Clocks, DBMs and States in Timed Systems

Johan Bengtsson

A Dissertation submitted
for the Degree of Doctor of Philosophy
Department of Information Technology
Uppsala University

June 2002

Dissertation for the Degree of Doctor of Philosophy in Computer science with specialization in Real Time Systems presented at Uppsala University in 2002.

ABSTRACT

Bengtsson, J. 2002: Clocks, DBMs and States in Timed Systems. Acta Universitatis Upsaliensis *Uppsala Dissertations from the Faculty of Science and Technology* 39. 143 pp. Uppsala. ISBN 91-554-5350-3

Today, computers are used to control various technical systems in our society. In many cases, time plays a crucial role in the operation of computers embedded in such systems. This thesis is about techniques and tools for the analysis of timing behaviours of computer systems. Its main contributions are in the development and implementation of UPPAAL, a tool designed to automate the analysis process of systems modelled as timed automata.

As the first contribution, we present a software package for timing constraints represented as Difference Bound Matrices. We describe in details, all data-structures and operations for DBMs needed in state-space exploration of timed automata, as well as techniques for efficient implementation. In particular, we have developed two normalisation algorithms to guarantee termination of reachability analysis for timed automata containing constraints on clock differences, that transform DBMs according to not only maximal constants of clocks as in algorithms published in the literature, but also difference constraints appearing in the automata. The second contribution of this thesis is a collection of low level optimisations on the internal data-structures and algorithms of UPPAAL to minimise memory and time consumption. We present compression techniques to allow the state-space of a system to be efficiently stored and manipulated in main memory. We also study super-trace and hash-compaction methods for timed automata to deal with system-models for which the size of available memory is too small to store the explored state-space. Our experiments show that these techniques have greatly improved the performance of UPPAAL. The third contribution is in partial-order reduction techniques for timed-systems. A major problem in automatic verification is the large number of redundant states and transitions introduced by modelling concurrent events as interleaved transitions. We propose a notion of committed locations for timed automata. Committed locations are annotations that can be used for not only modelling of intermediate states within atomic transitions, but also guiding the model checker to ignore unnecessary interleavings in state-space exploration. The notion of committed locations has been generalised to give a local-time semantics for networks of timed automata, which allows for the application of existing partial order reduction techniques to timed systems.

Johan Bengtsson, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden.

ISSN 1104-2516
ISBN 91-554-5350-3
Printed in Sweden by Elanders Gotab, Stockholm 2002
Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala, Sweden

Till Erika och Simon

Acknowledgements

First of all I want to thank my supervisor, Wang Yi. Without his guiding and support this thesis would never have been completed. I have learnt a lot during the years we have been working together and I hope that someday I will be as good a reasearcher as he is. I would like to thank all current and former members of the UPPAAL group here in Uppsala, *i.e.* Tobias Amnell, Alexande David, Elena Fersman, Fredrik Larsson, Leonid Mokrushin, Paul Pettersson, and Justin Pearson, for the stimulating environment and all nice moments both in and outside the department. Specially I would like to thank Fredrik and Paul who were around from the very beginning of the UPPAAL-project. I would also like to thank Kim G. Larsen, Gerd Behrmann, and the rest of the UPPAAL group in Aalborg for fruitful collaboration over the years. Without their participation UPPAAL would not have been what it is today. I am also grateful to my co-authors, *i.e.* Pedro D'Argenio, Ansgar Fehnker, David Griffioen, Bengt Johnsson and Johan Lilius, for fruitful discussions. It has been fun working together with you.

I would like to thank everyone at DoCS for making the department such an enjoyable environment. In particular I would like to thank Björn Victor and Anders Berglund for their support.

To my wife Erika, I give my love and my deepest thanks. Without her by my side I would never have reached this point. Finally, I want to thank my son Simon, my pride and joy, and my best source of inspiration.

This work has been partially supported by the Swedish Board for Technical Development (NUTEK), the Swedish Technical Research Council (TFR), and EC via the AIT-WOODDES project.

This thesis includes, summarises and discusses mainly the results presented in five research papers written between 1996 and 2002. These papers are listed as follows:

Paper A: Johan Bengtsson. DBM: Structures, Operations and Implementation. Submitted for publication.

Paper B: Johan Bengtsson and Wang Yi. Reachability Analysis of Timed Automata Containing Constraints on Clock Differences. Submitted for publication.

Paper C: Johan Bengtsson and Wang Yi. Reducing Memory Usage in Symbolic State-Space Exploration for Timed Systems. Technical Report, 2001-009, Department of Information Technology, Uppsala University, 2001.

Paper D: Johan Bengtsson, Bengt Jonsson, Johan Lilius and Wang Yi. Partial Order Reductions for Timed Systems. In *Proceedings, Ninth International Conference on Concurrency Theory*, volume 1466 of Lecture Notes in Computer Science, Springer Verlag, 1998.

Paper E: Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. Automated Verification of an Audio-Control Protocol using UPPAAL. Accepted for publication in *Journal on Logic and Algebraic Programming*.

Comments on My Participation

Paper A: I implemented the major part of the DBM package in UPPAAL, and wrote the report.

Paper B: I participated in discussions, designed and implemented the algorithms. I wrote a large part of the paper.

Paper C: I participated in discussions, designed and implemented the optimisation techniques. I wrote the paper.

Paper D: I participated in discussions and wrote part of the paper. I made a prototype implementation which is not described in this paper.

Paper E: I participated in discussions and implemented committed locations in UPPAAL. I have also made minor revisions to the semantics for committed location.

Apart from the papers listed above, I have also participated in the following work:

- Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In *Proceedings of Modelling and Verification of Parallel Processes*, volume 2067 of LNCS, 2001.
- Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Wang Yi and Carsten Weise. New Generation of UPPAAL. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*, 1998
- Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Wang Yi. UPPAAL in 1995, In *Proceedings of Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of LNCS, 1996.
- Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, volume 1066 of LNCS, 1995.

Contents

Introduction

1 Background

The computer boom in the last decades has not only given us faster and more advanced equipment for word-processing, banking and scientific computations. The development of small, cheap, and powerful microprocessors has also enabled a range of new application areas. Today, computers are trusted to control all kinds of devices and technical systems used in our society, ranging from stereos and micro-wave ovens to life-support systems and nuclear power-plants. Currently this kind of computer applications, *i.e.* the *embedded systems*, forms the fastest growing market for microprocessors. In recent years 98%–99% of all produced processors have been used in embedded systems [Hal00]. Only around 2% are used in desktop computers.

In many embedded applications, a computer failure may have dire consequences, such as economical damage, environmental catastrophes and even loss of human lives. Thus it is of great importance that such systems operate correctly, according to their specifications. Traditionally this is accomplished by methods like reviewing of design documents and source code and by extensive simulation and testing of the system and its components. However, this process is often time-consuming and provides only statistical measures of correctness.

As a complement to the traditional methods for obtaining correct software, a large number of mathematically based techniques for reasoning about correctness of computer systems have been proposed in the literature, *e.g.* [Hoa69, Dij75, Hoa78, Rei85, Mil89, Hol91]. The common procedure in all these so called formal methods is to define the system under development in a formal framework and then apply rigorous methods within this framework to prove that the system meets its requirements. However, due to the complexity of real-life systems, applying formal methods is often considered too difficult. A way

to bridge this gap is to automate the analysis, for instance by using *model-checking* [CGP99]. In contrast to manual techniques, model-checking is fully automatic in the sense that the proof showing that a system satisfies a given requirement is constructed by the model-checker without manual interaction.

In this thesis we study and develop techniques for model-checking tools to verify systems where timing is important. The work presented in this thesis is in the context of the model-checker UPPAAL, which is a tool for analysing timing properties of systems modelled as timed automata.

2 Timed Automata

Timed automata [AD90, AD94, HNSY94] is one of the most successful formalisms for describing the timing behaviour of computer systems. Examples of other formal systems with the same purpose, are timed Petri-net models, timed process algebras or real time logics [BD91, RR88, Yi91, NS94, ACD93, AH94, SS95].

A Brief Introduction to Timed Automata

A timed automaton is essentially a finite automaton extended by a set of real valued clocks. All clocks are synchronised in the sense that they start with the value zero when the system is initialised and grow synchronously, with the same rate. The clocks influence the automaton by clock-constraints (guards) on the edges. An edge is only enabled when the values of the clocks satisfies the guard. The automaton can influence the clocks by letting the edges reset a subset of the clocks.

An example of a timed automaton is shown in Figure 1. The timing behaviour of this automaton is controlled by two clocks x and y. The clock x is used to control the small self-loop in the **loop** location. This loop is only possible when the value of x is exactly one. The clock y controls the execution of the entire automaton. The automaton may leave **start** when y is between 10 and 20, it can go from **loop** to **end** when y is between 40 and 50, *etc*.

In this simple timed-automata model, the guards on the edges only guide the automaton when it decides to act; they can not force it. The example automaton may stay forever in any location, just idling. In the initial work by Alur and Dill [AD90] the problem is solved by introducing Büchi-acceptance conditions;

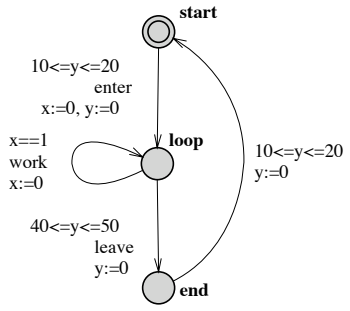


Figure 1: A Timed Automaton

a subset of the locations in the automaton are marked as accepting, and only executions passing through an accepting location infinitely often are considered valid. As an example, consider again the automaton in Figure 1 and assume that **end** is marked as accepting. This implies that all executions of the system enter **end** infinitely many times. This impose implicit conditions on **start** and **loop**. The location **start** must be left when the value of y is at most 20, otherwise the automaton would get stuck and never be able to enter **end**. Likewise, the automaton must leave **loop** when y is at most 50 to be able to enter **end**.

Büchi-acceptance conditions are theoretically elegant. However, as the example shows, an accepting location has global effects on the automaton, which is inconvenient for modelling and automatic analysis. A more intuitive notion of progress is used in timed safety-automata [HNSY94]. In timed safety-automata, the acceptance conditions are replaced by local timing constraints in each location, that is the so called location invariants. The automaton may only remain in a location as long as the clocks satisfies the invariant condition. This gives a more convenient view of the timing behaviour of particular parts of the automaton. For example, consider the automaton in Figure 2. This timed safety-automaton corresponds to the Büchi automaton in the previous example. In this automaton the conditions that **start** and **end** must be left when y is at most 20 and **loop** must be left when y is at most 50 are now stated locally in invariants. This gives a local view of the timing in each location in much more convenient way than Büchi-acceptance. For this reason we will use timed automata with location invariants to describe the timing behaviour of embedded systems.

To model concurrent systems, timed automata is often extended with parallel composition, giving *networks of timed automata*. In the early work on timed-automata [AD90, HNSY92], parallel composition is treated as logical con-

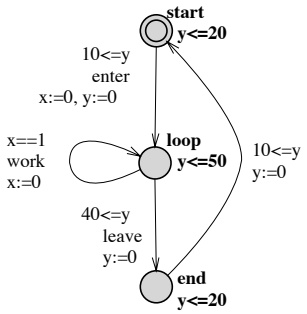


Figure 2: A Timed Safety-Automaton

junction, *i.e.* all processes in a system must synchronise on every action. In this thesis, as well as in UPPAAL, we adopt the parallel composition used in CCS [Mil89], which allows for internal actions in processes, as well as pair-wise synchronisation. An example of a system composed of two timed-automata is presented in Figure 3. The network models a time-dependent light-switch (to the left) and its user (to the right). The user and the switch communicate by the press labels. The user can press the switch (press!) and the switch waits to be pressed (press?). The product automaton, *i.e.* the automaton describing the combined system is shown in Figure 4. Building the product automaton is an entirely syntactical operation based on the component automata. In UPPAAL, the product automaton is computed on-the-fly during verification.

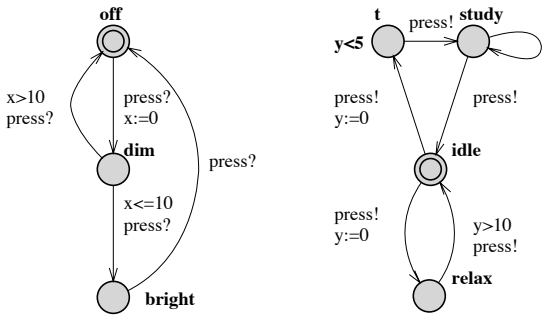


Figure 3: Network of Timed Automata

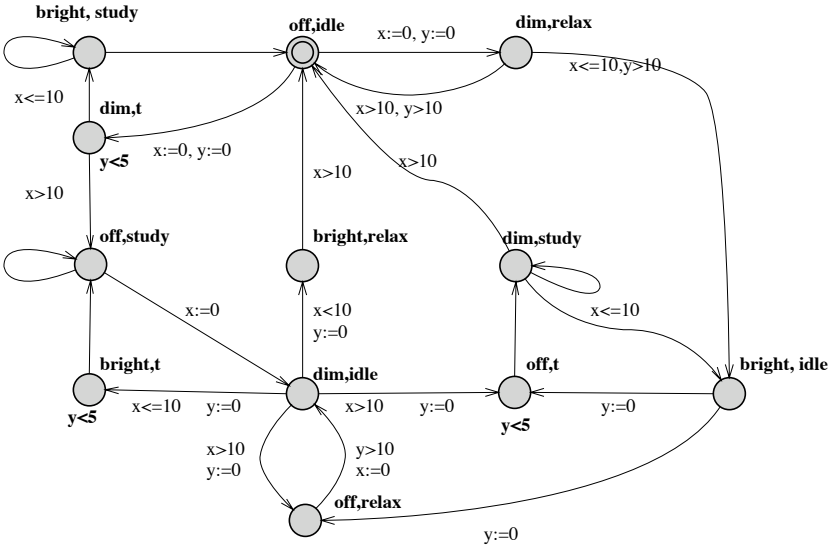


Figure 4: Product Automaton for the Network in Figure 3

Semantics of Timed Automata

A state or configuration of a timed automaton has two different parts, the current location and the current values of all clocks. There are two different types of transitions between states. The automaton may either delay for some time (a delay transition), or follow an enabled edge (an action transition). Since clocks in a timed automaton are real-valued the state-space will be infinite and not a good base for automatic verification. However, many verification problems such as language emptiness, reachability analysis and model-checking of timed logics can still be decided by constructing a *region graph* based on the automaton under consideration and perform the analysis on this graph [AD94, ACD93]. The downside with this solution is the large number of states in the region-graph. As an example, consider Figure 5. The figure shows the possible regions in each location of an automaton with two clocks x and y. The largest number compared to x is 3, and the largest number compared to y is 2. In the figure each line segment, each intersection and each area is one region. Thus, the number of possible regions in each location of this example is 60. It also grows rapidly with the number of clocks and the largest constants. In fact, the region graph is exponential in the number of clocks in the system as well as the in largest constants appearing in guards in the automaton.

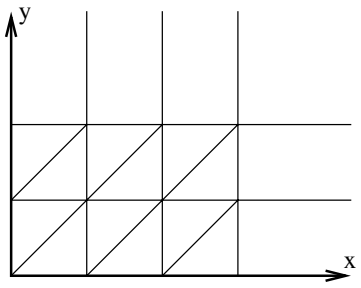


Figure 5: Regions for a System with Two Clocks

A more appealing representation of the state-space for timed automata is obtained by using *zone-graphs* [Dil89, HNSY94, ACH+95]. Instead of regions, the clock values are represented as constraints on clocks and clock differences. In practice this gives a coarser and thus more compact representation of the state space. The basic operations and algorithms to construct zone-graphs are described in Paper A. As an example, a timed automaton and the corresponding zone graph is shown in Figure 6. We note that for this automaton the zone graph has only eight states. The region-graph for the same example has over fifty.

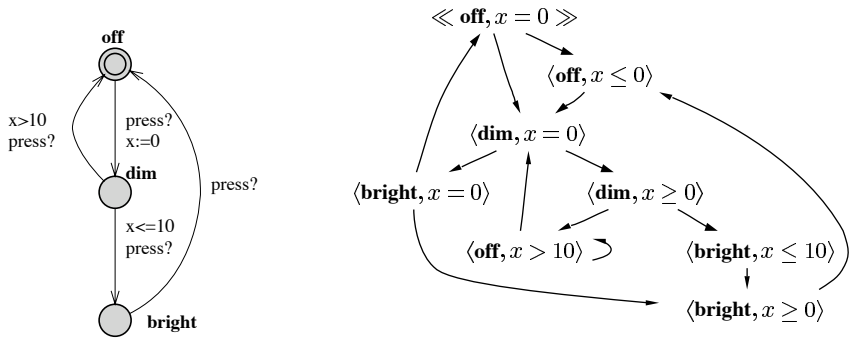


Figure 6: A Timed Automaton and the corresponding Zone Graph

The zone-graph of a timed automaton may unfortunately be infinite and verification based on zone-graphs is thus not guaranteed to terminate. As an example, consider the model in Figure 7. In this automaton the value of clock y drifts away unboundedly, giving an infinite graph. The solution is to normalise the zones in the zone-graph with respect to the maximum constants in the automaton, that is the so called k-*normalisation* [Rok93, Pet99]. The intuition is that once the value of a clock is larger than the maximum constant in the au-

tomaton it is no longer significant for the automaton to know its precise value, only that it is above the constant. As an example, the k-normalised zone-graph of the automaton in Figure 7 is given in Figure 8. Note that the k-normalisation only works for timed automata with guards on individual clocks. For automata with guards on clock differences a more elaborate normalisation procedure is needed (see Paper B).

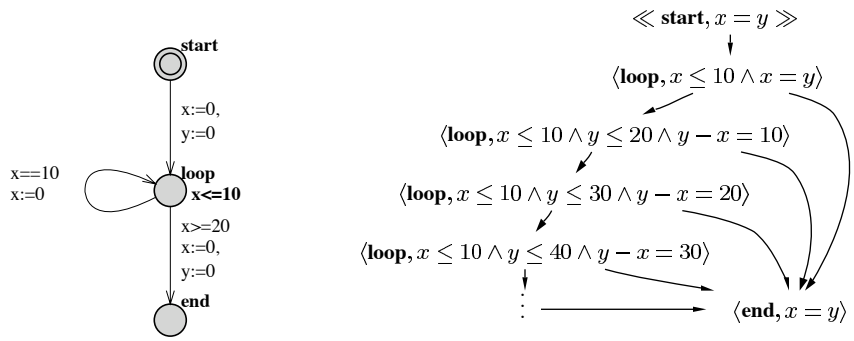


Figure 7: A Timed Automaton with an Infinite Zone Graph

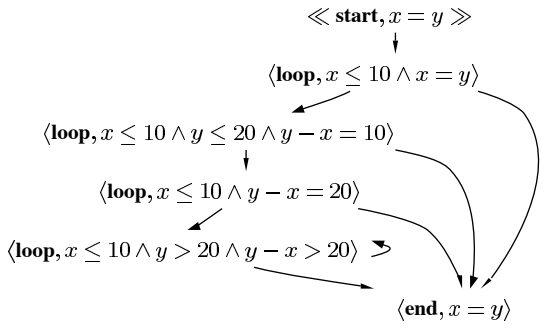


Figure 8: Normalised Zone Graph for the Automaton in Figure 7

3 Model Checking

Model-checking consists of two steps, computing the state-space of the system under consideration, and searching for states, or sets of states that satisfy certain logical properties. The first step can either be performed prior to the search,

or done *on-the-fly* during the search process. Computing the state-space on-the-fly has an obvious advantage over pre-computing, in that only the part of the state-space needed to prove (or disprove) the property is generated. As an example, consider a system with a million states. Generating the full state-space of this system will take a lot of time and may even be infeasible due to memory requirements. However, if the logical properties of interests can be proven by only examining the first hundred states of the system then an on-the-fly method will only need to generate those states and the result will be almost instantaneous. It should be noted though, that even on-the-fly methods will generate the entire state-space to prove certain properties, *e.g.* invariant properties.

The model-checking procedure itself is based on traversing the state-space in search for states that prove or contradict stated logical properties. As an example, we use one of the model-checking algorithms for timed-automata implemented in UPPAAL (see Algorithm 1).

Algorithm 1 Reachability analysis

PASSED $= \emptyset$, WAIT $= \{\langle l_0, D_0\rangle\}$
while WAIT $\neq \emptyset$ **do**
  take $\langle l, D\rangle$ from WAIT
  if $l = l_f \wedge D \cap D_f \neq \emptyset$ **then return** "YES"
  if $D \not\subseteq D'$ for all $\langle l, D'\rangle \in$ PASSED **then**
    add $\langle l, D\rangle$ to PASSED
    for all $\langle l', D'\rangle$ such that $\langle l, D\rangle \rightsquigarrow \langle l', D'\rangle$ **do**
      add $\langle l', D'\rangle$ to WAIT
    end for
  end if
end while
return "NO"

The algorithm is used for verifying *safety properties* (that something bad will never happen) for networks of timed automata. The properties are given to the algorithm as a set of bad states $\langle l_f, D_f\rangle$ (l_f denotes a location in the network and D_f denotes a clock zone). The algorithm will then compute the zone-graph of the system on-the-fly, in search for states intersecting with $\langle l_f, D_f\rangle$. The algorithm highlights some of the issues in developing a model-checker for timed automata. First, handling of states, or primarily zones, is crucial to the performance of the model-checker. Thus, devising algorithms and data-structures for zones is a major issue in developing a verification tool for timed automata, which is addressed in Paper A. Second, PASSED holds all states encountered so

far and its size puts a limit on how large systems we can verify. This makes it important to find compact representations for PASSED. However, since passed is searched each time a new state is processed it is crucial to performance that searching is cheap. This leads to a tradeoff between size and speed. We study this problem in Paper C.

A major obstacle in constructing the state-space of a network of timed automata is the so called state-explosion problem. The problem appears in all kinds of model-checkers, and the source of the problem is the practice of modelling concurrent systems using parallel composition of several automata. The effect is a rapid growth of the explored state-space due to the fact that each possible interleaving of the automata will be taken into account by the analysis.

A method for reducing the state-space used in model-checkers for untimed systems is partial-order reductions. With this technique the state-space is reduced by removing interleavings of independent transitions and choose one representative. However, in timed-automata the number of independent transitions is much less than in the untimed case due to the tight synchronisation of time. One way of breaking this synchronisation, without affecting stated reachability properties, is presented in Paper D.

