IT Licentiate theses
2000-003

Efficient Implementation of Model-Checkers for Networks of Timed Automata

FREDRIK LARSSON

UPPSALA UNIVERSITY
Department of Information Technology

Efficient Implementation of Model-Checkers for Networks of Timed Automata

Fredrik Larsson

May 8, 2000

Abstract

Since real-time systems often operate in safety-critical environments it is extremely important that they function correctly. UPPAAL is a tool that can be used for validation and verification of real-time systems. The user models the system using networks of timed automata and uses a simple logic to express safety requirements that the modelled system must satisfy to guarantee its correct behaviour. UPPAAL then performs reachability analysis using constraint solving techniques to check if the model satisfies the given requirements. In addition, the tool is also able to provide the user with a sample execution that explains why a requirement is (or is not) satisfied by the model. The analysis is fully automated.

This thesis describes various techniques adopted when implementing UPPAAL. Some of he techniques have improved the performance of UPPAAL significantly. We have studied the techniques with performance measurements in several case-studies. One of the main contributions is the comparison of different strategies in implementing the basic data structures and searching algorithms. The measurements can be used as hints on what parts of the model-checker that are most important to optimise. Though the techniques are studied in the context of timed automata, we believe that they are applicable to the implementation of general software tools for automated analysis.

Acknowledgement

First, thanks to my supervisor Wang Yi for starting the Uppaal project and providing valuable comments on this thesis. Thanks to all my colleges at the department of computer systems, Uppsala University, in particular Johan Bengtsson, Paul Pettersson, Tobias Amnell and Alexandre David. It has been a pleasure to discuss design principles and research problems and ending up with much better solutions than what is possible to come up with oneself. Also, thanks to Kim G. Larsen and his colleges in Aalborg for a very stimulating collaboration. Without all of you, Uppaal would not have been where it is today.

Other valuable feed-back is provided by all of you outside the development team who have used Uppaal in their research, such as [BFK+98, JLS96, BvKST97, BFM98]. With your input it is much easier to know what design decisions to take, what new features shall be added to Uppaal and what line the further development shall follow.

Last but not least, I want to thank my parents. Much of the writing has been done in their home, a place where no phone calls or E-mail could interrupt me. The food has been excellent as well. Also, thanks to all my friends that did not stop asking me when this thesis was supposed to be finished, despite that they always got the same answer: Soon.

Contents

Chapter 1

Introduction

1.1 Background

During the last two decades, the use of computers in our society has increased dramatically and continues to grow rapidly. Computers are no longer used only to help people with word processing, personal database management, statistical analysis and other heavy mathematical computations; they are also used to control industrial processes, cars, air-planes, medical equipment and other equipment that we use daily. The possibility to build small and powerful computers allow us to integrate them in so-called embedded systems. An important class of such systems is real-time systems [BW90]. They are characterised by the fact that it is not only important that the computer performs the correct things, they must also be performed in time. For example, even if the correct navigation information is derived from the controller, it is useless if it is not delivered on time.

When computers are used to control equipment in daily use the security requirements become very important. In fact, many of the tasks that computers perform today were earlier carried out entirely by humans. Even if a human still interacts with the computer and watches the information it produces, computers have at least partially replaced human capabilities in many situations. This increases the demand for secure operation of software and hardware even more. People will simply not accept that accidents occur due to malfunctioning technology but we all know that it happens. Despite that, the trend in our society goes in the direction that tasks previously done by humans are automated as much as possible and replaced by the so-called embedded computer systems.

1.1.1 The Traditional System Development Process

The traditional software development cycle is often described as a four-stage process involving design, implementation, testing and documentation. The first three phases are performed in sequence, often with revisiting of an earlier phase. Documentation is done in parallel producing appropriate documents for each phase as well as user instructions. The design phase aims at finding the functional requirements of the software and to make sure that the product to be produced will meet the customers requirements. This is part of the validation that must take place to answer the question: Are we building the right product, i.e. will the product suit our costumers needs? During the design phase one also makes decisions that will guide the implementation. The design stage may of course introduce errors and they must be find when the product is tested.

The implementation phase is the actual coding of the software. It also includes regularly presenting prototypes to get evaluational feed-back from the customers. Many errors may be introduced when producing thousands of lines of code and these errors must be detected during the test phase. There exists a lot of good tools that can assist the programmer in debugging the code but still, errors will remain and there are not many products that help the developers to find errors introduced in the design phase.

The testing phase is the final opportunity to find and correct errors. It aims to answer the question: are we building the product in a correct way, i.e. will it perform all the specified functions with no erroneous behaviour? While it may be possible to show that the requested functionality is implemented correctly it is not at all easy to find convincing indications that the software product does not contain any unwanted behaviour. After all, testing and simulation can never be complete and even if the software passes all tests it can not be guaranteed that untested functionality works correctly and that unwanted behaviour does not exist. Such faulty situations may occur during an unusual user scenario long time after the product has been delivered. Since the testing is the last phase, very short time may be devoted to it in many cases since delays in earlier stages must be compensated in order to meet the over-all deadline for the delivery.

To summarise, the great advances in hardware technology make it possible to use computers in many more application areas than earlier. Many of these areas are safety-critical and impose requirements that make the software and hardware more complex. In order to be competitive and save money and time, producers of computer systems are forced to produce more complex systems in shorter time. This clearly indicates that there is a need for a more thorough methodology when developing computer systems. One step in that direction is to use so-called formal methods.

1.1.2 Formal Methods and Automated Analysis

Formal methods is a term for methodologies where formal reasoning, for example based on logic and mathematics, is used to ensure that each step in the development process preserve the desired behaviour and do not introduce faulty ones. In many fields such frameworks exist, are well accepted and have been used for a long time. For example in physics, most phenomena known for long time have well-known physical laws that are mathematical models of the observed facts and the models enable us to use well-founded mathematics to reason about the phenomena. Another example is the use of thorough statistical methods, such as queueing theory and Markov models, to predict human behaviour when designing and allocating resources for service systems.

Computer science is a young science and would benefit a lot if it was possible to use concepts from other well-established sciences. A well-known example is the mathematical framework used to do scheduling analysis of multiple processes in a real-time multi-tasking environment. The method relies on the fact that there is a way to provide the execution time for each process in order to derive a perfect schedule according to a given criteria. Methods that can be used to find estimations of the execution time are called execution-time analysis. The attempt to provide safe estimations of the execution time for all possible behaviours of the processes to get schedules that always work is called WCET, worst-case execution time analysis. Another example is the complexity theory that is used to analyse space and time behaviour of algorithms and data structures for different sizes of input. Further, statistics is often used to analyse and predict packet traffic in networks such as telecommunications instead of picking parameters "out of the air" and let testing and every-day use tell if they work appropriately.

Formal frameworks all share some common properties. They are based on formalisms that allow reasoning using rules and facts, axioms and theorems. For example, one can construct a model in a modelling formalism of the behaviour of the software that shall be analysed and use some, maybe different, formalism to express the requirements that we want the software to satisfy. By using strictly defined computational rules it is then possible to decide if the model satisfies the requirements. The smaller the gap between the real software and the model the larger the probability that conclusions from the computations are applicable and tells something about the software.

In 1969 Hoare wrote a paper on the topic of axiomatisation of sequential programs [Hoa69] and the idea of proving computer programs correct using loop invariants. It was inspired by a paper published by Floyd [Flo67] about annotating edges in flow charts with invariants. In 1975 Dijkstra developed a guarded-command language [Dij75] together with the weakest-precondition analysis [Gri81] that is based on the propositional and predicate calculi and can be used to reason in a formal way about sequential programs written in imperative languages. Another framework is the Lambda calculus developed by Alonzo Church and introduced in the 1930s. The calculus is often used to analyse functional programs even if it is as expressible as Turing machines and hence can be used for other software as well.

However, the increasing use of multi-process operating systems and multi-processor computer architectures address a need for techniques that allow analysis of communicating processes executing concurrently or in parallel. Two process algebra approaches were introduced in the 1980s, CCS [Mil89] and CSP [Hoa78, Hoa85]. They were introduced for modelling of concurrent execution, e.g. multiple processes on a single processor. Another formalism is Petri nets [Pet77] introduced by C. A. Petri in the early 60s. They can be used to describe true parallelism, e.g. multi-processor systems and resource sharing in distributed systems. The main drawback was the lack of hierarchical constructs and constructs supporting data analysis. Petri nets were therefore revised in the late 70s. Petri nets has since then been extended in many ways and there exists many different dialects, e.g. coloured Petri nets, timed Petri nets and stochastic Petri nets. Other formalisms based on automata theory and different kinds of logics and formal models are available and it is important to pick a formalism as close as possible to what shall be modelled and analysed.

The computations required to analyse a model and its requirements are often difficult and it is likely to introduce errors. A way to reduce the complexity in the modelling and computations is to use abstraction. Abstraction is a way to provide different views of a system depending on the kind of analysis to perform. One shall concentrate on the details that seems most relevant for that particular analysis and do a less detailed, more abstract, model of the parts of the system that in this particular view, constitute the environment. The less abstractions used, the heavier the computations needed and it is easier to re-use conclusions drawn from the computations to find the corresponding faulty behaviour in the modelled system. Correctness proofs for a control-protocol used by Philips in their audio equipments were carried out by hand in 1994 [BPV94] which took several months. It would be nice if computers could assist in these computations so that the user just had to provide a model, a specification, the kind of analysis to perform and just press a button and get an answer.

One of the earliest tools was Spin [Hol91, Hol97, VW86] developed at Bell-Labs. The development started in 1980. The modelling language is called PROMELA which is heavily based on Dijkstras "guarded-command language" and Hoares CSP notation for I/O operations. The main usage is the analyse of reactive action-systems. The tool is based on automata theory for modelling and specifications are expressed in LTL either directly or indirectly through so-called Bchi Automata.

Another tool is Murphi [DDHY92], developed at Stanford in 1992. It has been successfully applied to analyse different kinds of protocols. The "Murphi description language" is also based on guarded commands and the algorithmic technique used relies on state-space enumeration. None of these tools are able to verify timing properties.

In 1991 HyTech was developed at Stanford University. HyTech is a tool able to analyse models based on hybrid systems. The model is automata-based and differential equations can be used to model progress of time and other continuous variables. The computations are performed by manipulations of n-dimensional poly-hedra. This made the constructs in the modelling language very powerful, which affected the performance of the tool negatively. The first version of HyTech verified the Philips' audio control protocol mentioned earlier in 7 hours and a later version of the tool in less than a minute. More about HyTech can be found in [HH95, HHWT95b, HHWT95a, HHWT97].

While HyTech can handle quite general problems there is often a cost in performance. The philosophy behind UPPAAL is to solve less general problems and gaining in performance. The modelling formalism used in UPPAAL is based on networks of timed automata with data variables. Requirements are expressed as propositional formulae extended with quantifiers and basic temporal logic operators. The computational analysis is based on reachability analysis of the control graph combined with constraint solving to manipulate time and data. Other tools that use the same framework did not perform the computations on-the-fly, instead they constructed the product automaton from the network of automata before the reachability analysis was performed, imposing severe restrictions on the size of systems they could cope with.

An early prototype written in Prolog was developed in 1994 at Uppsala University to test the ideas of constraint solving and on-the-fly generation of the state-space. The results seemed promising but performance was very bad. The first version of UPPAAL written in C^{++} was released in 1995 [BLL+95a, BLL+95b, LPY95a, BL96]. It was developed in Uppsala but a collaboration with Aalborg had just started. Since 1996 UPPAAL has gone through many changes [LH96, LPY97b, LPY97a, BLL+98] to become an even more useful tool and the joint collaboration between Uppsala and Aalborg has been very intensive. This thesis discusses the implementation of the verification engine of UPPAAL.

Another tool related to UPPAAL is Kronos [DOTY95, BDM+98] developed at Grenoble University, France. The first version of Kronos arrived in 1995. Kronos is based on timed automata as UPPAAL but Kronos was aimed at model-checking more complex logical formula, in fact all of TCTL [HNSY92].

The interest in automatic formal analysis has lead to the development of many different tools. The ones mentioned here are the ones that are most related to UPPAAL.

1.2 Contributions

One of the main contributions of this thesis is to compare the time and space performance of different algorithms and data structures that can be used when implementing a model-checker based on reachability analysis for networks of timed automata. In particular it focuses on the techniques studied when implementing UPPAAL but the covered materiel should be of interest to anyone building model-checkers for timed systems. The comparison of performance of the described algorithms and data structures are presented with measurements. Instead of measuring the exact time and space usage the over-all performance, i.e. total execution time and memory consumption, are measured. This method is suitable in many ways. It can be used to get an idea of what performs better and what is important to focus on when building a tool like UPPAAL. The purpose is also to get a feeling of what parts of a model-checker that is most important to optimise in order to gain the most in performance. If the goal is to derive exact formulae for the time and space complexity of the algorithms and data structures presented, it is probably better to do more accurate measurements and replace the time accounting routines, memory manager and task scheduler.

Another important contribution of the thesis is the comparison between different implementations of the same algorithms or data structures. The comparisons are made with measurements in the same way as earlier. This is done in order to emphasise the fact that it is not only the choice of a particular algorithm or data structure that is important; the choice of constructs in the programming language when coding them is at least as important. Hopefully the thesis also serves as a bridge between the theoretical concepts used when reasoning about model-checking and the available constructs in the implementation language that must be used to express these abstract notations in order to build software tools for automated analysis. It is definitively the case that today's existing hardware and software architectures, together with available theoretical results and practical experience, can be used to build tools that can aid system developers in the production process of industrial-sized real-time systems.

1.3 Outline

The thesis is organised as follows: Chapter 2 treats the basic concepts behind Uppaal, including timed automata and reachability analysis using constraint solving. There is also a discussion on what properties can be verified with reachability analysis and how "error traces", or sample executions, can be generated. The material is presented in a quite informal way with references to more thorough treatments of the theory.

Chapter 3 discusses algorithms and data structures for the transition relation. It covers representation of states, transitions and the synchronisation mechanism. It shows how the use of pre-processing and static analysis of the automata network can be used to speed-up verification.

Chapter 4 describes one of the most critical part of Uppaal's performance. It covers algorithms and data structures for symbolic states. It presents details comparing different implementations of constraint manipulation and develops a method for removing redundant constraints. Further it examines different representations of location vectors and data variables.

Chapter 5 deals with another important part critical for the performance of Uppaal. It contains algorithms and data structures for large sets of symbolic states. It discusses the use of hashing to search through these sets efficiently. It also covers approximations and heuristics to reduce the number of states to manipulate. It ends with a section on verification of multiple properties and re-use of earlier verification results.

Chapter 6 studies techniques to maintain the large amount of memory consumed by the huge state-spaces of models, i.e. sets of symbolic states. It presents experiments and propose a method for deallocating and traversing the state-space in a way that reduces swapping. It contains a method to let the application control how the operating system manages blocks of memory without writing an application-specific memory manager.

Chapter 7 ends the thesis. The most important conclusions and future work are summarised.

Finally appendix A contains a description of each of the examples used throughout the thesis in the experiments. It also presents details about the measurement scheme and equipment used.

Chapter 2

Timed Automata and Reachability Analysis

This chapter describes the underlying theory of Uppaal. The first section describes the modelling language in Uppaal based on networks of timed automata extended with data variables. It deals with the synchronisation mechanism used and some other constructs that affects the transition semantics. The presentation is informal with references to more extensive materiel. The next section describes the model-checking technique utilised in Uppaal, which is to perform reachability analysis based on constraint solving. The term symbolic state will be introduced as well as many other concepts used throughout the thesis. It also covers difference-bound matrices which is considered as a common way of representing the continuous part of a symbolic state. It also covers some elementary implementation issues about searching in graphs.

The next section is devoted to the query language of Uppaal, based on propositional logic extended with temporal quantifiers. The section also covers other properties that can be verified with the aid of reachability analysis. The chapter closes with a section describing how to get feedback to the user whether properties are satisfied or not. The diagnostic information reported by Uppaal is traces describing the execution of the model from the initial state to the reachable state in question.

2.1 Timed Automata with Data Variables

Timed automata [AD90] are ordinary finite automata extended with clocks. A state is a tuple $\langle L, C \rangle$ where L is the location of the automaton and $C \in \mathbf{R}^{n+}$ is a clock assignment describing the values of the clocks at the moment. Each transition is extended with a guard that determines when it is enabled. A guard is a conjunctive formula of constraints on clocks of the form $x_i \sim c$ or $x_i - x_j \sim c$ where x_i and x_j are clocks, c is an integer constant and $\sim \in \{<, <=, ==, >=, >\}$. The relational operators have the same meaning as in conventional programming languages. Transitions may also contain commands that can be used to reset clocks to integer values. If more than one transition is enabled, any of them can be performed. A timed automaton may also choose to omit performing a transition, just delaying in the location. This means that a timed automaton can delay an infinite time just omitting all enabled transitions. To guarantee progress each location can be marked with a location invariant constraining how long an automaton is allowed to stay in the location before it must choose an enabled transition. To avoid the possibility that invariants are false when a location is entered and become true after some delay, they are restricted to be downwards closed, i.e. they have a lower bound of zero. This restriction makes it possible to simplify computations and do model-checking faster. Thus, an invariant has the form $x_i < c$ or $x_i \leq c$, x_i and c have the same meaning as in the discussion of guards.

Data Variables

In the UPPAAL modelling language timed automata are further extended with data variables to make it easier to perform integer arithmetic. Hence, a state is a tuple $\langle L, C, V \rangle$ where L and C are as before and $V \in \mathbf{Z}^{n+}$ is an assignment of data variables. The guards and commands on the transitions in original timed automata are extended as well. Guards on data variables may be general relational expressions as in standard programming languages. As earlier, an automaton may only perform a transition if it is enabled i.e. its guards are true with respect to the current settings of clocks and values of data variables in that state. The assignment commands on a transition make it possible to update data variables using integer arithmetics similar to most programming languages. The variables have domains and wrap-around arithmetic is used. The term reset will be used to denote all assignment commands on a transition regardless of if they affect clocks or data variables.

Networks of Timed Automata

In the UPPAAL model, timed automata can be put in parallel to form networks. Each component can synchronise with other components in the network through two-way handshake synchronisation. This implies that transitions can have an optional synchronisation label or action. An action is of the form *chan*! or *chan*? where *chan* is the name of the channel. The exclamation mark indicates sending on the channel and the question mark indicates receiving. Channels are internal to the components in the network and not accessible or observable from the outside environment. A network performs a transition either when an individual automaton performs a transition or when two components synchronise and perform a compound transition. A compound transition can only be performed if the involved components both have enabled transitions with matching actions. Two synchronisation labels only match each other if the channel names are the same, i.e. one of them acts as sender and the other as a receiver on the same channel. The semantics of progress is easily extended for a network of timed automata; a network may only delay in a location as long as the conjunction of all location invariants for all components are true in the current state.

Clocks are local to each automaton, i.e. there value cannot be reset or observed by other automata in the network. In contrast, data variables are shared and available for both reading and writing by any component in the network. This means that implicit synchronisation is available through the use of the data variables as well. The fact that data variables are global, i.e. shared between components may cause consistency problems. For example if two synchronising automata both want to access the same variable on their compound transition[1], the result of such a computation will be dependent on the order of the accesses. There are at least three ways of dealing with this situation.

The simplest is to forbid accessing of the same variable on transitions with matching actions. This test may either be performed statically, before verification starts, or dynamically during the verification. The disadvantage of the static approach is that it may warn on some combinations of transitions that will never occur because they are in an unreachable part of the state-space. The advantage is that it may be performed only once per model. The dynamic approach would only warn on the necessary reachable combinations but would lead to some over-head in verification time. The second approach is to perform both accesses but in a well-defined order. This is what is used in UPPAAL today and the sender performs its accesses before the receiver. The advantage is that it can be utilised to make models more efficient but the draw-back is that it destroys symmetry properties that may be used to explore the state-space more efficiently. A third approach is to introduce non-determinism in the model and explore both possible orders. This would increase the size of the state-space and the construct would probably be of little use for the user.

One of the locations in each individual automaton is marked as initial and all clocks are zero the first time it is entered. Data variables may be initialised by the user and get that value the first time the initial location is entered. If no value is supplied by the user the variables have either value 0 or the minimum value in their respective domain if zero is not included. Domains may also be declared by the user and if no domain is declared, variables get a default domain of -32767 to $+32767$. The initial location of the network is derived from the initial location of the individual components.

Other Extensions

There are situations where it is required that a transition is performed as soon as it becomes enabled. This can be done by annotating the transition with a channel and declaring it as urgent. For efficiency reasons, no guards on clocks are allowed on a transition equipped with an urgent action. UPPAAL only handles convex constraints of a simple form that would be violated without this restriction. It also prevents the fact that transitions with urgent channels become true after some delay which also makes verification more efficient. It is also possible to mark a location as urgent and prohibit a delay of the network in this location.

