changed to the state given by the actual transition. If we do forward analysis the new state of the process is the state the transition leads to and if we do backward analysis the new state is the state the transition leads from. If the transition had an action this change is made for both processes.

If we do forward analysis we now compute the strongest postcondition of the clock region in gs'. This must be done because we don't know for how long the system will delay in gs before the transition is performed. If we do backward analysis we instead compute the weakest precondition of the region in gs because we don't know for how long the system has delayed in gs since the transition was performed.

If we do forward analysis we now conjunct the guards on the transition to the variable assignment of gs'. If the transition had an action there are two sets of guards to conjunct. If the variable assignment still is consistent we know that the guards are satisfied. If no guards are present the variable assignment will be unchanged and remains consistent. If there are any resets on the transition, or transitions, they are performed now. Now we know that gs' is a reachable global state and it is put on the wait list.

If we do backward analysis we must process the resets before we conjunct the guards. We start by checking if the present resets are possible, that is if they could lead to any of the variable assignments in gs'. If they can, we then use the free operation to compute all possible variable assignments that could have been in gs' before the resets were performed. Remember that there are two sets of resets if the transition had an action. Next, we conjunct the guards to the variable assignment in gs'. If they still are consistent, we know that their exists variable assignments satisfying the guards and hence gs' is a reachable state and can be put on the wait list.

The linear search now continues until all processes and transitions have been examined. This search can maybe be made more efficient if one use the information present in the system to construct some kind of table telling which processes can synchronize with each other. However, when the tool was profiled we discovered that not much time was spent on this search.

5.3.4 The Passed List

The passed list is a data structure where all passed global states are stored. It is used to prevent us from exploring a global state twice or to avoid exploring a global state that we already know will not lead to anything new. This list is searched through every time a global state is picked

from the wait list. It must therefor be as fast as possible to perform such a search. The passed list also grows exponentially and hence the memory usage by this data structure is the most critical problem in verifying a system. This things must be kept in mind when designing a data structure for the passed list.

The first approach is to store a linked list of all global states. This means however that the passed list will be slower and slower to search through as the verification proceeds. When we compare global states with each other we first look if the control states are identical. If so, we continue and test how the variable assignments are related to each other. This means that when a global state is compared to the contents of the passed list, most of the global states that we look through does not have an identical control state.

The solution to that problem used in our tool is to implement the passed list as an array of lists and use hashing on the control state to get the position in the array to store the state. This means that we apply a function that maps the control state to an integer which tells us which list to store the state in. How shall this function be chosen?

The function must be fast to compute and have a good deviation so that the states become as evenly distributed over the array as possible. Let $\vec{c}$ be the control state with components c_i. Let $\vec{m}$ be a vector with components m_i where m_i is the number of states of process p_i. We then know that c_i ranges from 0 to $m_i - 1$. The function we used in our tool was

$$f(\vec{c}) = c_1 + \sum_{i=2}^{N} c_i m_{i-1}$$

where N is the number of processes in the system. Note that m_n is not used. This function maps every control state vector to a unique integer. If there are M possible control states the function is optimal because every integer in the interval $[0, M-1]$ corresponds to a given vector $\vec{c}$. This hash function would have been perfect, have no collisions, if the hash table could have had size m. In practise we have to choose the size of the array to be much smaller than M so the array position used is the remainder when $f(\vec{c})$ is divided by the size of the array. This means that the used hash function will not be perfect for large systems because different control states will hash to the same position. The size must be chosen so that as few collisions as possible arise. We chosen the size to be 17001 [13]. We can compute the total number of control states as

$$M = \pi_{i=1}^{N} m_i$$

For systems with $m \leq 17001$ the hash function will be perfect.

The problem with the huge use of memory still exist because the number of global states is much greater than M. This is because many global states can have identical control states but different variable assignments. This is a problem that needs to be solved in future implementations. One way is to reduce the number of global states by pruning the state space with use of quotienting but this is not treated here. Some other approaches are proposed below.

One solution is to compress a global state before it is stored on the passed list and then uncompress it when its relation to other global states is tested. IF such a compression method could be found we would save some memory but lose some performance in time. It would be much better if we could find a transformation which mapped a variable assignment to a vector or a number and if that mapping preserved the relation between variable assignments. Then we would save both memory and time because we don't need to transform it back to test relations and the relation algorithm would have a complexity which is $O(n)$ or $O(1)$ respectively, n is the number of components in the

[13] If the size is a prime one seems to minimize collisions.

vector. If such savings in time could be done it doesn't matter if the transformation takes some time to perform.

We can also save some memory if we can reduce the number of global states on the passed list. An example will clarify the idea:

Example 5 *Assume that all global states we now discuss have the same control state so we only need to look at their variable assignments. Also, assume that we have a system with one clock x. Let gs_1, gs_2, and gs_3 be three global states with clock regions r_1, r_2 and r_3 of the form*

$$r_1 = 0 \leq x \leq 2, r_2 = 2 \leq x \leq 4, r_3 = 4 \leq x \leq 6$$

We now want to see if the global state gs_4 with the same control state as the other but with the region $r_4 = 1 \leq x \leq 3$ shall be added or not. When we apply the relation algorithm we find that r_4 is disjoint from all the other regions and hence conclude that it shall be added. However the three regions on the passed list can be merged to one region

$$r = 0 \leq x \leq 6$$

If that had been done we would not only have saved memory by storing the global states gs_1 to gs_3 as one global state, we would also have discovered that r_4 is included in r and hence didn't need to be explored.

The most important improvements are those that can be done by use of techniques that reduce the state space. This will save both memory and time. If we could reduce the size of the global states as much as a factor 2 we could only verify systems with the number of global states doubled. Because the state space grows exponentially this is only a small increase in the size of the systems that we could handle. It's not the size of the global states that is the main problem, it's the size of the state space.

5.3.5 The Principle of Maximum Delay

When doing forward analysis it's intuitively clear that the clock assignment of the initial global state shall be on the form $x_i = 0$. Are there any obvious clock assignments for the final states that we start in when we do backward analysis? We can argue that we don't know anything of the clock assignments in these states and therefor shall start with an unrestricted region $x_i - x_j \leq \infty$. However, this is not the whole truth.

The principle of maximum delay, even called the principle of maximum progress, states that a system may not choose to skip a transition it can perform. It must perform a transition if it can. That means that if we start the verification in a global state whose outgoing transition has a guard $x \leq 5$ we know that if this state is reachable the system must leave it before $x > 5$. Thus, it's the guards on the outgoing transitions that determines the clock assignment to start with. This assignment must make sure that the system can leave the global state if possible transitions are present. Therefor we want the maximum upper bound that each clock can have and the system still being able to leave the global state. Since the system is defined to be alive only when every component can perform a transition [14], the maximum delay for the system is the minimum of the maximum delays for each component. The maximum delay for a component is given by a

[14] One could also have chose a semantic stating that a system is alive if at least one component can perform a transition but this semantic is not used here.

disjunction of the guards on the outgoing transitions from the control state from which we start the backward analysis. This disjunction cannot in general be expressed as a region of the form $x_i - x_j \leq u_{i,j}$ and we must either do one reachability analysis for each outgoing transition or approximate the disjunction with a region of the form above. The first technique will increase the state space while the other either introduce invalid clock assignments or discards valid ones. We chose the approximation technique because it's only made once per verification. However, one must be aware of the fact that the number of invalid clock assignments might be larger than the number of valid ones.