Finally we shall point out that a powerful verification engine is not enough to get a successful tool and there are other issues in building verification tools. For a tool to be widely used it requires substantial work on the user interface. For instance, in UPPAAL a lot of effort has been put on improving the user interface. However, such issues are not the topic of this thesis.

4 Contributions of This Thesis

The main contributions of this thesis are in the development and implementation of UPPAAL, a tool-suite for timed-automata developed in collaboration between Uppsala University and Aalborg University. The first version of UPPAAL was released in spring 1995, and was then the first verification tool for timed automata where the system model could be drawn graphically (What You See Is What You Verify). In 1996, another step was taken in the same direction, when UPPAAL was equipped with a simulator that enable the user to test system models interactively and to visualise counterexamples generated by the model-checker. In 1999, the architecture of UPPAAL was completely changed, the tool was separated into two parts: A graphical user interface and a verification engine. Among other things this change enabled the GUI and the

verification engine to be run on separate machines, *e.g.* running the GUI on a local work-station and the verification engine on a large server. Today UPPAAL is one of the most popular tools for timed automata and it has been downloaded by over 1300 different users from over 70 countries.

The performance of the verification engine has always been a key issue when developing UPPAAL. This is also the part where most of the contributions described in this thesis are located.

A DBM Package: Data-Structures and Operations [Paper A,B]

Zones are the most important entities in a model-checker for timed systems and their representation and implementation are crucial to the performance. In implementing UPPAAL we discovered a number of techniques to improve performance of zone operations. In this thesis we present our experiences in form of a cook-book on Difference Bound Matrices [Dil89]. We describe all primitive operations and data-structures for zones needed to implement state-space exploration for time-automata in both forwards and backwards analysis, including operations for checking properties of zones, for computing the effect of delaying and resetting of clocks, and for constraining a zone with respect to a guard. We also present how to normalise the clock-zones with respect to guards in the automaton to guarantee that the model-checking procedure terminates. In particular we discovered that the existing k-normalisation algorithms published in the literature do not work for timed-automata with guards on clock-differences. We have developed two new normalisation algorithms for automata that may contain constraints on clock differences.

Low-Level Optimisations [Paper C]

The performance of UPPAAL has been greatly improved by low-level optimisations on its central algorithms and data-structures. In this thesis we develop and evaluate a number of techniques to minimise memory and time consumption. The techniques are implemented in UPPAAL and we believe that they are generic and applicable to model-checking of timed-systems in general.

We present two different methods for packing states. First, we code the entire state as one large number using a multiply-and-add algorithm. This method yields a representation that is canonical and minimal in terms of memory usage but the performance for inclusion checking between states is poor. The second method is mainly intended to use for the timing part of the state and it is based on concatenation of bit strings. Using a special concatenation of the bit string

representation of the constraints in a zone, ideas from [PS80] can be used to implement fast inclusion checking between packed zones.

The problem representing large state-spaces is addressed in two different ways. First, to get rid of states that do not need to be explored, as early as possible, we introduce inclusion checking already in the data structure keeping the states waiting to be explored. We also describe an implementation where time, as well as memory, is saved by this technique. Second, we investigate how super-trace [Hol91] and hash-compaction [WL93, SD95b] methods can be applied to timed systems. We present a variant of the hash compaction method, that allows termination of branches in the search tree based on probable inclusion checking between states. These techniques have been implemented in the UPPAAL tool, evaluated and compared by real-life examples; their strengths and weaknesses are described.

Partial Order Reduction [Paper D,E]

A major problem in automatic verification is the large number of states introduced by modelling concurrent events as interleaved transitions. In many cases, the order of the transitions are irrelevant for the investigated properties and one specific order can be chosen to represent all. In this thesis we address this problem in a setting of timed systems. We present a notion of committed locations for timed automata. Committed locations are annotations that can be used for not only modelling of intermediate states within atomic transitions, but also guiding the model checker to ignore unnecessary interleavings in state-space exploration. During modelling, intermediate locations in sequences of internal actions can be marked as committed. This prohibits delay in the locations and allows its transitions to be interleaved only with transitions of committed locations in other automata in the network. We present a modified algorithm for state space exploration for networks of timed automata which generate a reduced number of states when committed locations are used. Our experimental results demonstrate significant time and space-savings for a number of applications. For example, the audio control protocol presented in Paper E could not have been verified in 1995 without using committed locations.

Committed location is a simple case of getting rid of unnecessary interleavings in sequences of transitions without delays. Naturally, we want to extend the notion to sequences with non-zero delays. However, due to the tight synchronisation of time, delay in timed-automata has a global effect. To resolve this issue, we introduce a notion of local-time and let local clocks in each process advance independently of clocks in other processes. To avoid communication

between processes with different notion of the current time we require processes to resynchronise their local time scales whenever they communicate. A symbolic local-time semantics is developed in terms of predicate transformers, which enjoys the desired property that two predicate transformers are independent if they correspond to disjoint transitions in different processes. This allows existing partial order reduction techniques to be applied to the problem of reachability analysis for timed systems.

5 Related Work

In the past years, a large number of model-checkers have been developed by researchers for different application areas. For examples, we mention SPIN [Hol91, Hol97] for communication protocols and Murφ [DDHY92] for concurrent and reactive systems, UPPAAL [LPY97, ABB+01] and KRONOS [DOTY95, Yov97, BDM+98] for timed systems, and HYTECH [HHWT97] for hybrid systems. These tools have all been successfully applied to industrial-size case studies, *e.g.* [HLP98, JMMS98, SD95a, MMS97, LPY98, HSLL97, TY98, HWT96]. In this section we summarise the most successful and known techniques implemented in these tools.

Partial-order reduction: Partial-order methods [God90, Val90, Pel93] are based on the observation that in many cases the exact ordering of events affects neither the examined properties nor how the system evolves in the future. In these cases all equivalent orderings can be represented by one single ordering. For timed-automata there are two different approaches to applying partial order reductions. The first approach was introduced by Pagani in [Pag96] and later improved by Dams *et. al.* [DGKK98]. It is based on the global-time semantics of timed automata. This limits the possible reductions in that only transitions that can occur in exactly the same time-interval can be independent. The second approach is the local-time approach proposed in Paper D of this thesis. In [Min99], Minea extends our result for reachability analysis to LTL model-checking. Minea also shows that standard normalisation techniques for timing-constraints are applicable also in local-time semantics.

Symmetry reductions: Symmetry reduction [HJJJ84, ID96, ES97] is a method for exploiting the fact that many system-models have a large number of identical processes. It is often the case that these identical processes may be interchanged without noticeably affecting the system. As an example, in a communication protocol it may not matter if process A is sending and process B

is receiving or the other way around. In a model-checker this is exploited by defining equivalence classes for states based on different permutations of the identical processes. The model-checking algorithm is then applied to the graph of equivalence classes instead of the full state-space. As examples. we mention that this technique has been implemented in the tools Murφ [ID96] and SGM [HW98, WH02].

Symbolic model-checking: The key idea of symbolic model-checking is in representing and manipulating sets of states in terms of logical formulae. The best known symbolic technique, introduced in [BCM$^+$92], describe states and transitions as propositional formulae and use BDDs [Bry86] to store and manipulate them. The so called bounded model-checking is introduced in [BCCZ99]. In this work, the task of checking if a system-model has a certain logical property is transformed into checking satisfiability for a series of propositional formulae. Symbolic techniques for timed-systems are all based on zones [Dil89, HNSY94, AHH96, ACH$^+$95, YPD94], where sets of clock-values are represented and manipulated using clock constraints. This is the main topic of this thesis.

Approximation methods: Approximation methods are aimed at systems that are too large to be handled by precise methods, at the price that some results are inconclusive. There are two different types of approximation methods, under-approximations where part of the reachable state-space may be considered not reachable by the model-checker, and over-approximations where non-reachable states may be considered reachable by algorithm. Two examples of under-approximation are Supertrace [Hol91] and hash-compaction [WL93, SD95b]. These methods may conclude erroneously during verification that some unexplored state has been visited before. Thus, with under-approximations, a claim that invariant properties hold is inconclusive, since states violating the property may be lost in the approximation. Over-approximation techniques are often combined with symbolic model-checking. Often, the union of two symbolic-states can not be precisely represented by a single formula while it is possible to construct a formula including two symbolic states. In [Bal96] this type of method is applied to verification of timed automata. In the paper the convex-hull of time-zones are used as an approximation of union. In [WT94], Wong-Toi presents techniques for refining the results obtained by over-approximations by combining the results from forwards and backwards analysis.

Efficient Representation of Clock Constraints: In a verification tool, a large fraction of memory used during verification is spent on storing clock constraints. In the literature there are a number of techniques addressing this prob-

lem. In [DY96] live-range analysis is used to reduce the number of clocks in a model. The control-structure of the system-model is analysed to compute the set of active clocks for each location. To save space, only timing information regarding active clocks are stored. Another approach is taken in [LLPY97]. This work is based on the observation that the DBM representation of a time zone often contains redundant information, *i.e.* the same set of clock values can be represented using much fewer constraints. The paper presents an algorithm for computing the minimal set of constraints for a given DBM.

Inspired by the success of using BDDs to encode state-spaces in hardware verification, a number of similar techniques for representing clock zones were developed. In [Bal96], Balarin present a schema for encoding DBMs using DBBs. He also develop algorithms for performing essential DBM operations directly on the BDD. In [ABK+97], Asarin *et. al.* introduce Numerical Decision Diagrams (NDDs), a technique for representing sets of regions from the region-graph of timed automata as BDDs. The problem with both these representations is their sensitivity to time-granularity. This problem is not present for Clock Difference Diagrams (CDDs) [LPWY99, BLP+99] and Difference Decision Diagrams (DDDs) [MLAH99a, MLAH99b], which are two similar types of decision diagrams based on difference constraints. They allow for non-convex unions of zones to be represented by a single diagram, and the main difference is that in DDDs each node represents a single difference constraint, while a node in a CDD represents all difference constraints on a clock-pair. Thus, DDDs will, in general, have more nodes than the corresponding CDD, while the nodes in a CDD have a larger and variable number of edges.

Low-Level Optimisations: For many applications, it is not feasible to store all explored states in main memory. In [SD98], Stern and Dill present a method for storing PASSED on magnetic disk. To compensate for the increased time to access PASSED, they collect a large number of states in WAIT and check all of them in one sequential sweep through PASSED. Other alternatives when dealing with large state-spaces is to store only enough states to guarantee termination. In [LLPY97] static analysis of the system-model is used to compute the set of loop-entry location in the processes. A state is then stored in PASSED only if one of the processes enter such a location. This guarantee that at least one state in each dynamic loop will be present on PASSED and thus termination is ensured. A method for throwing identifying and removing states that no longer can be revisited is presented in [CKM01]. A progress measure is used by the model-checker to identify such states.

6 Conclusions and Future Work

This thesis summarises our experiences in developing and implementing UPPAAL. We have studied and developed techniques to improve the performance of UPPAAL in the verification process. Our main contributions are in three directions. First we have presented a DBM package including all data-structures and operations needed in symbolic state-space exploration of timed automata. We hope that the included reports may serve as a cook-book for tool developers of timed-systems. Second, we have presented and implemented a collection of low-level optimisations on the central algorithms and data-structures of UPPAAL. They have given significant performance improvements for the tool. Though these techniques are developed for UPPAAL, we believe that they are generic and applicable to other model-checkers for timed-systems. As the third contribution we have presented partial-order reduction techniques for timed systems. The notion of committed locations has been a useful mechanism for not only modelling atomic sequences but also guiding the model-checker to avoid redundant interleavings in state-space exploration. Though the notion of committed locations has been generalised to give a local-time semantics for networks of timed automata, which allows for partial order reductions in state-space exploration of timed systems, the technique has not yet been fully explored. This is a challenging area for future work. The local-time semantics is just a step on the way. A challenge is to develop an efficient implementation of the technique.

The work presented in this thesis can be extended in several directions. Hierarchical extensions to timed automata is a direction that is currently being pursued within the UPPAAL group. A challenging problem is how to take advantage of the hierarchical structures in state-space exploration. As future work, we will further study techniques to reduce memory requirements without losing performance in time, *e.g.* to develop packing methods and hash functions for zones, that preserve inclusion checking.

References

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In *Modelling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 100–125. Springer-Verlag, 2001.

[ABK+97] Eugene Asarin, Marius Bozga, Alain Kerbrat, Oded Maler, Amir Pnueli, and Anne Rasse. Data structures for the verification of timed automata. In *Proceedings, Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360. Springer-Verlag, 1997.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Journal of Information and Computation*, 104(1):2–34, 1993.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicholas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Journal of Theoretical Computer Science*, 138(1):3–34, 1995.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings, Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2):183–235, 1994.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[AHH96] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[Bal96] Felice Balarin. Approximate reachability analysis of timed automata. In *Proceedings, 17*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1996.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings, Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Journal of Information and Computation*, 98(2):142–170, 1992.

[BD91] Bernard Berthomieu and Michel Diaz. Modeling and verification of timed dependent systems using timed petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[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 *Proceedings, Tenth International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[BLP+99] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proceedings, Eleventh International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer-Verlag, 1999.

[Bry86] Randal E. Bryant. Graph-based algorithm for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[CKM01] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In *Proceedings, Seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001.

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

[DGKK98] Dennis Dams, Rob Gerth, Bart Knaack, and Ruurd Kuiper. Partial-order reduction techniques for real-time model checking. *Formal Aspects of Computing*, 10(5–6):469–482, 1998.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings, Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[DOTY95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool Kronos. In *Proceedings, Hybrid Systems III: Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[DY96] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *Proceedings, 17*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1996.

[ES97] E. Allen Emerson and A. Prasad Sistla. Using symmetry when modelchecking under fairness assumptions: An automata theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4), 1997.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings, Second International Conference on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer-Verlag, 1990.

[Hal00] Tom R. Halfhill. Embedded market breaks new ground. *Microprocessor Report*, January 2000.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *Journal on Software Tools for Technology Transfer*, pages 110–122, 1997.

[HJJJ84] Peter Huber, Arne M. Jensen, Leif O. Jespen, and Kurt Jensen. Towards reachability trees for high-level petri nets. In *Proceedings, Advances on Petri Nets '84*, volume 188 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[HLP98] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space craft controller using Spin. In *Proceedings, Fourth International SPIN Workshop*, 1998. Proccedings available online. URL: http://netlib.bell-labs.com/netlib/spin/ws98/program.html.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Journal of Information and Computation*, 111(2):193–244, 1994.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

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

[Hol97] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proceedings, 18*th *IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society Press, 1997.

[HW98] Pao-Ann Hsiung and Farn Wang. A state-graph manipulator tool for real-time system specification and verification. In *Proceedings, Fifth International Conference on Real-Time Computing Systems and Applications*, 1998.

[HWT96] Thomas A. Henzinger and Howard Wong-Toi. Using HYTECH to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, number 1165 in Lecture Notes in Computer Science, pages 265–282. Springer-Verlag, 1996.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Journal of Formal Methods in System Design*, 9, 1996.

[JMMS98] Wil Janssen, Radu Mateescu, Sjouke Mauw, and Jan Springintveld. Verifying business processes using SPIN. In *Proceedings, Fourth International SPIN Workshop*, 1998. Proccedings available online. URL: http://netlib.bell-labs.com/netlib/spin/ws98/program.html.

[LLPY97] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structure and state space reduction. In *Proceedings, 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, 1997.

[LPWY99] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 1999.

[LPY97] Kim G. Larsen, Paul Petterson, and Wang Yi. UPPAAL in a nutshell. *Journal on Software Tools for Technology Transfer*, 1997.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proceedings, Fourth 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, 1998.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[Min99] Marius Minea. *Partial Order for Verification of Timed Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.

[MLAH99a] Jesper Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In *Proceedings, 13th International Workshop on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[MLAH99b] Jesper Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology, Technical University of Denmark, 1999.

[MMS97] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proceedings, 1997 Conference on Security and Privacy*, pages 141–153. IEEE Computer Society Press, 1997.

[NS94] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, ATP: Theory and application. *Journal of Information and Computation*, 114(1):131–178, 1994.

[Pag96] Florence Pagani. Partial orders and verification of real-time systems. In *Proceedings, Fourth International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems.*, volume 1135 of *Lecture Notes in Computer Science*, pages 327–346. Springer-Verlag, 1996.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Proceedings, Fifth International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.

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

[PS80] Wolfgang J. Paul and Janos Simon. Decision trees and random access machines. In *Logic and Algorithmic*, volume 30 of *Monographie de L'Enseignement Mathématique*, pages 331–340. L'Enseignement Mathématique, Université de Genève, 1980.

[Rei85] Wolfgang Reisig. Petri nets. An Introduction. In *EATCS Monographs on Theoretical Compute Science*, volume 4. Springer Verlag, 1985.

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

[RR88] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261, 1988.

[SD95a] Ulrich Stern and David L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[SD95b] Ulrich Stern and David L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[SD98] Ulrich Stern and David L. Dill. Using magnetic disk instead of main memory in the Murφ verifier. In *Proceedings, Tenth International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[SS95] Oleg V. Sokolsky and Scott A. Smolka. Local model checking for real-time systems. In *Proceedings, Seventh International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[TY98] Stavros Tripakis and Sergio Yovine. Verification of the fast reservation protocol with delayed transmission using the tool Kronos. In *Proceedings, Fourth IEEE Real-Time Technology and Applications Symposium*. IEEE Computer Society Press, 1998.

[Val90] Antti Valmari. A stubborn attack on state explosion. In *Proceedings, Second International Conference on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer-Verlag, 1990.

[WH02] Farn Wang and Pao-Ann Hsiung. Efficient and user-friendly verification. *IEEE Transactions on Computers*, 51(1):61–83, 2002.

[WL93] Pierre Wolper and Dennis Leroy. Reliable hashing without collision detection. In *Proceedings, Fifth International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993.

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

[Yi91] Wang Yi. CCS + time = an interleaving model for real time systems. In *Proceedings, Eighteenth International Colloquium on Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Yov97] Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1, October 1997.

[YPD94] Wang Yi, Paul Petterson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings, Seventh International Conference on Formal Description Techniques*, pages 223–238, 1994.

Paper A:

DBM: Structures, Operations and Implementation

Johan Bengtsson.

DBM: Structures, Operations and Implementation

Johan Bengtsson

Abstract. A key issue when building a verification tool for timed systems is how to handle timing constraints arising in state-space exploration. Difference Bound Matrices (DBMs) is a well-studied technique for representing and manipulating timing constraints. The goal of this paper is to provide a cook-book and a software package for the development of verification tools using DBMs. We present all operations on DBMs needed in symbolic state-space exploration for timed automata, as well as data-structures and techniques for efficient implementation.

1 Introduction

During the last ten years, timed automata [AD90, AD94] has evolved as a common model for timed systems and one of the reasons behind its success is the availability of verification tools, such as UPPAAL [LPY97, ABB+01] and KRONOS [DOTY95, Yov97], that can verify industrial-size applications modelled as timed automata. The major problem in automatic verification is the large number states induced by the state explosion. For timed systems this problem has an even larger impact since both the number of states and the size of a single state are significantly larger than in the untimed case due to timing constraints. This makes devising data-structures for states and timing constraints one of the key challenges in developing verification tools for timed systems.

In this paper we describe a software package for representing timing constraints, arising from state space exploration of timed systems, as difference bound matrices [Dil89]. The package is based on the DBM implementation of [BL96] but it has been significantly improved using implementation experiences gained from developing the current version of UPPAAL. The paper is intended as a cook-book for developers of verification tools and the goal is to provide a basis for the implementation of state-of-the-art verification tools.

The paper is organised as follows: In Section 2 we introduce the DBM structure and its canonical form; we also describe how to find the minimal number of constraints needed to represent the same set of clock assignments as a given DBM. Section 3 lists all DBM operations needed to implement a verification

tool such as UPPAAL together with efficient algorithms for these operations. Some suggestions on how to store the structures in memory are given in Section 4 and finally Section 5 concludes the paper.

2 DBM basics

The key objects for representing timing information in symbolic state-space exploration for timed systems are a special class of constraint systems referred to as difference-bound constraint-systems or, more commonly, zones. A zone for a set of clocks $\mathcal{C}$ is a conjunction of atomic constraints of the form $x \sim n$ and $x - y \sim n$ where $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. What make zones so important is their simple structure and that the set of zones is closed with respect to the strongest postcondition of all operations needed in state-space generation.

Since zones are frequently used objects, their representation is a major issue when building a verification tool for timed automata. In this section we describe the basic concepts behind Difference Bound Matrices (DBM) [Dil89], which is one of the most effective data structures for zones.

To have a unified form for clock constraints we introduce a reference clock $\mathbf{0}$ with constant value 0. Let $\mathcal{C}_0 = \mathcal{C} \cup \{\mathbf{0}\}$. Then any zone $D \in \mathcal{B}(C)$ can be rewritten as a conjunction of constraints of the form $x - y \preceq n$ for $x, y \in \mathcal{C}_0$, $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z}$.

Naturally, if the rewritten zone has two constraints on the same pair of variables only the tightest of them is significant. Thus, a zone can be represented using at most $|\mathcal{C}_0|^2$ atomic constraints of the form $x - y \preceq n$ such that each pair of clocks $x - y$ is mentioned only once. We can then store zones using $|\mathcal{C}_0| \times |\mathcal{C}_0|$ matrices where each element in the matrix corresponds to an atomic constraint. Since each element in such a matrix represents a bound on the difference between two clocks, we call them *Difference Bound Matrices* (DBMs). In the following presentation we let D_{ij} denote element (i, j) in the DBM representing the zone D.

To compute the DBM representation for a zone D, we start by numbering all clocks in $\mathcal{C}_0$ to assign one row and one column in the matrix to each clock. The row is used for storing lower bounds on the difference between the clock and all other clocks while the column is used for upper bounds. The elements in the matrix are then computed in three steps.

- For each bound in $x_i - x_j \prec n$, in D, let $D_{ij} = (n, \prec)$.
- For each clock difference $x_i - x_j$ that is unbounded in D, let $D_{ij} = \infty$. Where ∞ is a special value denoting that no bound is present.
- Finally add the implicit constraints that all clocks are positive, *i.e.* $\mathbf{0} - x_i \leq 0$, and that the difference between a clock and itself is always 0, *i.e.* $x_i - x_i \leq 0$.