The synchronisation scheme used in UPPAAL ensures that at most two components can change locations at the same time, i.e. on the same transition. However there exist situations where

[1]Here, the term access means that at least one of the components changes the value of the variable

you want multi-synchronisation [BGK+96] e.g. allowing an automaton to send messages to multiple recipients. With this pair-wise communication scheme this can only be modelled using consecutive transitions sending the message to one recipient at a time. No delay may occur in the intermediate locations and the transitions really are performed as an atomic transaction. UPPAAL lets you do that by declaring the intermediate locations to be committed. More formally, the semantics of a committed location is that all components in a committed location must participate in the next transition performed by the network. Since at most two components can participate simultaneously in a transition it is considered a modelling error if more than two components are in committed locations at the same time. It is also considered an error if two components are in committed locations and cannot synchronise and perform a transition together. However this will only occur if at most two components are in committed locations in the initial location of a network as stated in the following fact.

Fact 1 *If at most two components are in committed locations in the initial location of the network, a situation with more than two components in committed locations will never occur*

Proof The proof is by induction on the length of transition sequences (see e.g. [Mil89]) of a network. There are three possible situations depending on the number of components initially in committed locations in a network:

1. Assume that no components in the initial location are in committed locations. Since no more than two components may participate in a transition at most two will change locations. No components are in committed locations before the transition is performed and hence at most two may enter a committed location after performing the transition.

2. Assume that one component is in a committed location in the initial location. According to the semantics of committed location this component must perform a transition. Either it performs a transition by itself without synchronising, or it synchronises with another component not in a committed location. In the first case only one component changes location and since this component was the only one residing in a committed location, only one component may be in a committed location after the performed transition. In the second case the component must synchronise with a component not in a committed locations. These are the only two components that may end up in committed locations.

3. Assume two components in committed locations. According to the semantics they will be the only components taking a transition. Afterwards, no more than two components may have entered a committed location.

□

In later versions the notion of committed locations is re-implemented based on a more relaxed semantics. It is enough that at least one of the components performing the next transition is in a committed location.

2.2 Reachability Analysis and Constraint Solving

This section presents the underlying theory of the UPPAAL verification engine. It covers reachability analysis in general and for networks of timed automata in particular. It explains how infinite state-spaces may be handled by grouping states into a finite number of equivalence classes and manipulated via constraint operations.

2.2.1 Reachability Analysis for timed Automata

Reachability analysis is a method for analysing the state-space of automata to determine what states are reachable from a given state, e.g. the initial state. It has many similarities with standard finite graph-searching to determine if two nodes are connected. The main difference in analysing timed models is that the clocks are real-valued making the state-space of timed-automata models infinite. To get a finite state-space we group clock assignments together in equivalence classes. Two clock assignments belong to the same equivalence class if they cannot be distinguished using the logic operations in a property or a guard. We call these equivalence classes time zones. In each state, for each encountered transition, the state-space is partitioned to separate clock assignments that cannot enable the transition from those that can.

The infinite state-space, caused by the real-valued clocks, can now be viewed as a finite state-space consisting of symbolic states. A symbolic state can be thought of as a triple $\langle L, V, U \rangle$ where L is the location vector, V is a vector with a data variable assignment and U is a time zone. L and V constitute the discrete part of a symbolic state and U the continuous part. Throughout the thesis U will be referred to as a time zone or a constraint system; these terms will be used interchangeably even if it is more correct to say that a constraint system is a way of describing the time zone. U may also denote the solution set of a constraint system but the meaning should be clear from the context.

2.2.2 Operations on Constraint Systems

This section describes the necessary constraint operations needed to explore the state-space, using forward reachability analysis, and manipulate the continuous part of the symbolic states.

Delays and Strongest Post-Condition

In the timed model used in UPPAAL time progresses in the states and the computations on the transitions are performed in no time. All clocks move with the same speed. Since a timed automaton can choose not to perform an enabled transition and simply stay in the states forever, if no state invariant forces it away, delays might be infinite.

Definition 1 *(The Strongest Post-Condition) Let U be a time zone and d a delay, x and z are vectors describing clock assignments in time zones. The strongest post-condition of U for all possible delays is $sp(U) = \{z \mid z_i = x_i + d, d \geq 0, x \in U\}$.* □

Conjuncting Constraints

To evaluate properties and checking if transitions are enabled and state invariants hold we have to conjunct constraints to a constraint system describing a time zone.

Definition 2 *(Constraint conjunction) If U is a time zone and G is a set of constraints the time zone after the conjunction is the intersection of the solution sets: $G \cap U$.* □

We must also be able to check consistency of a constraint system, i.e. if the time zone it describes is non-empty, $U \neq \emptyset$.

Reseting clocks

Automata may update clocks when performing a transition. To do that they make use of a reset operation.

Definition 3 *(Reset operation) Let U be a time zone and x_k the clock to be reset to v_k, x and z are vectors describing clock assignments in a time zone. $reset(U, x_k, v_k) = \{z \mid z_k = v_k \wedge z_i = x_i, i \neq k, x \in U\}$.* □

Relationship between Time Zones

When exploring the symbolic state-space we need to determine the relationship between time zones for example to see if some or all of the states described by it have been explored or not.

Definition 4 *(Inclusion) A time zone U is included in a time zone U' if $U \subseteq U'$.* □

Canonical Form of Constraint Systems

The implementation of the operations on constraint systems will be purely syntactical manipulations and comparisons if the constraint systems describing time zones are in a canonical form.

Definition 5 *(Canonical form of a constraint system) Let x be a vector describing a clock assignment in a time zone U of the form $x_i - x_j \leq u_{i,j}, x_0 = 0$. U is on canonical form if $\forall d_{i,j} \leq u_{i,j} \exists x \in U \;:\; x_i - x_j = d_{i,j}$.* □

This is equivalent to $\forall k(u_{i,j} \leq u_{i,k} + u_{k,j}), i, j, k \in [0..n]$, i.e. all constraints are the strongest possible and contains no redundancy.

Normalisation

The normalisation operation is needed to get a finite partition, i.e. a finite number of symbolic states, and to guarantee termination of the reachability algorithm. It relies on the fact that for each combination of system and property there exists a maximum constant for each clock that the system is sensitive to. Clock assignments larger than this constant vector cannot be distinguished during the state-space search and shall be considered equivalent. In short, two time zones have the same normalised form if all clock assignments represented by their solution sets can reach the same set of states in the future.

Definition 6 *(Normalisation) Let x and z be vectors describing a clock assignments in time zones. Let U be a time zone, in canonical form, $x_i - x_j \leq u'_{i,j}, x_0 = 0$ and k be a vector with components k_i containing the maximal constant that a clock x_i is compared to during the reachability analysis, $k_0 = 0$. The normalised time zone $U' = z_i - z_j \leq u'_{i,j}$ is $norm(U, k) = \{z \mid z_i - z_j \leq u'_{i,j}\}$ where*

$$u'_{i,j} = \begin{cases} u_{i,j} & -k_j \leq u_{i,j} \leq k_i \\ \infty & u_{i,j} > k_i \\ -k_j & u_{i,j} < -k_j \end{cases}$$

□

The definition relies on the fact that the real-valued vectors in U' will never be placed in different equivalence classes during the state-space exploration. The only way to partition U' is to conjunct a constraint of the form $x_i - x_j \sim c_{i,j}$ where $c_{i,j} < -k_j$ or $c_{i,j} > k_i$. If such a constraint can occur it will contradict the fact that K contains the maximal constants that the clocks are compared to in the model or property. For a more detailed proof of the fact see [Pet99].

2.2.3 The Algorithm for Reachability Analysis in UPPAAL

UPPAAL checks reachability properties for networks of timed automata by state-space exploration [YPD94, LPY97a] using Forward reachability analysis. The main reason for choosing forward reachability analysis instead of backward is that implementations perform much better given our modelling language. Forward reachability analysis handles data variables and expressions on them much more efficiently than backward reachability analysis for two major reasons: First, it is possible to give each data variable a prescribed value in the initial state of the network thus making it possible to determine a unique value for all expressions on the transitions. For a backward reachability analysis we must start with the assumption that any value in a domain of a variable may be possible for that variable. This will consume twice as much memory. Second, it is not possible to symbolically invert general expressions on integers that is needed to determine the possible domain for a variable when computing a pre-condition for a transition given the allowed domain. This will involve prime factoring and solving of non-linear Diophantic equations which are extremely time-consuming calculations.

We are now in a position to present an algorithm performing forward reachability analysis on timed automata via constraint solving. The algorithm checks if a state $\langle L_f, V_f, U_f \rangle$ is reachable from the initial state $\langle L_0, V_0, U_0 \rangle$ or not. When searching the state-space we need two sets of symbolic states called WAIT and PAST respectively. The WAIT set holds the states not yet explored; the PAST set holds the states explored so far. The algorithm may easily be adapted to backward reachability analysis. For a thorough treatment of the required adaptions, the additional constraint operations required and related implementation issues see [BL96].

Algorithm 1 *(Reachability analysis using constraint solving)*
Initial conditions: $\mathsf{WAIT} = \langle L_0, V_0, U_0 \rangle, \mathsf{PAST} = \emptyset$.

1. *Pick a state* $\langle L_i, V_i, U_i \rangle$ *from* WAIT.
2. *If* $L_i = L_f \wedge V_i = V_f \wedge U_i \subseteq U_f$ *return* yes.
3. *If* $L_i = L_j \wedge V_i = V_j \wedge U_i \Leftrightarrow U_j$, *for some* $\langle L_j, V_j, U_j \rangle \in \mathsf{PAST}$ *drop* $\langle L_i, V_i, U_i \rangle$ *and go to step 1. Otherwise save* $\langle L_i, V_i, U_i \rangle$ *in* $\mathsf{PAST}, \mathsf{PAST} = \mathsf{PAST} \cup \langle L_i, V_i, U_i \rangle$.
4. *Find all* L_k *that are reachable from* L_i *in one step regardless of guards and resets, taking only actions into account. For all such transitions do*
   (a) *Let* G *be the guard and* R *the resets on the performed transition.* i_k *is the location invariant of* L_k.
   (b) *If* V_i *satisfies the guard, let* $U_k = sp(reset(U_i \cap G, R)) \cap i_k$ *and update* V_i. *If* $U_k \neq \emptyset$, *store* $\langle L_k, V_k, U_k \rangle$ *in* WAIT, $\mathsf{WAIT} = \mathsf{WAIT} \cup \langle L_k, V_k, U_k \rangle$.
5. *If* $\mathsf{WAIT} \neq \emptyset$ *go to step 1.*
6. *Return* no.

Observe that neither PAST nor WAIT contain more than one copy of the same symbolic state. Also, note that the delay operation and invariant conjunction are performed before a state is stored in WAIT and not when it is picked from WAIT. This makes it easier to find invariant violations in an efficient way. Invariant violation are described in 2.3.4.

Committed Locations and Urgency

In section 2.1 we discussed some extensions utilised in the modelling language of Uppaal. This section describes the implications they have for the reachability algorithm presented earlier. There are three extensions that change the transition semantics and their implications are summarised below:

- The automata may not delay in an urgent location.
- The automata may not delay in a location if there are enabled outgoing transitions synchronising with an urgent action.
- No delay is allowed in a committed location.
- Not all outgoing transitions may be performed if the network is in a committed location even if their guards are true.

The first three conditions may be summarised as a condition on the networks possibility to allow delay transitions and will affect the *sp*-operation. The last requirement imposes a restriction on what components shall be inspected when searching for successor states. We can now define four predicates on symbolic states.

Definition 7 *(Predicates on symbolic states) Let $s = \langle L, V, U \rangle$ be a symbolic state.*

1. *$committed(L)$ is true if at least one of the components is in a committed location.*
2. *$urgs(L)$ is true if at least one of the components is in an urgent location.*
3. *$urgt(L, V)$ is true if at least one of the enabled compound transitions synchronises on an urgent channel.*
4. $$delay(s) = \neg(committed(L) \vee urgs(L) \vee urgt(L, V))$$

$\square$

In fact, algorithm 1 is a bit simplified to make the presentation of all reachability algorithms easier. In order to utilise the described extensions it is necessary to replace step 4 in *all* reachability algorithms with the ones below.

4' If $\not{c}ommitted(L_i)$ then find all L_k that are reachable from L_i in one step regardless of guards and resets, taking only actions into account. For all such transitions do

(a) Let G be the guard and R the resets on the performed transition. i_k is the location invariant of L_k.

(b) If V_i satisfies the guard, and $delay(\langle L_i, V_i, U_i \rangle)$, let $U_k = sp(reset(U_i \cap G, R)) \cap i_k$ and update V_i.

(c) Else if V_i satisfies the guard, let $U_k = sp(reset(U_i \cap G, R)) \cap i_k$ and update V_i.

(d) If $U_k \neq \emptyset$, store $\langle L_k, V_k, U_k \rangle$ in WAIT, WAIT $=$ WAIT $\cup \langle L_k, V_k, U_k \rangle$.

4" Otherwise, find all L_k that are reachable from L_i in one step regardless of guards and resets taking only actions into account. For all such transitions where at least one component is in a committed location in L_i do

(a) Let G be the guard and R the resets on the performed transition. i_k is the location invariant of L_k.

(b) If V_i satisfies the guard, and $delay(\langle L_i, V_i, U_i \rangle)$, let $U_k = sp(reset(U_i \cap G, R)) \cap i_k$ and update V_i.

(c) Else if V_i satisfies the guard, let $U_k = sp(reset(U_i \cap G, R)) \cap i_k$ and update V_i.

(d) If $U_k \neq \emptyset$, store $\langle L_k, V_k, U_k \rangle$ in WAIT, WAIT $=$ WAIT $\cup \langle L_k, V_k, U_k \rangle$.

Observe that committed locations are not saved in PAST in the current UPPAAL implementation of algorithms 1 and 2. This is an optimisation to keep PAST smaller. It is justified by the fact that the intended use of committed locations, described earlier, encourage that they will never be revisited by the state-space search. However, a mis-use of committed locations in loops so that all states in a loop are committed could cause the verifier not to terminate. The loop detecting algorithms in section 5.3 can be used to detect such mis-use and still keep PAST small.

2.2.4 Implementation Issues

The reachability algorithm is a standard graph-searching algorithm. It is implemented using two important data structures: WAITand PAST. These structures are very large and it is extremely important that they are efficient to access and do not consume unnecessary memory. If the WAITdata structure is a queue the search order is breath-first; if it is organised as a stack, the search become depth-first. The PAST data structure is implemented as a hash table so it can be searched and updated efficiently. These structures will be discussed in much greater detail in chapter 5.

Difference-Bound matrices

One of the main principles when designing the UPPAAL modelling language was to keep it as simple as possible but still expressive. It should be possible to implement model-checking efficiently. The class of time constraints needed to model the time behaviour of timed automata can always be expressed as $l_i \leq x_i \leq u_i, l_{i,j} \leq x_i - x_j \leq u_{i,j}$ where $x_i, i = 1, ..., n$ are the clocks in the automata. By introducing an extra constant clock $x_0 = 0$ and observing that $l_{i,j} = -u_{j,i}$ we can express the constraints as $x_i - x_j \leq u_{i,j}, i = 0, ..., n, j = 0, ..., n$. This equivalent representation makes it possible to develop and formalise all the constraint operations using only upper bounds. The representation can be viewed as a $(n+1) \times (n+1)$ matrix where element (i, j) contains $u_{i,j}$, the upper bound of $x_i - x_j$. These kind of matrices are called difference-bound matrices or DBM for short. It can also be thought of as a weighted graph with $n+1$ nodes, one for each clock, and directed edges with weight $u_{i,j}$. For a detailed treatment of the DBMalgorithms computing the constraint operations in section 2.2.2 see [BL96].

2.3 Safety Properties

This section describes the UPPAAL query language and different properties that can be verified with reachability analysis. We start with a discussion on safety properties and continue with other properties of more diagnostic nature.

2.3.1 Simple Reachability Properties

The essential class of properties that UPPAAL check is reachability properties. The query language of UPPAAL is based on propositional logic extended with quantifiers that let us formalise properties about paths and time. If φ is a propositional formula on states the following safety properties can be checked:

- $\forall \Box \varphi$ is true if φ holds in all states along all paths.
- $\exists \diamond \varphi$ is true in a state if there exists a path such that φ will eventually hold.

In fact UPPAAL only checks properties of the form $\exists \diamond \varphi$ because the other formula can be transformed to that one. A restricted class of liveness properties, called bounded liveness, and bounded leads-to properties may also be verified using the algorithm described in the previous section by use of test automata or decoration [LPY98].

The following algorithm is an adaption of algorithm 1 suitable for verification of safety properties. PAST and WAIT are as before, $\langle L_0, V_0, U_0 \rangle$ is the initial state and the property to check is $\exists \diamond \varphi$.

Algorithm 2 *(Model-checking a property)*
Initial conditions: WAIT $= \langle L_0, V_0, U_0 \rangle$, PAST $= \emptyset$.

1. *Pick a state $\langle L_i, V_i, U_i \rangle$ from* WAIT.
2. *If $\varphi(L_i, V_i, U_i)$, return* yes.

3. *If* $L_i = L_j \wedge V_i = V_j \wedge U_i \subseteq U_j$*, for some* $\langle L_j, V_j, U_j \rangle \in \mathsf{PAST}$*, drop* $\langle L_i, V_i, U_i \rangle$ *and go to step 1. Otherwise save* $\langle L_i, V_i, U_i \rangle$ *in* $\mathsf{PAST}, \mathsf{PAST} = \mathsf{PAST} \cup \langle L_i, V_i, U_i \rangle$*. If* $U_j \subset U_i$ *we can replace the state* $\langle L_j, V_j, U_j \rangle$ *with* $\langle L_i, V_i, U_i \rangle$ *to reduce the generated state-space and speed-up termination.*

4. *Find all* L_k *that are reachable from* L_i *in one step regardless of guards and resets, taking only actions into account. For all such transitions do*

   (a) *Let* G *be the guard and* R *the resets on the performed transition.* i_k *is the location invariant of* L_k*.*

   (b) *If* V_i *satisfies the guard, let* $U_k = sp(reset(U_i \cap G, R)) \cap i_k$ *and update* V_i*. If* $U_k \neq \emptyset$*, store* $\langle L_k, V_k, U_k \rangle$ *in* WAIT*,* $\mathsf{WAIT} = \mathsf{WAIT} \cup \langle L_k, V_k, U_k \rangle$*.*

5. *If* $\mathsf{WAIT} \neq \emptyset$ *go to step 1.*

6. *Return* no.

Note that the formula φ is evaluated in each state. All resets on a transition are assumed to be performed atomically and the system is only observable in the states. The propositional formula φ may contain the usual boolean connectives together with expressions on locations, clocks and data variables and location formulas. The expressions on clocks and variables are of the same form as guards on transitions while location formulas make it possible to ask if the network is in a certain combination of locations.

Note that there are some optimisations that can be performed as well. Let $\langle L_i, V_i, U_i \rangle$ be a symbolic state that satisfies a property φ. If $suc(L_i, V_i, U_j)$ is a symbolic state such that $U_j \subseteq U_i$ we know that that state satisfies φ as well. It means that we can terminate a search-branch in the state-space, i.e. stop exploring a state $\langle L_i, V_i, U_i \rangle$, as soon as we find a state $\langle L_i, V_i, U_k \rangle \in \mathsf{PAST}$ such that $U_i \subseteq U_k$ since the result of φ will not be affected and no new successors will be generated. This is utilised in step 3 of the algorithm above and is also the reason for the swapping of states in the PAST set in step 4. These optimisations can also be used when determining if a generated successor state needs to be stored in the WAIT set. This optimisation is important since the number of symbolic states generated when terminating on equality, like in algorithm 1, is very large compared to what is generated when terminating on inclusion.

The following sections deal with properties of a more diagnostic nature that can be verified using reachability analysis. The algorithms that can be used to check them are very similar to either algorithms 1 or 2 and will be described as modifications of those.

2.3.2 Unreachable Locations and Transitions

The main reason for the success of on-the-fly verification is that many locations in the product automaton will never be reachable. It may be the case that certain locations in the individual automata will never be reachable. To implement a check for unreachable locations each automaton is equipped with a table holding information about what states have been visited. These state tables are updated during a complete state-space exploration. After termination the state tables will indicate what states were visited.

The algorithm is obtained by modifying algorithm 2 as follows: We remove steps 2 and 6 since we want to explore the whole state-space. Further, we insert an extra step 4' after step 4 in algorithm 2:

4' Update the state tables for all components by marking state L_i as visited.

The algorithm may be easily modified to check for transitions never performed. Each transition is marked with a flag indicating if it has been performed or not. Transitions may be unreachable either because their guards never become true or they connect unreachable locations to reachable or unreachable locations. The optimisations used when checking propositional-logic properties may also be used here as long as we do not want exact measures on how many times a transition is performed or a location is visited. The search order might affect such measures as well so it is probably not useful.

2.3.3 Dead-locks and Live-locks

If a network of timed automata enters a symbolic state it can never leave it must be considered an error in the model and should be reported to the user as a dead-lock state. Some of these dead-lock states may be reported as live-lock states instead if the model may delay an infinite time because no outgoing transitions are enabled and no invariant is associated with the location. Note that the network has a possibility to skip enabled transitions and possibly delay for an infinite time in a location. According to the semantics of timed automata, such behaviour is not considered a live-lock.