The algorithm we used to find the maximum delay can be sketched as follows:

Algorithm 11 *1. For each process and its control states do the following:*

2. *Find the maximum delay for each transition. This step is not so easy as it seams and will be discussed in greater detail below.*
3. *The maximum delay for the process in the examined control state is approximated as the maximum of the different maximum delays found for each transition.*
4. *The maximum delay for the system is the minimum of the maximum delays for each process and is derived by conjuncting each such delays to an unrestricted time region.*

To find the maximum delay d_i for a clock x_i on a given transition we start with $d_i = -1$ because every maximum delay is greater than -1. When going through the guards on the transition we keep the maximum upper bound encountered so far as the current value of d_i. After that, if $d_i = -1$ we can conclude that no guard on that transition involved the clock x_i and thus set its d_i to ∞ because the transition has no time restriction for this clock. But this attempt doesn't work!

If we have a guard of the form $2 \leq x_i \leq 3$ that guard is represented as two guards $x_i \geq 2$ and $x_i \leq 3$. When the guard $x_i \geq 2$ is found d_i is set to ∞ which will be the answer no matter of which other upper bounds for x_i that will be found. The correct answer should be $d_i = 3$. We therefor conclude that we cannot look at guards with an upper bound equal to ∞. We can only look at the guards of the form $x_i \leq u_i$ and $x_i = v_i$.

But this leaves us with another problem. Since we do not look at all guards involving x_i we can no longer conclude that no guard involving x_i exists on a given transition if $d_i = -1$ after the guards have been searched through. However, we don't need to distinguish the case when d_i shall be set to ∞, caused by a guard $x_i \geq l_i$, from that when d_i shall be set to ∞ because no guard involving x_i has been encountered on the transition. If there still are clocks those $d_i = -1$ when all transitions have been searched through, they shall have their values of d_i set to ∞ because they do not restrict the process in that state in any way at all.

This algorithm to find the maximum delay works fine in most cases but there are some examples which will cause errors. One problem is illustrated in the example below.

Example 6 *Let us consider a system with one clock x and an integer i. Assume that we are in a control state which has two outgoing transitions t_1 and t_2 with the sets of guards g_1 and g_2 respectively. Let $g_1 = x \leq 2$ and $g_2 = x \leq 3, i \leq 1, i \geq 2$. The algorithm will find that the maximum delay for x to be 3 but the correct value is 2 because the integer guards prevent the transition t_2 to ever be possible.*

One might argue that this error in g_2 is so trivial that it can be discovered before the verification. But, one might also argue that a verification toll at least shall be able to find the trivial errors

because they ought to be more easily found. However the error in the system shown in the next example are not as easy to find. It illustrates a problem that arise because we treat each process separately and don't look at the actions on the transitions.

Example 7 *Let us consider the same system as in the example above but with the addition that t_1 and t_2 now synchronize on an action. We modify g_1 and g_2 so that*

$$g_1 = x \leq 2, i \leq 1, g_2 = x \leq 3, i \geq 2$$

The algorithm will still find the maximum delay for x to be 3 but the combined transition is in fact not possible. The correct conclusion is in this case that the maximum delay for x shall be ∞ or -1 if other outgoing transitions not yet searched through exists.

A more serious problem is sketched in the following example which shows that it is not always clear how to make the approximation.

Example 8 *Consider a system with two clocks x and y and the other symbols as in the previous example. Let*

$$g_1 = x \leq 1, y \leq 3, g_2 = x \leq 3, y \leq 1$$

The algorithm will compute the maximum delay for x and y to be 3. However we can also choose to set the maximum delay for x and y to be 1. Both are of course approximations but if we want a too wide or too narrow approximation can depend on what kind of verification we will perform.

There are many more cases when this algorithm does not behave correctly. The only way to avoid all problems pointed out above is probably to and do one step of forward analysis to first get the possible transitions and then apply some algorithm similar to ours only on the possible transitions. The initial region in that step of forward analysis shall be unrestricted. The problem will be even more important when we do forward analysis. In order to guarantee maximal progress we will have to compute such a maximum delay for each step and treat it as an invariant in the state. The problem in the last example can be avoided, or at least passed to the user, by requiring the user to explicitly specify invariants that must hold in the states. In that way, the user can choose, for each verification, if an approximation that is too big or an approximation that is too small will be the best.

5.3.6 The Diagnostic Trace

When a global state is reachable the tool shows a path from the initial state to that global state. It shows the the system's control state, the actions it performs and the values of all variables in each global state. In the initial state a clock assignment $x_i = 0$ is shown and unbound integers are shown to have a value of ??. For each global state it shows how long the system delayed there before it reached the next global state. It is the minimum delay that is shown. The new clock assignment in that state is found by adding the delay to the clock assignment in the previous global state. How is this minimum delay computed?

Let gs_1 and gs_2 be two contiguous global states on the path. Let the time region in gs_1 be $r_1 = x_i - x_j \leq u_{i,j}$ and the clock assignment computed so far in gs_1 be $\vec{d}$ with components d_i. The time region in gs_2 is r_2 of the same form as r_1. The delay in gs_1 before the system proceeds to gs_2 can be computed as

$$D = max_i(-u_{0,i} - d_i)$$

We must take the maximum because all clocks have the same rate and hence delay the same amount of time. By taking the maximum we ensure that all constraints in r_2 will be satisfied.

To compute the clock assignment in gs_2 we first add D to d_i. If there were any resets on the transition to gs_2 we must change d_i accordingly. Even integers and other variables must be handled in the same way. This means that we must save the resets and actions when the analysis proceed in order to generate a correct diagnostic trace.

The trace is generated by storing a list in each global state. The list contains the global states that has been passed on the way backwards from the final global state which means the state we now check the reachability for. The list in the initial global state is the trace stored in forward direction. It is one trace, not necessarily the trace which take shortest time or contains a minimum number of global states. If we do a breath-first search we will get the shortest trace, that with the minimum number of global states. This consumes a lot of memory because almost no global states can be deleted during the search and there are maybe more efficient ways of generating the trace. Although the method is fast.

Chapter 6

Performance

The current version of UPPAAL have been tested both on the verification of Fischers Protocol and the verification of the Philips Audio Control Protocol. It has also been compared to other existing tools for verification of real–time systems: HyTech from Cornell, Kronos from Grenoble and Epsilon from Aalborg. The experimental results show that UPPAAL is not only faster, but also capable of handling much larger systems.