As an example, consider the zone $D = x - \mathbf{0} < 20 \wedge y - \mathbf{0} \leq 20 \wedge y - x \leq 10 \wedge x - y \leq -10 \wedge \mathbf{0} - z < 5$. To construct the matrix representation of D, we number the clocks in the order $\mathbf{0}, x, y, z$. The resulting matrix representation is shown below:

$$M(D) = \begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) & (5, <) \\ (20, <) & (0, \leq) & (-10, \leq) & \infty \\ (20, \leq) & (10, \leq) & (0, \leq) & \infty \\ \infty & \infty & \infty & (0, \leq) \end{pmatrix}$$

To manipulate DBMs efficiently we need two operations on bounds: comparison and addition. We define that $(n, \prec) < \infty$, $(n_1, \prec_1) < (n_2, \prec_2)$ if $n_1 < n_2$ and $(n, <) < (n, \leq)$. Further we define addition as $b_1 + \infty = \infty$, $(m, \leq) + (n, \leq) = (m + n, \leq)$ and $(m, <) + (n, \prec) = (m + n, <)$.

2.1 Canonical DBMs

Usually there are an infinite number of zones sharing the same solution set. However, for each family of zones with the same solution set there is a unique constraint system where no atomic constraint can be strengthened without losing solutions. We say that such a zone is *closed under entailment* or just *closed*. Since there is a unique closed zone for each solution set we use closed zones as canonical representation of entire families of zones.

To compute the closed representative of a given zone we need to derive the tightest constraint on each clock difference. To solve this problem, we use a graph-interpretation of zones. If we see zones as a weighted graphs where the clocks in $\mathcal{C}_0$ are nodes and the atomic constraints are edges, deriving bounds corresponds to adding weights along paths in the graph. Note that DBMs form adjacency matrices for this graph-interpretation.

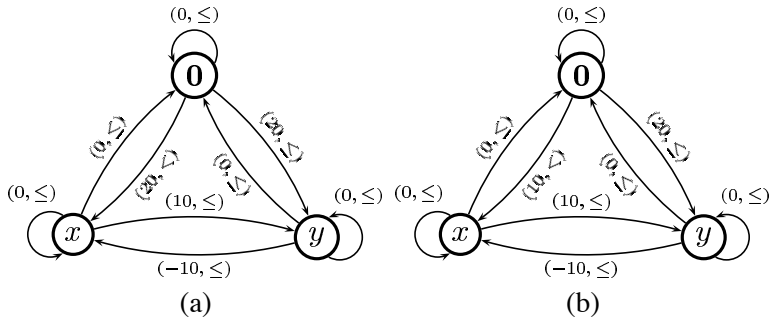


Figure 1: Graph interpretation of the example zone and its closed form

As an example, consider the zone $x-\mathbf{0} < 20 \wedge y-\mathbf{0} \leq 20 \wedge y-x \leq 10 \wedge x-y \leq -10$. By combining the atomic constraints $y - \mathbf{0} \leq 20$ and $x - y \leq -10$ we derive that $x - \mathbf{0} \leq 10$, *i.e.* the bound on $x - \mathbf{0}$ can be strengthened. Consider the graph interpretation of this zone, presented in Figure 1(a). The tighter bound on $x - \mathbf{0}$ can be derived from the graph, using the path $\mathbf{0} \rightarrow y \rightarrow x$, giving the graph in Figure 1(b). Thus, deriving the tightest constraint on a pair of clocks in a zone is equivalent to finding the shortest path between their nodes in the graph interpretation of the zone. The conclusion is that a canonical, *i.e.* closed, version of a zone can be computed using a shortest path algorithm. Many verifiers for timed automata use the Floyd-Warshall algorithm [Flo62] (Algorithm 1) to transform zones to canonical form. However, since this algorithm is quite expensive (cubic in the number of clocks), it is desirable to make all frequently used operations preserve the canonical form, *i.e.* the result of performing an operation on a canonical zone should also be canonical.

Algorithm 1 Floyds algorithm for computing shortest path

```
for k := 0 to n do
  for i := 0 to n do
    for j := 0 to n do
      D_ij := min(D_ij, D_ik + D_kj)
    end for
  end for
end for
```

2.2 Minimal Constraint Systems

A zone may contain redundant constraints. Obviously, it is desirable to remove such constraints to store only the minimal number of constraints. Consider,

for instance, the zone $x - y \leq 0 \wedge y - z \leq 0 \wedge z - x \leq 0 \wedge 2 \leq x - \mathbf{0} \leq 3$, which is in minimal form. This zone is completely defined by only five constraints. However, the closed form contains no less than 12 constraints. It is known, *e.g.* from [LLPY97], that for each zone there is a minimal constraint system with the same solution set. By computing this minimal form for all zones and storing them in memory using a sparse representation we might reduce the memory consumption for state-space exploration. This problem has been thoroughly investigated in [LLPY97, Pet99, Lar00] and this presentation is a summary of the work presented there.

The goal is to find an algorithm that computes the minimal form of a closed DBM. However, closing a DBM corresponds to computing the shortest path between all clocks. Thus, we want to find an algorithm that computes the minimal set of bounds with a given shortest path closure. For clarity, the algorithm is presented in terms of directed weighted graphs. However, the results are directly applicable to the graph interpretation of DBMs.

First we introduce some notation: we say that a cycle in a graph is a *zero cycle* if the sum of weights along the cycle is 0, and an edge $x_i \xrightarrow{n_{ij}} x_j$ is *redundant* if there is another path between x_i and x_j where the sum of weights is no larger than n_{ij}.

In graphs without zero cycles we can remove all redundant edges without affecting the shortest path closure [Pet99]. Further, if the input graph is in shortest path form (as for closed DBMs) all redundant edges can be located by considering alternative paths of length two.

As an example, consider Figure 2. The figure shows the shortest path closure for a zero-cycle free graph (a) and its minimal form (b). In the graph we find that $x_1 \xrightarrow{9} x_2$ is made redundant by the path $x_1 \xrightarrow{7} x_4 \xrightarrow{2} x_2$ and can thus be removed. Further, the edge $x_3 \xrightarrow{15} x_2$ is redundant due to $x_3 \xrightarrow{5} x_1 \xrightarrow{9} x_2$. Note that we consider edges marked as redundant when searching for new redundant edges. The reason is that we let the redundant edges represent the path making them redundant, thus allowing all redundant edges to be located using only alternative paths of length two. This procedure is repeated until no more redundant edges can be found.

This gives the $O(n^3)$ procedure for removing redundant edges presented in Algorithm 2. The algorithm can be directly applied to zero-cycle free DBMs to compute the minimal number of constraints needed to represent a given zone.

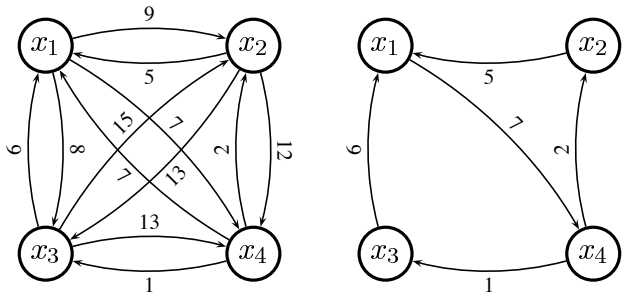


Figure 2: A zero cycle free graph and its minimal form

Algorithm 2 Reduction of Zero-Cycle Free Graph G with n nodes

```
for i := 1 to n do
  for j := 1 to n do
    for k := 1 to n do
      if G_ik + G_kj ≤ G_ij then
        Mark edge i → j as redundant
      end if
    end for
  end for
end for
Remove all edges marked as redundant.
```

However, this algorithm will not work if there are zero-cycles in the graph. The reason is that the set of redundant edges in a graph with zero-cycles is not unique. As an example, consider the graph in Figure 3(a). Applying the above reasoning on this graph would remove $x_1 \xrightarrow{3} x_3$ based on the path $x_1 \xrightarrow{-2} x_2 \xrightarrow{5} x_3$. It would also remove the edge $x_2 \xrightarrow{5} x_3$ based on the path $x_2 \xrightarrow{2} x_1 \xrightarrow{3} x_3$. But if both these edges are removed it is no longer possible to construct paths leading into x_3. In this example there is a dependence between the edges $x_1 \xrightarrow{3} x_3$ and $x_2 \xrightarrow{5} x_3$ such that only one of them can be considered redundant.

The solution to this problem is to partition the nodes according to zero-cycles and build a super-graph where each node is a partition. The graph from Figure 3(a) has two partitions, one containing x_1 and x_2 and the other containing x_3. To compute the edges in the super-graph we pick one representative for each partition and let the edges between the partitions inherit the weights from edges between the representatives. In our example, we choose x_1 and x_3 as

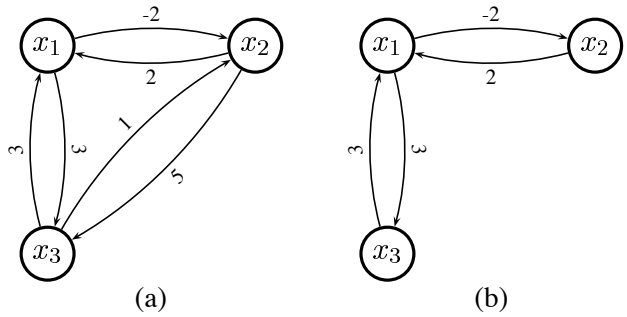


Figure 3: A graph with a zero-cycle and its minimal form

representatives for their equivalence classes. The edges in the graph are then $\{x_1, x_2\} \xrightarrow{3} \{x_3\}$ and $\{x_3\} \xrightarrow{3} \{x_1, x_2\}$. The super-graph is clearly zero-cycle free and can be reduced using Algorithm 2. This small graph can not be reduced further. The relation between the nodes within a partition is uniquely defined by the zero-cycle and all other edges may be removed. In our example all edges within each equivalence class are part of the zero-cycle and thus none of them can be removed. Finally the reduced super-graph is connected to the reduced partitions. In our example we end up with the graph in Figure 3(b). Pseudo-code for the reduction-procedure is shown in Algorithm 3.

Now we have an algorithm for computing the minimum number of edges to represent a given shortest path closure that can be used to compute the minimum number of constraints needed to represent a given zone.

3 Operations on DBMs

This section presents all operations on DBMs needed in symbolic state space exploration of timed automata, both for forwards and backwards analysis. Note that even if a verification tool only explores the state space in one direction all operations are still useful for other purposes, *e.g.* for generating diagnostic traces. The effects of the operations are shown graphically in Figure 5.

In the following section we do not distinguish between DBMs and zones, and the terms are used alternately. To simplify the presentation we assume that the clocks in $\mathcal{C}_0$ are numbered $0, \ldots, n$ and the index for $\mathbf{0}$ is 0. We assume that the input zones are consistent and in canonical form.

Algorithm 3 Reduction of negative-cycle free graph G with n nodes

```
for i := 1 to n do
  if Node_i is not in a partition then
    Eq_i = ∅
    for j := i to n do
      if G_ij + G_ji = 0 then
        Eq_i := Eq_i ∪ {Node_i}
      end if
    end for
  end if
end for
Let G' be a graph without nodes.
for each Eq_i do
  Pick one representative Node_i ∈ Eq_i
  Add Node_i to G'
  Connect Node_i to all nodes in G' using weights from G.
end for
Reduce G'
for each Eq_i do
  Add one zero cycle containing all nodes in Eq_i to G'
end for
```

The operations on DBMs can be divided into three different classes:

1. **Property-Checking:** Operations in this class include checking if a DBM is consistent, checking inclusion between zones, and checking whether a zone satisfies a given atomic constraint.

2. **Transformation:** This is the largest class containing operations for transforming zones according to guards, delay and reset.

3. **Normalisation:** They are used to normalise zones in order to obtain a finite zone-graph. In this paper we only describe one operation in this class, the so called k-normalisation. For more normalisation operations we refer to [BY01].

3.1 Checking Properties of DBMs

`consistent`(D)

The most basic operation on a DBM is to check if it is consistent, *i.e.* if the solution set is non-empty. In state-space exploration this operation is used to remove inconsistent states from exploration.

For a zone to be inconsistent there must be at least one pair of clocks where the upper bound on their difference is smaller than the lower bound. For DBMs this can be checked by searching for negative cost cycles in the graph interpretation. However, the most efficient way to implement a consistency check is to detect when an upper bound is set to lower value than the corresponding lower bound and mark the zone as inconsistent by setting D_{00} to a negative value. For a zone in canonical form this test can be performed locally. To check if a zone is inconsistent it will then be enough to check whether D_{00} is negative.

`relation`(D, D')

Another key operation in state space exploration is inclusion checking for the solution sets of two zones. For DBMs in canonical form, the condition that $D_{ij} \leq D'_{ij}$ for all clocks $i, j \in \mathcal{C}_0$ is necessary and sufficient to conclude that $D \subseteq D'$. Naturally the opposite condition applies to checking if $D' \subseteq D$. This allows for the combined inclusion check described in Algorithm 5.

`satisfied`($D, x_i - x_j \preceq m$)

Sometimes it is desirable to non-destructively check if a zone satisfies a constraint, *i.e.* to check if the zone $D \wedge x_i - x_j \preceq m$ is consistent without altering D. From the definition of the `consistent`-operation we know that a zone is consistent if it has no negative-cost cycles. Thus, checking if $D \wedge x_i - x_j \preceq m$ is non-empty can be done by checking if adding the guard to the zone would introduce a negative-cost cycle. For a DBM on canonical form this test can be performed locally by checking if $(m, \preceq) + D_{ji}$ is negative.

3.2 Transformations

`up`(D)

The `up` operation computes the strongest post condition of a zone with respect to delay, *i.e.* `up`(D) contains the time assignments that can be reached from D

by delay. Formally, this operation is defined as $\texttt{up}(D) = \{u + d \mid u \in D, d \in \mathbb{R}_+\}$.

Algorithmically, up is computed by removing the upper bounds on all individual clocks (In a DBM all elements D_{i0} are set to ∞). This is the same as saying that any time assignment in a given zone may delay an arbitrary amount of time. The property that all clocks proceed at the same speed is ensured by the fact that constraints on the differences between clocks are not altered by the operation.

This operation preserves the canonical form, *i.e.* applying up to a canonical DBM will result in a new canonical DBM. The reason is that to derive an upper bound on a single clock x we need at least an upper bound on another clock y and the relation between x and y, and all upper bounds have been removed by the operation. The up operation is also presented in Algorithm 6.

down(D)

This operation computes the weakest precondition of D with respect to delay. Formally $\texttt{down}(D) = \{u \mid u + d \in D, d \in \mathbb{R}_+\}$, *i.e.* the set of time assignments that can reach D by some delay d. Algorithmically, down is computed by setting the lower bound on all individual clocks to $(0, \leq)$. However due to constraints on clock differences this algorithm may produce non-canonical DBMs. As an example, consider the zone in Figure 4(a). When down is applied to this zone (Figure 4(b)), the lower bound on x is 1 and not 0, due to constraints on clock differences. Thus, to obtain an algorithm that produce canonical DBMs the difference constraints have to be taken into account when computing the new lower bounds.

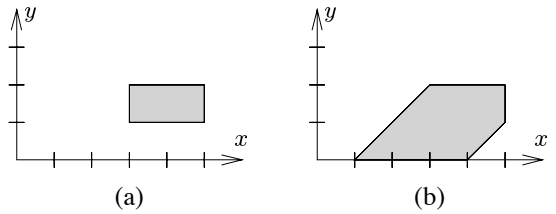


Figure 4: Applying down to a zone.

To compute the lower bound for a clock x, start by assuming that all other clocks y_i have the value 0. Then examine all difference constraints $y_i - x$ and compute a new lower bound for x under this assumption. The new bound on

$\mathbf{0} - x$ will be the minimum bound on $y_i - x$ found in the DBM. Pseudo-code for `down` is presented in Algorithm 7.

`and`$(D, x_i - y_j \preceq b)$

The most useful operation in state-space exploration is conjunction, *i.e.* adding a constraint to a zone. The basic step of the `and` operation is to check if $(b, \preceq) < D_{ij}$ and in this case set the bound D_{ij} to $(b, \preceq)$. If the bound has been altered, *i.e.* if adding the guard affected the solution set, the DBM has to be put back on canonical form. One way to do this would be to use the generic shortest path algorithm, however for this particular case it is possible to derive a specialisation of the algorithm allowing re-canonicalisation in $O(n^2)$ instead of $O(n^3)$.

The specialised algorithm take advantage of the fact that D_{ij} is the only bound that has been changed. Since the Floyd-Warshall algorithm is insensitive to how the nodes in the graph are ordered, we may decide to treat x_i and x_j last. The outer loop of Algorithm 1 will then only affect the DBM twice, for $k = i$ and $k = j$. This allows the canonicalisation algorithm to be reduced to checking, for all pairs of clocks in the DBM, if the path via either x_i or x_j is shorter than the direct connection. The pseudo code for this is presented in Algorithm 8.

`free`(D, x)

The `free` operation removes all constraints on a given clock, *i.e.* the clock may take any positive value. Formally this is expressed as $\texttt{free}(D, x) = \{u[x = d] \mid u \in D, d \in \mathbb{R}_+\}$. In state-space exploration this operation is used in combination with conjunction, to implement reset operations on clocks. It can be used in both forwards and backwards exploration, but since forwards exploration allows other more efficient implementations of reset, `free` is only used when exploring the state-space backwards.

A simple algorithm for this operation is to remove all bounds on x in the DBM and set $D_{0x} = (0, \leq)$. However, the result may not be on canonical form. To obtain an algorithm preserving the canonical form, we change how new difference constraints regarding x are derived. We note that the constraint $\mathbf{0} - x \leq 0$ can be combined with constraints of the form $y_i - \mathbf{0} \preceq m$ to compute new bounds for $y_i - x$. For instance, if $y - \mathbf{0} \leq 5$ we can derive that $y - x \leq 5$. To obtain a DBM on canonical form we derive bounds for D_{ix} based on D_{i0} instead of setting $D_{ix} = \infty$.In Algorithm 9 this is presented as pseudo code.

reset$(D, x:=m)$

In forwards exploration this operation is used to set clocks to specific values, *i.e.* $\texttt{reset}(D, x:=m) = \{u[x = m] \mid u \in D\}$. Without the requirement that output should be on canonical form, `reset` can be implemented by setting $D_{x0} = (m, \leq)$, $D_{0x} = (-m, \leq)$ and remove all other bounds on x. However, if we instead of removing the difference constraints compute new values using constraints on the other clocks, like in the implementation of `free`, we obtain an implementation that preserve the canonical form. Such an implementation is presented in Algorithm 10.

copy$(D, x:=y)$

This is another operation used in forwards state-space exploration. It copies the value of one clock to another. Formally, we define $\texttt{reset}(D, x:=y)$ as $\{u[x = u(y)] \mid u \in D\}$. Analogous with `reset`, `copy` can be implemented by assigning $D_{xy} = (0, \leq)$, $D_{yx} = (0, \leq)$, removing all other bounds on x and re-canonicalise the zone. However, a more efficient implementation is obtained by assigning $D_{xy} = (0, \leq)$, $D_{yx} = (0, \leq)$ and then copy the rest of the bounds on x from y. For pseudo code, see Algorithm 11

shift$(D, x:=x+m)$

The last reset operation is shifting a clock, *i.e.* adding or subtracting a clock with an integer value, *i.e.* $\texttt{shift}(D, x:=x+m) = \{u[x = u(x)+m] \mid u \in D\}$. The definition gives a hint on how to implement the operation. We can view the shift operation as a substitution of $x - m$ for x in the zone. With this reasoning m is added to the upper and lower bounds of x. However, since lower bounds on x are represented by constraints on $y_i - x$, m is subtracted from all those bounds. This operation is presented in pseudo-code in Algorithm 12

3.3 Normalisation Operations

norm$_k(D, [k_0, k_1, \ldots, k_n])$

To obtain a finite zone graph most verifiers use some kind of normalisation of zones. One of the key steps in most normalisation algorithms is the so called k-normalisation, *i.e.* the zone is normalised with respect to the maximum constant each clock is compared to in the automaton.

The procedure to obtain the k-normalisation of a given zone is to remove all bounds $x - y \preceq m$ such that $(m, \preceq) > (k_x, \leq)$ and to set all bounds $x - y \preceq m$ such that $(m, \preceq) < (-k_y, <)$ to $(-k_y, <)$. This corresponds to removing all upper bounds higher than the maximum constant and lowering all lower bounds higher than the maximum constant down to the maximum constant.

The k-normalisation operation will not preserve the canonical form of the DBM, and in this case the best way to put the result back on canonical form is to use Algorithm 1. Pseudo-code for k-normalisation is given in Algorithm 13

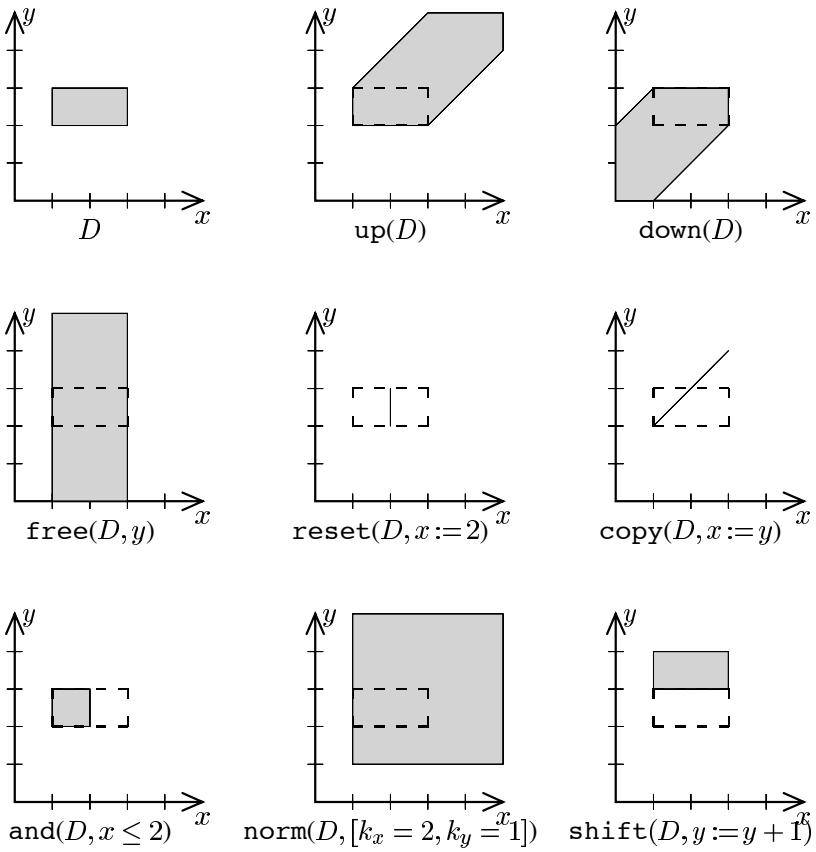


Figure 5: All DBM operations applied to the same zone

4 Zones in Memory

This section describes a number of techniques for storing zones in computer memory. The section starts by describing how to map DBM elements on machine words. It continues by discussing how to place two-dimensional DBMs in one-dimensional memory and ends by describing how to store zones using a sparse representation.