Definition 8 *(Dead-lock states) Let $\langle L_i, V_i, U_i \rangle$ be a symbolic state and $suc(L_i, V_i, U_i)$ be the set of successors. $\langle L_i, V_i, U_i \rangle$ is a dead-lock state iff $suc(L_i, V_i, U_i) = \emptyset$* □

Definition 9 *(Live-lock states) Let $s = \langle L_i, V_i, U_i \rangle$ be a symbolic state that is a dead-lock state. Let x be a time assignment in U_i. s is a live-lock state iff $z_i = x_i + d, d \in \mathbf{R}^+$ is a time assignment in U_i for all values of d.* □

To detect if a time zone allows the possibility of delaying an infinite amount of time we transform the corresponding constraint system to the canonical form. If u_i is the upper bound for clock x_i we have such possibility if $\exists i \, : \, u_i = \infty$.

An algorithm to check a model and report dead-locks and live-locks according to the definitions above can be based on algorithm 1. Again, we remove steps 2 and 6 and insert an extra step 4' shown below. The algorithm traverses the state-space of the model and counts the number of successor states for each state.

4' If $\langle L_i, V_i, U_i \rangle$ has zero successors and does not contain infinite delays, report it as a state containing dead-locks. If $\langle L_i, V_i, U_i \rangle$ has zero successors and contains infinite delays, report it as a state containing a live-lock.

This algorithm is very expensive since it only allows us to terminate state-space exploration on equality between time zones and not inclusion. This is the main reason for modifying algorithm 1 and not algorithm 2. If we choose to terminate on inclusion the algorithm will not find all dead-locks but it would be efficient and still give some feed-back to the user. The reason why not all dead-locks will be detected is the following: If $\langle L_i, V_i, U_i \rangle$ and $\langle L_i, V_i, U_j \rangle$ are two symbolic states satisfying $U_j \subset U_i$ and $\langle L_i, V_i, U_i \rangle$ does not contain any dead-locks it is still possible that $\langle L_i, V_i, U_j \rangle$ or its successors may do so. The discarding of states in WAIT and PAST cannot be used either since it relies on the same reasoning as terminating on inclusion and prevents successors from being explored; successor states with "small" time zones containing very few clock assignments are possible candidates for such dead-lock states; the smaller a time zone, the larger the probability that dead-locks occur. The search order, i.e. breath-first or depth-first, will affect the reporting of dead-locks and live-locks. The reason is that the contents of WAIT is affected by the search strategy used and its contents influences what successor states will be explored later.

It must be remembered that our dead-lock check is a partial check. Even if it does not find any dead-locks the model may still contain them in practise. This has to do with the fact that symbolic states group multiple real states together and the algorithm only detects if all such states in a symbolic state are states without enabled out-going transitions. Some of the time assignments in a time zone may still produce dead-lock situations if the model is simulated or implemented an executed as a computer program.

2.3.4 Violation of Invariants

When computing successors of a symbolic state each outgoing transition is examined and those leading to inconsistent time zones or unsatisfied guards on the data variables are considered disabled. However it may be the case that transitions lead to consistent time zones after conjunction of guards and handling of resets but become inconsistent when the invariant of the successor location is added. Of course these transitions shall be treated as disabled but it may be viewed as a modelling error if it is the invariant, and not the guard, that makes the time zone inconsistent and not reachable and such states shall be reported to the user. The algorithm can be derived from algorithm 1 by removing steps 2 and 6 and replacing step 4 by the one shown below.

4 Find all L_k that are reachable from L_i in one step regardless of guards and resets, taking only actions into account. For all such transitions do

(a) Let G be the guard and R the resets on the performed transition. i_k is the location invariant of L_k.

(b) If V_i satisfies the guard, let $U'_k = reset(U_i \cap G, R), U_k = sp(U'_k) \cap i_k$ and update V_i. If $U_k \neq \emptyset$, store $\langle L_k, V_k, U_k \rangle$ in $\mathsf{WAIT}, \mathsf{WAIT} = \mathsf{WAIT} \cup \langle L_k, V_k, U_k \rangle$.

(c) If $U'_k = \emptyset$, report it as invariant violation state.

The remarks made in the discussion of the dead-lock check also applies here. No optimisations for faster termination such as terminating on inclusion or replacing of states as in algorithm 2 can be used if we want to report all invariant violations. The method of finding invariant violations is also partial and does not detect if there exists some states in a symbolic state violating the invariant. However it is easy to extend the algorithm to discover that as well. If $U'_k \subset U_k$ we can deduce that U'_k contained states violating the invariant. The algorithm will run a little bit slower because we must save U'_k and compare it to U_k. However, this extra cost may be worth its price because the optimisations for faster termination is applicable.

2.3.5 Bounded Domains of Data Variables

As mentioned earlier, data variables have domains and wrap-around semantics is used when computations yield results outside the domains. Even if this is a well-defined behaviour, wrap-arounds may cause errors and a user should be aware of all wrap-arounds that occur in the model. This is accomplished by performing all computations internally in UPPAAL using temporary variables with domains large enough to catch wrap-arounds in the model.

Definition 10 *(Domain wrap-around) Assume that data variable d has a domain $[d_{min}, d_{max}]$. Let $d_w = d_{max} - d_{min} + 1$ be the size of the domain of d. A wrap-around in the expression $d = E$ occurs if $E > d_{max}$ or $E < d_{min}$. The new value of d is x such that $d_{min} \leq x \leq d_{max}$ and $\exists n \in \mathbf{Z} \,:\, e = d_w n + x$.* □

How do we choose domains large enough internally? For efficiency reasons the data variables in UPPAAL are mapped to a built-in data type in the verifier's implementation language instead of performing computations with dynamic bit fields. This means that the allowed domains in the model will be limited by the domain for that built-in data type. Choosing the largest built-in data type for the temporary variables, e.g. 64 bits, and restricting the allowed domain size for variables in the modelling language, e.g. 16 bits, is safe in all cases if each intermediate result of a binary operator is range-checked. Also, if a wrap-around occurs for the temporary variables in the implementation language we know that a wrap-around must occur in the modelling language as well.

Algorithm 2 can be modified to perform range checking during a state-space exploration. Remove steps 2 and 6 and replace step 4 with a new one shown below. For easier notation assume that $top(d_i)$ and $bot(d_i)$ return the maximum and minimum values in the domain of d_i respectively. A subexpression is a binary arithmetic expression that is part of a larger expression. As long as an arithmetic expression is built-up by binary operators it is possible to evaluate it by evaluating one binary subexpression at a time. If an expression is not partitioned into subexpressions there is a risk of undetected wrap-arounds of the data variables used in the implementation language.

4 Find all L_k that are reachable from L_i in one step regardless of guards and resets, taking only actions into account. For all such transitions do

   (a) Let G be the guard and R the resets on the performed transition. i_k is the location invariant of L_k.
   (b) If V_i satisfies the guard, let $U_k = sp(reset(U_i \cap g_k, r_k)) \cap i_k$.
   (c) For all expressions $d_i = E$ found in R when updating V_i do
      i. Let $tmp = e_i$ for all subexpressions e_i of E.
      ii. If $tmp > top(d_i)$ report domain overflow.
      iii. If $tmp < bot(d_i)$ report domain underflow.
   (d) If $U_k \neq \emptyset$, store $\langle L_k, V_k, U_k \rangle$ in $\mathsf{WAIT}, \mathsf{WAIT} = \mathsf{WAIT} \cup \langle L_k, V_k, U_k \rangle$.

Another question the user must answer is what are appropriate domains for the data variables in the model? It is possible to derive domains large enough given initial values and all computations. Let MIN and MAX be vectors containing the domains large enough so far for the data variables. They are initialised to the initial values of the corresponding data variables and updated while the whole state-space of the model is explored. An algorithm that performs a reachability analysis and derive appropriate domains for the data variables can be found by removing steps 2 and 6 from algorithm 2 and replace step 4 with the step shown below. min and max are standard functions present in all programming languages that return the minimum and maximum value of their arguments respectively.

4 Find all L_k that are reachable from L_i in one step regardless of guards and resets, taking only actions into account. For all such transitions do

   (a) Let G be the guard and R the resets on the performed transition. i_k is the location invariant of L_k.
   (b) if V_i satisfies the guard, let $U_k = sp(reset(U_i \cap G, R)) \cap i_k$.
   (c) For all expressions $d_i = E$ found in R when updating V_i do
      i. Let
      $$min_i = min(min_i, e_i), max_i = max(max_i, e_i)$$
      for all subexpressions e_i of E.
   (d) if $U_k \neq \emptyset$, store $\langle L_k, V_k, U_k \rangle$ in $\mathsf{WAIT}, \mathsf{WAIT} = \mathsf{WAIT} \cup \langle L_k, V_k, U_k \rangle$.

The optimisations used for faster termination are valid here. If $\langle L_i, V_i, U_i \rangle$ and $\langle L_i, V_i, U_j \rangle$ are two symbolic states satisfying $U_j \subseteq U_i$ their successors will produce the same data variable assignments. Thus, these algorithms can easily be combined with the verification of propositional-logic properties in algorithm 2.

2.4 Reporting Diagnostic Information

The most important feed-back from UPPAAL to the user is the diagnostic information [LPY95b]. that provides information about for example why a safety property is not satisfied. When the user verifies a model and some states are unexpectedly reachable it is of great importance to know what execution took the system to that state. The information provided by UPPAAL is a trace showing what transitions and synchronisations are performed and how clocks and data variable values change. To keep track of such information, each symbolic state maintains information about its parent state and what transition that was performed to advance the model from the parent state to that state. This information can then be used to extract information for the user.

The next issue is to decide what kind of trace to present to the user. It is easy to show a list of control locations and data variables together with the constraint systems describing when the system may stay in a location. This symbolic trace contains all details about the possible executions of the model but is hard to interpret. It is of greater help to see one sample execution showing real clock assignments and delays extracted from each time zone in the symbolic trace.

The question to answer is then how to extract such a diagnostic trace, i.e. how to find a point in each time zone. Finding a point in an arbitrary multi-dimensional convex polytope is computationally hard, especially when strict inequalities are involved. The approach taken in UPPAAL is totally different. Instead of trying to find a point given a time zone, we start with a clock assignment that we know is included in a time zone. We then use the transition to the succeeding state together with a time zone to derive how the clock assignment has changed. We find a start value for a vector x containing a time assignment by utilising that $x = \vec{0}$ is an assignment in the first time zone, namely the one associated with the initial state.

When calculating a trace we cannot utilise the symbolic states computed during the reachability analysis as they are. It is possible to reuse the discrete part, i.e. location vector and data variable assignment, without any complications but the time zones must be replaced. The zones derived during the verification does not contain enough information to produce correct delays in the trace. If we ask if a state is reachable within b time units, and it is, we do not want any delays in the trace to be larger than b time units even though such delays may be perfectly valid in the sense that the transitions to perform will be enabled. This information may only be derived from a backward calculation of time zones starting in U_{n+1} and ending in U_0. If these zones are denoted V_i and the zones derived from forward calculations are W_i the zones to use in the algorithm is $U_i = V_i \cap W_i$. Further, the zones U_i must not be normalised with respect to the maximal constants as such time zones are too abstract and does not contain enough information to compute appropriate delays. This means that we cannot re-use the zones calculated during forward reachability analysis either since normalisation is required for termination. To guarantee that the modified U_i will not be normalised it is enough to ensure that W_i or V_i is not normalised.

The algorithm below shows how to produce a diagnostic trace from a sequence of modified symbolic states according to the description above.. It uses the *delay*-predicate, see definition 7.

Algorithm 3 *(Computing a diagnostic trace)*

Input *A clock assignment* $x = \vec{0}$ *describing the clocks initial values together with a sequence* s_i *of symbolic states with modified time zones* U_i. *Assume* U_{n+1} *satisfy* φ, *the property we want more information on.*

Output *A trace showing one possible execution taking the system from the initial state* s_0 *to the state* s_{n+1} *ending the trace.*

1. *Let* G *be the guard and* R *the resets on the transition from* s_i *to* s_{i+1}.
2. *for* $i = 0$ *to* n *do*
   (a) *Display the discrete part of* s_i, *i.e. the control location and data variable assignment.*
   (b) *Display the current clock assignment* X.
   (c) *if delay*(s_i), *use* U_i, G *and* U_{i+1} *together with* X *to calculate a delay* d *and display it.*
   (d) *Update the current clock assignment* X *by adding the delay* $d, x_i = x_i + d$.
   (e) *Display synchronisation label on the transition from* s_i *to* s_{i+1} *if present, otherwise display a* τ*-action.*
   (f) *If there are any resets,* $x_k = v$ *in* R, *update* X *accordingly, set* $x_k = v$.

Note that this method of saving symbolic states consumes a lot of memory because we have to save all encountered states in the case they are part of a trace even though the reachability algorithms allow us to through them away from PAST. We have all information needed to recompute the symbolic states if we wish to save space by storing only the performed transitions or a part of a symbolic state. This is not described any further in this thesis.

To compute a delay between two states we observe that all clocks move with the same speed implying that bounds on clock differences do not affect the delay. Assume a time zone U with DBM $u_{i,j}$. Assume the transition between the symbolic state associated with U and the adjacent symbolic state with time zone U' has a guard G. The next algorithm shows how to obtain a delay d such that when it is added to $X \in U$ it gives a vector in U'.

Algorithm 4 *(Computing delays between symbolic states)*

input *The clock assignment X together with the associated time zone U, its adjacent time zone U' and the corresponding transition with guard G and resets R.*

output *A delay d that when added to $X \in U$ gives a vector in the adjacent time zone U'.*

1. *Let $U' = U \cap G$.*
2. *Set $d_{min} = max_i(-u'_{0,i} - x_i)$*
3. *Set $d_{max} = min_i(u_{i,0} - x_i)$*
4. *If d_{min} and d_{max} both are non-strict bounds, choose d to be an integer in the interval, e.g. d_{min}.*
5. *if d_{min} or d_{max} is strict and $d_{max} - d_{min} \geq 1$ choose d to be an integer in the interval, there must exist one.*
6. *if d_{min} or d_{max} is strict and $d_{max} - d_{min} < 1$ choose d to be a number in the interval with as few decimals as possible, in the worst case choose $d = \frac{d_{min}+d_{max}}{2}$.*

The algorithm above first finds an allowed interval for the delay and then tries to minimise the number of decimals in the fractional part of d. That is the main reason for not using interval splitting setting $d = \frac{d_{min}+d_{max}}{2}$. An alternative approach would be to strive for a minimal number of decimals in x_i, the result is expected to be very similar in most cases.

Chapter 3

Network Structure

This chapter looks at data structures for representing the system and processes, i.e. the automata network and algorithms for finding enabled transitions from a state. It discusses how states and transitions can be represented and some pre-processing that may be done to speed up the access of transitions in the transition relation. Further it covers how to represent the synchronisation mechanism and how to use static analysis to build data structures speeding up searches for transitions matching a given synchronisation label.

3.1 States and Transitions

3.1.1 Elementary Data Representation

In the UPPAAL model, no processes can be created or destroyed automatically i.e. the process model is static and the number of automata in the network known before the verification, after parsing the system. A suitable data structure for representation of the network is therefore an array containing the processes. Such a structure allows fast access to individual components which is important when searching the network to find transitions from a symbolic state. In the literature, automata are often described as tuples of the form $\langle S, T, i \rangle$ where S is the set of states, $T \subseteq S \times S$ the set of transitions and $i \in S$ the initial state. Further, transitions in networks of timed automata not only has start and ending states but also guards, resets and synchronisation labels. In the UPPAAL modelling language synchronisation channels also have attributes such as urgent. States have attributes as well, e.g. committed and urgent, see 2.

According to the mathematical description of an automaton we could represent a transition as follows assuming that the abstract data type for a state already is defined:

- reference to the start state of the transition
- reference to the end state of the transition

- information about guards
- information about resets
- synchronisation information

References are often represented with pointers in the implementation language. However, it is unnecessary space-consuming to store pointers to states. Instead we let each process keep an array of all states and store an integer, holding the position of the state in the array, in the transition instead of a pointer to the state. The abstract data type describing a state must contain information about attributes and invariants. The attributes may be represented with a bit field or Boolean variables. This information is then used to evaluate the predicates $urgs$ and *committed* defined in definition 7. Invariants may be represented as a vector with upper bounds for the clocks.

Space become even more important when finding representations for symbolic states. Each symbolic state must contain information on what the locations of the individual automata are as well as clock constraints and the assignment of data variables see chapter 4. As pointed out earlier, the number of automata are known and we can reference each component with a number and use an array in analogy with states and transitions. An array of integers is much more space-efficient than an array of references to each component in each symbolic state.

Regarding the guards and resets on each transition, we might represent them using linked lists. It is possible to use array representations here as well but this information will only be represented once so a linked list structure will do. The number of guards and resets varies for each transition and when computing successors to a symbolic state, we will traverse the lists. We postpone the discussion on synchronisation to the next section.

To summarise, a process now contains an array of states, an integer holding the position of the initial state and a set of transitions. The set of transitions can be represented as a $n \times n$-matrix if there are n states in the automaton. Each matrix entry (i, j) contains a list of transitions connecting states i and j. However, this matrix will probably contain many empty entries and instead of a large sparse matrix a linked lists with all transitions will be used.

We are now ready to write an algorithm searching all pairs of components to find outgoing transitions, either with matching synchronisation labels or no labels at all, from a symbolic state. For a backward analysis we would have searched for incoming transitions instead. For the moment we ignore the time zone and data variable assignment parts of the symbolic state. Thus, we only inspect the control structure of the networks for possible candidates of enabled transitions. Let $s(t)$ and $e(t)$ be functions returning the states a transition t connects. Depending of the direction of the reachability analysis the definitions for $s()$ and $e()$ shall be interchanged. Further, let tl_i be the list of transitions belonging to automaton i. Let L be a data structure with components L_i holding the current locations of all automata. This notation is not consistent with the notation used in the previous chapter but is convenient in this context.

Algorithm 5 *(Finding transitions from the current location)*

Input *A data structure describing a system consisting of n automata.*

Output *A set of transitions T.*

- *for $i = 1$ to n do*
  - *for $j = i$ to n do*
    - * *if $i = j$, find all $t \in tl_i$ with τ-actions satisfying $s(t) = L_i$ and let $T = T \cup t$.*
    - * *else if $i \neq j$, find all pairs $(t_i, t_j) \in tl_i \times tl_j$ with matching synchronisation labels satisfying $s(t_i) = L_i \wedge s(t_j) = L_j$. Let $T = T \cup (t_i, t_j)$.*

The algorithm goes through each individual component and all pairs of components in the network searching for transitions with no synchronisation label or pairs of transitions with matching labels. Note that the second loop starts at $j = i$ to avoid searching for transitions common to components i and j twice. For each pair and single component it scans the lists of transitions comparing their start state to the corresponding component of the location vector of the current symbolic state. When a match is found, it indicates that outgoing transitions have been found. The algorithm is almost independent of the direction of the reachability analysis.

3.1.2 Sorting Transitions

The algorithm described above contains some unnecessary overhead. It searches all the transitions of the component pair every time in order to find transitions with states matching the location vector before it can check for matching synchronisation labels. If we pre-process the list of transitions for each component and split them into arrays of lists where each list only contains the transitions whose start states are identical we might speed up the algorithm. This sorting of transitions only needs to be done once for each system. We build the array in a way that matches the order of the states in the process's array of states and the location vectors of the symbolic states. We shall expect a noticeable speed-up, especially if the number of states and transitions in the system is large. The modified algorithm is shown below. We use the same notion as in the previous section but there is no longer any need for the $s()$ and $e()$ functions. Instead of the transition lists tl_i we now use a structure $tl_i(j)$ containing the transitions of component i leading from or to state j depending on the direction of the reachability analysis. As before, L is the location vector with components L_i.

Algorithm 6 *(Finding transitions from the current location)*

Input *A data structure describing a system consisting of n automata.*

Sample	Time sorted sec	Time unsorted sec	red %
audio	0.12	0.13	7.7
audio_bus	2.3	3.3	30.3
B&O	26.5	36.1	26.6
brp	1.1	1.3	15.4
dacapo_s	6.4	9.0	28.9
dacapo_b	13.7	19.2	28.7
engine	4.7	5.8	19.0
mplant	0.8	0.9	11.1
fischer4	0.6	0.6	0
fischer5	17.8	18.0	1.1
fischer6	1492	1504	0.8

Table 3.1: Performance of Transition Sorting

Output *A set of transitions T.*

- *for $i = 1$ to n do*
  - *for $j = i$ to n do*
    - *if $i = j$, find all $t \in tl_i(L_i)$ with τ-actions. Let $T = T \cup t$.*
    - *else if $i \neq j$, find all pairs $(t_i, t_j) \in tl_i(L_i) \times tl_j(L_j)$ with matching synchronisation labels. Let $T = T \cup (t_i, t_j)$.*

Note that the algorithm now depends on the direction of the reachability analysis even though the preprocessing is easy to adapt. Table 3.1 compares verification time for two versions of Uppaal one using the sorting of transitions, and one that does not.

As can be observed in the Table, it is well worth the extra effort building this slightly more complex data structure for storing transitions. The time spent on inspecting the control structure is reduced a lot leading to an decrease in total verification time between 25 and 30 %. The more states in the individual automata the more is the speed-up. In the next section we look at additional optimisations of the algorithm.

3.2 Handling Synchronisation

3.2.1 Synchronisation Labels

Each transition carries information about its synchronisation label. Since no multi-synchronisation is supported it is enough to compare synchronisation labels for components two at a time. To achieve a fast test we assign an integer $n(c)$ to each channel c. We then represent $c!$ as $+n(c)$ and $c?$ as $-n(c), c = 0$ if no synchronisation label is present. The test of two transitions for matching synchronisation labels is shown in the following definition.