6.1 Fischers Protocol Revisited

Using the current version of UPPAAL, installed on a SUN SparcStation 10 with 64MB of primary memory and 64MB of swap space running SunOS 4.1.2, we have verified the mutual exclusion property of the simple version of Fischers Protocol[1] from two up to eight[2] concurrent processes.

On the same machine we also made experiments where the same type of protocol was verified using the three other tools. The versions used in this experiment was HyTech 0.6, Kronos 1.1c and Epsilon 3.2.

As illustrated in table 6.1 and figure 6.1 the experiments showed that UPPAAL is both significantly faster than, and capable of handle larger experiments than all the other tools. For HyTech and Kronos we have measured both the total time (including the time for generating the product automaton), as well as the time for the actual verification (marked with v in table 6.1).

6.2 Philips Audio Control Protocol Revisited

To test if UPPAAL is capable of verifying more realistic examples, we tried to verify the Philips Audio Control Protocol as it was modeled by Ho and Wong–Toi [HWT95][3]. During this verification we were much helped by the WYSIWYV[4] and diagnostic features of the toolkit.

[1] See section 2.3.1 for a description of how the protocol is modeled.
[2] We have verified the case with nine concurrent processes, but that was on a different machine.
[3] We used an early version of this paper, available at that time from the web server at Cornell University.
[4] What You See Is What You Verify

	2	3	4	5	6	7	8	9
HyTech	6.0	83.5	$\perp$					
HyTech^v	3.6	26.4	$\perp$					
Epsilon	0.8	10.6	242.6	$\perp$				
Kronos	0.5	4.0	50.5	$\perp$				
Kronos^v	0.2	3.4	46.9	$\perp$				
UPPAAL	0.2	0.2	0.7	5.5	18.8	145.0	1107.5	$\perp$

Table 6.1: Execution Times (seconds).

The first version we tried to verify was an adjusted version of the description in the paper by Ho and Wong-Toi. The adjustments made were necessary due to differences between HYTECH and UPPAAL. That included: transforming and moving the invariant conditions of the states to enabling conditions for the transitions; adding an edge $\mathsf{stop} \xrightarrow{\mathtt{leng} \geq 1} \mathsf{error}$ in the receiver automaton; and modeling the modulo-2 counters `k` and `m` as integers.

Some obvious typing errors were also corrected in this stage. A graphical representation of this version of the protocol is shown in figure 6.2

We then attempted to verify the correctness property of the protocol, but failed. Using the diagnostic trace generated by UPPAAL the system was further improved; an `input_1?` action was added in the receiver automaton and the `output_neq_0?` and `output_neq_1?` actions was swapped in the output acknowledge automaton.

A graphical representation of this version of the protocol can be seen in figure 6.3. The changes from the first version are shown in boldface.

Again we attempted to verify correctness of the protocol, and again it failed. After studying the new diagnostic trace we found a timing error, caused by swapped enabling conditions on some of the transitions leading from the `last_is_1` state of the receiver automaton. This was easily fixed (see figure 6.4), and we tried to verify the protocol again.

Once again the verification failed. Looking at the trace, we found out that the value of the parity counter `m` got wrong when the bit sequence `01` was part of the message sent. We found that the reason was that when the sequence `01` was received the counter was only updated one step, even though two bits was received. We changed that and tried again to verify the protocol, this time the protocol satisfied the correctness property. The final version of the protocol can be seen in figure 6.5.

The verification of the final version took 46 seconds to perform on a SS–10, and the verification attempts of the erroneous versions took between 2 and 30 seconds before they stopped and showed a trace.

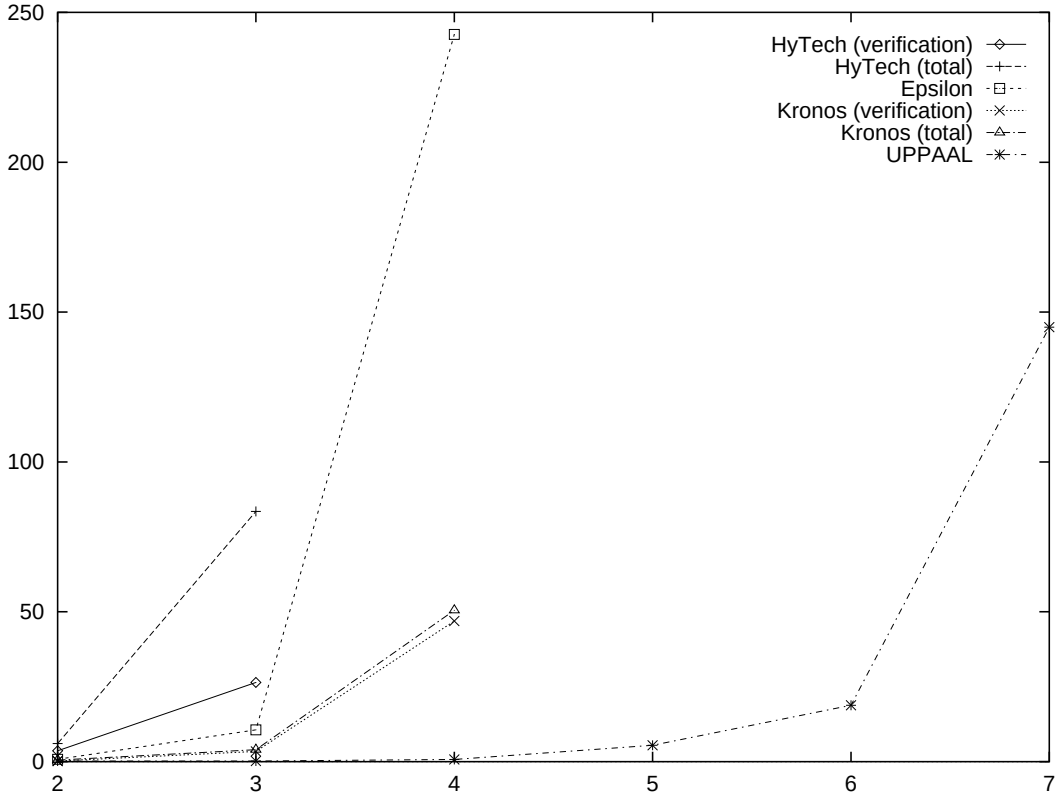

Figure 6.1: Execution Times (seconds).