4.1 Storing DBM Elements

To store a DBM element in memory we need to keep track of the integer limit and whether it is strict or not. The range of the integer limit is typically much lower than the maximum value of a machine word and the strictness can be stored using just one bit. Thus, it is possible to store both the limit and the strictness in different parts of the same machine word. Since comparing and adding DBM elements are frequently used operations it is crucial for the performance of a DBM package that they can be efficiently implemented for the chosen encoding. Fortunately, it is possible to construct an encoding of bounds in machine words, where checking if b_1 is less than b_2 can be performed by checking if the encoded b_1 is smaller than the encoded b_2.

The encoding we propose is to use the least significant bit (LSB) of the machine word to store whether the bound is strict or not. Since strict bounds are smaller than non-strict we let a set (1) bit denote that the bound is non-strict while an unset (0) bit denote that the bound is strict. The rest of the bits in the machine word are used to store the integer bound. To encode ∞ we use the largest positive number that fit in a machine word (denoted `MAX_INT`).

For good performance we also need an efficient implementation of addition of bounds. For the proposed encoding Algorithm 4 adds two encoded bounds b_1 and b_2. The symbols $\&$ and $|$ in the algorithm are used to denote bitwise-and and bitwise-or, respectively.

Algorithm 4 Algorithm for adding encoded bounds

if $b_1 =$ `MAX_INT` or $b_2 =$ `MAX_INT` **then**
  return `MAX_INT`
else
  return $b_1 + b_2 - ((b_1 \& 1) | (b2 \& 1))$
end if

4.2 Placing DBMs in Memory

Another implementation issue is how to store two-dimensional DBMs in linear memory. In this section we present two different techniques and give a brief comparison between them. The natural way to put matrices in linear memory is to store the elements by row (or by column), *i.e.* each row of the matrix is stored consequently in memory. This layout has one big advantage, its good performance. This advantage is mainly due to the simple function for computing the location of a given element in the matrix: $loc(x, y) = x * (n + 1) + y$. This function can (on most computers) be computed in only two instructions. This is important since all accesses to DBM elements use this function. How the different DBM elements are place in memory with this layout if presented in Figure 6(a).

The second way to store a DBM in linear memory is based on a layered model where each layer consists of the bounds between a clock and the clocks with lower index in the DBM. In this representation it is cheap to implement local clocks, since all information about the local clocks are localised at the end of the DBM. The drawback with this layout is the more complicated function from DBM indices to memory locations. For this layout we have:

$$loc(x, y) = \begin{cases} y * (y + 1) + x & \text{if } x \leq y \\ x * x + y & \text{otherwise} \end{cases}$$

This adds at least two instructions (one comparison and one conditional jump) to the transformation. This may not seem such a huge overhead, but it is clearly noticeable. The cache performance is also worse when using this layout than when storing the DBMs row-wise. This layout is illustrated in Figure 6(b).

The conclusion is that unless the tool under construction supports adding and removing clocks dynamically the row-wise mapping should be used. On the other hand, if the tool supports local clocks the layered mapping may be preferable since no reordering of the DBM is needed when entering or leaving a clock scope.

4.3 Storing Sparse Zones

In most verification tools, the majority of the zones are kept in the set of states already visited during verification. They are used as a reference to ensure termination by preventing states from being explored more than once. For the states

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \quad \begin{pmatrix} 0 & 2 & 6 & 12 \\ 1 & 3 & 7 & 13 \\ 4 & 5 & 8 & 14 \\ 9 & 10 & 11 & 15 \end{pmatrix}$$

(a) Row wise (b) Layered

Figure 6: Different layouts of DBMs in memory

in this set we may benefit from storing only the minimal number of constraints using a sparse representation.

A straight forward implementation is to store a sparse zone as a vector of constraints of the form $\langle x, y, b \rangle$. We may save additional memory by omitting implicit constraints, such as $\mathbf{0} - x \leq 0$. A downside with using sparse zones is that each constraint require twice the amount of memory needed for a constraint in a full DBM, since the sparse representation must store clock indices explicitly. Thus, unless half of the constraints in a DBM are redundant we do not gain from using sparse zones.

A good feature of the sparse representation is that checking whether a zone D_f represented as a full DBM is included in a sparse zone D_s may be implemented without computing the full DBM for D_s. It suffices to check for all constraints in D_s that the corresponding bound in D_f is tighter. However, to check if $D_s \subseteq D_f$ we have to compute the full DBM for D_s.

5 Conclusions

In this paper we have reviewed Difference Bound Matrices as a data-structure for timing constraints. We have presented the basics of the data-structure and all operations needed in symbolic state-space exploration of timed automata. We have also discussed how to save space by computing the minimum number of constraints defining a given zone and saving them using a sparse representation.

References

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In *Modelling and Verification*

of Parallel Processes, volume 2067 of *Lecture Notes in Computer Science*, pages 100–125. Springer-Verlag, 2001.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings, Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2):183–235, 1994.

[BL96] Johan Bengtsson and Fredrik Larsson. UPPAAL a tool for automatic verification of real-time systems. Technical Report 96/67, Department of Computer Systems, Uppsala University, 1996.

[BY01] Johan Bentsson and Wang Yi. Reachability analysis of timed automata containing constraints on clock differences. Submitted for publication., 2001.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings, Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[DOTY95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool Kronos. In *Proceedings, Hybrid Systems III: Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

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

[Lar00] Fredrik Larsson. Efficient implementation of model-checkers for networks of timed automata. Licentiate Thesis 2000-003, Department of Information Technology, Uppsala University, 2000.

[LLPY97] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structure and state space reduction. In *Proceedings, 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, 1997.

[LPY97] Kim G. Larsen, Paul Petterson, and Wang Yi. UPPAAL in a nutshell. *Journal on Software Tools for Technology Transfer*, 1997.

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

[Yov97] Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1, October 1997.

A Pseudo-Code

Algorithm 5 `relation`(D, D')

$f_{D \subseteq D'} := \mathbf{t}$
$f_{D \supseteq D'} := \mathbf{t}$
for $i := 0$ **to** n **do**
  for $j := 0$ **to** n **do**
    $f_{D \subseteq D'} := f_{D \subseteq D'} \wedge (D_{ij} \leq D'_{ij})$
    $f_{D \supseteq D'} := f_{D \supseteq D'} \wedge (D_{ij} \geq D'_{ij})$
  end for
end for
return $\langle f_{D \subseteq D'}, f_{D \supseteq D'} \rangle$

Algorithm 6 `up`(D)

for $i := 1$ **to** n **do**
  $D_{i0} := \infty$
end for

Algorithm 7 `down`(D)

for $i := 1$ **to** n **do**
  $D_{0i} = (0, \leq)$
  for $j := 1$ **to** n **do**
    if $D_{ji} < D_{0i}$ **then**
      $D_{0i} = Dji$
    end if
  end for
end for

Algorithm 8 `and`(D, g)

if $D_{yx} + (m, \preceq) < 0$ **then**
  $D_{00} = (-1, \leq)$
else if $(m, \preceq) < D_{xy}$ **then**
  $D_{xy} = (m, \preceq)$
  for $i := 0$ **to** n **do**
    for $j := 0$ to n **do**
      if $D_{ix} + D_{xj} < D_{ij}$ **then**
        $D_{ij} = D_{ix} + D_{xj}$
      end if
      if $D_{iy} + D_{yj} < D_{ij}$ **then**
        $D_{ij} = D_{iy} + D_{yj}$
      end if
    end for
  end for
end if

Algorithm 9 `free`(D, x)

for $i := 0$ **to** n **do**
  if $i \neq x$ **then**
    $D_{xi} = \infty$
    $D_{ix} = D_{i0}$
  end if
end for

Algorithm 10 `reset`$(D, x := m$

for $i := 0$ **to** n **do**
  $D_{xi} := (m, \leq) + D_{0i}$
  $D_{ix} := D_{i0} + (-m, \leq)$
end for

Algorithm 11 `copy`$(D, x := y)$

for $i := 0$ **to** n **do**
  if $i \neq x$ **then**
    $D_{xi} := D_{yi}$
    $D_{ix} := D_{iy}$
  end if
end for
$D_{xy} := (0, \leq)$
$D_{yx} := (0, \leq)$

Algorithm 12 `shift`$(D, x := x + m)$

for $i := 0$ **to** n **do**
  if $i \neq x$ **then**
    $D_{xi} := D_{xi} + (m, \leq)$
    $D_{ix} := D_{ix} + (-m, \leq)$
  end if
end for
$D_{x0} := max(D_{x0}, (0, \leq))$
$D_{0x} := min(D_{0x}, (0, \leq))$

Algorithm 13 `norm`$_k(D, [k_0, k_1, \ldots, k_n])$

for $i := 0$ to n **do**
  for $j := 0$ to n **do**
    if $D_{ij} \neq \infty$ and $D_{ij} > (k_i, \leq)$ **then**
      $D_{ij} = \infty$
    else if $D_{ij} \neq \infty$ and $D_{ij} < (-k_j, <)$ **then**
      $D_{ij} = (-k_j, <)$
    end if
  end for
end for
`close`(D)

Paper B:

Reachability Analysis of Timed Automata Containing Constraints on Clock Differences

Johan Bengtsson and Wang Yi.

Reachability Analysis of Timed Automata Containing Constraints on Clock Differences

Johan Bengtsson and *Wang Yi*

Department of Computer Systems, Uppsala University, Sweden.
E-mail: {johanb,yi}@docs.uu.se

Abstract. The key step to guarantee termination of reachability analysis for timed automata is the normalisation algorithms for clock constraints (i.e. symbolic states represented as DBM's). It transforms DBM's which may contain arbitrary constants into their equivalent according to maximal constants appearing in clock constraints of an automaton. A restriction of the existing algorithms is that clock constraints in timed automata must be in the form [AD94] of $x \sim n$ where x is a clock, $\sim \in \{\leq, <, =, >, \geq\}$, and n is a natural number. It was discovered recently that the existing tools were either providing incorrect answers or not terminating when they are used to verify automata containing difference constraints of the form: $x - y \sim n$ (that are indeed needed in many applications e.g. in solving scheduling problems). In this paper, we present new normalisation algorithms that transforms DBM's according to not only maximal constants of clocks but also difference constraints appearing in an automata. To our knowledge, they are the first published normalisation algorithms for timed automata containing difference constraints. The algorithms have been implemented in UPPAAL, demonstrating that little extra overhead is needed to deal with difference constraints.

1 Introduction

Following the work of Alur and Dill on timed automata [AD94], a number of model checkers have been developed for modelling and verification of timed systems with timed automata as the core of their input languages [DOTY95, Yov97, LPY97, ABB+01] based on reachability analysis. The foundation for decidability of reachability problems for timed automata is Alur and Dill's region technique, by which the infinite state space of a timed automaton due to the density of time, may be partitioned into finitely many equivalence classes i.e. according to *regions* in such a way that states within each class will always evolve to states within the same classes. However, analysis based on the region technique is practically infeasible due to the large number of equivalence

classes [LPY95], which is highly exponential in the number of clocks and their maximal constants.

One of the major advances in the area after the pioneering work of Alur and Dill is the symbolic technique [Dil89, YL93, HNSY94, YPD94, LPY95], which transforms the reachability problem to that of solving simple constraints. It adopts the idea from symbolic model checking for untimed systems, which uses logical formulas to represent set of states and operations on formulas to represent state transitions. It is proven that the infinite state-space of timed automata can be finitely partitioned into symbolic states which are represented and manipulated using a class of linear constraints known as zones and represented as *Difference Bound Matrices* (DBM) [Bel57, Dil89]. The reachability relation over symbolic states can be represented and computed by a few efficient operations on zones. From now on, we shall not distinguish the terms: constraint, zone and DBM.

The technique can be simply formulated in an abstract reachability algorithm[1] as shown in Algorithm 1. The algorithm is to check whether a timed automaton may reach a final location l_f. It explores the state space of the automaton in terms of *symbolic states* in the form (l, D) where l is a location and D is a zone (represented as a DBM).

Algorithm 1 Symbolic reachability analysis

```
PASSED = ∅, WAIT = {⟨l0, D0⟩}
while WAIT ≠ ∅ do
  take ⟨l, D⟩ from WAIT
  if l = lf ∧ D ∩ Df ≠ ∅ then return "YES"
  if D ⊈ D' for all ⟨l, D'⟩ ∈ PASSED then
    add ⟨l, D⟩ to PASSED
    for all ⟨l', D'⟩ such that ⟨l, D⟩ ⇝ ⟨l', D'⟩ do
      add ⟨l', D'⟩ to WAIT
    end for
  end if
end while
return "NO"
```

Having a closer look at the algorithm, one will realize that termination is not guaranteed unless the number of constraints generated is finite or the constraints

[1] Several verification tools for timed systems (e.g. UPPAAL [BLL+96]) have been implemented based on this algorithm.

form a well quasi-ordering with respects to set-inclusion (over solution sets for clock constraints) [Hig52]. There have been several normalisation algorithms for clock constraints represented as DBMs (e.g. [Rok93, Pet99]) that are the key step to guarantee termination for the existing tools. They transform DBMs which may contain arbitrary constants into their equivalent with maximal constants appearing in clock constraints. The transformation respects region equivalence and therefore the number of DBMs explored is finite. However a restriction of the existing normalisation algorithms is that clock constraints in the syntax of timed automata must be in the form [AD94] of $x \sim n$ where x is a clock variable, $\sim$ is a relational operator and n is a natural number. It was discovered recently that the existing tools were either providing incorrect answers or not terminating when they are used to verify automata containing difference constraints of the form: $x - y \sim n$ (that are indeed needed in many applications e.g. in solving scheduling problems).

A normalisation algorithm based on region equivalence treats clock values above a certain constant as equivalent. This is correct only when no guard of the form: $x - y \sim n$ is allowed in an automaton. Otherwise the normalisation operation may enlarge a zone so that the guard (a difference constraint) labelled on a transition is made true and thus incorrectly enables the transition. For automata containing difference constraints as guards, we need a finer partitioning, since the difference constraints introduce diagonal lines that split the entire clock space, even above the maximum constants for clocks. The partitioning and related normalisation operation based on region construction is too crude.

We demonstrate this by an example. Consider the example shown in Figure 1. The final location of the automaton is not reachable according to the semantics. This is because in location s_2, the clock zone is ($x - y > 2$ and $x > 2$) where the guard is ($x - z < 1$ and $z - y < 1$) which is equivalent to ($x - z < 1$ and $z - y < 1$ and $x - y < 2$) can never be true and thus disables the last transition. However, because the maximal constants for clock x is 1 (and 2 for y), the zone in location s_2: ($x - y > 2$ and $x > 2$) will be normalised to ($x - y > 1$ and $x > 1$) by the maximal constant 1 for x, which enables the guard ($x - z < 1$ and $z - y < 1$) leading to the final location. Thus the symbolic reachability analysis based on a standard normalisation algorithm would incorrectly conclude that the last location is reachable.

In [BDGP98], it has been proved that a timed automaton with constraints on clock differences can be transformed to an equivalent automaton without constraints on differences. However, this approach is impractical to be implemented in the existing tools that support debugging of models because the transfor-

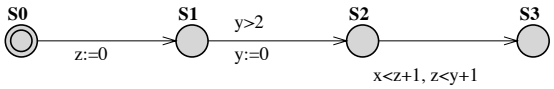


Figure 1: Bug example

mation will change the syntax of the original automaton. In this paper, we present two normalisation algorithms that allow not only clock comparison with naturals but also comparison between clocks i.e. constraints on clock differences. The algorithms transform DBMs according to not only the maximal constants of clocks but also difference constraints appearing in an automaton. To our knowledge, they are the first published normalisation algorithms for timed automata containing difference constraints. The algorithms have been implemented in UPPAAL. Our experiments demonstrate that almost no extra overhead is added to deal with difference constraints.

The paper is organised as follows: Section 2 reviews timed automata and reachability analysis. Section 3 introduces the problem in normalising symbolic states for timed automata with constraints over clock differences. Section 4 presents two new normalisation algorithms. Section 5 concludes the paper.

2 Preliminaries

In this section we briefly review the notation for timed automata and its semantics. More extensive descriptions can be found in *e.g.* [AD94, Yov98, Pet99].

2.1 Timed Automata Model

Let Σ be a finite set of labels, ranged over by a, b etc. A timed automaton is a finite state automaton over alphabet Σ extended with a set of real valued clocks, to model time dependent behaviour. Let $\mathcal{C}$ denote a set of clocks, ranged over by x, y, z, and define $\mathcal{B}(\mathcal{C})$ as the set of conjunctions of atomic constraints of the form $x \sim n$ and $x - y \sim n$ for $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. We use $\mathcal{B}_{\text{df}}(\mathcal{C})$ for the subset of $\mathcal{B}(\mathcal{C})$ where all atomic constraints are of the form $x \sim n$ and let g range over this set.

Definition 1 (Timed Automaton) *A timed automaton A is a tuple $\langle N, l_0, \rightarrow, I\rangle$ where N is a set of control nodes, $l_0 \in N$ is the initial node, $\rightarrow \in N \times \mathcal{B}_{\mathrm{df}}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}\times\mathbb{N}} \times N$ is the set of edges and $I : N \longrightarrow \mathcal{B}(\mathcal{C})$ assign invariants to locations. As a simplification we will use $l \xrightarrow{g,a,r} l'$ to denote $\langle l, g, a, r, l'\rangle \in \rightarrow$.*

The clocks values are formally represented as functions, called clock assignments, mapping $\mathcal{C}$ to the non-negative reals $\mathbb{R}_+$. We let u, v denote such functions, and use $u \in g$ to denote that the clock assignment u satisfy the formula g. For $d \in \mathbb{R}_+$ we use $u+d$ for the clock assignment that map all clocks x in $\mathcal{C}$ to the value $u(x)+d$, and for $r : \mathcal{C} \to \mathbb{N}$ we let $[r]u$ denote the clock assignment that map all clocks, x, in the domain of r to $r(x)$ and agree with u for the other clocks in $\mathcal{C}$

The semantics of a timed automaton is a timed transition-system where the states are pairs $\langle l, u\rangle$, with two types of transitions, corresponding to delay transitions and discrete action transitions respectively:

- $\langle l, u\rangle \xrightarrow{\epsilon(t)} \langle l, u+t\rangle$ if $u \in I(l)$ and $(u+t) \in I(l)$
- $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle$ if $l \xrightarrow{g,a,r} l', u \in g, u' = [r]u$ and $u' \in I(l')$

It is easy to see that the state space for such a transition system is infinite and thus not adequate for algorithmic verification. However, efficient algorithms may be obtained using a *symbolic semantics* based on *symbolic states* of the form $\langle l, D\rangle$, where $D \in \mathcal{B}(\mathcal{C})$ [HNSY92, YPD94]. The symbolic counterpart of the transitions are given by:

- $\langle l, D\rangle \rightsquigarrow \langle l, D^{\uparrow} \wedge I(l)\rangle$
- $\langle l, D\rangle \rightsquigarrow \langle l', r(D \wedge g) \wedge I(l')\rangle$ if $l \xrightarrow{g,a,r} l'$

where $D^{\uparrow} = \{u + d \mid u \in D \wedge d \in \mathbb{R}_+\}$ and $r(D) = \{[r]u \mid u \in D\}$. It can be shown that the set of constraint systems is closed under these operations, in the sense that the result of the operations can be expressed by elements of $\mathcal{B}(\mathcal{C})$.

Moreover the symbolic semantics correspond closely to the standard semantics in the sense that if $\langle l, D\rangle \rightsquigarrow \langle l', D'\rangle$ then, for all $u' \in D'$ there is $u \in D$ such that $\langle l, u\rangle \rightarrow \langle l', u'\rangle$.

2.2 Reachability Analysis

Given a timed automaton with symbolic initial-state $\langle l_0, D_0\rangle$ and a symbolic state $\langle l, D\rangle$, $\langle l, D\rangle$ is said to be *reachable* if $\langle l_0, D_0\rangle \rightsquigarrow^* \langle l, D_n\rangle$ and $D \cap D_n \neq$

$\emptyset$ for some D_n. This problem may be solved using a standard reachability algorithm for graphs. However the unbounded clock values may render an infinite zone graph and thus might the reachability algorithm not terminate. The solution to this problem is to obtain a finite symbolic semantics by normalising the states with respect to the maximum constant each clock is compared to in the automaton. For details we refer the reader to [Pet99, Rok93] but the main fact and the intuition behind it is described here. In order to do this we first have to introduce the notion of closed constraint systems. We say that a constraint system $D \in \mathcal{B}(C)$ is *closed under entailment* or just closed, for short, if no constraint in D can be strengthened without reducing the solution set.

Proposition 1 *For each constraint system $D \in \mathcal{B}(C)$ there is a unique constraint system $D' \in \mathcal{B}(C)$ such that D and D' have exactly the same solution set and D' is closed under entailment.*

From this proposition we conclude that a closed constraint system can be used as a canonical representation of a zone.

Given a zone D and a set of maximal constants $k = \{k_x, k_y, \ldots\}$ where k_x denotes the maximal constant for clock x, the normalisation of D, denoted $\mathsf{norm}_k(D)$, is computed from the closed representation of D by

1. Removing all constraints of the form $x < m$, $x \leq m$, $x - y < m$ and $x - y \leq m$ where $m > k_x$,

2. Replacing all constraints of the form $x > m$, $x \geq m$, $x - y > m$ and $x - y \geq m$ where $m > k_x$ with $x > k_x$ and $x - y > k_x$ respectively.