Definition 11 *(Matching synchronisation labels) Assume two components i and j of a network and two of their transitions with synchronisation labels $c_i \neq 0$ and $c_j \neq 0$. The transitions match if $n(c_i) + n(c_j) = 0$.* □

This test will be very efficient since it can be realised using built-in data types and only relies on an addition and a comparison to zero. However, there are still room for some improvements of the algorithms presented in the previous section.

In most real-world systems, not all components synchronise with all the other and components do not synchronise with each other in every state. This can be utilised by building a data structure keeping track of which components synchronise on which channels in which states. Again, this is a structure that only needs to be built once for a system and all information needed is known before verification.

We extend each component with two arrays $ib(l)$ and $ob(l)$ holding bit fields for each location l. If the network has m channels the bit fields are of size m. A bit B in $ob(l)$ is set to 1 if the component wants to send a message in location l on the channel mapped to integer B, i.e. any of its outgoing transitions has a synchronisation label $c!$ and $n(c) = B$. All other bits are zero. Similarly, a bit in $ob(l)$ is set to one if a component wants to receive a message on the corresponding channel in location l. We can now define a Boolean predicate $sync(i, j, L)$ in the following way:

Definition 12 *(Common synchronisation for two components) Assume two components i and j and a symbolic state with location vector L.*

$$sync(i, j, L) = ((ib(L_i) \wedge ob(L_j)) \vee (ob(L_i) \wedge ib(L_j)))$$

□

This is a predicate that can be evaluated using bitwise logic and the tables it relies on are easily represented using built-in data types. Hence the test is cheap to execute. Below is the algorithm using the *sync* predicate.

Sample	Time without *sync* sec	Time with *sync* sec	red %
audio	0.12	0.12	0
audio_bus	2.5	2.3	8.0
B&O	28.1	26.5	5.7
brp	1.2	1.1	8.3
dacapo_s	6.6	6.4	3.0
dacapo_b	14.2	13.6	4.2
engine	4.8	4.6	4.2
mplant	0.8	0.8	0
fischer4	0.6	0.6	0
fischer5	17.9	17.8	0.6
fischer6	1495	1472	1.6

Table 3.2: Performance of the *sync* Predicate

Algorithm 7 *(Finding transitions from the current location)*

Input *A data structure describing a system consisting of* n *automata.*

Output *A set of transitions* T*.*

- *for* $i = 1$ *to* n *do*
  - *for* $j \in [i, n]$ *satisfying* $sync(i, j)$ *do*
    * *if* $i = j$*, find all* $t \in tl_i(L_i)$ *with* τ*-actions. Let* $T = T \cup t$*.*
    * *else if* $i \neq j$*, find all pairs* $(t_i, t_j) \in tl_i(L_i) \times tl_j(L_j)$ *with common synchronisation labels. Let* $T = T \cup (t_i, t_j)$*.*

The static information is in fact an over-approximation of whether the components really can synchronise; we have no guarantee that the transitions really are enabled. It depends on the values of the clocks and data variables and that information is generally not available before verification. Thus, the Boolean predicate *sync* is a necessary condition for synchronisations to occur but not sufficient. Table 3.2 shows a comparison of verification times when *sync* is used. We expect speed-ups for examples with many states and transitions and modest synchronisation behaviour.

The Table clearly indicates that it is well worth doing the extra preprocessing building the data structures needed by the predicate for networks with a low amount of synchronisation. In other cases the increase in performance is more modest. The reason is mainly because the transition lists for each of the locations are relatively short. Further, the time spent by UPPAAL examining the network structure is low compared to other tasks. As soon as the number of transitions and the number of independent components increase, so will the performance of this optimisation. The results had been more noticeable if the synchronisation predicate is used with the first representation of the transition relation where transitions were not sorted on states.

It is possible to speed up the search for matching synchronisation labels when the predicate indicates that such exists by sorting the transition lists for each state using $n(c)$ as sorting criteria. Instead of a linear search we may perform a binary search or at least terminate the linear search earlier. Since most UPPAAL models does not have that many channels, this technique has not been implemented.

3.2.2 Urgent channels

As mentioned in chapter 2 urgent channels differs from other channels since they affect the delay transitions, see definition 7. This means that before we compute delay transitions we must test if any of the components currently are in locations with enabled transitions with urgent channels. We can build a similar structure with bit fields telling if components are in locations with matching urgent channels but since we also want to know if the guards on the data variables on the transitions are enabled we need to know what guards to evaluate, not only if urgent channels exists.

To develop a test that still is efficient we construct a table according to the scheme above with bit fields describing in what locations urgent channels exists. We then define a predicate similar to *sync* that describe if two components are in states containing transitions equipped with urgent channels. If that predicate indicate that such transitions exist we scan through all transitions leaving that state and evaluate the guards on the transitions having urgent channels. In this way we achieve a test that is as cheap as the *sync* predicate when it fails, as will be the most common case, and only requires computation when transitions exists. This is how the *urgt* predicate in definition 7 is implemented. We still have the problem that the table will indicate possible transitions with urgent channels even if the transition is not enabled.

Chapter 4

Symbolic States

One of the most important parts of UPPAAL is the handling of symbolic states. As defined in chapter 2 a symbolic state contains the location vector, clock constraint system and data variable assignment. The time and space performance is heavily dependent on what algorithms and data structures used to represent and manipulate them. This is the subject of this chapter. The first two sections deal with the continuous part of the symbolic state; discussing the representation and manipulation of clock constraints. It describes how to extend the DBM and its operations for handling strict inequalities. It also covers a graph-theoretical approach that makes it possible to remove redundant constraints and how it influences the performance of verification. The closing section is about the discrete part of a symbolic state; the control location and data variables. The main point is the use of some restrictions that enable us to use built-in data types and operators leading to major increases of space and time performance.

4.1 Constraint Operations

In chapter 2 the model of timed automata was described together with the operations needed to perform forward reachability analysis. The timing constraints could be described as

$$x_i - x_j \sim b_{i,j}$$

where x_i and x_j are clocks, $b_{i,j}$ is an integer and $relop \in \{<, \leq, =, \geq, >\}$. Obviously a matrix is a suitable data structure for the clock constraints. In chapter 2 we introduced such a matrix structure called a DBM. Further, it is possible to find efficient matrix operations corresponding to the constraint operations needed. The following sections describe the DBM operations that correspond to the constraint operations defined in chapter 2 as well as how to adopt them to be able to handle strict inequalities almost as efficient as non-strict.

Conjunction

Conjunction of constraints to a constraint system is used when evaluating state invariants and transition guards. The following algorithm shows how to calculate the new DBM when a non-strict guard is added. The function $min(x, y)$ is a standard function to return the minimum of the integers x and y. As long as no non-strict constraints are involved it can be efficiently implemented using built-in data types and comparison operators.

Algorithm 8 *(Conjuncting constraints)*

Input *A* DBM U *corresponding to a constraint system* $x_i - x_j \leq u_{i,j}$ *and a constraint* G *of the form* $x_k - x_l \leq c$ *to conjunct.*

Output *A* DBM U' *describing the constraint system after conjunction.*

- *set* $u_{k,l} = min(u_{k,l}, c)$

The algorithm only inspects the DBM element corresponding to the constraint involving the same clocks as the conjunct. This makes the conjunction a constant-time operation and very efficient. However, in order to know if the constraint system satisfies the conjuncted constraint or not it is not sufficient to check one DBM element. The test for inconsistency of a DBM is computationally heavy and it is therefore more efficient to conjunct all guards on a transition, or all invariants in a state, before checking for consistency. This approach does not allow us to use short-circuit evaluation of multiple guards or invariants but since the conjunction itself is not an expensive operation we do not need short-circuit evaluation for efficiency and the inconsistency check can be postponed and done once for each set of guards on a transition or invariants in a state.

Computing the canonical form of a Constraint System

As pointed out in chapter 2, many of the constraint operations require the DBM to be in a canonical form. Intuitively it means that we want to derive the lowest possible upper bound for each clock difference using the constraints in the system. To achieve that we model the constraint system as a weighted graph as described in [YL93, Bel57] and also used in the context of analysing Petri nets described in [Rok93, RM94].

We have one node for each clock, including x_0, and an edge from node i to node j labeled by $u_{i,j}$. The different upper bounds for a clock difference $x_i - x_j$ now correspond to sums of weights along paths between node i and j. The upper bounds in the canonical representation are the shortest paths, the paths with the lowest sum of weights, in the corresponding graph.

There are several efficient ways of finding all shortest paths for a weighted graph represented as a matrix. We will use Floyd's algorithm [Flo62] since it is more suitable for a dense graph represented that way. Dijkstra's algorithm is more suitable for sparse graphs represented as adjacency lists. Initially, the matrix contains weights corresponding to the lengths of the paths from node i to node j without visiting any other node. We now compare these paths with the paths we get if we are allowed to visit node 0 when moving from node i to node j and if any such path is shorter we store that one instead. In that way the matrix element in position (i, j) always contains the shortest path from node i to node j encountered so far. We now repeat the procedure but now we are 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 the constraint system in its canonical form.

Algorithm 9 *(Canonical form)*

Input *A* DBM U *containing a constraint system.*

The same DBM U *containing the same constraint system on the canonical form.*

- *For every node k do*
  - *For every node i do*
    - *For every node j do*
      - *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 the transformation will be of complexity $O(n^3)$.

There is one requirement that the graph must satisfy when using 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. It can be shown that a graph modelling a constraint system never contain negative cycles if the constraint system is consistent. If a negative cycle exists in the graph it will be preserved by the shortest-path algorithm.

Consistency Checking

An inconsistency in a constraint system arises if a lower bound for a clock difference is greater than the corresponding upper bound, i.e. the length of the allowed interval for a clock difference is less than zero. The algorithm below shows how to detect such inconsistencies in a DBM.

Algorithm 10 *(Consistency)*

Input *A* DBM U *representing a constraint system on canonical form.*

Output true *if the constraint system is inconsistent,* false *otherwise.*

- *If* $u_{i,i} < 0$ *for any* i*, return* true*, else return* false.

Fact 2 *The above algorithm returns an indication of whether the constraint system on canonical form, represented by the input* DBM*, is consistent or not.*

Proof In a DBM $u_{i,j}$ representing a constraint system U on canonical form the length of an interval for $x_i - x_j$ can be expressed as $u_{i,j} + u_{j,i}$. The smallest interval for a clock x_i is

$$min_j(u_{i,j} + u_{j,i}) = u_{i,i}$$

$u_{i,i}$ is also the shortest cycle from node i back to itself in the graph model of U. If $u_{i,i} \geq 0$ for all i the graph is free of negative cycles and the constraint system is consistent. If a negative cycle occurs for any node i the length of the allowed interval for that clock is negative and the constraint system is inconsistent. □

The above algorithm examines the diagonal elements in the DBM and is of complexity $O(n)$ if we have n clocks in the constraint system. If we restrict guards and invariants to expressions on individual clocks rather than clock differences an inconsistency may only occur among the elements in the first row and column and it is enough to look at the matrix element $(0, 0)$ yielding an $o(1)$ algorithm.

Relationship Checking

When performing a state-space exploration we need to determine if the solution set of one constraint system is contained in another or not. We say that a constraint system U is included in U' if all solutions in U also appears in U'. If there exist solutions in U not present in U' and vice versa none of the constraint systems is included in the other. If both U and U' are on canonical form there is an easy way to determine their relationship by comparing the elements in their DBMs.

Algorithm 11 *(Relationship)*

Input *Two* DBM*s, representing two constraint systems* U *and* U' *on canonical form.*

Output *A value expressing the relation between the constraint systems.*

- *If* $u_{i,j} \leq u'_{i,j}$ *for all* i *and* j *then* U *is included in* U'.
- *If* $u'_{i,j} \leq u_{i,j}$ *for all* i *and* j *then* U' *is included in* U.
- *If neither of the above conditions hold then* U *and* U' *both have solutions not present in the other.*

Fact 3 *If both constraint systems are on canonical form the above algorithm will correctly determine if one of them is included in the other or not.*

Proof We have the following cases: Either all elements in one DBM are less than or equal to the corresponding elements in the other or they are not.

Let $x \in U$ and $z \in U'$ be vectors containing time assignments. 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 $x \in U'$. The same reasoning can be applied if $u'_{i,j} \leq u_{i,j}$ to show that $z \in U$.

Next, 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. Since the DBMs are on canonical form there exists x satisfying $x_k - x_l = u_{k,l}$ and z satisfying $z_i - z_j = u'_{i,j}$. Since $x \notin U'$ and $z \notin U$ 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. We can speed it up if one uses the fact that if some bounds have a relation indicating that none of the constraint systems is included in the other, the other bounds do not need to 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 element because this will slow the whole algorithm down more than if we omitted the optimisation.

Computing Strongest Post-Condition

The strongest postcondition $U' = sp(U)$ is a constraint system containing all clock assignments that are reachable from assignments in U when an arbitrary positive delay $d \geq 0$ is added to them. It simulates that the clocks in the model moves with the same speed. An assignment in U corresponds to multiple assignments in U', one for each d. The operation is sometimes called the up operation because the upper bounds for the clocks are relaxed.

Algorithm 12 *(Strongest post-condition)*

Input *A* DBM U *describing a constraint system on canonical form.*

Output *A* DBM U' *describing a constraint system on canonical form representing the strongest post-condition of* U*.*

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

Fact 4 *The algorithm described above will give a constraint system representing the strongest postcondition of a given constraint system according to definition 1.*

Proof Let U be $x_i - x_j \leq u_{i,j}$ and $U' = sp(U)$ be $z_i - z_j \leq u'_{i,j}$. Let D be a vector satisfying $d_i = d, d \geq 0$. Note that no delay is added to x_0 and bounds where $i = 0$ or $j = 0$ must be treated separately.

According to definition 1 in chapter 2 we can write all clock assignments z that after some delay d are reachable from a clock assignment $x \in U$ as $z = x + D$. We first calculate $u'_{i,j}, i \neq 0, j \neq 0$.

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

Next, we calculate $u'_{i,0}$ and $u'_{0,i}$

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

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

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

The algorithm has complexity $O(n^2)$ for a constraint system on canonical form where n is the number of clocks.

Reseting of Clocks

We want an operation that performs a reset $x_k = v_k$. Even though the case $v_k = 0$ is the most common we will present a general algorithm for resetting a clock to any $v \geq 0$.

Algorithm 13 *(Reset)*

Input *A* DBM U *describing a constraint system on canonical form and a reset* $x_k = v, v \geq 0, x \in U$ *to perform.*

Output *A* DBM U' *describing a constraint system on canonical form when* x_k *is reset to* $v, x \in U$.

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

Fact 5 *The algorithm above gives a constraint system where clock* x_k *is set to a value* v *according to definition 3.*

Proof Let U be the constraint system before the reset operation and of the form $x_i - x_j \leq u_{i,j}$. Let U' be the constraint system after reseting x_k to v and of the form $z_i - z_j \leq u'_{i,j}$.

Let us first derive $u'_{i,j}$ the upper bound of $z_i - z_j, i \neq k, j \neq k$. According to definition 3 in chapter 2 x_i and x_j does not change when x_k is reset to v.

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

Let us now compute $u'_{i,k}$ and $u'_{k,i}$.

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

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

This completes the proof that our algorithm is correct. □

If n is the number of clocks the complexity of the reset operation is $O(n^2)$.

Normalisation

It was pointed out in chapter 2 that the maximal constants that a clock is compared to is important when determining if two states are equivalent. To avoid constructing a larger state-space than necessary each DBM is normalised with respect to these maximal constants of its clocks. Assume a vector C, of the same dimension as the number of clocks, with components c_i that contains the maximal constant for clock x_i. The constants may be obtained by inspecting the guards and invariants in the network and any constraints in the property. Note that since we rewrite all constraints as $x_i - x_j \leq u_{i,j}$ it is important to consider the numeric values when comparing constants that appear in the constraints. As an example, the guards $x \leq 3$ and $x \geq 10$ will be represented as $x \leq 3$ and $-x \leq -10$. The maximal constant for x is 10. It is also important that a constraint on a clock difference, $x_i - x_j \leq c$ where $i \neq 0$ and $j \neq 0$ is handled in a way that c is considered a candidate for c_i and c_j. The following fact shows why the algorithm below is correct.

Fact 6 *Assume a constraint system of the canonical form $x_i - x_j \leq u_{i,j}$ where clock x_i has maximal constant c_i. The greatest clock difference $x_i - x_j$ that the verification is sensitive to is c_i and the smallest is $-c_j$.*

Proof Since x_i and x_j are always non-negative the observation is trivial:

$$x_i - x_j \leq x_i \leq c_i$$
$$x_i - x_j \geq -x_j \geq -c_j$$

$\square$

It is not possible to guarantee that U' is in the canonical form.

Algorithm 14 *(Normalisation)*

Input *A* DBM *U representing a constraint system on canonical form.*

Output *A* DBM *U' representing a normalised constraint system.*

- *if $u_{i,j} > c_i$ set $u'_{i,j} = \infty$*
- *if $u_{i,j} < -c_j$ set $u'_{i,j} = -c_j - 1$*

Concluding Remarks

This section has shown algorithms for DBM manipulations that corresponds to the different operations defined in chapter 2. They are designed in a way that makes it possible to implement them in standard programming languages using integer manipulation instructions. The next section discusses how to introduce strict constraints and what to do to still keep the algorithms efficient even though the data structure to manipulate changes slightly.

4.1.1 Handling Strict Inequalities

The DBM data structure for representation of binary constraints assumes that the same binary operation and the same relational operation occur in all constraints. In our case all constraints are of the form $clock_i - clock_j$ so the first requirement is full-filled. If no strict inequalities are used it is possible to rewrite all constraints involving $=$ and $\geq$ to constraints involving $\leq$ but since we want to use strict constraints as well we also need $<$. To be able to use DBMs we need to keep some extra information for each constraint telling if it is strict or not. The easiest way is to borrow a bit from each matrix element and use it as a flag indicating if the relational operator is $<$ or $\leq$, i.e. if the comparison is strict or not. Unfortunately this will slow down the operation of many DBM algorithms as shown in the following example.

Example 1 *(Canonical form) Assume a constraint system of the form* $x_i - x_j \leq u_{i,j}, i, j = 0...n$ *where* x_i *and* x_j *are clocks. The* DBM *will be the* $(n+1) \times (n+1)$ *matrix with elements* $u_{i,j}$*. The algorithm for transforming a* DBM *to its canonical form can be written as follows:*

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

Now look at a constraint system that contains a mix of constraints either of the form $x_i - x_j < u_{i,j}$ *or* $x_i - x_j \leq u_{i,j}$*. The* DBM *will still have* $u_{i,j}$ *as elements but we also have the extra information about strictness to represent. Let* $s_{i,j} = 1$ *if the comparison operator for* $u_{i,j}$ *is* $<$ *and* 0 *otherwise. We still want to use the relational operators built-in to our programming language for efficiency reasons and the algorithm must now be written as follows:*

- *For every node* k *do*
  - *For every node* i *do*
    - *For every node* j *do*
      - *Set* $tmp = u_{i,k} + u_{k,j}$
      - *if* $u_{i,j} < tmp$*, set* $u_{i,j} = tmp, s_{i,j} = s_{i,k} \vee s_{k,j}$
      - *if* $u_{i,j} = tmp$*, set* $s_{i,j} = s_{i,j} \vee s_{i,k} \vee s_{k,j}$

The second algorithm involves a lot of extra computations and is slower.

The example shows how the handling of strict inequalities affects the computation of the canonical form and an analogous reasoning shows how other DBM operations are affected. To summarise, the reason for the slow-down is that the handling of strict inequalities introduces extra comparisons and assignments in order to compute a new DBM element correctly. Since the constraint operations are performed many times for each symbolic state generated during the verification it is important that these operations are as optimised as possible. A major increase of performance is possible if we can store something in the DBM elements that may be derived from both $u_{i,j}$ and $s_{i,j}$ and make it possible to use the algorithms for non-strict constraints and get the handling of the strict ones for free.

Define

$$v(u_{i,j}, s_{i,j}) = 2u_{i,j} + 1 - s_{i,j}$$

The value returned by v is of the same data type as $u_{i,j}$ and may be handled using built-in data types and assignment operators. The choice of v also makes it possible to use built-in comparison operators because $v(u_{i,j}, 1) < v(u_{i,j}, 0)$.

Sample	$time_{u,s}$ sec	$time_v$ sec	red %
audio	0.22	0.12	45.4
audio_bus	5.0	2.3	54.0
B&O	47.5	27.7	41.7
brp	1.6	1.1	31.3
dacapo_s	9.8	6.7	31.6
dacapo_b	20.9	14.5	30.6
engine	7.3	4.9	32.9
mplant	1.3	0.8	38.5
fischer4	0.8	0.6	25.0
fischer5	27.9	18.8	32.6
fischer6	2272	1737	23.6

Table 4.1: Comparing methods for Handling of Strict inequalities

The price we have to pay is that we get a slightly more complex addition operator. Suppose we want to calculate an upper bound for $x_i - x_j$ by computing $u'_{i,j} = u_{i,k} + u_{k,j}$. To determine $s'_{i,j}$ we compute $s_{i,k} \vee s_{k,j}$ The problem is that v is not additive, i.e.

$$v(u_{i,k} + u_{k,j}, s_{i,k} \vee s_{k,j}) \neq v(u_{i,k}, s_{i,k}) + v(u_{k,j}, sk, j)$$

It is easy to show that

$$v(u'_{i,j}, s'_{i,j}) = v(u_{i,k}, s_{i,k}) + v(u_{k,j}, s_{k,j}) + s_{i,k}s_{k,j} - 1$$

We have now achieved our goal of being able to processing constraints with both strict and non-strict relational operators almost as efficiently as constraints involving only one of them. Table 4.1 shows the difference in verification time when the new DBM algorithms are used compared to the second algorithm in the example above.