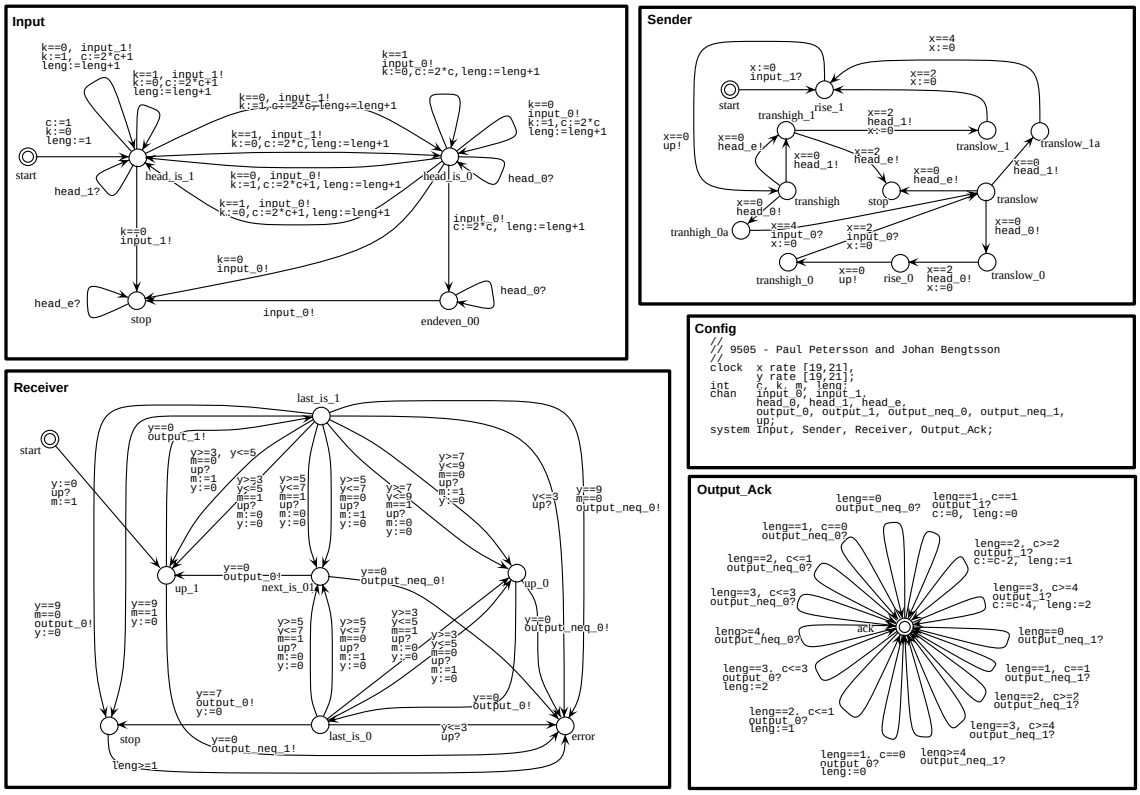

Figure 6.2: The First Version of the Audio Control Protocol

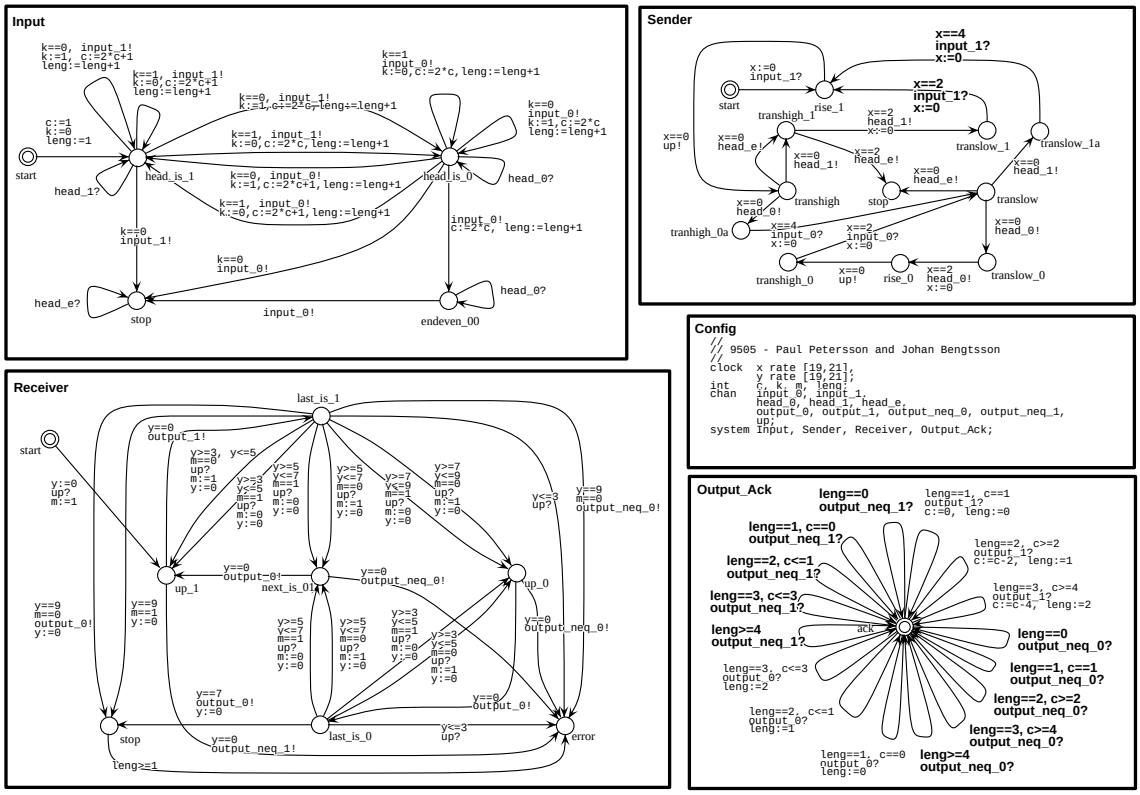

Figure 6.3: The Second Version of the Audio Control Protocol

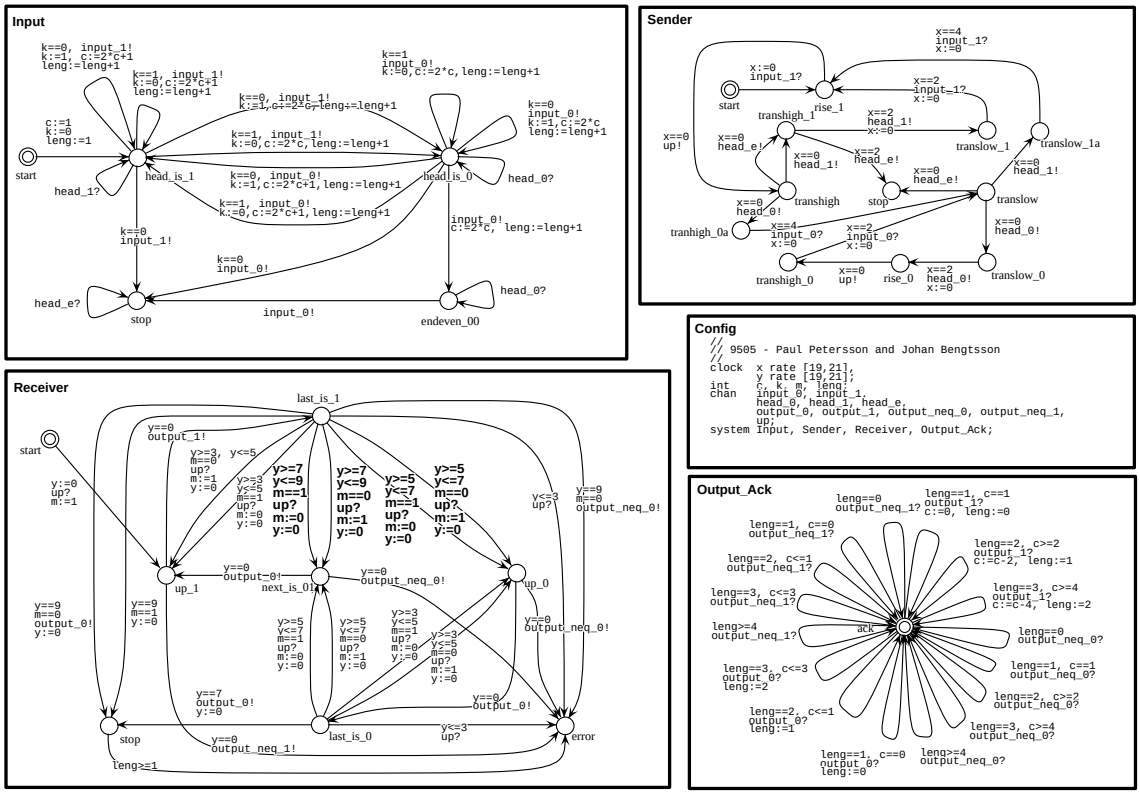

Figure 6.4: The Third Version of the Audio Control Protocol

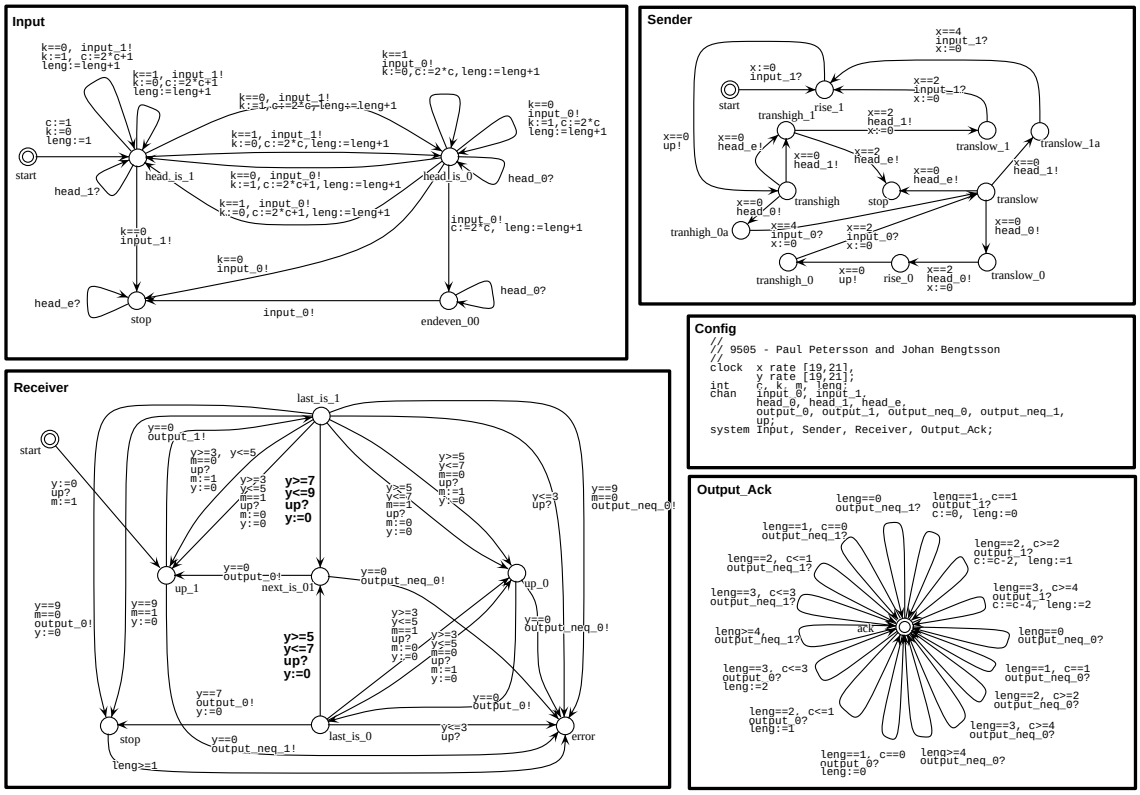

Figure 6.5: The Final Version of the Audio Control Protocol

Chapter 7

Conclusions

We have described a formal language that can be used to model a certain class of real-time systems and a simple logic for expressing properties of such systems Algorithms performing backward and forward reachability analysis using constraint-solving techniques have been presented together with efficient algorithms to handle the constraints. The ideas described in this report have been implemented in UPPAAL: a tool for automatic verification of real-time systems. The implementation of all its parts have been described in detail and the user's guide is included as an appendix. UPPAAL has been tested on various benchmark examples such as Fischer's mutual-exclusion protocol and Philips audio protocol. UPPAAL is fast and can handle quite large systems.

7.1 Future Work

There are still many features that can be added in order to make UPPAAL more powerful. Below, we present some of them.

7.1.1 Other Constraint Solvers