This can then be used to define a notion of normalised symbolic transitions ($\leadsto_k$) by modifying the transitions of the standard symbolic semantics to preserve normalisation. The discrete action transition already preserves this so there is no need to modify it, but the delay transition should be modified to $\langle l, D\rangle \leadsto_k \langle l, \mathsf{norm}_k(D^{\uparrow} \wedge I(l))\rangle$.

Proposition 2 *Assume a timed automaton A with initial-state $\langle l_0, D_0\rangle$ and let k be the set of maximal constants used to compare with respective clocks in A. Then $\langle l, D\rangle$ is reachable from $\langle l_0, D_0\rangle$ if and only if there is a sequence of normalised transitions $\langle l_0, D'_0\rangle \leadsto_k^* \langle l, D'_n\rangle$ such that $D \cap D'_n \neq \emptyset$.*

Using this we get a finite symbolic state-space where we can apply a standard reachability algorithm for graphs, such as the one in Algorithm 1 with the symbolic transition relation $\leadsto$ being replaced with the normalised version $\leadsto_k$.

3 Constraints on Clock Differences and Normalisation

Timed automata can easily be extended to allow guards where the difference between two clocks is compared, *i.e.* allowing the guards to be taken from the full set $\mathcal{B}(\mathcal{C})$, not only from $\mathcal{B}_{\text{df}}(\mathcal{C})$. However, this extension do not give more expressive power; it has been shown (*e.g.* in [BDGP98]) that a timed automaton with difference constraints can be transformed into an equivalent automaton without difference constraints. From now on, we will call the timed automata described in Section 2 for *diagonal free timed automata* and widen the term timed automata to include also automata with difference constraints.

We note that for diagonal-free timed automata the normalisation algorithm described earlier is based on the so called region equivalence.

Definition 2 (Region Equivalence) *For a clock $x \in \mathcal{C}$, let k_x be a constant (the ceiling of clock x). For a real number t, let $\{t\}$ denote the fractional part of t, and $\lfloor t \rfloor$ denote its integer part. Two clock assignments u, v are* region-equivalent, *denoted $u \stackrel{\cdot}{\sim} v$, iff*

1. *for each clock x, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or $(u(x) > k_x$ and $v(x) > k_x)$, and*
2. *for all clocks x, y if $u(x) \leq k_x$ and $u(y) \leq k_y$ then*
   (a) $\{u(x)\} = 0$ *iff* $\{v(x)\} = 0$ *and*
   (b) $\{u(x)\} \leq \{u(y)\}$ *iff* $\{v(x)\} \leq \{v(y)\}$

For the extended version we need a finer partitioning, since the difference constraints in the guards introduce diagonal lines that split the entire clock space, even above the maximum constants for the clocks. The partitioning used for diagonal-free automata, and the connected normalisation operation norm_k is too crude. We demonstrate this by an example: Consider the timed automaton in Figure 2. The automaton has two clocks, x (with maximum constant 0) and y (with maximum constant 1). The only difference constraint in the example is $x \leq y$. This will be used as a running example throughout the paper since the small number of clocks make it possible to show the zones using graphs.

The problem is easily detected by comparing the clock zones of the unnormalised symbolic semantics (presented in Figure 3) with the corresponding clock zones in the normalised symbolic semantics (presented in Figure 4). To highlight the cause of the problem even further we show borders between equivalence classes in the clock space ($y = 1$ from the maximum constant for y and

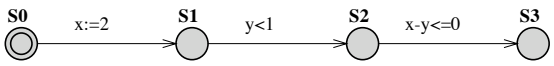


Figure 2: Example automaton with difference guards

$x = y$ from the difference constraint) as dashed lines. The initial state of the automaton is $\langle S_0, x = y\rangle$ After performing the first action step it will reach the state $\langle S_1, x = 2\rangle$. Idling in location S_1 leads to the state $\langle S_1, x \geq 2 \wedge x - y \leq 2\rangle$. Since this is above the maximum constant of x the corresponding state in the normalised semantics will be $\langle S_1, x > 0\rangle$. Thus, the normalised state will, erroneously, contain time assignments from the triangle $y \leq 1 \wedge x \leq y$ even though no state in the real semantics has a time assignment in this equivalence class. Performing the next action step will take the automaton to the state $\langle S_2, y < 1 \wedge x \geq 2 \wedge x - y \leq 2\rangle$. The next step in the normalised semantics is the state $\langle S_2, y < 1 \wedge x > 0\rangle$. The automaton may now idle to the state $\langle S_2, x > 2 \wedge 1 < x - y \leq 2\rangle$. The corresponding state in the normalised semantics is $\langle S_2, y - x < 1\rangle$. In the real semantics no further step is possible since the guard $x - y \leq 0$ is not satisfied while the normalised semantics may proceed to location S_3 since the time assignments satisfying $0 \leq y - x < 1$ are present in the normalised states. The conclusion is that the normalisation procedure have to be adapted to handle timed automata with difference constraints.

4 New Normalisation Algorithms

In this section we will present how to normalise the symbolic states for a timed automaton with difference constraints together with two slightly different algorithms to do this. The first algorithm is simpler, and may still yield an infinite zone graph. However, the zones form a well quasi-order with respect to inclusion checking and thus the reachability algorithm is guaranteed to terminate (see *e.g.* [AČJT00]). The second algorithm is more complex and does not suffer from this problem, but it may require splitting of symbolic states.

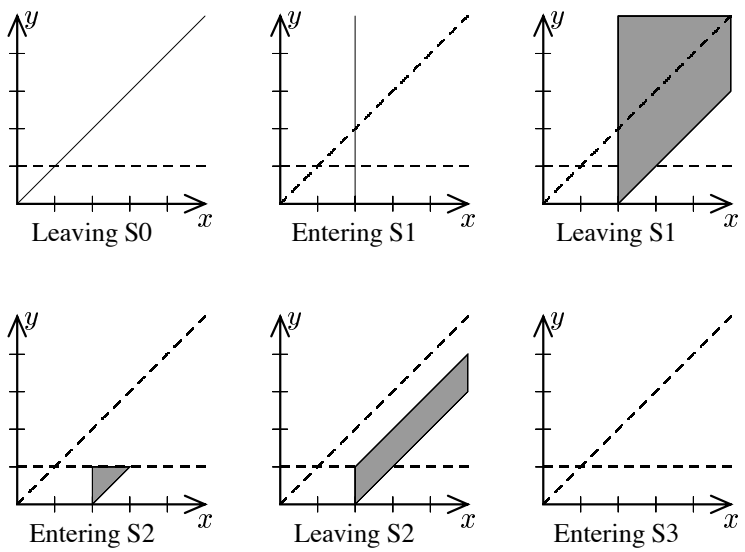


Figure 3: Zones for the example automaton, without normalisation

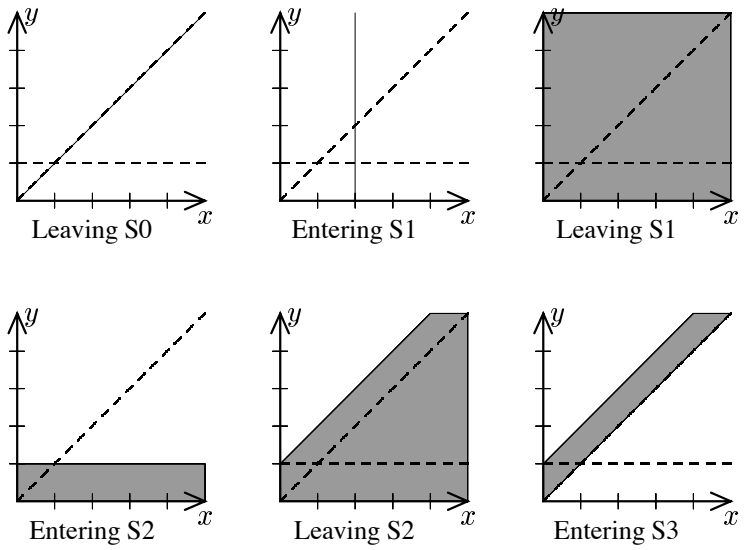


Figure 4: Zones for the example automaton, with incorrect normalisation

4.1 Region Equivalence Refined by Difference Constraints

The key issue for the extended normalisation algorithms is to honour the equivalence classes that are introduced by difference constraints in the guards. We note that difference constraints in the guards may introduce equivalence classes in the clock space that reach beyond any maximum constant. Thus we need to refine the region equivalence from Definition 2 to take the difference constraints into account.

Definition 3 (Refined Region Equivalence) *Let G be a finite set of constraints of the form $x - y \sim n$ for $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. Two clock assignments u, v are equivalent, $u \dot{\approx} v$ iff $u \dot{\sim} v$ and $\forall g \in G : u \in g \Leftrightarrow v \in g$*

We note that since the number of regions defined by $\dot{\sim}$ is finite and there are only finitely many constraints in G this refined region equivalence will define finitely many regions.

4.2 The Core of Normalisation

We can now use the refined region equivalence from Definition 3 to obtain the core of a normalisation algorithm. From the region equivalence we get the need to ensure that if a difference constraint is not satisfied by any point in the unnormalised zone, D, then it should not be satisfied by any point in the normalised zone, $\mathsf{norm}_d(D)$, and if all points in D satisfy a difference constraint then so should all points in $\mathsf{norm}_d(D)$. This leads to a core normalisation algorithm consisting of three stages:

1. Collect all difference constraints from A that are not satisfied by any point in the zone and the negation of all difference constraints that are satisfied by all points in the zone.
2. Perform normalisation with respect to the maximum constants of A.
3. Apply the negation of all the collected constraints to the normalised zone to make sure that none of the collected constraints are satisfied after normalisation.

In Algorithm 2 this core normalisation is given as pseudo code. The set G_{d} referred to in the algorithm is the set of difference constraints in A and the operation norm_k refers to normalisation with respect to the maximum constants of A.

Algorithm 2 Core normalisation algorithm

$G_{\text{unsat}} := \emptyset$
for all g such that $g \in G_{\text{d}}$ or $\neg g \in G_{\text{d}}$ **do**
  if $D \wedge g = \emptyset$ **then**
    $G_{\text{unsat}} := G_{\text{unsat}} \cup \{g\}$
  end if
end for
$D := \mathsf{norm}_k(D)$
for all $g \in G_{\text{unsat}}$ **do**
  $D := D \wedge \neg g$
end for
return D

However, there are cases where this algorithm is incorrect with respect to the equivalence classes. For some cases when a difference constraint split the zone to be normalised, the perfect normalisation may not be represented using a single zone. One instance of such a zone is the "leaving S1" zone in our running example. If this zone was extended to contain the set of equivalence classes, from which it contains points, it would be the whole clock space except for the small triangle defined by $y \leq 1 \wedge x \leq y$. The main difference between the two proposed algorithms is how this is resolved.

4.3 Algorithm: Normalisation without Zone Splitting

The first normalisation algorithm is based on the observation that due to the geometry of the equivalence classes, the problem will only occur when the zone is stretched downwards, *i.e.* when the lower bound of the zone is lowered down to the maximum constant. Thus, if the zones are not stretched downwards when normalising the problem can be avoided. This gives an algorithm that is similar to core normalisation presented in Algorithm 2, with the only exception that the step of replacing constraints of the form $x > m, x \geq m, x - y > m$ and $x - y \geq m$ for $m > k_x$ is removed from the norm_k-operation.

The problem with this solution is that the normalised zone graph is no longer finite, since delay may cause the lower bound of the zone to increase unboundedly. However, the normalised zones are well quasi-ordered with respect to zone inclusion and thus the termination of the reachability algorithm can be guaranteed [AČJT00].

Using this normalisation procedure for our example automaton will give the zones presented in Figure 5. We note from the figure that now the normalised zones does not contain time assignments from equivalence classes not in the corresponding unnormalised zone. Further we see that the zones are not expanded to fill all the equivalence classes where it has points.

4.4 Algorithm: Normalisation with Zone Splitting

The second normalisation procedure is based on the observation that the problem only occurs when a difference constraint divides the unnormalised zone, *i.e.* some of the time assignments in the zone satisfy the difference constraint and some do not. Thus, if all such zones are split along dividing difference constraints, *e.g.* using Algorithm 3, before normalisation this problem can be avoided.

Algorithm 3 Zone splitting algorithm

$Q := \{D\}, Q' := \emptyset$
for all $g \in G_{\mathrm{d}}$ **do**
  for all $D' \in Q$ **do**
    if $D' \wedge g$ and $D' \wedge \neg g$ **then**
      $Q' := Q' \cup \{D' \wedge g, D' \wedge \neg g\}$
    else
      $Q' := Q' \cup \{D'\}$
    end if
  end for
  $Q := Q', Q' := \emptyset$
end for
return Q

The full normalisation procedure is presented in Algorithm 4. The splitting, denoted by split in the description, is used as a preprocessing step and then the basic normalisation algorithm, norm_d, is applied to all the resulting zones. We use norm_s to denote this normalisation operation and we use this operation to define a normalised symbolic transition relation.

Definition 4 *Let A be a timed automaton with the symbolic semantics $\rightsquigarrow$. The s-normalised version of $\rightsquigarrow$ ($\rightsquigarrow_s$) for A is defined by: whenever $\langle l, D\rangle \rightsquigarrow \langle l', D''\rangle$ then $\langle l, D\rangle \rightsquigarrow_s \langle l', D'\rangle$ for all $D' \in \mathsf{norm}_s(D'')$.*

Algorithm 4 Splitting normalisation algorithm

$Q := \emptyset$
for all $D' \in \mathsf{split}(D)$ **do**
  $Q := Q \cup \{\mathsf{norm}_d(D')\}$
end for
return Q

To demonstrate the normalisation procedure we apply it to our running example. In Figure 6 we see the zones from the s-normalised transition relation. The two first states are not affected by normalisation so they are the same as in the unnormalised semantics. The zone for the next state, the "Leaving S_1" state, have both time assignments satisfying the constraint $x \leq y$ and time assignment satisfying $x > y$. Thus we have to split the state before normalisation. The two resulting states are $\langle S_1, x \geq 2 \wedge y \geq x\rangle$ and $\langle S_2, x \geq 2 \wedge 0 < x - y \leq 2\rangle$. When normalising these states we get $\langle S_1, x > 0 \wedge y > 1 \wedge y \geq x\rangle$ and $\langle S_2,$ $x > 0 \wedge x > y\rangle$ respectively. After the next step only one of the states remain, the other did not satisfy the guard on the transition. The state is now $\langle S_2, x > 0 \wedge y < 1 \wedge x > y\rangle$. After idling in S_2 the state is $\langle S_2, x > 0 \wedge x > y\rangle$. This state does not satisfy the constraint $x \leq y$, and thus location S_3 is not reachable.

Before proving the correctness of the s-normalised transition relation, we need to establish some properties of the norm_s operator.

Lemma 1 *Assume a timed automaton A, with associated norm_s operator. For any zone D the following holds.*

(1) For all constraints g mentioned in A, $\mathsf{norm}_s(D \wedge g) = \{D' \wedge g \mid D' \in \mathsf{norm}_s(D)\}$

(2) $\mathsf{norm}_s(D^{\uparrow}) = \{(D')^{\uparrow} \mid D' \in \mathsf{norm}_s(D)\}$

(3) $D' \in \mathsf{norm}_s(D) \Rightarrow \mathsf{norm}_s(r(D')) \subseteq \mathsf{norm}_s(r(D))$

Proof: (sketch) These properties are proved by reasoning about how the $\wedge, \uparrow$ and r operations modify the zones with respect to the two types of constraints that effect normalisation, *i.e.* non-difference constraints with bounds above the maximum constants and difference constraints.

(1) Adding a guard of the form $x_i - x_j \sim n$ will cut the zone along one of the normalisation split lines. If this is done before normalisation the result will be that normalisation produce a subset of the zones that it

would originally have produced. If the guard is added after normalisation a number of entire zones from the normalisation will be removed giving the same final result.

Adding a guard of the form $x_i \sim n$ will cut away a part of the zone that is not affected by the normalisation since, by definition, $n \leq k_i$

(2) Difference constraints are not effected at all by the $\uparrow$ operation. Further $\uparrow$ do not introduce any new non-difference constraints.

(3) $r(D)$ operations are projections of a D on a hyperplane defined by r. This projection has the property that points that were added by normalisation are mapped to other that would be added by renormalisation.

Finally we prove that the s-normalised transition relation is correct.

Theorem 1 *Let A be a timed automaton and for each clock $x_i \in \mathcal{C}$ let k_i be the largest number x_i is compared to in A.*

- *(Soundness) whenever $\langle l_0, \{u_0\}\rangle \leadsto_s^* \langle l_f, D_f\rangle$ then $\langle l_0, u_0\rangle \rightarrow^* \langle l_f, u_f\rangle$ for all $u_f \in D_f$ such that $u_f(x_i) \leq k_i$*
- *(Completeness) whenever $\langle l_0, u_0\rangle \rightarrow^* \langle l_f, u_f\rangle$ then $\langle l_0, \{u_0\}\rangle \leadsto_s^* \langle l_f, D_f\rangle$ for some D_f such that $u_f \in D_f$*

Proof: Both soundness and completeness are proven by induction on the length of the transition sequences.

(Soundness) As induction hypothesis, assume $\langle l_0, \{u_0\}\rangle \leadsto_s^n \langle l_n, D_n^s\rangle \Rightarrow \exists D_n$ such that $\langle l_0, \{u_0\}\rangle \leadsto^n \langle l_n, D_n\rangle$ and $D_n^s \in \mathsf{norm}_s(D_n)$. Further assume $\langle l_n, D_n^s\rangle \leadsto_s \langle l_{n+1}, D_{n+1}^s\rangle$. We now need to prove that $\exists D_n$ such that $D_n^s \in \mathsf{norm}_s(D_n)$, $\langle l_n, D_n\rangle \leadsto \langle l_{n+1}, D_{n+1}\rangle$ and $D_{n+1}^s \in \mathsf{norm}_s(D_{n+1})$. We have two cases: delay transitions and action transitions.

- (Delay) By the assumption $\langle l_n, D_n^s\rangle \leadsto_s \langle l_n, D_{n+1}^s\rangle$ by delay, and the definition of $\leadsto_s$ we get $D_{n+1}^s \in \mathsf{norm}_s(D_n^{s\uparrow} \wedge I(l_n))$. Combining this with Lemma 1 (1+2) gives $D_{n+1}^s \in \mathsf{norm}_s(\{D \wedge I(l_n) \mid \{(D')^\uparrow \mid D' \in \mathsf{norm}_s(D_n^s)\}\})$, and since D_n^s is already normalised we get $D_{n+1}^s \in \{D_n^{s\uparrow} \wedge I(l_n)\}$, *i.e.* $D_{n+1}^s = D_n^{s\uparrow} \wedge I(l_n)$. Now assume that for all D_n^i such that $D_n^s \in \mathsf{norm}_s(D_n^i)$ and $\langle l_n, D_n\rangle \leadsto \langle l_n, D_{n+1}\rangle$ by delay, $D_{n+1}^s \notin \mathsf{norm}_s(D_{n+1})$. By the definition of $\leadsto$ we have $D_{n+1} = D_n^{i\uparrow} \wedge I(l_n)$, which gives $\mathsf{norm}_s(D_{n+1}) = \mathsf{norm}_s(D_n^{i\uparrow} \wedge I(l_n))$. Expansion using Lemma 1 (1+2) yields $\mathsf{norm}_s(D_{n+1}) = \{D \wedge I(l_n) \mid D \in \{(D')^\uparrow \mid D' \in$

$\mathsf{norm}_s(D_n^i)\}\}$. By our assumption, $D_n^s \in \mathsf{norm}_s(D_n^i)$ and $D_{n+1}^s \notin \mathsf{norm}_s(D_{n+1})$, for all D_n^i, but this lead to a contradiction.

- (Action) By assumption we know that $\langle l_n, D_n^s\rangle \rightsquigarrow_s \langle l_{n+1}, D_{n+1}^s\rangle$ by $l_n \xrightarrow{gar} l_{n+1}$. From the definitions of norm_s and $\rightsquigarrow$ we can derive that for all D_n^i such that $D_n^s \in \mathsf{norm}_s(D_n^i)$, $\langle l_n, D_n^i\rangle \rightsquigarrow \langle l_{n+1}, D_{n+1}\rangle$ by $l_n \xrightarrow{gar} l_{n+1}$ Now we need to prove that $\exists D_{n+1}$ such that $D_{n+1}^s \in \mathsf{norm}_s(D_{n+1})$. By the definition of $\rightsquigarrow$, $\mathsf{norm}_s(D_{n+1}) = \mathsf{norm}_s(r(D_n^i \wedge g) \wedge I(l_{n+1}))$. Expansion by Lemma 1(1) gives $\mathsf{norm}_s(D_{n+1}) = \{D \wedge I(l_{n+1}) \mid D \in \mathsf{norm}_s(r(D_n^i \wedge g))\}$. According to Lemma 1(1+3) and $D_n^s \in \mathsf{norm}_s(D_n^i)$ we have $\mathsf{norm}_s(r(D_n^s \wedge g)) \subseteq \mathsf{norm}_s(r(D_n^i \wedge g))$. And since we know, by the definition of $\rightsquigarrow$, that $D_{n+1}^s \in \mathsf{norm}_s(r(D_n^i \wedge g))$ we conclude that $D_{n+1}^s \in \mathsf{norm}_s(D_{n+1})$.

(Completeness) As induction hypothesis, assume $\langle l_0, u_0\rangle \rightarrow^n \langle l_n, u_n\rangle \Rightarrow \exists D_n$ such that $\langle l_0, \{u_0\}\rangle \rightsquigarrow_s^n \langle l_n, D_n\rangle$ and $u_n \in D_n$. Further assume $\langle l_n, u_n\rangle \xrightarrow{\alpha} \langle l_{n+1}, u_{n+1}\rangle$. We need to prove that $\exists D_n^i$ such that $u_n \in D_n^i$, $\langle l_n, D_n^i\rangle \rightsquigarrow_s \langle l_{n+1}, D_{n+1}\rangle$ and $u_{n+1} \in D_{n+1}$. There are two cases, $\alpha = \epsilon(d)$ or $\alpha \in \Sigma$.