As can be observed, the speed-up in the DBM operations are tremendous and the verification time decreases about 30%. UPPAAL spends most of its time doing DBM operations so it is well worth optimising them.

4.2 Minimising Constraint Systems

In a timed system with n clocks, the DBMs representing the constraint systems are of size $n+1 \times n+1$. The $+1$ comes from the extra zero-valued clock introduced to make all constraints binary. If the size of each element is 4 bytes [1], each DBM will consume $4(n+1)^2$ bytes. Also, the complexity of operations depends on the number of clocks or constraints in the DBM.

[1] 4 is a common size of built-in integer data types.

One way to reduce the size of a DBM is to remove the diagonal elements of value zero and storing the constraint system in a $n \times (n+1)$ matrix but that would destroy the nice regularity in the DBM algorithms increasing the number of tests needed to access the matrix and hence decreasing the algorithm's efficiency. Another way to reduce the size of the DBM is to use the maximal constant for each clock and derive the exact number of bits needed for each DBM element. The matrix would then be implemented as a bit field rather than an array. While bit operations are fast one shall not underestimate the efficiency of accessing array elements at multiple byte boundaries in the memory. DBM elements are accessed many times in each constraint operation and it is worth wasting some memory keeping the accesses efficient. A third way would be to reduce the number of clocks in the automata [DY96]. We will use another approach and simplify the constraint system, removing redundant constraints, just as when simplifying an algebraic expression. The example below shows a simplification of a constraint system and clarifies the idea.

Example 2 *(Removing redundant constraints) Assume a timed system with two clocks x and y. Assume a time zone described by the constraint system*

$$0 \leq x \leq 5, 1 \leq y \leq 6, y = x + 1$$

The DBM *for the canonical form of this constraint system will contain 9 elements.*

It is obvious that we have redundancy; the constraint system

$$0 \leq x \leq 5, y = x + 1$$

contains exactly the same information and has the same solution set but is much smaller. Instead of storing 9 elements it would be enough with 4 or in our case 3 since the constraint $x \geq 0$ is implicit, i.e. a clock is always greater than or equal to zero.

The next section discusses a graph-theoretical method that automatically finds the largest possible subexpressions and remove them, thus obtaining a minimal set of constraints whose solution set is equivalent to that of the original constraint system. The method will be presented for constraints involving only non-strict inequalities but using the method described in the previous section it is possible to handle constraints with strict inequalities as well.

4.2.1 Removing Redundant Constraints

Given a constraint system with DBM U on canonical form we want to find a constraint system U' equivalent to U containing a minimal number of constraints. Further we require that the DBM of the canonical form of U' is equivalent to U. In fact we want as few constraints as possible and if more than one such system exists with the same minimal number of constraints, it is enough to find one of them.

Using our graph model of constraint systems we may formulate the problem as follows: Find the largest set of edges that can be removed, still keeping the shortest-path closure unchanged.

The following method achieves this. The presentation of the method will be quite informal. For a more formal treatment see [LLPY97]. We recall some definitions and an important fact that the method relies on.

Definition 13 *(Zero-cycles and Redundant edges)*

- *In a directed weighted graph g a cycle is a zero-cycle if the sum of the weights on the edges along the cycle is zero.*
- *In a directed weighted graph g with shortest-path closure g' an edge e between vertices i and j is redundant if its weight is greater than the weight on the corresponding edge in g' between i and j.*

□

Fact 7 *In a graph free of zero-cycles its shortest-path closure is unique and will be preserved even if the redundant edges are removed.*

It is important that the graph we want to reduce does not contain any zero-cycles. If it does we might get into problems deciding what edges to remove. An edge might be redundant but if it is removed we could be forced to keep other redundant edges to maintain the shortest-path closure. Thus it would be difficult to achieve a minimal number of edges in the graph. The problem to solve is how to get a zero-cycle free graph to reduce and how it shall correspond to our original constraint graph.

First, the nodes are grouped in equivalence classes; two nodes belong to the same equivalence class if there is a zero-cycle between them. Since we already have the shortest-path closure stored in the DBM, a zero-cycle between nodes i and j can be found by the test $u_{i,j} + u_{j,i} = 0$. In the example above, x and y belong to one equivalence class. The extra zero-clock belongs to a second equivalence class because in this example there are no zero-cycles between it and the other clock nodes.

The next step is to construct a super-graph with nodes c_i corresponding to the equivalence classes and edges $c_{i,j}$ connecting each class to all the others classes; it is not important what nodes are chosen from the classes to connect. Note that this super-graph must be zero-cycle free according to the way we partitioned the nodes into equivalence classes and constructed it. This super-graph might have some redundant edges that we can remove. An edge $c_{i,j}$ is redundant if $c_{i,j} \geq c_{i,k} + c_{k,j}$ for some node k. We may also mark edges with a weight of ∞ and edges with weight 0 from the zero-clock to any other clock as redundant because clocks are always positive.

After removal of the redundant edges, the super-graph is extended with edges connecting every clock in an equivalence class to one other clock in the same class; it is not important which clock gets connected with which so any order might be chosen. The last node is connected to the first to form a cycle. This gives us a graph with a minimal number of edges preserving the shortest-path closure of the original one. Intuitively the equivalence classes corresponds to clock constraints of the form $x_i = x_j + c$ Hence it is enough to connect a clock in a class to only one other clock in the class. The non-redundant edges between the equivalence classes are used when information about a clock in one class shall be derived from information on a clock in another class.

When implementing the reduction method above there are some things to pay attention to: What is the most efficient way to partition the nodes into equivalence classes? How shall the relation between nodes and equivalence classes be represented? The naive way of partitioning the nodes is to go through each of the nodes and for check which of the other nodes that belong to its equivalence class. This yields an $O(n^2)$ algorithm assuming the number of nodes in the original graph is n. However, this method places each node in its equivalence class more than once. We can obtain an $O(n)$ algorithm by ensuring that each node is assigned an equivalence class only once by introducing a place-holder $place(i)$ keeping track of if a node i has been placed in an equivalence class or not. We then only inspects zero-cycles to nodes not yet assigned to any class.

Alternatively we can give each equivalence class a number and for each node, store the number corresponding to its equivalence class in an array-like structure. This is a space-efficient data structure but it has some drawbacks. The algorithm requires us to be able to easily find a nodes in a given equivalence class and we must also be able to iterate through the individual nodes in a given equivalence class. Neither of these things may be done efficiently with such a structure.

Instead we will use a Boolean matrix structure. We do not know how many nodes each equivalence class will contain but it would be at most n if n is the number of nodes in the graph. The number of equivalence classes is not known either but it will be at most n. The matrix is therefore of size $n \times n$. We can now easily find a node in a given equivalence class, or iterate through all of them, by a simple row or column traversal. We introduce two matrices: $eq(i,j)$ represents the assignment between nodes and equivalence classes and $red(i,j)$ keeps track of the edges that are redundant and could be removed. The algorithm can be stated as follows.

Algorithm 15 *(Constraint system reduction)*

Input *A* DBM *U describing a constraint system on canonical form and the corresponding graph model U_g.*

Output *A sparse graph U' describing a reduced constraint system.*

- Initialise $placed(i)$ to zero.
- Initialise $red(i,j)$ and $eq(i,j)$ to zero.

- Initialise a counter cl that will hold the number of equivalence classes to zero.
- for all nodes $i \in U$ do
  - if $placed[i] = 0$ then
    - for all nodes $j \in U$ do
      - if $u_{i,j} + u_{j,i} = 0$ then $placed[j] = 1, eq[cl, j] = 1$
  - Increase the equivalence class counter cl.
- Construct the complete super-graph, $cl_g[i, j]$, connecting all equivalence classes to all the other.
- for all nodes $i \in cl_g$ do
  - for all nodes $j \in cl_g$ do
    - for all nodes $k \in cl_g$ do
      - if $(cl_g[i, k] + cl_g[k, j]) < cl_g[i, j]$ then $red[i, j] = 1$
- for all nodes $i \in U_g$ do
  - if $u_{0,i} = 0$ then $red[0, i] = 1$
- for all nodes $i \in U_g$ do
  - for all nodes $j \in U_g$ do
    - if $u_{i,j} = \infty$ then $red[i, j] = 1$
- for all equivalence classes c do
  - for all nodes $i \in c$ do
    - $u'_{i,i+1} = u_{i,i+1}$
- for all nodes $i \in U_g$ do
  - for all nodes $j \in U_g$ do
    - if $red[i, j] = 0$ then $u'_{i,j} = u_{i,j}$

The complexity of the algorithm is $O(n^3)$ where n is the number of nodes in the graph. We have now developed the necessary model to implement the reduction technique described, a sparse graph. Next, we discuss how to represent a sparse graph.

4.2.2 Representing Sparse Graphs

The two most common methods of representing graphs are matrices and adjacency lists [Sed92]. Matrices are used to represent dense graphs and adjacency lists for sparse graphs. A matrix representation of a graph is exactly what we have when using the DBM data structure; adjacency lists of some form is normally the preferred choice when representing sparse graphs corresponding to reduced constraint systems.

How many constraints do we need to remove to gain any space? A system with n clocks produce constraint systems with $(n+1)^2$ constraints if a DBM representation is used. If each upper bound occupies a bytes the DBM will use $a(n+1)^2$ bytes of memory. An adjacency list is a linked list where each list node contains data and a pointer to the next node. Since we deal with weighted graphs describing constraints, the data in our case is the constraint, bound and indices, of the two clocks involved. If we restrict ourselves to systems not containing more than 256 clocks a constraint will occupy $a+2$ bytes. This means that each list node will occupy $a+6$ bytes in the adjacency list describing the sparse graph[2].

The reduced constraint system may at most contain $n(n+1)$ constraints if we do not get any reductions at all and occupy $(a+6)n(n+1)$ bytes of memory. Assume that our reduced constraint system contains $m = \alpha n(n+1)$ constraints, i.e. we get a reduction of $1-\alpha$ %. In order to save any space the following condition must hold:

$$(a+6)\alpha n(n+1) \leq a(n+1)^2$$

which means that

$$\alpha = \frac{an}{(a+6)(n+1)} \approx \frac{4}{10}$$

if a bound consumes 4 bytes of memory, i.e. $a = 4$. This indicates that we need a reduction of at least 60% in order to save space.

However, we may achieve a lower grade of needed reduction if we utilise that the number of constraints in the reduced system is known when the graph is built, i.e. the number of edges is known and we can get rid of the linked structure of the adjacency list. This way of implementing a sparse graph is known as an array in the literature [Sed92]. If we remove the pointers in the adjacency list, each constraint can now be represented using $a+2$ bytes instead yielding

$$\alpha = \frac{an}{(a+2)(n+1)} \approx \frac{2}{3}$$

thus reducing the required reduction ratio to $\frac{1}{3}$.

Table 4.2 shows the number of constraints in reduced constraint systems per saved state for our sample series.

We still have one more question to answer: Shall the “parallel array” be implemented as three different arrays, one with bounds and two with indices, or as an array of records each containing the bound and indices? Memory management in operating systems is much more efficient if a larger block is allocated rather than three smaller blocks; it may even save space since each block require some extra “book-keeping” information and applications requesting a block of a certain size often gets a larger block with a size of the nearest power of two. However, three blocks rounded up to the nearest power of two will in our case occupy less memory than

[2] 4 is a common size for pointer data types.

Sample	Reduced	Unreduced	red %
audio	2.3	9	74.5
audio_bus	17.2	36	52.1
B&O	6.6	16	58.8
brp	11.9	25	52.4
dacapo_s	10.3	36	71.3
dacapo_b	18.9	36	47.6
engine	14.0	36	61.1
mplant	24.7	36	31.3
fischer4	10.3	25	58.6
fischer5	15.8	36	56.2
fischer6	22.2	49	54.8

Table 4.2: Constraints in reduced and unreduced constraint systems

a large block with size rounded up to the nearest power of two. Further, compilers tend to pad records with extra bytes so that each field starts at a four byte boundary giving them sizes of powers of four for more efficient accesses; in our case 12 bytes instead of six. On the other hand, a bound and its corresponding indices are always accessed in sequence, before the next constraint is accessed, i.e. each field in the record is used before the next record meaning that machines with small data caches might benefit from one array of records rather than three arrays and time performance will increase.

In Table 4.3 the two different implementations of parallel arrays are compared.

The table indicates that it is not possible to tell which of the implementations is the best. On some example, the record implementation performs better and on others the version using parallel arrays consumes less memory. It is also possible to implement the graph as two parallel arrays instead of three by grouping the coordinates together in records and place them into an array. This would give us similar results. In the next section we will look at the DBM operations again and investigate their time complexity on reduced constraint systems.

4.2.3 DBM operations

An advantage with the matrix representation of constraint systems is that most of the constraint operations are very efficient. For example, guard conjunction only involves checking and changing one element in the matrix that may be found immediately from the indices of the clocks in the guard making the operation $O(1)$, not taking the consistency check and canonical transformation into account. Similarly, the reset operation involves accessing one row and one column. The only operation heavily dependent on the number of constraints is the one checking the relation between the solution sets of two constraint systems which is an

Sample	Time			Memory		
	Array sec	Record sec	Red %	Array MB	record MB	Red %
audio	0.12	0.12	0	0.8	0.8	0
audio_bus	2.8	2.8	0	2.3	2.4	4.2
B&O	28.5	28.7	0.7	12.2	11.6	-5.2
brp	1.3	1.3	0	1.6	1.6	0
dacapo_s	7.4	7.3	12.3	3.5	3.4	-3.0
dacapo_b	16.1	15.9	-1.3	8.0	8.3	3.6
engine	6.5	6.3	3.2	0.9	0.9	0
mplant	1.1	1.7	35.3	1.5	1.6	6.3
fischer4	0.7	0.7	0	1.2	1.2	0
fischer5	21.6	22.8	5.3	8.2	8.5	3.5
fischer6	1906	2822	32.5	138	147	6.1

Table 4.3: Comparing implementations of sparse graphs

$O(n^2)$ operation if the constraint systems are in canonical form.

For a reduced constraint system, conjuncting a guard is a linear operation, in the number of constraints, to search through the reduced constraint system for the appropriate matrix element and change it if it is present. It may be possible to lower the complexity of the search using some sorting or clever hashing but what shall we do if the proper constraint is not present in the reduced system? We can add the guard in case it will affect the solution set or we can transform it back to a DBM in canonical form, use the O(1) operation and reduce the constraint system again.

The first approach violates the assumption that the size of a reduced constraint system is unchanged and we cannot guarantee that the graph representating the reduced constraint system still has the minimum number of edges after the conjunction. This will make our array implementation very inefficient. The second approach will make conjunction of guards an expensive $O(n^3)$ operation instead of a cheap one with constant complexity. Further, there is no way of checking consistency of a reduced constraint system other than transforming it back to a DBM in the canonical form. Similar problems arise for the reset operation as well. We have to conclude that we are almost forced to reconstruct the original DBM in order to perform the operations discussed above.

The relation operation is also affected by the reduction. We can only compare bounds present in the reduced constraint system to the corresponding bounds in an unreduced one and have to use this information to draw conclusions about the relationship between the solution sets of the constraint systems. There is information enough to determine if an unreduced constraint system is included in a reduced one but not the other way around. This can be explained as follows:

Assume that $u'_{i,j}$ is a DBM corresponding to a reduced constraint system U_r and that $u_{i,j}$ is a DBM representing some other constraint system. If a constraint $x_i - x_j \leq u'_{i,j}, j \geq 0$ is not present in U_r we know that either $u'_{i,j} = \infty$ or it can be written as a sum of some of the constraints in U_r. This means that if all bounds present in U_r are greater than the corresponding elements in U we may safely conclude that the solution set of $x_i - x_j \leq u_{i,j}$ is included in the solution set of $x_i - x_j \leq u'_{i,j}$ but not vice versa. If the method of removing redundant constraints is changed so that constraints of the form $x_i - x_j \leq \infty$ are kept, we will be able to fully detect the relationship as when normal DBMs are used but this would reduce the compression ratio $1 - \alpha$ remarkably.

Below is an algorithm that decides the relationship between the solution sets of two constraint systems when one of them is on reduced form and the other one on its expanded form in a DBM. The reduced constraint system is represented as a set of triples $\{i, j, u\}$ describing the constraint $x_i - x_j \leq u$.

Algorithm 16 *(Relationship)*

Input *A reduced constraint system* U_r *and a* DBM $u'_{i,j}$ *representing a constraint system* U *on canonical form.*

Output true *if* U *is included in* U_r*,* false *otherwise*

- *for all constraints* $\{i, j, u\} \in U_r$ *do*
  - *if* $u < u'_{i,j}$ *then return* false
- *return* true

There is a way to detect equality between a reduced constraint system and a DBM by reducing the DBM and compare the two sparse graphs constraint by constraint. This will work since the algorithm is deterministic and will produce equivalent sparse graphs from equivalent DBMs. While the relation algorithm will be faster when fewer constraints are checked the fact that we may not detect their relationship in full might cause extra states to be explored preventing us from terminating the reachability analysis as early as is otherwise possible.

Since some operations perform much better on DBMs and others perform better utilising the reduced constraint systems the most efficient solution seems to be one using both. The question that arises is how to combine them. By inspecting the reachability algorithms in chapter 2 one can see that most of the constraint operations are used when computing successor states to states picked from WAIT. When searching through PAST, only the relation operation is used. One possibility would be to only use reduced constraint systems in PAST. This is the current solution implemented. Another possibility would be to store the constraint systems in reduced form in WAIT, transform them to DBMs when they are retrieved and computing successor states. This would probably be space-efficient since only the current explored state

Sample	Time			Memory		
	Reduced sec	Unreduced sec	Red %	Reduced MB	Unreduced MB	Red %
audio	0.12	0.12	0	0.8	0.8	0
audio_bus	2.8	2.3	-21.7	2.4	2.3	-4.3
B&O	28.7	26.6	-7.9	11.6	11.9	2.5
brp	1.3	1.1	-18.2	1.6	1.5	-6.7
dacapo_s	7.3	6.7	-9.0	3.4	3.8	10.5
dacapo_b	15.9	13.5	-17.8	8.0	7.9	-1.3
engine	6.3	4.7	-34.1	0.9	0.9	0
mplant	1.1	0.8	-37.5	1.5	1.4	-7.1
fischer4	0.7	0.6	-16.7	1.2	1.2	0
fischer5	21.6	18.6	-16.1	8.2	8.9	7.9
fischer6	1906	1496	-27.4	138	151	8.6

Table 4.4: Comparing performance for reduced and unreduced DBMs

is in its unreduced form while all the other are reduced. However, since reducing a constraint system and expanding it again are expensive operations we will suffer bad time performance.

Table 4.4 shows verification performance for a pure DBM implementation and an implementation storing reduced DBMs in PAST.

Unfortunately, according to the experimental results in Table 4.4, we do not gain any reductionin time or space by removing redundant constraints and representing reduced constraint systems as linked lists. The possible gain in time when checking the relationship between constraint systems is negligible compared to the very expensive operation that finds and remove redundant constraints. Further, it seems that we have to implement our own memory heap to get space reductions comparable to the reduction in the number of constraints shown in Table 4.2. The reason is that the memory manager in the under-lying operating system for efficiency reasons always allocate memory blocks whose size is a power of 2. It also has a minimum size for an allocated block and as long as the number of clocks in the system is small that size seems to exceed the actual space needed both by the DBM and the reduced graph. In our case a DBM seems to be the most efficient representation of a constraint system as long as the number of clocks in the model are small. However, there are cases when DBM's cannot be used and we are forced to use the linked structure all the time. This occurs when the number of clocks in each state varies. Such an algorithm is described in [DT98]. In such situations the technique of reducing constraints systems is expected to work very well.

4.3 Representing Location Vectors

The discrete part of a symbolic state consists of information holding the current location for each individual component and the current assignment for the data variables. The location information is represented as an array with elements holding a number indicating the state of each component in the network. The data variables are also represented as an array with elements corresponding to the current value.

These representations have two major drawbacks with respect to efficiency: It results in allocation of many small memory blocks and it prevents us from using built-in manipulation operators since data types, not intrinsic in the language, are used. Is it possible to find a more compact representation that can fit in a primitive built-in data type? If we restrict the allowed size for the automata networks we may get rid of the dynamic memory allocation and can use static allocation which is much more efficient and the restriction is not a severe one.

Restricting the data variables is hard because the users have no easy way to estimate domains for data variables and there is no static method that can be used for deriving appropriate domains. However, with the location vector things are different. We know how many states each component have and it is possible to issue a warning if there is a chance that the total number of locations does not fit into the built-in data type used for the representation.