Constraint solvers for other data types can be added. One has to define how guards are conjuncted, necessary and sufficient conditions for consistency and how to treat resets in backward and forward reachability analysis. One might also need a relation algorithm and there may exist a counterpart to the concept of closed constraint systems. The weakest precondition and strongest postcondition operations are associated with time and probably not needed in those constraint solvers.

One can also replace the current constraint solvers for integers and clocks. One application of this is if we want to implement parametrization and need to manipulate the constraints symbolically.

7.1.2 Richer Logics

The logic for expressing the properties of a system can be extended to a richer one, (*e.g.* the logic described in [LPY95b]) to allow bounded liveness properties to be expressed in the logic, without transforming them into safety properties.

Bibliography

[AD90] R. Alur and D. Dill. Automata for modeling real–time systems. In *Proc. of the* 17th International Colloquium on Automata, Languages and Programming, volume 443. Springer–Verlag, 1990.

[AL93] Martin Abadi and Leslie Lamport. An Old–Fashioned Recipe for Real Time. *Lecture Notes in Computer Science*, 600, 1993.

[AOY94] J. Sifakis A. Olivero and S. Yovine. Using abstractions for the verification of linear hybrid systems. In *Proc. 6th Int. Conf. on Computer Aided Verification*, number 818 in Lecture Notes in Computer Science, Springer Verlag, pages 81–94. Springer–Verlag, 1994.

[BPV93] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio–Control Protocol. *Lecture Notes in Computer Science*, 863, 1993. In Proceedings of FTRTFT'94.