- ($\alpha = \epsilon(d)$) By the assumption $\langle l_n, u_n\rangle \xrightarrow{\epsilon(d)} \langle l_n, u_n + d\rangle$ we know $(u_n + d) \in I(l_n)$, *i.e.* $\exists u_n : u_n \in D_n^i \wedge u_n + d \in I(l_n)$. From the definition of $\rightsquigarrow_s$ we have $\langle l_n, D_n\rangle \rightsquigarrow_s \langle l_n, D_{n+1}\rangle$ by delay, if $D_{n+1} \in \mathsf{norm}_s(D_n^{\uparrow} \wedge I(l_n))$. Expansion by the definition of $\uparrow$ yields $D_{n+1} \in \mathsf{norm}_s(\{u + d \mid u \in D_n \wedge u + d \in I(l_n)\})$. By the definition of norm_s we know that for all zones D, $D \Rightarrow \bigwedge_{D' \in \mathsf{norm}_s(D)} D'$. Thus there is a zone $D_{n+1} \in \mathsf{norm}_s(D_n^{\uparrow} \wedge I(l_n))$ such that $u_{n+1} \in D_{n+1}$

- ($\alpha \in \Sigma$) By the assumption $\langle l_n, u_n\rangle \xrightarrow{\alpha} \langle l_{n+1}, [r]u_n\rangle$ we know $l_n \xrightarrow{g,\alpha,r} l_{n+1}$, $u_n \in g$, $[r]u_n \in I(l_{n+1})$. From the definition of $\rightsquigarrow_s$ we have $\langle l_n, D_n^i\rangle \rightsquigarrow_s \langle l_{n+1}, D_{n+1}\rangle$ by $l_n \xrightarrow{g,\alpha,r} l_{n+1}$ if $D_{n+1} \in \mathsf{norm}_s(r(D_n \wedge g) \wedge I(l_{n+1}))$. Expanding this by the definition of the r-operation yields $D_{n+1} \in \mathsf{norm}_k(\{[r]u \mid u \in D_n^i \wedge u \in g \wedge [r]u \in I(l_{n+1})\})$. By the definition of norm_s we know that for all zones D, $D \Rightarrow \bigwedge_{D' \in \mathsf{norm}_s(D)} D'$. Thus there is a zone $D_{n+1} \in \mathsf{norm}_s(r(D_n \wedge g) \wedge I(l_{n+1}))$ such that $u_{n+1} \in D_{n+1}$.

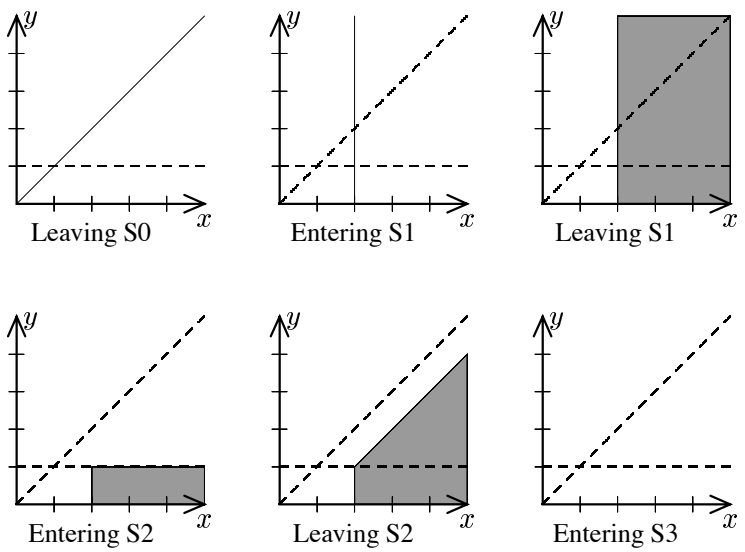


Figure 5: Zones for the example automaton, with non splitting normalisation.

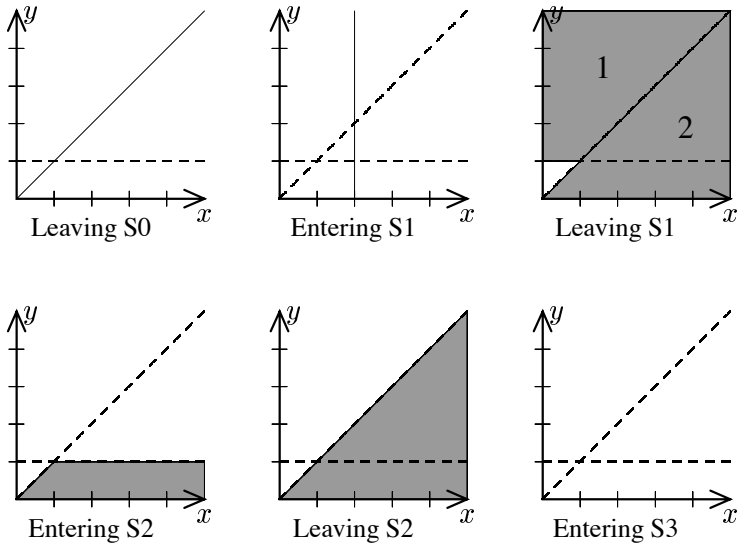


Figure 6: Zones for the example automaton, with splitting normalisation.

5 Conclusion

In modelling and verifying timed systems, using timed automata, constraints over clock differences are useful and needed in many applications e.g. solving scheduling problems. In this paper, we have reported a problem in the existing (published) symbolic reachability algorithms for timed automata. The problem is that the existing normalisation algorithms (implemented by several verification tools for timed automata e.g. UPPAAL) for clock constraints based on region equivalence are incorrect in the sense that they may provide wrong answers in verifying timed automata containing constraints on clock differences. The reason is that the normalisation operations may enlarge a zone so that the guard (a difference constraint) labelled on a transition is made true and thus incorrectly enables the transition. Thus the normalisation operation should be based on a finer equivalence relation than region equivalence. We propose to use the region equivalence which is further refined by difference constraints. Based on this, we have developed two normalisation algorithms that allow not only clock comparison with naturals but also comparison between clocks i.e. constraints on clock differences. The algorithms transform DBM's according to not only the maximal constants of clocks but also difference constraints appearing in an automaton. To our knowledge, they are the first published normalisation algorithms for timed automata containing difference constraints. The algorithms have been implemented in UPPAAL showing that almost no extra overhead is added to deal with difference constraints.

References

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science, pages 100–125. Springer-Verlag, 2001.

[AČJT00] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Johnsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Journal of Information and Computation*, 160:109–127, 2000.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2):183–235, 1994.

[BDGP98] Beatrice Bérard, Volker Diekert, Paul Gastin, and Antoine Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36:145–182, 1998.

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

[BLL+96] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, March 1996.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings, Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[DOTY95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool kronos. In *Proceedings, Hybrid Systems III: Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society, Ser. 3*, 2:326–336, 1952.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. Technical Report TR94-1404, Cornell Computer Science Technical Report Collection, 1994.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.

[LPY97] Kim G. Larsen, Paul Petterson, and Wang Yi. Uppaal in a nutshell. *Journal on Software Tools for Technology Transfer*, 1997.

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

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

[YL93] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. In *Proceedings, Fifth International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 210–224. Springer-Verlag, 1993.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1, October 1997.

[Yov98] Sergio Yovine. Model checking timed automata. In *European Educational Forum: School on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer-Verlag, 1998.

[YPD94] Wang Yi, Paul Petterson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings, Seventh International Conference on Formal Description Techniques*, pages 223–238, 1994.

Paper C:

Reducing Memory Usage in Symbolic State-Space Exploration for Timed Systems

Johan Bengtsson and Wang Yi.

Reducing Memory Usage in Symbolic State-Space Exploration for Timed Systems

Johan Bengtsson and *Wang Yi*

Department of Computer Systems, Uppsala University, Sweden.
Email: {johanb,yi}@docs.uu.se

Abstract. One of the major problems in scaling up model checking techniques to the size of industrial systems is memory consumption. This paper studies the problem in the context of verifiers for timed automata. We present a number of techniques that reduce the amount of memory used in symbolic reachability analysis. We address the memory consumption problem on two fronts. First, we reduce the size of internal representations for each symbolic state (clock constraints) by means of compression methods. Second, we reduce the explored state space (list of symbolic states) by *early-inclusion checking* between states and by probabilistic methods. These techniques have been implemented in the UPPAAL tool. Their strengths and weaknesses are evaluated and compared in experiments using real-life examples. Note that though these techniques are developed for timed systems, they are of general interests for verification tool development, in particular to handle large symbolic states based on constraint representation and manipulation.

1 Introduction

During the last ten years timed automata [AD90, AD94] have evolved as a common model to describe timed systems. This process has gone hand in hand with the development of verification tools for timed automata, such as KRONOS [DOTY95, Yov97] and UPPAAL [LPY97, ABB+01]. One of the major problems in applying these tools to industrial-size systems is the large memory consumption (*e.g.* [BGK+96]) when exploring the state space of a network of timed automata. The reason is that the exploration not only suffers from the large number of states to be explored, but also from the large size of each state. In this paper we address both these problems.

We will present techniques to reduce memory usage for internal representations of symbolic states by means of compaction. We use two different methods for packing states. First, we code the entire state as one large number us-

ing a multiply-and-add algorithm. This method yields a representation that is canonical and minimal in terms of memory usage but the performance for inclusion checking between states is poor. The second method is mainly intended to be used for the timing part of the state and it is based on concatenation of bit strings. Using a special concatenation of the bit string representation of the constraints in a zone, ideas from [PS80] can be used to implement fast inclusion checking between packed zones.

Furthermore, we attack the problem with large state spaces in two different ways. First, to get rid of states that do not need to be explored, as early as possible, we introduce inclusion checking already in the data structure holding the states waiting to be explored. We also describe how this can be implemented without slowing down the verification process. Second, we investigate how supertrace [Hol91] and hash compaction [WL93, SD95] methods can be applied to timed systems. We also present a variant of the hash compaction method, that allows termination of branches in the search tree based on probable inclusion checking between states.

The rest of the paper is organised as follows: In section 2 we introduce timed automata and describe briefly how to check reachability properties for timed automata. In section 3 we present methods to represent the key objects used in checking timed automata, namely the symbolic states. We also give a comparison between them. Section 4 addresses issues on the whole state space of an automaton. We describe how the wait and past lists are handled efficiently. We also describe an approximation method of the past list that can be used when the complete state space of an automaton is too big to be stored in memory. Finally, section 5 wraps up the paper by summarising the most important results and suggests some directions for future work.

2 Preliminaries

In this section we briefly review background materials including timed automata and reachability analysis based on clock constraints. A more extensive description can be found in *e.g.* [AD94, Pet99].

Let Σ be a finite set of labels, ranged over by a, b *etc*.A timed automaton is a finite state automaton over alphabet Σ extended with a set of real valued clocks, to model time dependent behaviour. Let $\mathcal{C}$ denote a set of clocks, ranged over by x, y, z. Let $\mathcal{B}(C)$ denote the set of conjunctions of atomic constraints of the

form $x \sim n$ or $x - y \sim n$ for $x, y \in \mathcal{C}, \sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. We use g and later D to range over this set.

Definition 5 (Timed Automaton) *A timed automaton A is a tuple $\langle N, l_0, \rightarrow, I\rangle$ where N is a set of control nodes, $l_0 \in N$ is the initial node, $\rightarrow \in N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$ is the set of edges and $I : N \longrightarrow \mathcal{B}(C)$ assign invariants to locations. As a convention we will use $l \xrightarrow{g,a,r} l'$ to denote $\langle l, g, a, r, l'\rangle \in \rightarrow$.*

The clock values are formally represented as functions, called clock assignments, mapping $\mathcal{C}$ to the non-negative reals $\mathbb{R}_+$. We let u, v denote such functions, and use $u \in g$ to denote that the clock assignment u satisfy the formula g. For $d \in \mathbb{R}_+$ we let $u + d$ denote the clock assignment that map all clocks x in $\mathcal{C}$ to the value $u(x) + d$, and for $r \subseteq \mathcal{C}$ we let $[r \mapsto 0]u$ denote the clock assignment that map all clocks in r to 0 and agree with u for all clocks in $\mathcal{C} \setminus r$.

The semantics of a timed automaton is a timed transition-system where the states are pairs $\langle l, u\rangle$, with two types of transitions, corresponding to delay transitions and discrete action transitions respectively:

- $\langle l, u\rangle \xrightarrow{\epsilon(t)} \langle l, u + t\rangle$ if $u \in I(l)$ and $(u + t) \in I(l)$
- $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle$ if $l \xrightarrow{g,a,r} l', u \in g, u' = [r \mapsto 0]u$ and $u' \in I(l')$

It is easy to see that the state space is infinite and thus not a good base for algorithmic verification. However, efficient algorithms may be obtained using a *symbolic semantics* based on *symbolic states* of the form $\langle l, D\rangle$ [HNSY92, YPD94] where l is the location vector of an automaton and $D \in \mathcal{B}(C)$ is a clock constraint (zone) specifying the clock values. They are symbolic in the sense that the clock zone represents a set of concrete states of the automaton with the same control location vector. The symbolic counterpart of the transitions are given by:

- $\langle l, D\rangle \rightsquigarrow \langle l, D^{\uparrow} \wedge I(l)\rangle$
- $\langle l, D\rangle \rightsquigarrow \langle l', r(D \wedge g)\rangle$ if $l \xrightarrow{g,a,r} l'$

where $D^{\uparrow} = \{u + d \mid u \in D \wedge d \in \mathbb{R}_+\}$ and $r(D) = \{[r \mapsto 0]u \mid u \in D\}$. It can be shown that the set of constraint systems is closed under these operations. Moreover the symbolic semantics correspond closely to the standard semantics in the sense that if $\langle l, D\rangle \rightsquigarrow \langle l', D'\rangle$ then, for all $u' \in D'$ there is $u \in D$ such that $\langle l, u\rangle \rightarrow \langle l', u'\rangle$.

Given a timed automaton with an initial symbolic state $\langle l_0, D_0\rangle$ and a final symbolic state $\langle l_f, D_f\rangle$, $\langle l_f, D_f\rangle$ is said to be *reachable* if $\langle l_0, D_0\rangle \rightsquigarrow^* \langle l_f, D_n\rangle$

and $D_f \cap D_n \neq \emptyset$ for some D_n. The reachability problem can be solved using a standard reachability algorithm as shown in Algorithm 1 for graphs with a proper normalisation algorithm for clock constraints [Pet99, Rok93] (to guarantee termination).

The algorithm uses two important data structures: WAIT and PASSED. WAIT is a list of states waiting to be explored and PASSED is the set of states already explored. Due to the size of the state space, these structures may consume a considerable amount of main memory. The main objective of this paper is to present techniques to reduce the memory usage of these two structures.

Algorithm 1 Symbolic reachability analysis

```
PASSED = ∅, WAIT = {⟨l_0, D_0⟩}
while WAIT ≠ ∅ do
  take ⟨l, D⟩ from WAIT
  if l = l_f ∧ D ∩ D_f ≠ ∅ then return "YES"
  if D ⊈ D' for all ⟨l, D'⟩ ∈ PASSED then
    add ⟨l, D⟩ to PASSED
    for all ⟨l', D'⟩ such that ⟨l, D⟩ ⇝_k ⟨l', D'⟩ do
      add ⟨l', D'⟩ to WAIT
    end for
  end if
end while
return "NO"
```

The core of the above reachability algorithm is manipulation and representation of symbolic states. So symbolic states are the core objects of state space search, and one of the key issues in implementing an efficient model checker is how to represent them. The desired properties of the representation also differ in parts of the verifier, and there are potential gains in using different representations in different places.

The encoding of the location vector and the integer assignment is straight forward. For the location vector we number the locations in each process, to get a vector of location numbers. Representing the clock zone is a bit trickier but starting from the constraint system representation of a zone it is possible to obtain an efficient intermediate representation. We start with the following observation: Let 0 be a dummy clock with the constant value 0. Then for each constraint system $D \in \mathcal{B}(\mathcal{C})$ there is a constraint system $D' \in \mathcal{B}(\mathcal{C} \cup \{0\})$ with the same solution set as D, and where all constraints are of the form $x - y < n$ or $x - y \leq n$, for $n \in \mathbb{Z}$.

We also note that to represent any clock zone we need at most $(|\mathcal{C} \cup \{0\}|)^2$ atomic constraints. One of the most compact ways to represent this is to use a matrix where each element represent a bound on the difference between two clocks. Each element in the matrix is a pair $\langle n, \sim \rangle$ where n is an integer and $\sim$ tells whether the bound is strict or not. Such a matrix is called a *Difference Bounds Matrix*, or *DBM* for short. More detailed information about DBMs can be found in [Dil89].

3 Representing Symbolic States

Recall that symbolic states are pairs in the form $\langle l, D \rangle$ where l is the location vector of an automaton represented as a vector of integers and D is a clock constraint (zone), represented as a matrix of integers (DBM). Logically such a state is a vector of integers representing the control locations and clock bounds. In the following, we study how to represent the vectors physically in the main memory for efficient storage and manipulation.

3.1 Normal Representation

The simplest way to physically represent a symbolic state is to use a machine word for each control location, integer value or clock bound. The implementation is straight forward, but a practical tip is that if the standard library functions for memory management are used all the memory needed for one state should, if possible, be allocated in the same chunk, to minimise the allocation overhead.

The strength of this representation is its simplicity and the speed of accessing an individual control location, integer value, or clock bound. In this representation the maximum time needed to reach any individual entity is the time needed to fetch a word from the memory. This makes the representation ideal to use when we have to do operations on individual entities, *e.g.* when calculating the successors of a state. The weakness is the amount of wasted space. Here a whole machine word, typically 32-bit wide, is used to store entities where all possible values may fit in many fewer bits.

However this is a good base representation for states. It is ideal for states that will be modified in the near future, such as intermediate states or states in WAIT. It also works reasonably well for states in PASSED, specially for small and medium sized examples.

This representation is used for both WAIT and PASSED in the current version of UPPAAL.

3.2 Packed States

The second representation is on the opposite side of the spectrum compared to the previous one and it can be used for the discrete part of the states, for the clock zone and for both together. The encoding builds on a simple multiply and add scheme, similar to the position system for numbers, and it is very compact. In the description we will focus on encoding an entire symbolic state, but the parts can also be encoded separately.

First, consider the state as a vector $v_1, \ldots, v_n$, where each element represents a control location, the value of a variable, or a clock bound. For each element v_i we can compute the number of possible values, $|v_i|$. For the location vector $|v_i|$ is the number of control locations in the corresponding process, for the integer assignment $|v_i|$ is the size of the domain of the corresponding variable and for the clock zone then $|v_i|$ can be computed using the maximum constant k.

Now consider the vector as a number written down in a position system with a variable base, *i.e.* each element v_i is a digit and the product $\prod_{j=0}^{i-1} |v_i|$ is its position value. Represent the state as the value of this number, *i.e.* encode the state as follows:

$$E(\langle l, D \rangle) = \sum_{i=0}^{n} \left(v_i \cdot \prod_{j=0}^{i-1} |v_i| \right)$$

Note that the representation of states using multiply-and-add encoding is canonical and minimal in terms of space usage. Note also that in this context $\langle l, D \rangle$ is a sequence of numbers. The encoding $E(\langle l, D \rangle)$ is a number. The resulted numbers are often too large to fit in a machine word and they have to be in fixed precision; thus we need some kind of arbitrary precision numbers for the encoding. In our prototype implementation we used the GMP package [Gra00].

The strength of this representation is the effective use of space and the weakness is that to access an individual integer value or clock bound a number of division and modulo operations must be performed. This results in small states that are expensive to handle.

Example	Time		Space	
	real (Sec)	relative	real (MB)	relative
Field Bus (Faulty 1)	541.32	3.70	23.05	0.30
Field Bus (Faulty 2)	956.55	4.34	33.80	0.29
Field Bus (Faulty 3)	10630.90	8.21	136.52	0.28
Field Bus (Fixed)	2890.30	5.88	60.59	0.29
B&O	52.44	2.81	11.17	0.51
DACAPO (big)	1379.45	4.81	34.01	0.37
DACAPO (small)	31.67	2.23	5.55	0.65
Fischer 5	94.31	5.82	4.18	0.45
Fischer 6	17387.56	12.88	40.12	0.28

Table 1: Performance for packed states with expensive inclusion checking

In order to test the performance of this representation, it is implemented in the PASSED structure in UPPAAL. The implementation is straight forward, however expensive division and modulo operations have to be used, in order to compare the DBMs bound by bound.

The result of the experiment is presented in Table 1 (as absolute figures and in relation to the current PASSED implementation in UPPAAL). We note that with this representation the space performance is very good, with reductions of up to 70% compared to the current PASSED implementation. However the time performance is poor, for one instance of Fischers protocol we notice a slowdown of almost 13 times and for one instance of the Field Bus protocol the slowdown is 8 times. The conclusion is that this representation should only be used in cases where main memory is a severe restriction.

3.3 Packed Zones with Cheap Inclusion Check

The main drawback of representing states using the number encoding given in section 3.2 is expensive inclusion checking. In this section we present a compact way of representing zones overcoming this drawback. The heart of this representation builds on an observation due to [PS80] that one subtraction can be used to perform multiple comparisons in parallel.

Let m denote the minimum number of bits needed to store all possible values for one clock bound. The DBM is then encoded as a long bit string, where each bound is assigned a $m+1$ bit wide slot. The value of the clock bound is put in

the m least significant bits in the slot and the extra, most significant bit, is used as a *test bit*.

Since a zone D is included in another zone D' if and only if each bound in the DBM representing D is as tight as the same bound in the DBM representation of D', inclusion checking is to check if all elements in one vector is less than or equal to the same bound in another vector. Using the new bit-string encoding of zones this can be checked using only simple operations like bitwise-and ($\&$), bitwise-or ($|$), subtraction and test for equality.

Given two packed zones $E(D)$ and $E(D')$, to check if $D \subseteq D'$ first set all the test bits in $E(D)$ to zero and all the test bits in $E(D')$ to one. In an implementation the test bits are usually zero in the stored states and setting them to one is done using a prefabricated mask M. The test is then performed by calculating $E(D') - E(D)$. The result is read out of the test bits. If a test bit is one the corresponding bound in D is at least as tight as in D' and if a test bit is zero the corresponding bound is tighter in D' than in D. Thus, if all test bits are one we can conclude that $D \subseteq D'$ and if all the test bits are zero $D \supset D'$. It is worth noting that "all test bits are one" is both necessary and sufficient to conclude $D \subseteq D'$ while "all test bits are zero" is only sufficient to conclude $D \supset D'$.