Assume a network of n automata; each symbolic state contains an n-dimensional location vector L with components L_i. Further, assume that automaton i has m_i states. The number of states in the product automata is $M = \pi_i m_i$. A compact representation of L can be found by assigning a unique integer $C(L)$ in the range $0...M-1$ to each location vector L:

$$C(L) = \sum_i L_i c_i$$

where $c_0 = 1$ and

$$c_i = \prod_{k=0}^{i-1} m_k$$

The constants c_i is independent of L_i and can be pre-computed before the verification starts.

If we can find an efficient way of recreating a location vector from a given integer we can use this technique everywhere in the tool, not just when storing states in the PAST data structure. All c_i satisfies the property c_i is a devisor of c_{i+1}. This may be used to obtain L_i from $C(l)$:

$$L_i = (C \mathsf{mod} c_{i+1})/c_i$$

When component i performs a transition and changes its location from s to s' we do not want to recreate the location vector, change L_i and compute the compact representation again. Instead we want to re-use the compact representation of the unchanged components. Let L' be the new location vector when L_i changes from s to s' and denote its compact representation by $C(l')$. Further, let $\delta_i = s' - s$ be the change of L_i. We get

$$C'(L') = C(L) + \delta_i c_i$$

Sample	Time			Memory		
	Red Loc sec	Unred Loc sec	Red %	Red Loc MB	Unred Loc MB	Red %
audio	0.12	0.12	0	0.9	0.9	0
audio_bus	2.6	2.3	-13.0	2.0	2.3	13.1
B&O	32.1	26.5	-21.1	11.1	12	7.5
brp	1.3	1.1	-18.2	1.5	1.5	0
dacapo_s	7.3	6.4	-14.1	3.7	3.8	2.6
dacapo_b	15.3	13.7	-11.7	7.5	7.9	5.1
engine	5.4	4.8	-12.5	0.9	0.9	0
mplant	0.9	0.8	-12.5	1.4	1.4	0
fischer4	0.6	0.6	0	1.2	1.2	0
fischer5	16.3	17.6	7.4	8.4	8.9	5.6
fischer6	1260	1472	14.4	144	151	4.6

Table 4.5: Comparing performance of compact and non-compact representation of location vectors

Table 4.5 shows a comparison of space and time performance of the compact versus non-compact representation of the location vector.

It is a bit surprising to find that there is almost no decrease in time performance. It should be faster to use a built-in comparison operator and compare built-in data types than comparing arrays element by element. One explanation is that element-wise comparison of arrays may be implemented as a comparison of two memory blocks instead of a loop that iterates through both arrays and compare the elements. Even though the comparison of memory blocks is slower than comparing two single values the difference is not sufficiently large to yield any dramatic increases in performance of the total verification time. The gain in performance is mostly observable for examples with many reachable control states. The memory savings are noticeable for examples with large control structure. This is not surprising since the difference between the compact storage of the control structure whose size is independent of the size of the control structure and the original one is proportional to the size of that data structure.

Chapter 5

Symbolic State-Spaces

Most of the verification time spent by UPPAAL is in calculating successor states and searching through the WAIT and PAST data structures, introduced in chapter 2. In chapter 4 we looked at techniques to speed-up the operations needed to compute successor states and we also discussed techniques to reduce the size of a symbolic state. This chapter discusses how to manipulate large sets of symbolic states, such as WAIT and PAST.

The first section examines the use of hashing to search through these large structures in an efficient way. The next three sections deal with reducing the memory usage of these structures. We will try to decrease the number of symbolic states stored in these structures. We examine heuristics, approximations and complete methods. The chapter ends with a section on how to save time by re-using parts of the state-space when verifying multiple properties. In contrast to the methods described in the preceding sections this is a method that requires more space to improve time performance.

5.1 Implementing WAIT and PAST

The major part of UPPAAL's memory utilisation is caused by storing symbolic states. These states are either stored in the WAIT or PAST data structures. These structures may be very large and the effectiveness of a verification tool is heavily dependent of how these large sets of states are maintained. It is not possible to say which of these structures are the largest and most important; it both depends on the search strategies used, e.g. breath-first or depth-first, and the structure of the system verified.

In chapter 2 we described the graph-searching algorithm used and realised that the data structure to use when representing the state-space stored in WAIT is a stack or a queue and it is determined by the search order used. These structures work fine because queues and stacks are still efficient even if they become large. The reason is that only the start and end parts of the structures are manipulated; no random access or traversal of the structure is performed.

However, the PASTdata structure is different.

Recalling the reachability algorithms in chapter 2 we note that the most important function of the PAST data structure is to guarantee termination, i.e. the same state is not explored infinitely many times. Its presence is also important to speed-up termination, i.e. the same set of concrete states in a symbolic state is not explored more than once. This clearly involves traversal of the elements in the structure to detect and the maintenance becomes critical. Usually, large sets of data can be handled efficiently using hashing and this is true for the state-space stored in PAST as well.

We need to design a hash function and a structure of the hash table that meets our requirements. Studying the reachability algorithms again gives us the following:

- Symbolic states with different discrete parts will never be examined at the same time. This is because the decision to store a new symbolic state or not, only depends on the other symbolic states with the same discrete parts that are already stored.
- All symbolic states with the same discrete parts should be stored together. If this is not the case, we have to search through the whole hash table, i.e. state-space, when deciding if a symbolic state shall be stored and its successors explored. Failing to do so will not guarantee termination.
- Chaining techniques might be more useful for us than open addressing since we run the risk that the hash table becomes full. If this happens we want to avoid increasing hash table size dynamically at run-time and re-hash the whole state-space.
- since this is a verification tool we want to implement exact hashing, i.e. if collisions occur we must be able detect and resolve them.
- Due to the large set of data to maintain, we want to have a hash function that is easy to compute and keep the chains in each table entry as short as possible.
- Because hash tables are random-access structures they should not consume too much memory, which may cause the memory manager to behave inefficiently when different parts of the table are inspected. Therefore the hash-table size should be chosen carefully.

A common data structure for a hash table for chaining is an array of linked lists and it is our choice as well. To be able to handle collisions we must store the entire symbolic state in the list, not just the DBM or the discrete part.

The first hash function we will experiment with is based on the idea presented in the context of reducing the discrete part of a symbolic state, discussed in chapter 4. From the requirements above, it is clear that the input to the hash function must contain at least the location vector of the state. If the table size is large, we might obtain a good hash function by mapping location vectors to unique integers. Of course, not all control locations in the product automata are reachable and this hash function might not be optimal. Also, each control vector is not reached with the same number of variable assignments causing the length of the chains to be unevenly distributed. Obviously we cannot choose tables as large as the number of control locations in the product automata and collisions will occur. However, these collisions may result in symbolic states occupying the empty slots in table, caused by the combinations of location vectors not reachable in the product automata.

We assume a hash-table size H. Using the notion of compact representation for control structure developed in section 4.3, we have the first hash function:

$$h_1(l) = (\sum_i L_i c_i) \mathsf{mod}\ H$$

where m_k is the number of states in automaton k, $c_0 = 1$, $c_i = \prod_{k=0}^{i-1} m_k$ and $M = \prod_i m_i$ is the total number of locations in the product automata. The hash function maps a location vector to an integer. That integer is divided by the table size and the remainder of the division is used as a hash value. This will map each location vector to a unique position if the table is large enough and collisions will only occur if there are more different location vectors than entries in the hash table. A drawback is that even unreachable location vectors, will be assigned a table position and that position will only be occupied if collisions occur.

To examine the performance of the hash function we will study the following measures:

- The numbers of used and empty slots in the hash table
- The longest and shortest chains in the hash table

Table 5.1 shows these measures for a table size of $H = 17609$ entries used in UPPAAL today when our example series is verified. As it is shown, the length of the chains has very high deviation and only a small part of the hash table is utilised. We may either reduce the size of the hash table or preferably, find a better hash function. It makes sense to extend our hash function so that more input data from the symbolic state may be taken into account.

The hash function discussed earlier does not fully comply to our requirement of hashing symbolic states with different discrete parts to different table positions; we must use the integer vector, describing the data variable assignment as well, as input data to the hash function.

Assume that V is a vector with components v_i containing values of the data variables. Let z_i be the size of the domain of variable i. Assuming n data variables in the system and defining $d_0 = 1$ and $d_i = \prod_{k=0}^{i-1} z_k$ we construct

$$h'(V) = (\sum_i v_i d_i) \mathsf{mod}\ H$$

where H is the size of the hash table. By combining h_1 and h' we get our second hash function

$$h_2(l, V) = h_1 + Mh' = ((\sum_i L_i c_i) + M \sum_i v_i d_i) \mathsf{mod}\ H$$

Sample	# Empty Entries	Max chain	Min chain
audio	17563	4	1
audio_bus	17468	211	2
B&O	17484	4711	1
brp	17598	1120	6
dacapo_s	17214	2460	1
dacapo_b	17214	7182	1
engine	17320	9	1
mplant	17502	90	1
fischer4	17496	192	6
fischer5	17286	1920	18
fischer6	16688	23040	60

Table 5.1: Hash function performance

where $c_0 = d_0 = 1$, $c_i = \prod_{k=0}^{i-1} m_k$ and $M = \prod_i m_i$ as before. This hash function also maps a given combination of location vector and data variable assignment to a unique integer in the same manner as described for h_1. The drawback is the same as earlier but since the number of different discrete parts are larger we might expect a better table utilisation.

Table 5.2 shows the same information as Table 5.1 for the combined hash function $h_2(l, V)$ and a table size of $H = 17609$ entries. Table 5.3 shows a comparison of total verification time between two versions of UPPAAL, one using the hash function h_1 and the other using hash function h_2. We can see that the extra time needed for the more complex calculations of h_2 is compensated by the better scattering of symbolic states and the access time of states in the hash table decreases.

We have managed to reduce the maximum and minimum lengths of the chains but unfortunately we still have very poor hash table utilisation. This means that either there is a lot of collisions occurring causing unnecessary long chains or the function is sparse due to the fact that only a small part of the possible combinations of location vectors and data variable assignments correspond to reachable symbolic states.

To find out which of the phenomena causing the low utilisation and the long chains we will change hashing strategy from chaining to open addressing and use linear probing. When saving a symbolic state in PAST we will try to find a new position in the table if other states with a different discrete part already occupies that table entry. We continue searching through the table until we find an empty entry or a chain with the same discrete part. If the hash table becomes full we either stops the verification, throw some symbolic state out of the passed list according to some strategy or traverse the hash table and rehash it using a larger hash table. The first two alternatives are a good approach if we want a limit on the size of the PAST structure. The approach that throws away certain states is interesting but is not studied any further here. The third alternative is expensive due to the large size of PAST. In

Sample	# Empty entries	Max chain	Min chain
audio	17520	2	1
audio_bus	15888	19	1
B&O	6421	61	1
brp	17149	21	1
dacapo_s	13250	44	1
dacapo_b	13250	50	1
engine	17228	3	1
mplant	17502	90	1
fischer4	17389	48	2
fischer5	16882	384	6
fischer6	15320	3840	18

Table 5.2: Hash function performance

Sample	Time (h_1) sec	Time (h_2) sec	red %
audio	0.12	0.12	0
audio_bus	2.8	2.3	17.9
B&O	121	27.5	77.3
brp	2.6	1.1	57.7
dacapo_s	13.2	6.5	50.8
dacapo_b	80.0	14.5	81.9
engine	4.6	4.8	-4.4
mplant	0.8	0.8	0
fischer4	0.6	0.6	0
fischer5	24.0	18.5	22.9
fischer6	2800	1580	43.6

Table 5.3: Comparing verification time using two different Hash functions

Sample	# Empty entries	Max chain	Min chain
audio	17520	2	1
audio_bus	15796	14	1
B&O	208	44	1
brp	17144	21	1
dacapo_s	12499	44	1
dacapo_b	12499	44	1
engine	17224	3	1
mplant	17502	90	1
fischer4	17389	48	2
fischer5	16882	384	6
fischer6	15320	3840	18

Table 5.4: Hash function performance

our experiments the first approach was used.

Table 5.4 shows the performance of the same hash function as in Table 5.2 but with use of the probing strategy just explained. Table 5.5 compares time performance for h_2 when chaining and open addressing is used respectively. The performance is nearly identical to that in Table 5.2 indicating that very few collisions occur at all.

Unfortunately we have a sparse hash function and the long chains occur because there are many DBMs associated with the same reachable combination of location vector and data variable assignment. The requirements that the hashing shall be exact and that all states with equal discrete part must be examined together prevent us from placing a maximum limit for the length of a chain and gain a better table utilisation. For other uses of hashing in verification tools, especially non-complete techniques that have a very low probability of collisions see [WL93, SD96].

The following section discusses other ways of decreasing the access time by decreasing the size of the state-space.

5.2 Reducing the Size of the PAST Structure – a Heuristic Approach

The use of the passed data structure serves two purposes: it guarantees termination and very often speeds up termination. The more states we save the higher the possibility of finding a state in PAST that let us terminate a search-branch of the state-space and terminating the verification faster. However, the more states we save the more time is required in searching

Sample	Time with probing sec	Time without probing sec	red %
audio	0.12	0.12	0
audio_bus	2.4	2.3	-4.4
B&O	29	27.1	-9.2
brp	1.1	1.1	0
dacapo_s	6.6	6.6	0
dacapo_b	13.8	14.3	3.5
engine	4.8	4.8	0
mplant	0.8	0.8	0
fischer4	0.6	0.6	0
fischer5	18.0	18.5	2.7
fischer6	1586	1566	-1.3

Table 5.5: Hash function performance

through and updating PAST. It is therefore worth to examine what happens if only parts of the state-space is saved. We may then generate more states than necessary to decide a reachability property, even the same state may be generated more than once, but on the other hand we keep PAST small and easier to maintain efficiently.

In this section we will study a method that simply discards symbolic states, i.e. omit saving them, according to a given pattern. The method is heuristic in the sense that it does not guarantee that the verification will terminate, neither are there any guarantees that the performance always will be better than if the strategy were not used.

We have tried two different patterns. The first one is simply avoid saving every nth state that should be saved according to the criteria in the standard reachability algorithm. The second pattern may be described as the opposite of the first, namely only save every nth state that should be saved and discard the rest. The strategies are implemented by extending the data structure used for PAST with a modulo-n counter and only discarding or saving states when the counter reaches zero. Tables 5.6 to 5.9 show the performance, when verifying our usual sample series using our two strategies. We study the number of generated states and the number of states saved in PAST for different values of n.

As can be observed, the size of PAST decreases. Even though the state-space overhead, i.e. the extra states generated due to our strategy, increases, PAST is still kept small. It seems like a quite large part of the state-space may be removed without loss of performance. Therefore it makes sense to ask if it is possible to derive a criteria for when a state needs to be saved or not, reducing the state-space overhead and still guarantee termination. The next section looks at a method that examines the control structure of the components in the automata network derives an on-the-fly criteria for what states need to be saved.

Sample	# Generated			# Saved		
	$n = \infty$	$n = 2$	$n = 3$	$n = \infty$	$n = 2$	$n = 3$
audio	150	184	176	90	55	69
audio_bus	13638	17208	15482	4913	3190	3800
B&O	154531	156578	155643	38351	19457	25788
brp	8293	9026	8558	3419	1879	2374
dacapo_s	45005	47107	46309	9704	5340	6856
dacapo_b	94530	100047	97472	22737	12428	15911
engine	1065	1260	1169	489	294	361
mplant	4380	4580	4468	2568	1362	1774
fischer4	5049	7373	6177	2273	1540	1756
fischer5	78466	116623	97706	30081	20551	23392
fischer6	1421041	2131333	1778864	474505	323752	368890

Table 5.6: Discarding every nth state

Sample	# Generated			# Saved		
	$n = 4$	$n = 5$	$n = 10$	$n = 4$	$n = 5$	$n = 10$
audio	156	154	152	71	75	83
audio_bus	14829	14536	13966	4080	4239	4566
B&O	155093	154855	154712	28903	30771	34569
brp	8471	8379	8345	2638	2784	3111
dacapo_s	45842	45680	45308	7578	8017	8869
dacapo_b	96469	96156	95270	17638	18701	20736
engine	1105	1096	1070	390	412	447
mplant	4460	4442	4412	1971	2100	2337
fischer4	5837	5709	5317	1883	1973	2115
fischer5	90926	88114	82826	24845	25869	27929
fischer6	1657224	1599110	1498141	392324	407879	439651

Table 5.7: Discarding every nth state

Sample	# Generated			# Saved		
	$n = 1$	$n = 2$	$n = 3$	$n = 1$	$n = 2$	$n = 3$
audio	150	171	193	90	52	38
audio_bus	13638	17370	23042	4913	3212	2840
B&O	154531	156598	159851	38351	19458	13224
brp	8293	8989	10139	3419	1871	1400
dacapo_s	45005	47107	48378	9704	5334	3726
dacapo_b	94530	99838	103476	22737	12423	8743
engine	1065	1229	1431	489	290	221
mplant	4380	4597	5068	2568	1368	1002
fischer4	5049	7487	9554	2273	1564	1320
fischer5	78466	116115	151957	30081	20437	17709
fischer6	1421041	2130833	2823752	474505	323606	283854

Table 5.8: Saving every nth state

Sample	# Generated			# Saved		
	$n = 4$	$n = 5$	$n = 10$	$n = 4$	$n = 5$	$n = 10$
audio	239	272	518	36	31	28
audio_bus	27084	32074	56464	2522	2393	2111
B&O	161767	163478	175556	10049	8126	4358
brp	11162	11764	17175	1147	971	706
dacapo_s	49147	49477	52994	2902	2363	1350
dacapo_b	104985	107205	111165	6761	5561	2998
engine	1597	1739	3092	190	163	146
mplant	5218	5881	8539	779	705	516
fischer4	11621	14263	25219	1204	1191	1060
fischer5	189665	222812	394727	16549	15622	13929

Table 5.9: Saving every nth state

5.3 Reducing the Size of PAST – A Non-heuristic Approach

There are essentially two reasons for symbolic states to be visited more than once. Either, the symbolic state is included in a loop[1], or it may be reached more than once due to interleaving between components. The first category of states are the ones that need to be saved in order to guarantee that the verification terminates. The second category of states are the ones that may cause speed-ups in termination if they are saved.

The method we have used relies on static analysis of each individual component. It cannot detect symbolic states that are visited more than once due to interleaving but it will detect whether a symbolic state may be a member of a loop or not. For a more formal description of the method read [LLPY97],[Pet99]. We recall some of the definitions and facts the method is based on and sketch proof ideas.

Definition 14 *(Static and dynamic loops)*

- *A dynamic loop is a sequence of symbolic states* $\langle L_1, V_1, U_1\rangle, ..., \langle L_n, V_n, U_n\rangle$ *such that* $\langle L_1, V_1, U_1\rangle \rightsquigarrow \langle L_2, V_2, U_2\rangle \rightsquigarrow ... \rightsquigarrow \langle L_n, V_n, U_n\rangle \rightsquigarrow \langle L_1, V_1, U_1'\rangle$ *where* $U_1' \subseteq U_1$.
- *For an automaton* A *we construct a graph* A_G. *Each location in* A *corresponds to a node in* A_G. *There is an edge between two nodes in* A_G *if there is a transition between the corresponding locations in* A. *Each cycle in* A_G *corresponds to a static loop in* A.

□

The following observation, stated as a fact, is the main reason for why the method works.

Fact 8 *(Relationships between static and dynamic loops) Every dynamic loop contains at least one static loop.*

The fact can be shown with a proof by contradiction.

Definition 15 *(Entry state of a loop) An entry state of a loop is a symbolic state that is reachable from at least one state outside the loop and at least one state inside. The initial state is always considered as reachable from outside a loop. The network is in an entry state if at least one of its components are.* □

It is enough to save only one state for each reachable dynamic loop in order to guarantee termination and hence it is sufficient, but not necessary, to store entry states of static loops.

[1]A loop is a sequence of symbolic states that is repeated during a state-space exploration

Sample	# Generated			# Saved		
	Entry	Standard	Red %	Entry	Standard	Red %
audio	157	150	-4.7	85	90	5.6
audio_bus	15030	13638	-10.2	2981	4913	39.3
B&O	154611	154531	-0.05	30136	38351	21.4
brp	8462	8293	-2.0	2476	3419	27.6
dacapo_s	45723	45005	-1.6	8511	9704	12.3
dacapo_b	95682	94530	-1.2	20957	22737	7.8
engine	2999	1065	-181.6	67	489	86.3
mplant	4391	4380	-0.3	1412	2568	45.0
fischer4	5049	5049	0	885	2273	61.1
fischer5	78466	78466	0	11571	30081	61.5
fischer6	1421041	1421041	0	181723	474505	61.7

Table 5.10: Generated and saved state-space when saving only loop entry states

The method finds entry states of static loops in the control structure of each component and mark them. The method is implemented by building a graph for each component as described in definition 14. This graph is then searched in depth-first order and all entry states of static loops are marked. An array in each component holds the information.