[Flo62] R. W. Floyd. ACM algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.

[Hal93] Nicolas Halbwachs. Delay Analysis in Synchronous Programs. *Lecture Notes in Computer Science*, 697, 1993. In Proceedings of CAV'93.

[HWT95] Pei-Hsin Ho and Howard Wong-Toi. Automated Analysis of an Audio Control Protocol. In *Proc. of CAV'95*, volume 939. Springer Verlag, 1995.

[Lam87] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transations on Computer Systems*, 5(1):1–11, February 1987.

[LPY95a] 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.

[LPY95b] 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.

[Sha93] N. Shankar. Verification of Real–Time Systems Using PVS. *Lecture Notes in Computer Science*, 697, 1993. In Proceedings of CAV'93.

[YL93] 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.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real–Time Systems By Constraint–Solving. *In the Proceedings of the* 7th International Conference on Formal Description Techniques, 1994.

Appendix A

UPPAAL User's Guide

A.1 Introduction

UPPAAL is a tool suit for automatic verification of safety and bounded liveness properties of real–time systems described as networks of timed automata. It consists of a user interface and a model–checker.

The user interface supports both graphical and textual representations of networks of timed automata and automatic transformation of graphical descriptions into textual descriptions. In addition, it can also deal with networks of linear hybrid automata that can be translated to timed automata using abstraction techniques [AOY94].

The model–checker is based on constraint-solving techniques. Model checking of real–time systems suffers from two state–space explosion problems: explosion in the region space and explosion in the space of control–nodes. The current version of UPPAAL deals with the region–space explosion problem by a symbolic technique reducing the verification problem to that of solving simple constraint systems, and the control–space explosion problem by on–the–fly verification techniques. Future versions of UPPAAL will be extended with a compositional quotient construction, allowing components of a real–time system to be gradually moved from the system description into the specification, thus avoiding explicit construction of a product automaton. The theoretical foundation for UPPAAL is described in [YPD94, LPY95a].

An overview of the tool suite structure is shown in Figure A.1.

A.2 Hard- and software required by UPPAAL

Currently UPPAAL is available for three common types of workstations; Sun SPARC–stations both with Sun–OS 4.1 and Sun–OS 5, HP-9000 with HP–UX, and Intel x86 with Linux.

To run UPPAAL, the BASH–shell is required, and to use the graphical interface Autograph[1] must be available.

[1] Autograph is a tool for drawing automata, developed at CMA, France.

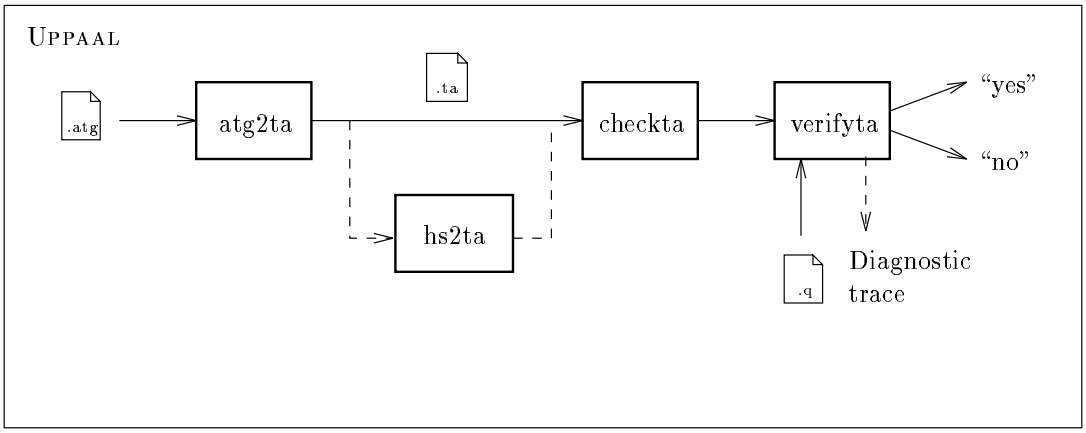

Figure A.1: Overview of UPPAAL

A.3 How to describe Real–Time Systems

UPPAAL allows real–time systems to be described using either of two types of models;

- Networks of Timed Automata
- Networks of Linear Hybrid Automata

A.3.1 Networks of Timed Automata

UPPAAL supports two different ways to describe the networks of timed automata. They can either be described graphically using Autograph, or textually. Both ways will be described below.

Textual description of Networks of Timed Automata

UPPAAL allows networks of timed automata to be described using a textual format (called `.ta`) which provides a basic programming language for timed automata.

The syntax of the `.ta` descriptions is quite simple. It consists of three basic parts:

- A global declaration part, where all global variables (*e.g.* clocks, integers and channels) are declared.
- A process description part, where all processes in the system are defined.
- A configuration part, where the global configuration of the system is defined. Right now the only information to be given in this part is a list of all processes in the system, but in the future it will also be possible to define which channels are to be hidden from the environment. (In the current version all channels are hidden from the environment)

To illustrate how to compose `.ta` description file, we consider the simple system description shown below.

```
//
// Generated from: /stud/docs/exjobb/tab/examples/big2.atg
//

//
// Global declarations
//
clock  x, y;                          } Global declarations.
chan   a;
//
// Processes
//
process p1 {                          }
  state start,loop,end;               }
  init  start;                        }
  trans start -> loop {               }
          guard  y == 0;              }
        },                            }
        loop -> loop {                }
          sync   a!;                  }
        },                            }
        loop -> end {                 }
          guard  y == 100;            }
        };                            }
}                                     }
                                      }
process p2 {                          } Process descriptions.
  state loop,start,end;               }
  init  start;                        }
  trans loop -> loop {                }
          guard  x==0;                }
          sync   a?;                  }
          assign x:=0;                }
        },                            }
        start -> loop {               }
          guard  x == 0;              }
        },                            }
        loop -> end {                 }
          guard  x == 0;              }
        };                            }
}                                     }
//
// Global configuration
//
system p1, p2;                        } System configuration.
```

Our example system contains two clocks (one for each process) and one synchronization channel. They are declared as follows:

```
clock  x, y;
chan   a;
```

The definition of a process starts with the keyword `process` followed by the name of the process and a process body, inside brackets. The process body consists of two parts, a state declaration part, and a transition declaration part.

```
process p2 {
  state  loop,start,end;                          } State declarations
  init   start;
  trans  loop -> loop {                           }
           guard    x==0;                         }
           sync     a?;                           }
           assign   x:=0;                         }
         },                                       }
         start -> loop {                          } Transition declarations
           guard    x == 0;                       }
         },                                       }
         loop -> end {                            }
           guard    x == 0;                       }
         };                                       }
}
```

As shown above, the state declaration part consists of a list of states in the process and a statement indicating the initial state.