In an implementation of this scheme the main issue is how to handle the bit strings. The easiest way is to let a bignum package handle everything. However, this may give a considerable overhead, specially in connection with memory allocation, since the bignum packages are often tailored towards other types of applications. In UPPAAL we share the memory layout of the bignum packages, but to reduce the overhead we have implemented our own operations on top of it.

In the physical representation, *i.e.* how the bit-string is stored in memory, the bit-string is chopped up into machine-word sized chunks, or *limbs*. The limbs are then packed in big-endian order, *i.e.* the least significant limb first, in an array. If the bit string doesn't fill an even number of machine words the last limb is padded with zero bits.

Noting that the effect of all operations needed for the inclusion check, except subtraction, is local within the limb and that subtraction only passes one borrow bit to the next more significant limb, we can implement the inclusion check in one pass through the array of limbs instead of one pass for each operation. The one pass inclusion check is shown in Algorithm 2. In the description we use $E(D)[i]$ to denote the limb with index i in $E(D)$ and $-_w$ to denote a binary subtraction of machine word size.

Algorithm 2 Inclusion check for packed zones

$b \Leftarrow 0$
for $i = 1$ **to** #limbs **do**
  cmp $\Leftarrow (M[i] \mid E(D')[i]) -_w (E(D)[i] + b)$
  if cmp $\neq M[i]$ **then return** "false"
  if $(M[i] \mid E(D')[i]) < (E(D)[i] + b)$ **then**
    $b \Leftarrow 1$
  else
    $b \Leftarrow 0$
  end if
end for
return "true"

Example	Time		Space	
	real (Sec)	relative	real (MB)	relative
Field Bus (Faulty 1)	142.47	0.97	49.69	0.64
Field Bus (Faulty 2)	212.82	0.97	71.77	0.61
Field Bus (Faulty 3)	1190.96	0.92	278.54	0.57
Field Bus (Fixed)	488.14	0.99	139.01	0.65
B&O	18.04	0.97	13.31	0.61
DACAPO (big)	278.58	0.97	43.39	0.48
DACAPO (small)	14.54	1.02	6.49	0.77
Fischer 5	12.46	0.77	4.66	0.50
Fischer 6	815.94	0.60	51.12	0.35

Table 2: Performance for packed states with cheap zone coding

To evaluate the performance of this technique, it was implemented in the PASSED structure in UPPAAL. In the experiment the discrete part of each state is stored in PASSED using the compact representation from the previous section and the zone is stored using this technique. The results are presented in Table 2, both as absolute figures and compared to the standard state representation. We note that using this method the space usage is typically reduced by about 40%, without increased verification time. The verification time is actually reduced a little using this scheme, even though the number of operations is increased. The reason for this is most certainly that the number of memory operations are reduced by the smaller memory footprint of the states[1].

[1]Memory operations are expensive compared to arithmetic operations, specially since there is no temporal locality in verifiers.

4 Representing the Symbolic State-Space

The two key data structures in a model checker are, as mentioned before, WAIT, that keeps track of states not yet explored, and PASSED, that keeps track of states already visited. Both these data structures tend to be large, and how to represent them is an important issue for performance. In this section we describe how to implement WAIT and how to improve its performance by adding inclusion checking. We also describe a standard implementation of PASSED as well as an implementation where space is saved at the price of possibly inconclusive answers.

4.1 Representing WAIT

In its most simple form WAIT is implemented as a linked list. This is easy to implement and it is easy to control the search order by adding unexplored states at the end, for breadth first search, or adding states at the beginning, for depth first search.

An optimisation in terms of both time and space is to check whether a state already occur in WAIT before adding it. For a verifier based on explicit states this will only give minor improvements, mainly by keeping down the length of WAIT, but for a verifier based on symbolic states this may actually prevent revisiting parts of the state space.

We know, *e.g.* from [Pet99], that if $\langle l, D\rangle \subseteq \langle l, D'\rangle$ then all states reachable from $\langle l, D\rangle$ are also reachable from $\langle l, D'\rangle$ and thus we only have to explore $\langle l, D'\rangle$. So before adding a new state $\langle l, D\rangle$ to WAIT we check all states already in WAIT. If we find any state including $\langle l, D\rangle$ we stop searching and throw away $\langle l, D\rangle$ since all states reachable from it are also reachable from a state already scheduled for exploration. If no such state is found we add $\langle l, D\rangle$ to WAIT. During the search through WAIT we also delete all states included in $\langle l, D\rangle$ in order to prevent revisiting parts of the state space.

There are some implementation issues that need consideration. The main issue is how to find all states in WAIT with same discrete part. The simplest way to do this is to do a linear search through WAIT every time a state is added. However, using this solution it will be expensive to add states, even for examples where WAIT is short. One solution to this is to implement WAIT using a structure where searching is cheap, *e.g.* a hash table. The problem with this solution is that picking up states from WAIT will be expensive, at least for search strategies

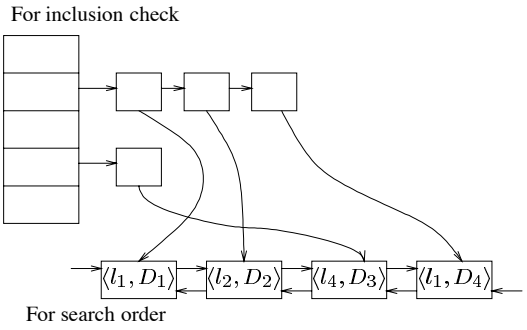


Figure 1: Structure of WAIT

Example	Time (Sec)			Space (MB)		
	no opt	inclusion	gain(%)	no opt	inclusion	gain(%)
Field Bus (Faulty 1)	335.53	152.66	54.50	83.00	78.02	6.00
Field Bus (Faulty 2)	610.87	226.31	62.95	128.32	117.97	8.07
Field Bus (Faulty 3)	2142.13	1342.23	37.34	510.77	489.94	4.08
Field Bus (Fixed)	1051.19	497.85	52.64	230.03	212.33	7.70
B&O	20.24	18.80	7.11	21.91	21.91	0.00
DACAPO (big)	828.53	296.87	64.17	104.84	90.91	13.29
DACAPO (small)	97.90	14.54	85.15	15.45	8.48	45.15
Fischer 5	14.75	16.86	−14.31	9.68	9.38	3.15
Fischer 6	1179.34	1456.64	−23.51	150.93	145.73	3.44

Table 3: Performance impact of inclusion check on WAIT

like breadth first and depth first, where the exploration order depends on the order in which the states were added WAIT.

In the implemented solution, each state in WAIT is indexed using both a list and a hash table. The list part is used to keep the depth or breadth first ordering of states and to make it cheap to pick up states to explore. The hash table part is used to index the states in WAIT based on their location vector, in order to speed up inclusion checking. A picture of this structure is shown in Figure 1.

To test the performance of this solution we compared the space and time needed to explore the state space of nine examples, for one version of UPPAAL without inclusion checking on WAIT and one version with the combined scheme. The result is shown in Table 3. It is worth noting that the version with inclusion checking is both significantly faster and less memory consuming than the version without inclusion checking, for all examples except Fischers protocol which is, as mentioned in the appendix, not typical.

4.2 Representing PASSED

The key feature needed by a representation of PASSED is that searching should be cheap. For a symbolic verifier it is also crucial, at least performance wise, that finding states which includes a given state is possible and cheap. In UPPAAL the standard PASSED is implemented as a hash table, where the key is computed from the discrete part of the state and collisions are handled by chaining. The reason for basing the hash key only on the discrete part is to simplify checking for inclusion between states by making all related states end up in the same hash bucket. It is easy to see that hashing only on the discrete part is as good as we can do if we want this property. The reason for using chaining instead of open addressing to resolve conflicts is, apart from keeping related states together, mainly simplicity and eliminating the need for expensive rehashing. Judging by performance the choice could go either way, at least if rehashing is not taken into account. More about this can be read in [Lar00].

For some models the memory needed for exact verification may exceed the amount of memory installed in the system where the verification takes place. This often occurs within the modelling phase before most bugs are removed from the model. During this phase the verification engine is often used as a tool to find the cause of unwanted behaviour and not primarily to prove the absence of such behaviour. Under these premises it is desirable to use a method that can handle larger systems but sometimes miss unwanted behaviour. Here we will describe two such methods. The first method is an application of the supertrace algorithm from [Hol91] on networks of timed automata. The second method is based on the hash compaction method from [WL93, SD95].

4.3 Supertrace PASSED for Timed Automata

The main idea behind supertrace PASSED is from the following observation: The purpose of PASSED is only to keep track of whether a state have been visited or not, *i.e.* for each state we only need one bit of information. Thus, PASSED for a system of n states can be implemented as an n-bit wide bit vector. However, if n is sufficiently large, even such a compact representation will be too large to fit the memory of the system running the verifier. A way to tackle this problem is to loosen the demand that the verification should be exact and allow false hits to be indicated, *i.e.* a previously unvisited state may, with some probability, be reported as already visited. Such a false hit will be called an *omission*, as it causes a part of the state space to be omitted from the state space

search. This affects the reachability search such that if a state is reported to be not reachable we can not conclude that it can not be reached since it might have been excluded by an omission.

The natural way to implement such a PASSED structure is to allocate a bit-vector of size k, where $k < n$, and hash each state to a value in $\{1, \ldots, k\}$. In the UPPAAL implementation of the supertrace algorithm the hash function is similar to the first packing technique described in Section 3:

$$H(\langle l, D \rangle) = \left(v_0 + \sum_{i=1}^{n} \left(v_i \cdot \prod_{j=0}^{i-1} |v_i| \right) \right) \mod k$$

Note that a variation of this hash function (applied only to the location vector and the integer assignment) is used in both the normal PASSED implementation and the cross-reference table of the WAIT list. It is also possible to enhance the supertrace algorithm by implementing a way to change the hash function between runs, in order to lower the probability that a part of the state space is omitted. A simple way to do this is to implement a generator of *universal*$_2$ hash functions [CW79] and provide the user with a way to choose among the functions in the class.

The main drawback of the supertrace algorithm, when applied to timed automata, is that inclusion between time zones can not be detected. The effect of this is that the number of explored states increases. This leads to longer verification times and more states to enter in PASSED, with an increased omission probability as result.

To investigate the performance of this algorithm we have implemented it in UPPAAL. In the experiment we test the supertrace PASSED structure for three different sizes: 16MB, 32MB and 64MB and compare it to the standard PASSED implementation of UPPAAL, to estimate the impact of collisions. The results of the experiment are presented in Table 4. For each of the examples the table shows the collision frequency and an estimation on the fraction of the state space not covered due to collisions.

In the table there are several interesting observations. First, for the Philips example the coverage is totally independent of the size of the PASSED structure. We get exactly the same collision frequency and coverage for all three runs. This is an indication that the hash function is far from optimal on this example.

Example	16MB		32MB		64MB	
	collision	omitted	collision	omitted	collision	omitted
Philips (Correct)	0.45	5.71	0.45	5.71	0.45	5.71
B&O	0.97	21.62	0.91	16.07	0.65	3.49
DACAPO (big)	4.40	13.05	2.75	13.74	1.24	4.39
DACAPO (small)	1.65	5.96	0.79	4.17	0.39	3.13
Fischer 5	0.18	0.64	0.07	0.35	0.04	0.04
Fischer 6	3.67	12.29	1.84	6.30	0.93	3.12

Table 4: Frequency of collisions and the fraction of state space not covered (in ‰) for three instances of the supertrace PASSED structure

Example	Supertrace			Classic
	16MB	32MB	64MB	
Philips (Correct)	2.39	2.58	2.97	1.91
Philips (Erroneous)	98.83	98.18	102.20	22.22
B&O	15.59	16.20	16.30	16.57
DACAPO (big)	268.32	268.04	269.67	254.01
DACAPO (small)	15.32	15.47	15.90	12.45
Fischer 5	11.57	13.01	12.20	14.36
Fischer 6	688.28	681.67	686.75	1217.88

Table 5: Time (in seconds) to explore the entire state space for three different supertrace PASSED lists for the standard PASSED list

We also note, when studying the big DACAPO example, that even though the collision frequency is decreased the fraction of the state space not covered in the search may increase. The reason for this may be that the collisions occur for different states in the different runs and that the number of children for these states differ. (If a state with many children is omitted the coverage will be less than if a state with few children is omitted.)

To see how the supertrace algorithm behave time-wise we made an experiment where the verification time was measured and compared to the standard PASSED implementation in UPPAAL. The setting of this experiment is a little different from the previous one. For this example we used inclusion checking on WAIT, to speed up verification. This is the most likely setting when using the tool in practice. The result of this experiment is presented in Table 5. As we see in the table the times for the supertrace is in the same order of magnitude as the standard PASSED implementation.

4.4 Hash Compaction for Timed Automata

Hash compaction evolved from the supertrace ideas as a way to lower the probability of omissions in the verification process. It was first investigated in [WL93] and then further developed in [SD95].

The key observation for hash compaction is that the supertrace PASSED list can be seen as representation of a set of hash values, where a set bit (1) in the table represents that this hash value is in the set; while an unset bit (0) in the table represent that it is not. Under the assumption that the set is sparse, *i.e.* the number of elements in the set is small compared to the number of elements not in it, a table of the elements might be a more compact representation of the set. With this solution the number of possible hash values is no longer bounded by the number of bits in the main memory.

In the work presented in [WL93] a normal hash table is used to store the elements and the key into this table is computed from the elements themselves. In [SD95] the technique is developed further. As a way to decrease the probability of false collisions the key into the table is computed from the state itself, instead of from the hash signature, using a different hash function. Since the hash signature and its entry in the table are computed using different hash functions two states have to collide in both the hash functions for a false collision to occur.

There is an alternative way to view this second variation of hash compaction. Start with the supertrace PASSED list. To lower the probability of classifying an unvisited state as already visited we increase the number of bits in each entry of the hash table. (Given a fixed amount of memory this is done at the expense of the number of entries in the table.) To separate different states that end up at the same position in table we build a signature, *e.g.* a checksum, of the states and store this. To compute the checksum we choose a function with a low probability that two different states have the same signature, *i.e.* $P\left(H(\langle l_1, D_1\rangle) = H(\langle l_2, D_2\rangle) | \langle l_1, D_1\rangle \neq \langle l_2, D_2\rangle\right)$ should be as small as possible. For this we use a hash function. If we take this one step further the combination of the signature and the index into the hash table can be seen as different parts of the same hash value. Some bits of this value are used to index into the hash table and some bits are stored in the table. A sketch of this is shown in Figure 2.

Given a fixed amount of memory there is a tradeoff where to put the border between the index part and the signature part. For each bit we take away from

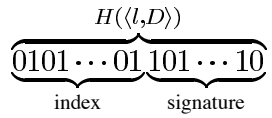


Figure 2: The table index and the signature as one hash value.

the index part we may double the number of bits in the signature, but at the price of less entries in PASSED.

So far we have not mentioned how to handle collisions within the hash table. Since there are now several possible values for the entries in the hash table, it is possible to get collisions in the hash table. Since the main priority of this solution is space, collisions are resolved using open addressing instead of chaining. This will save one pointer for each state entered into PASSED, and since the signatures are, more or less, as big as a pointer we may fit twice as many states in the same amount of memory with open addressing than with chaining. The price we pay for this choice is that the hash table might get full. Normally this would only lead to an expensive rehashing but in our case the information needed to rehash an entry in the hash table is no longer available. This leaves us with two choices, we can either stop the verification and say that PASSED is full and advice the user to try with a larger PASSED, or we can just skip adding the state to PASSED and hope that the search will terminate anyway. In the prototype implementation we have chosen the first alternative.

To evaluate hash compaction for timed automata, we used two slightly different PASSED implementations. The difference between them lies in what we store in the hash table. In the first implementation we store signatures of entire symbolic states. This solution gives a very compact representation of each state in PASSED, but it has the drawback that inclusion between states in PASSED can not be detected. This leads to potentially larger state spaces resulting in a higher pressure on the PASSED structure.

In the second PASSED implementation we try to get around this problem by separating the discrete part and the clock zone. In this implementation we apply the hash function only to the discrete part of the state. The clock zone is compressed using the method from Section 3.3 and stored in the hash table together with the signature. With this solution we aim at minimising the number of states stored in PASSED. However, storing the full zone has a big drawback. The entries in PASSED are much bigger than for the other type. For a fixed

memory-size this will give less entries in PASSED. A way around this would be to compress the zones further using a method that, with some probability, might report false inclusions. However, this has not been investigated in this paper.

As an introductory experiment all the examples are run with 47-bit signatures[2] for three different sizes of the hash table (16MB, 32MB and 64MB), and an estimate of the covered part of the state space is computed. In order to prevent interference from the inclusion check on WAIT, this feature is turned off. In this experiment we experienced no omissions, but for some examples the verification procedure did not terminate correctly.

The faulty Philips example can not be handled at all by PASSED implementation based only on signatures; while it can be handled by the combined scheme when the size of the passed list is at least 32MB. The reason is that large parts of the state space of this example is revisited since the first PASSED implementation only can detect equal states and not inclusion between states. In contrast to this example, the large instance of the DACAPO example terminates for all sizes using the first PASSED implementation, while it fails to do so for 16MB and 32MB using the second. This is because, for each state, the zone information is an order of magnitude larger than the size of the hash signature. This, in combination with the fact that (for this example) the number of explored states are almost the same in both variations, lead to that 16MB is large enough when using signatures only while 64MB is needed for the combined scheme.

To study what the impact of the signature length on the fraction of the state space that is omitted from exploration we perform an experiment with 7-bit signatures.[3] The result of this experiment can be seen in Table 6. As we see in the table there are still problem instances where no omissions occur. We also note that where omissions occur, in all cases except one, less than one per mille of the state space is omitted from exploration.

As a final experiment we measure the run time and memory use for state space exploration with a PASSED structure based on hash compaction with 47-bit signatures and compare it to the run time for state space exploration using the classic PASSED implementation in UPPAAL. To get as close as possible to a normal use situation, inclusion checking for WAIT is enabled in this experiment. The measured run times are listed in Table 7. We note from the table that the combined scheme (signatures of the discrete part + packed zone) is some-

[2] The size of the signature may seem a little odd, but in the implementation one bit is sacrificed to ensure that no used slot in the hash table can be mistaken for an empty.

[3] This is the smallest possible signature size in the current implementation.

what faster, for all examples, than using only signatures. The reason for this is the smaller number of states that is visited using the combined scheme. We also note that using hash compaction is somewhat slower than using the classic PASSED implementation (for all examples except Fischers protocol). This is partly due to the extra work needed to compute the signatures and partly due to that the hash compaction implementation within UPPAAL is a prototype.

The measured memory use for the different examples is listed in Table 8. From this table we note that for the large examples, *i.e.* Field Bus, the large DACAPO instance and Fischer 6, there are significant reductions in memory usage. We also note that for some of the smaller examples the classic PASSED implementation use less memory than the hash compaction. This suggests that the chosen size of the hash compaction is too large, and that these examples can be verified using much smaller PASSED. A further observation is that the measured numbers for hash compaction are larger than the requested size for PASSED. The reason for this is that the listed values are the total memory used in the verification, *i.e.* the numbers also include WAIT, temporary storage and the binary code. In a real application, this should be taken into account when deciding how much memory to reserve for PASSED.

5 Conclusions

This paper describes and evaluates three different ways to physically represent symbolic states in PASSED, in implementing verifiers for timed automata. The evaluation shows that if space consumption is a main issue rather than time consumption then the multiply-and-add scheme can be used. For the evaluated examples this optimisation reduces the memory usage by up to 70% compared to the current representation used in UPPAAL, at the price of 3–13 times slowdown due to expensive inclusion checking between states. In all other cases the state should be represented using a mixed representation where the discrete part is represented using the multiply-and-add scheme and the zone is represented by concatenated bit strings separated by test bits. This packing scheme reduces the memory usage with 35%–65% compared to the current version of UPPAAL. In most cases this representation also gives a minor speedup (1%–3%) compared to the current UPPAAL implementation.

Further the paper describes how to improve performance by checking for already visited states not only on PASSED, but also on WAIT. For the evaluated

Example	signature			signature+pack		
	16MB	32MB	64MB	16MB	32MB	64MB
Philips (correct)	0.00	0.00	0.00	0.00	0.00	0.00
Philips (faulty)	⊥	⊥	⊥	0.00	0.00	0.00
B&O	0.80	1.17	0.81	2.25	0.29	0.00
DACAPO (big)	0.94	0.65	0.21	⊥	⊥	0.19
DACAPO (small)	0.54	0.19	0.10	0.13	0.14	0.02
Fischer 5	0.00	0.00	0.05	0.00	0.00	0.00
Fischer 6	0.95	0.44	0.19	0.08	0.02	0.03

Table 6: Fraction of state space (in ‰) omitted from exploration for hash compaction with 7-bit signatures.

Example	signature			signature+pack			Classic
	16MB	32MB	64MB	16MB	32MB	64MB	
Field Bus (Faulty 1)	⊥	⊥	⊥	⊥	140.86	144.73	148.12
Field Bus (Faulty 2)	⊥	⊥	⊥	⊥	211.41	215.45	218.17
Field Bus (Faulty 3)	⊥	⊥	⊥	⊥	⊥	⊥	1296.95
Field Bus (Fixed)	⊥	⊥	⊥	⊥	⊥	464.89	488.39
Philips (correct)	3.30	3.46	3.82	2.19	2.32	2.61	1.89
Philips (faulty)	⊥	⊥	⊥	23.11	23.05	23.40	22.84
B&O	22.62	22.78	22.83	20.00	20.11	20.32	16.45
DACAPO (big)	304.24	305.09	303.44	⊥	256.77	256.35	259.87
DACAPO (small)	19.11	19.25	19.64	13.70	13.86	14.15	12.60
Fischer 5	13.05	13.18	13.50	12.80	13.05	13.35	14.42
Fischer 6	641.39	643.09	646.26	1252.79	995.90	963.30	1210.97

Table 7: Run time (in seconds) for state space exploration using a PASSED list based on hash compaction with 47-bit signatures.