Even if it is enough to save all entry states in order to guarantee termination we can in fact do better. consider a network that contains a component consisting of only one state with a self-loop, e.g. a test automaton, and some other components as well. In such a system, every symbolic state is an entry state. Clearly the component with the self-loop shall not be considered when selecting which entry states to store in PAST. We can achieve this by only checking the component or components that just performed the transition and see if any of them reaches an entry state, i.e. entered a static loop. Thus, A state that shall be saved according to the standard reachability algorithm is only saved if it is a possible entry state of a dynamic loop and the state was just entered from outside that loop. The method will not guarantee to find the minimal number of states that need to be saved. In fact it is an over-approximation in the number of states since not all static loops in the control structure give rise to reachable dynamic loops in the compound network but the approximation is safe.

Tables 5.10 and 5.11 compare the performance when verifying our examples using this strategy to verification using the standard reachability algorithm. Table 5.10 shows the number of generated states and the number of states saved in PAST. Table5.11 shows the actual performance difference in time and memory consumption.

As illustrated in the table, we get reductions in both space and time and the state-space overhead is lower than for the strategies tried in the previous section. The method can be further improved by not marking a state as an entry state if it is an entry state of a loop totally included in an other loop of the same component. In the next section we will look at another

Sample	Time			Memory		
	Entry sec	Standard sec	Red %	Entry MB	Standard MB	Red %
audio	0.12	0.12	0	0.8	0.8	0
audio_bus	2.4	2.3	-4.4	1.7	2.3	26.1
B&O	26.4	26.6	0.8	9.6	11.9	19.3
brp	1.1	1.1	0	1.3	1.5	13.3
dacapo_s	6.5	6.7	3.0	3.5	3.8	7.9
dacapo_b	13.8	13.5	-2.2	7.3	7.9	7.6
engine	12.6	4.7	-168.1	0.8	0.9	11.1
mplant	0.7	0.8	12.5	1.1	1.4	21.4
fischer4	0.5	0.6	16.7	0.9	1.2	25.0
fischer5	13.8	18.6	25.8	4.2	8.9	52.8
fischer6	963	1496	35.6	64.3	151	57.4

Table 5.11: Memory and time performance when saving only loop entry states

approach to reducing the size of the PAST structure by saving an approximation of the DBMs corresponding to a given combination of location vector and data variable assignment. The method can be combined with any of the reduction methods discussed in this chapter.

5.4 Convex Hull Approximation

For small examples where we can expect very few collisions, the length of the chains in the hash table is a measure of the number of DBMs occurring for a discrete part of a symbolic state. For systems with a large number of clocks a DBM consumes quite a lot of memory compared to the other parts of the state. Some memory can be saved if we could store only one DBM for each different discrete part of the symbolic state. In this section we will study a convex-hull approximation of DBMs corresponding to a given location vector and data variable assignment.

There are many different convex hulls one can use and for efficiency reasons we require that we are able to represent the convex hull as a DBM. We have studied the following convex hull:

Definition 16 *(Convex hull of* DBM*s) Assume* m DBM*s* $U_1, \ldots, U_m$*. The* DBM U_c *is the convex hull of* $U_k, k = 1 \ldots m$ *if* $\forall k : U_k \subseteq U_C$ *and*

$$\{U' \mid U' \subset U_C : \exists U_k \not\subset U'\}$$

□

Intuitively we compute a DBM containing all solutions present in any of the DBMs U_1 to U_m. The following algorithm shows how to compute it.

Algorithm 17 *(Computing convex hull of* DBM*s)*

Input *a set of* DBM*s in canonical form* U_1 *to* U_m

Output *a* DBM U_C *on cannonical form representing the convex hull of* U_1 *to* U_m

- *for all pairs* i *and* j *do*
  - $U_c(i,j) = max_{k=1}^{m} u_k(i,j)$

Fact 9 *(Convex hull) The algorithm above gives a* DBM *corresponding to definition 16 of the convex hull.*

Proof To prove that U_c contains all solutions present in $U_k, k = 1...m$ we utilise that $u_k(i,j) \leq max_{k=1}^{m} u_k(i,j)$. Hence the constraint system $x_i - x_j \leq u_c(i,j)$ where $u_c(i,j) = max_{k=1}^{m} u_k(i,j)$ describes all solutions present in any of the constraint systems U_k. Also note that we may lose solutions if a bound $u_c(i,j)$ is lowered because there $\exists l : u_l(i,j) = max_{k=1}^{m} u_k(i,j)$ for all i and j. □ □

When implementing the algorithm the computation of the hull is done incrementally, i.e. the convex hull of all DBMs encountered so far for a given discrete part of a symbolic state is updated as soon as a new DBM corresponding to that part is encountered. Thus no actual traversal of a whole chain occurs. To obtain a reachability algorithm using convex-hull approximations we can choose either algorithm 1 or 2 and replace steps 3 and 4 with the steps below.

3' If $L_i = L_j \wedge V_i = V_j \wedge U_i \subseteq U_j$, for some $\langle L_j, V_j, U_j \rangle \in \mathsf{PAST}$ drop $\langle L_i, V_i, U_i \rangle$ and go to step 1. Otherwise find $\langle L_i, V_i, U_j \rangle$ in PAST and let $U_i' = chull(U_i, U_j)$. Save $\langle L_i, V_i, U_i' \rangle$ in PAST, $\mathsf{PAST} = \mathsf{PAST} \cup \langle L_i, V_i, U_i' \rangle$ and discard $\langle L_i, V_i, U_j \rangle$.

4' Find all L_k that are reachable from L_i in one step regardless of guards and resets, taking only actions into account. For all such transitions do

   (a) Let G be the guard and R the resets on the performed transition. i_k is the location invariant of L_k.

   (b) If V_i satisfies the guard, let $U_k = sp(reset(U_i' \cap G, R)) \cap i_k$ and update V_i.

   (c) If $U_k \neq \emptyset$, store $\langle L_k, V_k, U_k \rangle$ in WAIT $\mathsf{WAIT} = \mathsf{WAIT} \cup \langle L_k, V_k, U_k \rangle$.

Sample	Time			Memory		
	Chull sec	Standard sec	Red %	Chull MB	Standard MB	Red %
audio	0.12	0.12	0	0.8	0.8	0
audio_bus	1.6	2.3	30.4	1.4	2.3	39.1
B&O	25.4	27.4	7.3	6.6	11.9	44.5
brp	0.3	1.1	72.7	0.9	1.5	40.0
dacapo_s	311	6.8	-4473.5	22.7	3.8	-497.7
dacapo_b	867	14.0	-6092.9	61.4	7.9	-677.2
engine	4.2	4.8	12.5	0.9	0.9	0
mplant	0.05	0.8	94.5	0.9	1.4	35.7
fischer4	0.1	0.6	83.3	0.9	1.2	25.0
fischer5	0.4	18.3	97.8	1.0	8.9	88.8
fischer6	2.0	1496	99.9	1.7	151	98.9
fischer7	8.6	-	-	4.6	-	-
fischer8	36.8	-	-	15.3	-	-
fischer9	159	-	-	56	-	-
fischer10	797	-	-	203.8	-	-

Table 5.12: Performance of convex-hull approximation

Note that the important step of changing U_i to the convex hull of itself and the other DBMs in states with the same discrete part. By storing the convex hull in PAST we claim that we have visited all clock assignments it contains and that we will generate all the successor states from them. This means that we have to change the current constraint system from U' to U_C before continuing exploration of the state-space. This might lead to the exploration of a much larger reachable state-space than what is present in the original model. If we stored U_C but continued generating successors only to assignments in U', the termination test might cause that parts of the actual state-space would not be generated.

Another thing worth emphasising is the fact that this approximation explores a larger state-space than that of the original model. This means that we cannot trust a verification of a $\exists \diamond \varphi$ property if the answer is yes or a verification of an $\forall \Box \varphi$ property if the answer is no. In the implementation we also use the approximation for the WAIT data structure as well, not only for PAST. Table 5.12 shows the memory and time reduction when our convex hull approximation is used during verification.

There is an enormous decrease of both time and space utilisation in most cases. This can be explained by the fact that there is a large number of DBMs for each reachable combination of control location and data variable values. This is consistent with the observation in the section on hashing where we came to the same conclusion. However, in one of the examples we see the drawback of our over-approximating technique. Since we assume that a too large part of the state-space is reachable we run the risk of exploring parts that the actual model cannot reach. This causes the number of generated states to grow and even though each saved symbolic state consumes less memory we are forced to save many more because of the extra state-space generated. For a study of the effect of using other convex hull approximations see [WT94, Bal96, DT98].

5.5 Re-Use of the Generated State-Space

When verifying reachability properties the state-space generated so far is saved in the PAST data structure, it is used during the rest of the verification and discarded before verification of a new property begins. When a new property is verified, in many cases the same parts of the state-space is generated again. Ideally it should be enough to generate the state-space only once. To avoid this waste of resources we can keep PAST and WAIT after verification of a property and before verification of a new property, check if PAST already contains any states satisfying that property. If there are no such states we continue the state-space exploration as usual. Below is a reachability algorithm modified to take advantage of the described method. Essentially step 3 to 7 can be replaced by any of algorithms 1 or 2.

Algorithm 18 *(Model-checking multiple properties)*
Initial conditions: $\mathsf{WAIT} = \langle L_0, V_0, U_0 \rangle, \mathsf{PAST} = \emptyset$.

1. *Pick the first property* φ *from the group of properties to be verified.*
2. *If* $\exists \langle L, V, U \rangle \in \mathsf{PAST} \, : \, \varphi(L, V, U)$, *return* yes.
3. *Pick a state* $\langle L_i, V_i, U_i \rangle$ *from* WAIT.
4. *If* $\varphi(L_i, V_i, U_i)$ *return* yes.
5. *If* $L_i = L_j \wedge V_i = V_j \wedge U_i \subseteq U_j$, *for some* $\langle L_j, V_j, U_j \rangle \in \mathsf{PAST}$ *drop* $\langle L_i, V_i, U_i \rangle$ *and go to step 3. Otherwise save* $\langle L_i, V_i, U_i \rangle$ *in* PAST, $\mathsf{PAST} = \mathsf{PAST} \cup \langle L_i, V_i, U_i \rangle$. *If* $U_j \subset U_i$ *we can replace the state* $\langle L_j, V_j, U_j \rangle$ *with* $\langle L_i, V_i, U_i \rangle$ *to speed-up termination.*
6. *Find all* L_k *that are reachable from* L_i *in one step regardless of guards and resets, taking only actions into account. For all such transitions do*
   (a) *Let* G *be the guard and* R *the resets on the performed transition.* i_k *is the location invariant of* L_k.
   (b) *If* V_i *satisfies the guard, let* $U_k = sp(reset(U_i \cap G, R)) \cap i_k$ *and update* V_i.
   (c) *If* $U_k \neq \emptyset$, *store* $\langle L_k, V_k, U_k \rangle$ *in* WAIT $\mathsf{WAIT} = \mathsf{WAIT} \cup \langle L_k, V_k, U_k \rangle$.
7. *If* $\mathsf{WAIT} \neq \emptyset$ *go to step 3.*
8. *Return* no.
9. *If more properties exists, go to step 1.*

Sample	Time			Memory		
	Re-Use sec	Standard sec	Red %	Re-Use MB	Standard MB	Red %
audio	0.05	0.12	58.3	0.8	0.8	0
audio_bus	2.1	2.3	8.7	3.2	2.3	-39.2
B&O	28.5	27.2	-4.8	42.2	11.9	-254.6
brp	1.1	1.1	0	2.3	1.5	-53.4
dacapo_s	6.4	6.5	1.5	9.8	3.8	-150.0
dacapo_b	13.6	14.0	2.9	18.6	7.9	-135.4
engine	0.6	4.8	87.5	1.0	0.9	-11.1
mplant	0.8	0.8	0	1.5	1.4	-7.2
fischer4	0.6	0.6	0	1.2	1.2	0
fischer5	18.2	18.2	0	8.9	8.9	0
fischer6	1496	1496	0	151	151	0

Table 5.13: Increase in performance when avoiding regeneration of state-space

However, we need to think carefully about the issue of maximal constants. The generated state-space is normalised with respect to certain clock constants and we may only re-use it if the maximal constants of the model and the new property does not exceed the old ones. We can solve the problem if we know the properties a priori by computing the maximal constants for all properties before verifying them. If properties are not known we can try to make a safe estimation of the maximal constants before verifying the properties. Then we only need to regenerate the state-space if our estimation fails. Table 5.13 shows the speed-up when verifying multiple properties of our sample models. We assume that all properties are known before verification.

Observe that the speed-up is high when multiple properties are verified. The standard reachability algorithm in UPPAAL does not save committed locations and the reason for the higher memory consumption when the state-space is re-used is that these states are now saved in PAST. Also, the time spent by searching through PAST could increase because more states are saved but the time can decrease as well because of faster termination. It is worth noting that there is no actual slow-down when only one property is verified or because verification of some properties are performed with unnecessary high maximal constants. If there are large differences in the maximal constants of the properties we can group them in a way that properties with merely the same maximal constants are placed in the same group. The state-space is then re-used in each group but regenerated for each group. We do not necessarily get the shortest trace out of the verifier if we pick the first state found in PAST satisfying the property. As mentioned in the section about diagnostic traces in chapter 2 a breath-first search always finds states reachable with the smallest number of transitions performed but if such traces shall be reported here we have to go through all of the PAST data structure and count the length of the trace for each state satisfying the given property.

Chapter 6

Memory Management

In many cases, the state-space of a model consumes a huge amount of memory, which often results in swapping. To reduce swapping, the part of state-space that is not needed for further analysis should be thrown away, which in turn generates a lot of operating system operations for memory allocation and deallocation. The deallocation process involves a traversal of the state-space which itself is very inefficient if memory accesses are not performed in an adequate order. The time spent by the operating system managing memory becomes significant.

Memory deallocation is a special case of the more general problem of traversing a large amount of memory blocks, such as a state-space. This chapter describes a technique that lets UPPAAL control how the operating system accesses memory without implementing an own memory manager. For a large example, consuming 335MB of memory, system-time overhead for memory deallocation was cut down from seven days to 1 hour, on a machine with 256MB of physical memory. The method collects information during the verification process and uses it to estimate a traversal order with better locality. It introduces very little overhead in space and time. For the example described, the introduced overhead in verification time was about 1 hour. For other work in the context of memory management see [Boe93, Wil92, SD98].

6.1 Memory Usage: the Problem

The memory architecture of today's computer systems is hierarchical. There are instruction an data caches with very low access time, and a virtual memory that provides the application programmer with a transparent interface to physical memory with even slower access time and magnetic storage, i.e. discs, which are very slow in comparison to caches and physical RAM memory. Processors become faster for every year and cache design advances as well. However, the media used for physical memory and discs have not gone through this evolution and hence the gap between the processor performance and the performance of the virtual memory system increases. Operating system designers try to come up with good memory management strategies for the virtual memory by adopting a paging algorithm that performs well with existing applications in most cases.

The amount of memory used during verification oscillates. It increases when the state-space exploration starts. It decreases when parts of the state-space are thrown away. It decreases even more when the WAIT and/or the PAST data structures are cleared and increases again when a new state-space exploration begins. This behaviour makes the time spent by the operating system to perform memory deallocation very important. The problem we want to solve is how to control memory accesses without writing our own memory manager. Since the memory consumption is large, swapping cannot be avoided.

When swapping is involved, it is very important how the state-space is traversed, i.e. in what order symbolic states are accessed. It is necessary to localise memory accesses.

In chapter 5 we proposed a hash table as a suitable data structure for WAIT and PAST. A common way to traverse the states in these structures is to go through the table, in consecutive hash value order, and access them one by one. This is by far not the most efficient strategy even if it is convenient to implement. The example in Table 6.1 and Table 6.2 illustrates the operations involved when traversing a state-space, performing an operation *op* on each state. The state-space is traversed in hash value order and reverse allocation order respectively.

Example 3 *In this example We assume two memory pages, each containing two states. Initially one page is in main memory and one is in a part of the virtual memory currently on disc. Tables 6.1 and 6.2 show the page layout in main memory and on disc together with the operations an operating system may perform when the application requests to access the states.*

In table 6.1 the allocation order is s1, s3, s2, s4 and the hash table order is s1, s2, s3, s4. SWAP *is a very expensive operation and the access order in Table 6.1 requires four such operations in order to traverse all states. In Table 6.2 the allocation order is the same as in Table 6.1 but the traversal order is different; s4, s2, s3, s1 i.e. reverse allocation order. By using this access strategy the number of* SWAP *operations can be reduced to one and op can be performed immediately after the access request in most cases.*

The next section describes an experiment with memory deallocations. It is performed to test if the scenario shown in the example above has any impact.

6.2 Comparing Deallocation Orders: an Experiment

Before implementing anything in UPPAAL an experiment was made to test if the deallocation order really had a big influence on execution time. 32MB of memory were allocated in different ways. The blocks simulate states. They were placed randomly in a table, simulating the PAST data structure and then deallocated according to three different strategies: The first is hash table, or hash value, order. The second is deallocation in the same order as the memory is

Memory	Disc	Operation
{s2,s4}	{s1,s3}	accessReq(s1)
		SWAP
{s1,s3}	{s2,s4}	op(s1)
{-,s3}	{s2,s4}	
{-,s3}	{s2,s4}	accessReq(s2)
		SWAP
{s2,s4}	{-,s3}	op(s2)
{-,s4}	{-,s3}	
{-,s4}	{-,s3}	accessReq(s3)
		SWAP
{-,s3}	{-,s4}	op(s3)
{-,-}	{-,s4}	
{-,-}	{-,s4}	accessReq(s4)
		SWAP
{-,s4}	{-,-}	access(s4)
{-,-}	{-,-}	

Table 6.1: State-Space Traversal Example

allocated. The third order is deallocation in reverse allocation order. Table 6.3 shows the deallocation time (in seconds) for the three strategies described above. The number and size of the memory blocks were varied. The experiment was performed on a machine with a 75 MHz Pentium processor, 8 MB of physical RAM, running a Linux 2.0.x kernel.

The experimental results, shown in Table 6.3, clearly indicates that memory deallocation time really matters when swapping is involved. Both access orders, correlated with the order of allocation are superior to the hash table order and reverse deallocation seems to have the best performance. The reason why reverse allocation order means less swapping than allocation order is because we start deallocating states already in the physical memory and hence get rid of some swapping.

6.3 Heuristic Solution: the Implementation

The last section clearly indicates that there might be a severe increase in performance by implementing a different access order. Is it possible to implement a more suitable memory management strategy that suits the requirements of UPPAAL better without writing our own memory manager? We need to keep track of the allocation order and it must be done using an efficient data structure that is easy to update. The structure must not consume a large amount of memory itself.

To deallocate states in reverse allocation order every symbolic state is extended with pointers

Memory	Disc	Operation
{s2,s4} {s2,s4} {s2,-}	{s1,s3} {s1,s3} {s1,s3}	accessReq(s4) op(s4)
{s2,-} {s2,-} {-,-}	{s1,s3} {s1,s3} {s1,s3}	accessReq(s2) op(s2)
{-,-} {s1,s3} {s1,-}	{s1,s3} {-,-} {-,-}	accessReq(s3) SWAP op(s3)
{s1,-} {s1,-} {-,-}	{-,-} {-,-} {-,-}	accessReq(s1) op(s1)

Table 6.2: State-space Traversal Example

Blocks	Size	Table	Alloc	Rev Alloc
32768	1024	192.6	116.6	23.5
65536	512	467.9	117.4	21.8
131072	256	1191.9	106.7	26.8
262144	128	3035.2	95.6	38.0
524288	64	8142.6	94.8	54.2
1048576	32	28367.6	121.1	88.0

Table 6.3: Experimental Results

to the previous and next allocated states. We get a doubly linked list. This list is then traversed backwards when the state-space is deallocated. This means that we will be able to deallocate all states on a given memory page before it is swapped out. In fact, the solution is heuristic because it may not estimate the exact allocation order. This is because some operations performed by the reachability algorithm changes parts of a symbolic state. This destroys the assumption that states allocated consecutively will have all its data collected on the same page.

6.4 Performance

How well will our solution perform in UPPAAL? To measure performance Eight examples were verified on a machine with 8MB of physical memory and a Pentium 75MHz processor. One of the example used does not involve swapping at all and we will then be able to estimate if their will be any overhead associated with the collection of information needed to keep track of the memory allocation order. The operating system used was Linux with a 2.0.x kernel. Table 6.4 shows memory usage together with verification time (t_1) and termination time (t_2), in seconds, for the different deallocation orders described earlier. The time spent by UPPAAL deallocating the main part of the state-space can be estimated a $t_2 - t_1$.

table		Alloc		Rev Alloc		Mem
t_1	t_2	t_1	t_2	t_1	t_2	
1400	33378	1486	2613	1497	2564	12.9
804	14902	861	1553	853	1476	10.5
135	2130	138	428	143	388	8.7
126	1244	132	339	133	310	8.2
70	72	64	65	66	67	5.4
1830	1831	1937	1938	1967	1968	7.9
4654	42017	5031	13126	5046	7045	15.9
139	2417	155	593	155	538	10.0

Table 6.4: Performance Measurements

We can observe that significant time savings were achieved when a deallocation order correlated with the allocation order was used. The more swapping, the greater the time reductions. We also note that deallocation in reverse allocation order is slightly better than deallocating memory in allocation order. Regarding the space-consumption it is almost not affected at all. The reason is that the memory manager always provides the application with memory blocks whose size is rounded up to the nearest power of two. For most examples it is possible to extend the symbolic states with the two extra pointers without affecting the sizes of the memory blocks at all. For some examples the pointers causes larger blocks to be allocated. For a more thorough study involving two other operating systems as well, see [LPY00].