The transitions between the control nodes are declared in the transition declaration part. The declaration starts with the keyword **trans** followed by a list of transitions. Each transition takes the following form:

<from> -> <to> { [<guardlist>] [<sync>] [<assignlist>]}

<from> and <to> are the names of the control nodes connected by the transition.

The third item,<guardlist>, is an optional item containing the conditions that have to be satisfied for the transition to be enabled. The <guardlist> starts with the keyword **guard** followed by a list of predicates. The predicates supported today are atomic constraints of the form: $x \prec n$, where x is a variable (*i.e.* a clock or an integer), n is a natural number, and $\prec$ is a non strict comparison operator (*i.e.* <=, >= or ==)

The <sync> item is also an optional item, telling if the process has to synchronize on a channel to be able to take the transition. A synchronization is declared as the keyword **sync** followed by the name of the channel on which the process must synchronize, and either a question mark (representing a query) or an exclamation mark (representing an answer).

Finally <assignlist> is also an optional item, containing a list of variable assignments made when the transition is taken. The item is started with the keyword **assign** followed by a list of assignments to the variables. For clocks, the only assignment allowed is resetting them to 0, while integers can be set to anything on the form $n_1 * x + n_0$ where x is the integer variable to be set, and n_0, n_1 are natural numbers.

The configuration part is, simply the keyword **system** followed by a list of names of processes working in parallel in the system. For our example system the configuration is as follows:

```
system  p1, p2;
```

A context free grammar for the **.ta** format is shown in appendix A.B.

Graphical Description of Networks of Timed Automata

A system may also be defined graphically, in terms of the following rules:

- The processes of a system must be put in boxes, with the names as structural labels.
- All states in a process must have a name associated. The name can be given in any visible label. The state names of a process are local, and can be reused in other processes.
- All transitions must be between states in the same process. Transitions between processes are not allowed.
- Guards, synchronizations and assigns for the transitions can be given in any visible label. they are written as a list of atomic constraints and assignments, in the same syntax as the textual format. Note that a transition can have only one synchronization.
- There must be a box describing the system configuration. It is a box with the word "config" as structural label, and declarations of the variables and system configuration, in the behavioral label. The declarations and system configuration should be given in the form described in the textual description format.
- A box containing no states, and no "config" label, is considered as a comment, and is ignored by the translation program.

The graphical description must be saved in the Autograph `.atg` format. It can then be compiled into the textual description using the `atg2ta` compiler from the UPPAAL toolbox. An Autograph description of the simple system we used when describing the textual format, is shown in figure A.2

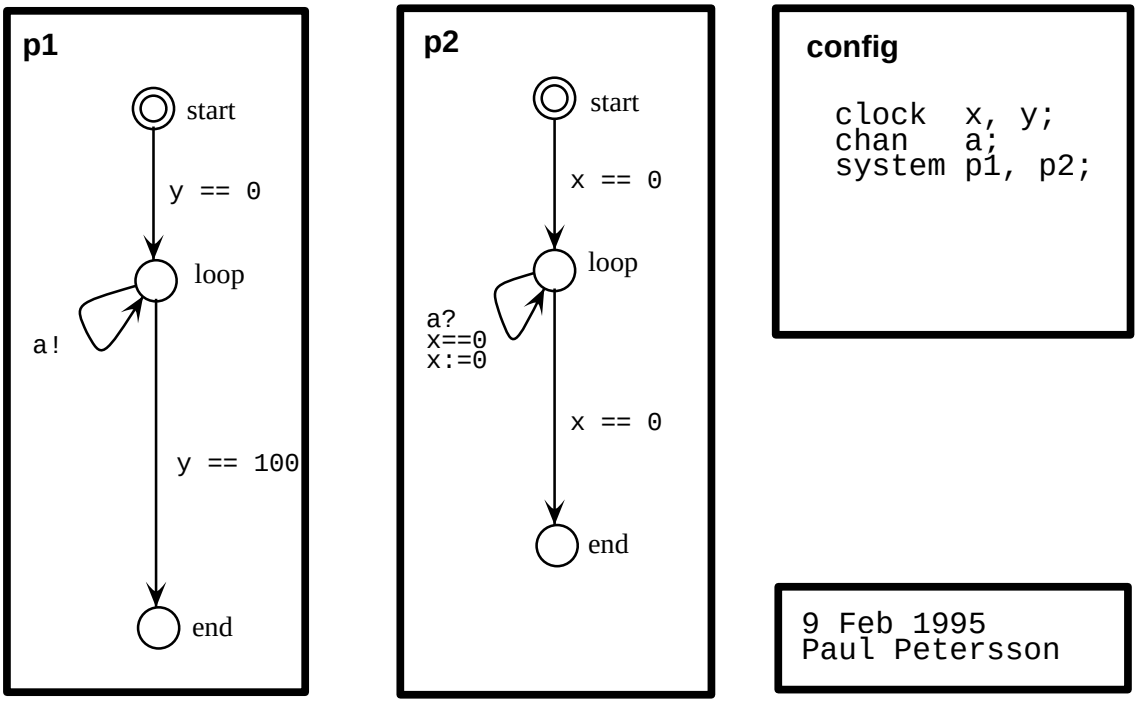

Figure A.2: Autograph Description of a Simple System.

A.3.2 Networks of Linear Hybrid Automata

UPPAAL also contains tools for translating networks of simple linear hybrid automata into networks of timed automata, using the method described in [AOY94]. The descriptions of linear hybrid systems doesn't differ very much from the descriptions of ordinary timed automata. The only difference is that the rate has to be given for all the hybrid variables. That is done by adding rate information when the clocks (*i.e.* the hybrid variables) are declared. For graphical descriptions that is in the configuration box, and for textual descriptions that is in the global declaration part. The syntax is as follows:

`clock` <name> `rate [`<lower>`,`<upper>`],`...

The translation is done by a preprocessor called `hs2ta`. The preprocessor is only capable of handling systems on textual format, if the systems are described graphically they have to be compiled to the textual format first, using `atg2ta`. There is a shorthand command, `atg2hs2ta`, that both compiles and transforms a system given on the graphical form.

A.3.3 Syntax Checks

UPPAAL has a syntax checker called `checkta`, that takes a file on `.ta` format, and checks the following:

- That all datatypes used are supported. (Today, the only supported datatypes are clocks and integers.)
- That all variables and channels are declared.
- That the processes have unique names.
- That all states in the processes are declared, and that all processes have an initial state.
- That both source and destination of the transitions are states, and that the states are declared in the same process as the transition. Note that state declarations are local to processes, so the same state name can be reused in other processes.
- That the guard and assign labels on the transitions are well formed.
- That only declared processes are used in the system.

A.4 How to Specify Properties of Real–Time Systems

The properties that can be verified with the current version of UPPAAL are simple reachability properties, *i.e.* if a specified state[2] is reachable or not.

The properties must be expressed as $\forall\Box P$, for invariantly P, and $\exists\Diamond P$, for possibly P. The predicate P is either a single atomic predicate, or composed of several atomic predicates connected

[2] In this case, a state is a configuration of the network, *i.e.* which control node each process is in.

using conjunction, disjunction, implication or negation. An atomic predicate is in one of the following forms:

$P_i.S_j$ Where P_i is a process in the system, and S_j is a state of P_i.
The predicate is true, iff P_i is in state S_j

$v_i \prec n$ Where v_i is a variable, $\prec \in \{\leq, \geq, =\}$ and n is a natural number.
The predicate is true iff the value of v_i is related to n by $\prec$

The translation of logical formulae into a syntax understood by UPPAAL is shown in table A.4. For instance, take the property expressing that process `p1` in our simple example eventually will terminate (*i.e.* reach it's end state). In logic it is expressed as $\exists\Diamond$ `p1.end`. Translated into UPPAAL syntax it will be "`E<> p1.end`". A context free grammar for the query language used in UPPAAL is given in appendix A.C.

Logical formula	UPPAAL **syntax**	**Comments**
$\exists\Diamond P$	`E<>` P	P is a predicate, without sub-predicates containing $\exists\Diamond$ or $\forall\Box$
$\forall\Box P$	`A[]` P	P is a predicate, without sub-predicates containing $\exists\Diamond$ or $\forall\Box$
$P_1 \wedge P_2$	P_1 `and` P_2	P_1, P_2 are predicates, without sub-predicates containing $\exists\Diamond$ or $\forall\Box$
$P_1 \vee P_2$	P_1 `or` P_2	P_1, P_2 are predicates, without sub-predicates containing $\exists\Diamond$ or $\forall\Box$
$P_1 \Rightarrow P_2$	P_1 `imply` P_2	P_1, P_2 are predicates, without sub-predicates containing $\exists\Diamond$ or $\forall\Box$
$\neg P$	`not` P	P is a predicate, without sub-predicates containing $\exists\Diamond$ or $\forall\Box$
$P_i.S_j$	`P`$_i$`.S`$_j$	The predicate is true, iff P_i is in state S_j
$v_i \leq n$	v_i`<=`n	
$v_i \geq n$	v_i`>=`n	
$v_i = n$	v_i`==`n	

Table A.1: Translations of formulae into UPPAAL syntax

A.5 Using the integrated environment

The "integrated environment", `uppaal`, is a shell–script made to simplify the verification process. When the uppaal script is started, the settings from last time is reloaded and a prompt is shown. From that prompt a number of commands are available.

exit Save all settings and exit the integrated environment.

help Display a small help screen.

set Set an internal variable.

show Show all settings.

verify Start a verification.

The different internal variables that can be set using set are:

check If this variable is set to `ON`, the specifications are syntactically checked before they are fed to the verifier. If there are syntactical errors in the specification, they are reported and the verification process is never started.

debug If this variable is set to `ON`, the possibility to show traces is enabled.

filter This variable specifies a filter for preprocessing of the specifications, before feeding them to the verifier. Typical filters are: `atg2ta`,`atg2hs2ts`,`hs2ta`. If no preprocessing is required, `cat` must be used as filter.

prop This variable specifies a file holding the property to be checked. If this variable is not set, the verifier will be asking for properties interactively.

silent If this variable is set to `ON`, no progress indicator will be shown.

spec This variable specifies a file holding the specification to be verified.

An example verification, where the example property is verified for our simple system is shown in appendix A.A

A.A Verification of a Simple System Using the Integrated Environment

```
Vivien> uppaal
Welcome to UPPAAL (version 0.9, Sep 1995). Copyright (c) 1995,
Uppsala University and Aalborg University. All right reserved.
UPPAAL> set file big2.atg
UPPAAL> set filter atg2ta
UPPAAL> set prop
UPPAAL> set debug on
UPPAAL> show
Current settings are:
   File    - big2.atg
   Prop.   -
   Filter  - atg2ta
Silent mode is OFF.
Syntax check is ON.
Debug mode is ON.
UPPAAL> go
UPPAAL version 0.5-Beta, May 1995 -- atg2ta.
Copyright (c) 1995, Uppsala University. All right reserved.
File is Ok !
Prop> E<> p1.end
The property is satisfied.
Showing example trace.

(p1.start p2.start )
clock x is 0
clock y is 0
```

```
(p1.loop p2.start )
clock x is 0
clock y is 0

(p1.loop p2.loop )
clock x is 0
clock y is 0

(p1.loop p2.end )
clock x is 0
clock y is 0
delay is 100

(p1.end p2.end )
clock x is 100
clock y is 100
Prop>
UPPAAL> quit
Vivien>
```

A.B Context Free Grammar for the Textual Format

Ita	→	*VarList ProcList Globals*
VarList	→	ϵ | *Channel VarList* | *Var VarList*
ProcList	→	*Proc* | *Proc ProcList*
Globals	→	`system` *IdList* `;` *OpHide* | *OpHide* `system` *IdList* `;`
Channel	→	`chan` *IdList* `;`
Var	→	*Type IdList* `;`
Proc	→	`process` *Id* `{` *ProcBody* `}`
IdList	→	*Id* | *Id* `,` *IdList*
OpHide	→	ϵ |`hide` *IdList* `;`
ProcBody	→	*StateDecls TransDecls*
StateDecls	→	`state` *IdList* `;` `init` *Id* `;` | `state` *IdList* `;` `init` *Id* `;` `final` *Id* `;`
Transdecls	→	`trans` *TransList* `;`
TransList	→	*Trans* | *Trans* `,` *TransList*
Trans	→	*Id* `->` *Id* `{` *OpGuard OpSync OpAssign* `}`
OpGuard	→	ϵ | `guard` *GuardList* `;`
OpSync	→	ϵ | *Id* `!` | *Id* `?`
OpAssign	→	ϵ | `assign` *AssignList* `;`
GuardList	→	*Guard* | *Guard* `,` *GuardList*
AssignList	→	*Assign* | *Assign* `,` *AssignList*
Type	→	`clock` | `int` | ...
Guard	→	*ClockGuard* | *IntGuard* | ...
Assign	→	*ClockAssign* | *IntAssign* | ...
ClockGuard	→	*Id RelOp Nat*
IntGuard	→	*Id RelOp Int*
ClockAssign	→	*Id* `:=` *Nat*
IntAssign	→	*Id* `:=` *IntExpr*
IntExpr	→	*Int* `*` *Id* + *Nat* | *Int* `*` *Id* - *Nat* | *Id* + *Nat* | *Id* - *Nat* | *Id* | *Int*
RelOp	→	`<=` | `>=` | `==`
Id	→	*Alpha* | *Id AlphaNum*
Nat	→	*Num* | *Num Nat*
Int	→	*Nat* | `-` *Nat*
Alpha	→	`A` | ... | `Z` | `a` | ... | `z`
Num	→	`0` | ... | `9`
AlphaNum	→	*Alpha* | *Num* | `_`

A.C Context Free Grammar for the Query Language

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