6.5 Re-Using the State-Space

In chapter 5 a technique that re-uses the generated state-space to speed up verification. This approach would obviously suffer bad performance due to swapping as well. There are other reasons as well for traversing PAST or WAIT without e deallocating them. In chapter 5 we discussed the possibility to dynamically adjust the size of the hash table used. This would Involve a traversal of the structures and rehashing them. The same reasoning that was used to find a better deallocation order of states may be used here as well since it is not the deallocation that is time consuming but rather the badly distributed accesses to the pages containing the symbolic states. The same thing will occur for any traversal of the state-space with bad locality if swapping is involved. Measurements on the speed-up in verification time of multiple properties are presented in [LPY00].

Chapter 7

Conclusions and Future Work

7.1 Conclusions

During the work of implementing UPPAAL many discussions and decisions arises that are worth paying extra attention to. We summarise a few of the most important here.

One of the most important conclusions is that today's techniques and technology allow us to build tools that are capable of handling industrial-sized case-studies. In order to fully utilise the new possibilities that now arises a more important problem for the future seems to be of educational and pedagogical nature. How can we incorporate formal methods in the development cycle of real-time systems?

Another very important conclusion is that there is a quite huge gap between the formal notation used when reasoning in logic and the available constructs in the implementation language. A relatively simple mathematic expression may require very complex code in the implementation language. An example is the reduction of DBMs discussed in chapter 4 that would require us to replace the memory manager used in the operating system to get results that corresponds to the theoretically computed measures. To summarise it is important to choose appropriate algorithms and data structures but at least as important to implement them in a correct way in the programming language.

Complex Modelling Language vs Primitive

When developing a modelling language one often gets into discussions about what to incorporate in the modelling language and what to provide as external translators. Data variables and the operations performed on them may be modelled as automata with states and transitions but the users would find it inconvenient. Data variables are very useful to incorporate in the language but we could still provide them and translate them to automata before verifying the model. However, it is more efficient to support data variables even in the verifier and get rid of states, transitions and synchronisations which are more difficult to deal with efficiently.

On the other hand, we choose to support other constructs such as complex data types and hierarchical design of components externally and translate them to plain timed automata. The reason is that these constructs are too complex to handle in the verifier at least as long as compositional techniques that utilise the hierarchical design in the verification process are developed.

The issue of a complex user-friendly expressive modelling language versus a more primitive language easy to verify efficiently must be carefully considered. What shall be supported in the language? What shall be supported in the verifier? What shall be supported using external translators? We may make an analogy with compilers for high-level languages and byte code. While high-level languages are more user-friendly and allow the programmer/designer to express more easily what shall be performed they would not be efficient to interpret and execute directly. Instead the program is compiled to some primitive language which is efficient to execute but not useful to write programs and develop models in.

We believe that the approach of translation to a more primitive language works well. It is then also possible to support many different modelling languages as long as their semantics is expressible with timed automata.

Time and Space - Not Only a Trade-off

It is often claimed that there is a trade-off between time and space utilisation. One may save computation time by use of more space to save intermediate results or save space by recomputing partial results instead of storing them. While this is true for applications with a small or medium memory consumption things are different when the memory consumption is huge, and swapping is involved. It is almost always the case that an optimisation that saves space also leads to a gain in verification time.

Static or Dynamic Analysis of the Model

It is well worth the effort of spending time and space on static analysis to be able to speed up verification. Even though static analysis of a dynamic behaviour does not give the most accurate information its time and space consumption is negligible compared to the verification time. Examples of this is the pre-processing of the network, described in chapter 3, to sort the transitions and build the data structures used by the *sync* predicate and the test for transitions with urgent channels. Another example worth mentioning is the depth-first search to find loop entry states described in chapter 5.

Even Small Optimisations Pay Off

The major part of the execution time of Uppaal is spent during verification on operations on symbolic states, either during state-space traversals or computation of successor states. These operations are called at least one time for every symbolic state generated. Even if these operations consumes very little time one by one they are called so many times that a small speed-up of such an operation will increase the performance quite noticeably. Therefore even small optimisations pay off. Five or six optimisations that reduce the verification time about 10% each will together cut down the verification time by a factor of two.

Imposing Restrictions

Improvements in performance may be achieved if dynamic memory allocation is kept to a minimum. It is better to impose restrictions on how many components that can be verified, how many clocks may a DBM contain, how large may domains of data variables be, etc. In chapter 4 we studied the result of compressing the discrete part of a symbolic state. By restricting the number of reachable states the tool can handle we were able to use the built-in data types and operators in the implementation language more efficiently. We might loose the possibility of verifying all possible timed automata, which we cannot do anyway, but we obtain good performance for many examples. One way of achieving the goal of efficient mapping to static data types and still keep generality would be to use partial evaluation and given a model and set of properties produce a verifier optimised for exactly that model and set of properties.

7.2 Future Work

There are certainly much more that may be done in the subject of finding efficient algorithms and data structures for verification of timed automata. A few of them worth examining are summarised here.

Search Orders

The search orders used in Uppaal so far is breath-first and depth-first. It would be interesting to study other search orders, for example is it possible to find some heuristic that utilise knowledge about the property to select what transitions that are interesting to check.

Symbolic States

In chapter 3 we examined the use of static analysis to build data structures that helped us to speed up the verification. Similarly the technique of finding entry states in chapter 5 led to a better performance with respect to both space and time. Perhaps it is possible to find an even more efficient representation of the transition relation or the state-space by utilising such static pre-processing.

A transformation that preserves the relationship between matrix elements would be useful to reduce the size of the DBMs and increase the efficiency of handling them.

Symbolic State-Spaces

The hash function used in UPPAAL today assumes that all states will be visited merely the same number of times. This is obviously not the case and the question if it is possible to find better hash functions that utilise the network structure to determine what states will be reachable more often and hence spreading the states better reducing the chains in the hash table, is motivated. The use of heuristics and approximations for probabilistic verification might be worth studying further.

Memory Management

One interesting approach to examine is the use of free-lists to avoid the "reallocation" of memory blocks that occurs. However the studies in chapter 6 will still be useful because we still need a strategy to determine what part of the free-list to re-use at a given time, in order to achieve good locality.

Garbage collection can be used to reduce fragmentation. We can allocate larger chunks of memory, that can hold many symbolic states, at one time. Deallocation of a symbolic state is postponed until a sufficiently large block of consecutive states is ready for deallocation.

It may be the case that we end up writing our own Special-purpose Memory manager. It has a lot of advantages: We get total control over the allocation order and memory layout. We also reduce memory consumption and fragmentation because we will be able to allocate memory blocks no larger than what is needed to store a symbolic state. We saw in chapter 4 that the reduction technique for DBMs seems to rely on a memory manager in order to achieve a lower memory consumption.

Architectures of Verifiers

We have recently started to look into client-server design of the verifier. Verification tools require large computer resources when performing the computations and users often need to buy expensive machines. However, the tools used for drawing and developing the model does not require that much of computer power. It therefore makes sense to run the verification engine on one machine and the user interface part on another. The two parts do not need to be implemented in the same programming language either. When a model is developed it is done interactively and it is an iterative process of verifying, simulating and refining the model. To make this development more efficient it makes sense to ask what parts of the state-space may be re-used if the model or property changes slightly.

Appendix A

Measurements and Examples

This appendix describes the examples used throughout the thesis in the performed experiments. There is a short overview of each example together with references to more extensive treatments. The appendix also contains information on how the measurements were performed, e.g. details about the equipment used etc.

A.1 The Measurement Scheme

Memory consumption is measured by counting the maximum number of pages allocated by the process during execution and multiply it with the page size of the operating system. This will not give us the exact number of bytes required to represent the data in UPPAAL, it rather gives the number of bytes that the operating system has returned in response to memory allocation requests from the process running UPPAAL. Since the counting of pages is done by polling, there is a small probability that the process requests and get a few additional pages during the remaining execution time after the last polling has occurred. By choosing short polling intervals the error can be reduced to a few pages.

When measuring process execution time in a multi-tasking environment one has to be a bit careful. The elapsed time, from when UPPAAL starts until it exits, reported by the operating system may be a bad measure of the actual execution time. This is because other processes get scheduled, interrupts UPPAAL and executes. To avoid that, we use the sum of the user time and system time reported for the UPPAAL process by the operating system. The user time is the CPU time devoted to the process and the system time is the time that the process is blocked waiting for the operating system to perform requests from the application but not including time spent by other processes. To make sure that the times are accurate the process is executed multiple times and a mean value is used. Due to the complex cache management strategies used by operating systems it sometimes happens that a single execution produces a result with high deviation from the others. Such statistically insignificant values have been removed before computing the mean value. It makes sense to estimate errors to a part of a percentage.

The measurements in chapters 3 to 5 all follow the described method and are performed on a 200 MHz Pentium Pro with 256 MB of physical ram running Linux with 2.2.x kernels. The measurements in chapter 6 on memory management are performed in a slightly different way. The hardware used is a 75 MHz Pentium with 8 MB of physical ram running Linux with 2.0.x kernels. More important is the fact that the time spent by the operating system in deallocating memory used by a process is not accounted for correctly in the system time returned by the operating system. Therefore we are forced to measure the elapsed time instead and attempts have been made to reduce unnecessary errors by removing as many other executing processes as possible. Therefore one can expect a bit lower confidence but the error shall still be small and not more than a few percentages.

A.2 Examples

A series of examples are used throughout the thesis to perform the measurements and produce the tables in chapters 3 to 5. They will be described briefly in this section and there will be references to more detailed descriptions of the examples themselves and their modelling in Uppaal. The tables in chapter 6 are produced using slightly modified versions of many of the examples described here but the details about the modifications are not considered in the thesis.

A.2.1 Audio Control Protocols from Philips

Philips' audio equipment consists of many different components and all of them can be controlled with the same remote controller. This requires some kind of protocol to enable the remote controller and the components to communicate with each other. Our example series contain two models of control protocols for audio equipment. The simplest consist of a sender that transmits messages to a receiver. The messages are coded with Manchester coding. The environment consists of two components, one that feeds messages to the sender and another that delivers messages from the receiver and checks that the received messages are correct. The model uses four automata and most of them have between five and 12 states. It uses two clocks, four data variables and 12 synchronisation channels.

The more complex model allow multiple senders sharing a bus and hence involve mechanisms for detecting and resolving collisions. It consists of two senders with 13 states in each, one receiver, a wire describing the bus and an environment with a message generator, using 17 states, and an automaton checking that the messages that reach the receiver are correct. Apart from the six automata There are five clocks, 11 data variables and 18 synchronisation channels altogether. We verify six safety properties for the smaller model and one for the larger. More about the protocols and models may be found in [BPV94, HWT95, BGK$^+$96, Gri94]. The protocols have become somewhat of standard bench-marks for model checkers. The table label for the smaller one is audio and for the larger example it is audio_bus.

A.2.2 An Audio Control Protocol from Bang and Olufsen

Bang & Olufsen is a Danish company also making home HIFI equipment. This example is based on a control protocol used in their equipment. The model consists of nine automata. Five automata models the two senders, the two detectors and the bus. The environment is modelled by four automata, two generates frames for the senders and two observers check that the frames are handled correctly by the protocol. The model uses three clocks, 26 data variables and 12 synchronisation channels. The senders consist of 20 states each and the additional components have an average of about five states. We verify one safety property. More about the protocol and model may be found in [HSLL97]. The table label is B&O.

A.2.3 A Bounded Re-transmission Protocol

This example models a two layer computer communication. A sender transfers a file to a receiver by dividing it into multiple packets with sequence numbers. The under-lying layer consists of another sender and receiver responsible for transmitting the packets over a lossy media using a variation of the alternating-bit protocol. The model consists of six automata with about five states in each, four clocks, seven data variables and 12 synchronisation channels. We verify one safety property. More information about this example can be found in [DKRT97, DKRT96]. The table label is brp.

A.2.4 Synchronising Start-Up protocols using TDMA

This example describes a synchronisation protocol for start-up of 3 units and get them synchronised. The units communicate with TDMA over a bus. We verify two models of the protocol. The main difference is in their abstraction levels. Both models use six automata including one for each of the three stations with about 25 states in each and one for the bus. Further they have four clocks, 12 data variables and six synchronisation channels. However, the more detailed model has a lot more transitions than the other. We verify one safety property. More information is in [LP97]. The table label for the more abstract model is dacapo_sim and the label for the more detailed one is dacapo_system.

A.2.5 A Gear-Controller from Mecel

Mecel is a Swedish company manufacturing car equipment. This model describes a fictions gear controller developed in collaboration with them. The model consists of a gear box, a gear controller and a clutch. The model of the environment consists of an engine and an interface to the gear controller. The model uses five clocks, four data variables and 15 synchronisation channels. There is one automaton for each component and the number of states in the five automata ranges from 6 in the simplest one to 25 in the most complex. We verify 46 safety properties. More details about the model may be found in [LPY98]. The table label is engine.

A.2.6 A Manufacturing Plant

This example models an abstract manufacturing plant. There is one robot picking boxes from a service station and placing them on a belt. Another robot take the boxes off from the belt after a certain time and hand them over to the service station again. The model uses five clocks and 9 synchronisation channels but no data variables. There are two automata modelling two boxes and three additional automata for the robots and the service station. In average there is about seven states in each automaton. We verify one safety property. A more detailed description can be found in [DY95]. The table label is mplant.

A.2.7 Fischer's Protocol for Mutual Exclusion

Mutual exclusion is a well-known problem used to guarantee that only one of several processes executing in parallel at a time gets access to a critical resource that all processes are competing for. There are many protocols to achieve this and one of them is Fischer's protocol. The example series contains seven models of different sizes ranging from four processes to ten. There is one automaton and one clock for each process. Each automaton has four states and the model contains one data variable. No synchronisation channels are used. Fischer's protocol is a standard bench-mark for model checking tools. We verify one safety property. Information about the protocol can be found in [Lam87, KLL$^+$97]. The table labels are fischer4 to fischer10.

Bibliography

[AD90] Rajeev Alur and David Dill. Automata for Modelling Real-Time Systems. In *Proc. of Int. Colloquium on Algorithms, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 322–335, July 1990.

[Bal96] Felice Balarin. Approximate Reachability Analysis of Timed Automata. In *Proc. of the 17*th *IEEE Real-Time Systems Symposium*, pages 52–61. IEEE Computer Society Press, December 1996.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *Proc. of the 10*th *Int. Conf. on Computer Aided Verification*, number 1427 in Lecture Notes in Computer Science, pages 546–550. Springer–Verlag, 1998.

[Bel57] Richard Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BFK+98] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic Verification of a Lip Synchronisation Algorithm using Uppaal. In *In Proc. of the 3*rd *Int. Workshop on Formal Methods for Industrial Critical Systems*, 1998.

[BFM98] H. Bowman, G. Faconti, and M. Massink. Specification and Verification of Media Constraints using Uppaal. In *Proc. of the Eurographics Workshop on the Design, Specification and Verification of Interactive Systems*, Eurographics Series. Springer-Verlag, 1998.

[BGK+96] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using Uppaal. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of the 8*th *Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[BL96] Johan Bengtsson and Fredrik Larsson. Uppaal a Tool for Automatic Verification of Real-time Systems. Master's thesis, Uppsala University, 1996.

[BLL+95a] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — A Tool Suite for Symbolic and Compositional Verification of Real-Time Systems. Presented at the 1*st* Workshop on Tools and Algorithms for the Construction and Analysis of Systems, May 1995.

[BLL$^+$95b] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[BLL$^+$98] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Wang Yi, and Carsten Weise. New Generation of Uppaal. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[Boe93] Hans-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 863 in Lecture Notes in Computer Science, 1994.

[BvKST97] K. Brink, J. van Katwijk, R.F. Lutje Spelberg, and W.J. Toetenel. Analyzing Schedulability of Astral Specifications using Extended Timed Automata. In *Third International Euro-Par Conference*, number 1300 in Lecture Notes in Computer Science, pages 1290–1297. Springer–Verlag, 1997.

[BW90] Allan Burns and Andy Welling. *Real-time systems and their programming languages.* Addison-Wesley, 1990.

[DDHY92] David. L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[Dij75] E. W. Dijkstra. Guarded commands, non-determinacy and a calculus for the derivation of programs. *ACM SIGPLAN Notices*, 10(6):2–14, June 1975.

[DKRT96] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. Modeling and Verifying a Bounded Retransmission Protocol. In *Proc. of COST 247, Int. Workshop on Applied Formal Methods in System Design*, June 1996. Also appears as Technical Report CTIT 96-22, University of Twente, July 1996.

[DKRT97] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proc. of the 3*rd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science, pages 416–431. Springer–Verlag, April 1997.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 208–219. Springer–Verlag, October 1995.

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In Bernard Steffen, editor, *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 313–329. Springer–Verlag, 1998.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with Kronos. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 66–75. IEEE Computer Society Press, December 1995.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proc. of the 17*th *IEEE Real-Time Systems Symposium*, pages 73–81. IEEE Computer Society Press, 1996.

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

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[Gri81] David Gries. *The Science of Programming.* Springer–Verlag, 1981.

[Gri94] W.O. David Griffioen. Analysis of an Audio Control Protocol with Bus Collision. Master's thesis, University of Amsterdam, Programming Research Group, 1994.

[HH95] Thomas A. Henzinger and Pei-Hsin Ho. HyTech: The Cornell HYbrid TECHnology Tool. In *Proc. of TACAS, Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1995. BRICS report series NS–95–2.

[HHWT95a] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A Users Guide to HyTech. Technical report, Department of Computer Science, Cornell University, 1995.

[HHWT95b] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: The Next Generation. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 56–65. IEEE Computer Society Press, December 1995.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A Model Checker for Hybird Systems. In Orna Grumberg, editor, *Proc. of the 9*th *Int. Conf. on Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 460–463. Springer–Verlag, 1997.

[HNSY92] Thomas. A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE Symp. on Logic in Computer Science*, 1992.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *ACM Communications*, 12(10):576–583, 1969.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[Hol91] Gerard Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hol97] Gerald J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1997.

[HWT95] Pei-Hsin Ho and Howard Wong-Toi. Automated Analysis of an Audio Control Protocol. In *Proc. of the 7th Int. Conf. on Computer Aided Verification*, number 939 in Lecture Notes in Computer Science. Springer–Verlag, 1995.

[JLS96] Henrik E. Jensen, Kim G. Larsen, and Arne Skou. Modelling and Analysis of a Collision Avoidance Protocol Using SPIN and UPPAAL. In *Proc. of 2nd Int. Workshop on the SPIN Verification System*, pages 1–20, August 1996.

[KLL+97] Kåre J. Kristoffersen, Francois Laroussinie, Kim G. Larsen, Paul Pettersson, and Wang Yi. A Compositional Proof of a Real-Time Mutual Exclusion Protocol. In *Proc. of the 7th Int. Joint Conf. on the Theory and Practice of Software Development*, April 1997.

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

[LH96] Kim G. Larsen and Hans Hüttel. UPPAAL — An Automatic Tool for Verification of Real Time and Hybrid Systems. Seminar slides from *Livslang Uddannelse 96*, 1996. Email: {kgl,hans}@cs.auc.dk.

[LLPY97] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[LP97] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.

[LPY95a] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.

[LPY95b] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 575–586. Springer–Verlag, October 1995.

[LPY97a] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LPY97b] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL: Status and developments. In Orna Grumberg, editor, *Proc. of the 9*th *Int. Conf. on Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 456–459. Springer–Verlag, June 1997.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer–Verlag, March 1998.

[LPY00] Fredrik Larsson, Paul Pettersson, and Wang Yi. On memory-block traversal problems in model checking timed systems. In *Proc. of the 6*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer–Verlag, March 2000.

[Mil89] R. Milner. *Communication and Concurrency.* prentice, Englewood Cliffs, 1989.

[Pet77] James L. Peterson. Petri-Nets. *Computer Surveys*, 9(3), September 1977.

[Pet99] Paul Pettersson. *Modelling and Analysis of Real-Time Systems Using Timed Automata: Theory and Practice.* PhD thesis, Department of Computer Systems, Uppsala University, February 1999.

[RM94] Tomas G. Rokicki and Chris J. Myers. Automatic verification of timed circuits. In David L. Dill, editor, *Proc. of the 6*th *Int. Conf. on Computer Aided Verification*, number 818 in Lecture Notes in Computer Science, pages 468–480. Springer–Verlag, 1994.

[Rok93] Tomas Gerhard Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.

[SD96] U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90, 1996.

[SD98] Ulrich Stern and David L. Dill. Using Magnetic Disk instead of Main Memory in the Murphi Verifier. In *Proc. of the 10*th *Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science. Springer–Verlag, June 1998.

[Sed92] Robert Sedgewick. *Algorithms in C++.* Addison-Wesley, 1992.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. of the International Workshop on Memory Management*, number 637 in LNCS. Springer–Verlag, 1992.

[WL93] Pierre Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. of the 5*th *Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science, pages 59–70. Springer–Verlag, 1993.

[WT94] Howard Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Stanford University, November 1994.

[YL93] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real–time transition systems. In *Proc. of the 5*th *Int. Conf. on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 210–224, 1993.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.