Example	signature			signature+pack			Classic
	16MB	32MB	64MB	16MB	32MB	64MB	
Field Bus (Faulty 1)	⊥	⊥	⊥	⊥	40.41	72.41	77.92
Field Bus (Faulty 2)	⊥	⊥	⊥	⊥	40.61	72.62	117.88
Field Bus (Faulty 3)	⊥	⊥	⊥	⊥	⊥	⊥	489.87
Field Bus (Fixed)	⊥	⊥	⊥	⊥	⊥	75.82	212.21
Philips (correct)	17.91	33.91	65.91	17.89	33.89	65.89	4.21
Philips (faulty)	⊥	⊥	⊥	18.35	34.35	66.35	18.23
B&O	17.85	33.85	65.85	17.85	33.85	65.85	21.81
DACAPO (big)	24.93	40.93	72.93	⊥	40.01	72.01	92.77
DACAPO (small)	18.44	34.44	66.44	18.40	34.40	66.40	8.38
Fischer 5	18.91	34.91	66.91	18.88	34.88	66.88	9.28
Fischer 6	40.13	56.13	88.13	38.38	54.38	86.38	145.64

Table 8: Space (in MB) for state space exploration using a PASSED list based on hash compaction with 47-bit signatures.

examples this optimisation reduces the verification time by up to 85% and the memory usage with up to 45%.

Finally we study PASSED representations based on supertrace and hash compaction effect the performance of UPPAAL. The gain from this technique is significantly reduced memory usage for large examples, but at the price of possibly omitting parts of the state space from exploration. For the evaluated examples a supertrace PASSED cause between 22 per mille and 0.04 per mille of the state space to be omitted from the exploration. The evaluation show also that supertrace PASSED representations only work for examples where the number of revisited states (that can't be detected without inclusion checking) is small.

For hash compaction we evaluate two, slightly different, methods. One method where a hash key, signature and probe sequence is computed using both the discrete part of the states and the time zone, and one method where the hash key, signature and probe sequence is computed only from the discrete part of the states while the time zone is compressed and stored together with the signature. The evaluation shows that in terms of coverage both these methods outperform the supertrace method. For 47-bit signatures there are no omissions at all (in the evaluated examples) and for 7-bit signatures the number of omissions are less than $1/10$:th of the number of omissions in the supertrace PASSED representation.

A future extension of this part of the work is to investigate how the size of timing region can be reduced while still maintaining the possibility of inclusion checking between states.

References

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science, pages 100–125. Springer-Verlag, 2001.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings, Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2):183–235, 1994.

[BGK+96] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Petterson, and Wang Yi. Verification of an audio protocol with bus collision using UPPAAL. In *Proceedings, Eigth International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings, Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[DOTY95] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool kronos. In *Proceedings, Hybrid Systems III: Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[DY00] Alexandre David and Wang Yi. Modelling and analysis of a commercial field bus protocol. In *Proceedings, Twelfth Euromicro Conference on Real Time Systems*, pages 165–174. IEEE Computer Society Press, 2000.

[Gra00] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library*, 3.0.1 edition, 2000.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

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

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using uppaal. In *Proceedings, 18*th *IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society Press, 1997.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[Lar00] Fredrik Larsson. Efficient implementation of model-checkers for networks of timed automata. Licentiate Thesis 2000-003, Department of Information Technology, Uppsala University, 2000.

[LP97] Henrik Lönn and Paul Pettersson. Formal verification of a tdma protocol startup mechanism. In *Proceedings of 1997 IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242. IEEE Computer Society Press, 1997.

[LPY97] Kim G. Larsen, Paul Petterson, and Wang Yi. Uppaal in a nutshell. *Journal on Software Tools for Technology Transfer*, 1997.

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

[PS80] Wolfgang J. Paul and Janos Simon. Decision trees and random access machines. In *Logic and Algorithmic*, volume 30 of *Monographie de L'Enseignement Mathématique*, pages 331–340. L'Enseignement Mathématique, Université de Genève, 1980.

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

[SD95] Ulrich Stern and David L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[WL93] Pierre Wolper and Dennis Leroy. Reliable hashing without collision detection. In *Proceedings, Fifth International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1, October 1997.

[YPD94] Wang Yi, Paul Petterson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings, Seventh International Conference on Formal Description Techniques*, pages 223–238, 1994.

A Examples and Experiment Environment

The experiments are run on a Sun Ultra Enterprise 450 with four[4] 400MHz CPUs and 4GB of main memory. The operating system on the machine was Solaris 7.

All the ideas have been implemented on top of the current development version of UPPAAL (3.1.26), and to measure the performance we used five different applications: Field Bus, B&O, DACAPO, Philips and Fischer. During all the experiments the memory limit for the run is set to 1GB which is at least twice the amount of memory needed for UPPAAL 3.1.26 to check any of the examples, using the standard representation of states. If any run exceeds this limit the run is marked as unsuccessful ($\perp$).

The Field Bus application is a model of the data link layer of a commercial field bus protocol. The protocol and the models we use (three erroneous and one corrected version) are described in [DY00].

B&O is a highly time-sensitive protocol devolved by Bang & Olufsen to transmit control messages between audio/video components. The model used in the experiments is described in [HSLL97].

DACAPO is a model of the start-up algorithm of the so-called DACAPO protocol. The DACAPO protocol is TDMA (time division multiple access) based and intended for local area networks inside modern vehicles. For a more thorough description of this application, see [LP97]. In these experiments we use two different models: a small one with three stations and drifting clocks, and larger one with four stations and perfect clocks.

The Philips example is a model of the physical layer of a protocol used by Philips to connect different parts of stereo equipment. This model was one of the first larger case studies made with UPPAAL. The model is thoroughly described in [BGK$^+$96]. This example is only used in the experiments with probabilistic passed lists. Two versions of this protocol are used in the experiments, the correct model and one faulty model.

The last application is Fischers protocol for mutual exclusion [Lam87]. This simple protocol for mutual exclusion has, unfortunately, become a standard benchmark for verification tools for timed systems, since the state space grows rapidly with the number of processes. The reason that this example is not a good

[4]The version of UPPAAL used in the experiments is not multi threaded, so we only use one CPU for each run.

benchmark example is that it is not very realistic and it behaves differently, verification-wise, from examples based on real case studies. In the experiments we have used two different sizes of this problem, one with five processes and one with six processes.

Paper D:

Partial Order Reductions for Timed Systems

Johan Bengtsson, Bengt Jonsson, Johan Lilius and Wang Yi.

Partial Order Reductions for Timed Systems

Johan Bengtsson[1] *Bengt Jonsson*[1] *Johan Lilius*[2] *Wang Yi*[1]

[1] Department of Computer Systems, Uppsala University, Sweden.
Email: {bengt,johanb,yi}@docs.uu.se
[2] Department of Computer Science, TUCS, Åbo Akademi University, Finland.
Email: Johan.Lilius@abo.fi

Abstract. In this paper, we present a partial-order reduction method for timed systems based on a *local-time* semantics for networks of timed automata. The main idea is to remove the implicit clock synchronisation between processes in a network by letting local clocks in each process advance independently of clocks in other processes, and by requiring that two processes *resynchronise* their local time scales whenever they communicate. A symbolic version of this new semantics is developed in terms of predicate transformers, which enjoys the desired property that two predicate transformers are independent if they correspond to disjoint transitions in different processes. Thus we can apply standard partial order reduction techniques to the problem of checking reachability for timed systems, which avoid exploration of unnecessary interleavings of independent transitions. The price is that we must introduce extra machinery to perform the resynchronisation operations on local clocks. Finally, we present a variant of DBM representation of symbolic states in the local time semantics for efficient implementation of our method.

1 Motivation

During the past few years, a number of verification tools have been developed for timed systems in the framework of timed automata (e.g. KRONOS and UPPAAL) [HH95, DOTY95, BLL+96]. One of the major problems in applying these tools to industrial-size systems is the huge memory-usage (e.g. [BGK+96]) needed to explore the state-space of a network (or product) of timed automata, since the verification tools must keep information not only on the control structure of the automata but also on the clock values specified by clock constraints.

Partial-order reduction (e.g., [God96, GW90, HP94, Pel93, Val90, Val93]) is a well developed technique, whose purpose is to reduce the usage of time and memory in state-space exploration by avoiding to explore unnecessary inter-

leavings of independent transitions. It has been successfully applied to finite-state systems. However, for timed systems there has been less progress. Perhaps the major obstacle to the application of partial order reduction to timed systems is the assumption that all clocks advance at the same speed, meaning that all clocks are implicitly synchronised. If each process contains (at least) one local clock, this means that advancement of the local clock of a process is not independent of time advancements in other processes. Therefore, different interleavings of a set of independent transitions will produce different combinations of clock values, even if there is no explicit synchronisation between the processes or their clocks.

A simple illustration of this problem is given in Figure 1. In (1) of Figure 1 is a system with two automata, each of which can perform one internal local transition (α_1 and α_2 respectively) from an initial local state to a synchronisation state (m, s) where the automata may synchronise on label a (we use the synchronisation model of CCS). It is clear that the two sequences of transitions $(l, r) \xrightarrow{\alpha_1} (m, r) \xrightarrow{\alpha_2} (m, s)$ and $(l, r) \xrightarrow{\alpha_2} (l, s) \xrightarrow{\alpha_1} (m, s)$ are different interleavings of two independent transitions, both leading to the state (m, s), from which a synchronisation on a is possible. A partial order reduction technique will explore only one of these two interleavings, after having analysed that the initial transitions of the two automata are independent.

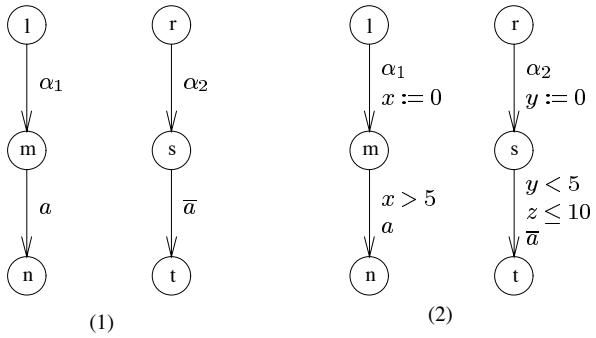


Figure 1: Illustration of Partial Order Reduction

Let us now introduce timing constraints in terms of clocks into the example, to obtain the system in (2) of Figure 1 where we add clocks x, y and z. The left automaton can initially move to node m, thereby resetting the clock x, after waiting an arbitrary time. Thereafter it can move to node n after more than 5 time units. The right automaton can initially move to node s, thereby resetting the clock y, after waiting an arbitrary time. Thereafter it can move to node t within 5 time units, but within 10 time units of initialisation of the

system. We note that the initial transitions of the two automata are logically independent of each other. However, if we naively analyse the possible values of clocks after a certain sequence of actions, we find that the sequence $(l,r) \xrightarrow{\alpha_1} (m,r) \xrightarrow{\alpha_2} (m,s)$ may result in clock values that satisfy $x \geq y$ (as x is reset before y) where the synchronisation on a is possible, whereas the sequence $(l,r) \xrightarrow{\alpha_2} (l,s) \xrightarrow{\alpha_1} (m,s)$ may result in clock values that satisfy $x \leq y$ (as x is reset after y) where the synchronisation on a is impossible. Now, we see that it is in general not sufficient to explore only one interleaving of independent transitions.

In this paper, we present a new method for partial order reductions for timed systems based on a new local-time semantics for networks of timed automata. The main idea is to overcome the problem illustrated in the previous example by removing the implicit clock synchronisation between processes by letting clocks advance independently of each other. In other words, we *desynchronise* local clocks. The benefit is that different interleavings of independent transitions will no longer remember the order in which the transitions were explored. In this specific example, an interleaving will not "remember" the order in which the clocks were reset, and the two initial transitions are independent. We can then import standard partial order techniques, and expect to get the same reductions as in the untimed case. We again illustrate this on system (2) of Figure 1. Suppose that in state (l,r) all clocks are initialised to 0. In the standard semantics, the possible clock values when the system is in state (l,r) are those that satisfy $x = y = z$. In the "desynchronised" semantics presented in this paper, any combination of clock values is possible in state (l,r). After both the sequence $(l,r) \xrightarrow{\alpha_1} (m,r) \xrightarrow{\alpha_2} (m,s)$ and $(l,r) \xrightarrow{\alpha_2} (l,s) \xrightarrow{\alpha_1} (m,s)$ the possible clock values are those that satisfy $y \leq z$.

Note that the desynchronisation will give rise to many new global states in which automata have "executed" for different amounts of time. We hope that this larger set of states can be represented symbolically more compactly than the original state-space. For example, in system (2), our desynchronised semantics gives rise to the constraint $y \leq z$ at state (m,s), whereas the standard semantics gives rise to the two constraints $x \leq y \leq z$ and $y \leq x \wedge y \leq z$. However, as we have removed the synchronisation between local time scales completely, we also lose timing information required for synchronisation between automata. Consider again system (2) and look at the clock z of the right automaton. Since $z = 0$ initially, the constraint $z \leq 10$ requires that the synchronisation on a should be within 10 time units from system initialisation. Implicitly, this then becomes a requirement on the left automaton. A naive desynchronisation of lo-

cal clocks including z will allow the left process to wait for more than 10 time units, in its local time scale, before synchronising. Therefore, before exploring the effect of a transition in which two automata synchronise, we must explicitly "resynchronise" the local time scales of the participating automata. For this purpose, we add to each automaton a local *reference clock*, which measures how far its local time has advanced in performing local transitions. To each synchronisation between two automata, we add the condition that their reference clocks agree. In the above example, we add c_1 as a reference clock to the left automaton and c_2 as a reference clock to the right automaton. We require $c_1 = c_2$ at system initialisation. After any interleaving of the first two independent transitions, the clock values may satisfy $y \leq z$ and $x - c_1 \leq z - c_2$. To synchronise on a they must also satisfy the constraint $c_1 = c_2$ in addition to $x > 5$, $y < 5$ and $z \leq 10$. This implies that $x \leq 10$ when the synchronisation occurs. Without the reference clocks, we would not have been able to derive this condition.

The idea of introducing local time is related to the treatment of local time in the field of parallel simulation (e.g., [Fuj90]). Here, a simulation step involves some local computation of a process together with a corresponding update of its local time. A snapshot of the system state during a simulation will be composed of many local time scales. In our work, we are concerned with verification rather than simulation, and we must therefore represent sets of such system states symbolically. We shall develop a symbolic version for the local-time semantics in terms of predicate transformers, in analogy with the ordinary symbolic semantics for timed automata, which is used in several tools for reachability analysis. The symbolic semantics allows a finite partitioning of the state space of a network and enjoys the desired property that two predicate transformers are independent if they correspond to disjoint transitions in different component automata. Thus we can apply standard partial order reduction techniques to the problem of checking reachability for timed systems, without disturbance from implicit synchronisation of clocks.

The paper is organised as follows: In section 2, we give a brief introduction to the notion of timed automata and its standard semantics i.e. the global time semantics. Section 3 develops a local time semantics for networks of timed automata and a *finite* symbolic version of the new semantics, analogous to the region graph for timed automata. Section 4 presents a partial order search algorithm for reachability analysis based on the symbolic local time semantics; together with necessary operations to represent and manipulate distributed symbolic states. Section 5 concludes the paper with a short summary on related work, our contribution and future work.

2 Preliminaries

2.1 Networks of Timed Automata

Timed automata was first introduced in [AD90] and has since then established itself as a standard model for timed systems. For the reader not familiar with the notion of timed automata we give a short informal description. In this paper, we will work with *networks of timed automata* [YPD94, LPY95] as the model for timed systems.

Let Act be a finite set of *labels* ranged over by a, b etc. Each label is either *local* or *synchronising*. If a is a synchronising label, then it has a *complement*, denoted $\overline{a}$, which is also a synchronising label with $\overline{\overline{a}} = a$.

A timed automaton is a standard finite–state automaton over alphabet Act, extended with a finite collection of real–valued *clocks* to model timing. We use x, y etc. to range over clocks, C and r etc. to range over finite sets of clocks, and $\mathbf{R}$ to stand for the set of non-negative real numbers.

A *clock assignment* u for a set C of clocks is a function from C to $\mathbf{R}$. For $d \in \mathbf{R}$, we use $u + d$ to denote the clock assignment which maps each clock x in C to the value $u(x) + d$ and for $r \subseteq C$, $[r \mapsto 0]u$ to denote the assignment for C which maps each clock in r to the value 0 and agrees with u on $C \backslash r$.

We use $\mathcal{B}(C)$ ranged over by g (and later by D), to stand for the set of conjunctions of atomic constraints of the form: $x \sim n$ or $x - y \sim n$ for $x, y \in C$, $\sim \in \{\leq, <, >, \geq\}$ and n being a natural number. Elements of $\mathcal{B}(C)$ are called *clock constraints* or *clock constraint systems* over C. We use $u \models g$ to denote that the clock assignment $u \in \mathbf{R}^C$ satisfies the clock constraint $g \in \mathcal{B}(C)$.

A *network of timed automata* is the parallel composition $A_1 \mid \cdots \mid A_n$ of a collection $A_1, \ldots, A_n$ of timed automata. Each A_i is a timed automaton over the clocks C_i, represented as a tuple $\langle N_i, l_i^0, E_i, I_i \rangle$, where N_i is a finite set of (control) *nodes*, $l_i^0 \in N_i$ is the *initial node*, and $E_i \subseteq N_i \times \mathcal{B}(C_i) \times Act \times 2^{C_i} \times N_i$ is a set of *edges*. Each edge $\langle l_i, g, a, r, l_i' \rangle \in E_i$ means that the automaton can move from the node l_i to the node l_i' if the clock constraint g (also called the enabling condition of the edge) is satisfied, thereby performing the label a and resetting the clocks in r. We write $l_i \xrightarrow{g,a,r} l_i'$ for $\langle l_i, g, a, r, l_i' \rangle \in E_i$. A *local action* is an edge $l_i \xrightarrow{g,a,r} l_i'$ of some automaton A_i with a local label a. A *synchronising action* is a pair of matching edges, written $l_i \xrightarrow{g_i,a,r_i} l_i' | l_j \xrightarrow{g_j,\overline{a},r_j} l_j'$ where a is a synchronising label, and for some $i \neq j$, $l_i \xrightarrow{g_i,a,r_i} l_i'$ is an edge of

A_i and $l_j \xrightarrow{g_j,\overline{a},r_j} l'_j$ is an edge of A_j. The $I_i : N_i \to \mathcal{B}(C_i)$ assigns to each node an *invariant condition* which must be satisfied by the system clocks whenever the system is operating in that node. For simplicity, we require that the invariant conditions of timed automata should be the conjunction of constraints in the form: $x \leq n$ where x is a clock and n is a natural number. We require the sets C_i to be pairwise disjoint, so that each automaton only references local clocks. As a technical convenience, we assume that the sets N_i of nodes are pairwise disjoint.

Global Time Semantics.

A *state* of a network $A = A_1|\cdots|A_n$ is a pair (l, u) where l, called a *control vector*, is a vector of control nodes of each automaton, and u is a clock assignment for $C = C_1 \cup \cdots \cup C_n$. We shall use $l[i]$ to stand for the ith element of l and $l[l'_i/l_i]$ for the control vector where the ith element l_i of l is replaced by l'_i. We define the invariant $I(l)$ of l as the conjunction $I_1(l[1]) \wedge \cdots \wedge I_n(l[n])$. The initial state of A is (l^0, u^0) where l^0 is the control vector such that $l[i] = l_i^0$ for each i, and u^0 maps all clocks in C to 0.

A network may change its state by performing the following three types of transitions.

- Delay Transition: $(l, u) \longrightarrow (l, u + d)$ if $I(l)(u + d)$
- Local Transition: $(l, u) \longrightarrow (l[l'_i/l_i], u')$ if there exists a local action $l_i \xrightarrow{g,a,r} l'_i$ such that $u \models g$ and $u' = [r \mapsto 0]u$.
- Synchronising Transition: $(l, u) \longrightarrow (l[l'_i/l_i][l'_j/l_j], u')$ if there exists a synchronising action $l_i \xrightarrow{g_i,a,r_i} l'_i | l_j \xrightarrow{g_j,\overline{a},r_j} l'_j$ such that $u \models g_i$, $u \models g_j$, and $u' = [r_i \mapsto 0][r_j \mapsto 0]u$.

We shall say that a state (l, u) is *reachable*, denoted $(l^0, u^0) \longrightarrow^* (l, u)$ if there exists a sequence of (delay or discrete) transitions leading from (l^0, u^0) to (l, u).

2.2 Symbolic Global–Time Semantics

Clearly, the semantics of a timed automaton yields an infinite transition system, and is thus not an appropriate basis for verification algorithms. However, efficient algorithms may be obtained using a *symbolic* semantics based on *symbolic states* of the form (l, D), where $D \in \mathcal{B}(C)$, which represent the set of

states (l, u) such that $u \models D$. Let us write $(l, u) \models (l', D)$ to denote that $l = l'$ and $u \models D$.

We perform symbolic state space exploration by repeatedly taking the strongest postcondition with respect to an action, or to time advancement. For a constraint D and set r of clocks, define the constraints $D^{\uparrow}$ and $r(D)$ by

- for all $d \in \mathbf{R}$ we have $u + d \models D^{\uparrow}$ iff $u \models D$, and
- $[r \mapsto 0]u \models r(D)$ iff $u \models D$

It can be shown that $D^{\uparrow}$ and $r(D)$ can be expressed as clock constraints whenever D is a clock constraint. We now define predicate transformers corresponding to strongest postconditions of the three types of transitions:

- For global delay, $sp(\delta)(l, D) \stackrel{def}{=} \left(l, D^{\uparrow} \wedge I(l)\right)$
- For a local action $l_i \stackrel{g,a,r}{\longrightarrow} l_i'$, $sp(l_i \stackrel{g,a,r}{\longrightarrow} l_i')(l, D) \stackrel{def}{=} \left(l[l_i'/l_i], r(g \wedge D)\right)$
- For a synchronising action $l_i \stackrel{g_i,a,r_i}{\longrightarrow} l_i' | l_j \stackrel{g_j,\overline{a},r_j}{\longrightarrow} l_j'$,
  $sp(l_i \stackrel{g_i,a,r_i}{\longrightarrow} l_i' | l_j \stackrel{g_j,\overline{a},r_j}{\longrightarrow} l_j')(l, D) \stackrel{def}{=} \left(l[l_i'/l_i][l_j'/l_j], (r_i \cup r_j)(g_i \wedge g_j \wedge D)\right)$

It turns out to be convenient to use predicate transformers that correspond to first executing a discrete action, and thereafter executing a delay. For predicate transformers τ_1, τ_2, we use $\tau_1; \tau_2$ to denote the composition $\tau_2 \circ \tau_1$. For a (local or synchronising) action α, we define $sp_t(\alpha) \stackrel{def}{=} sp(\alpha); sp(\delta)$.

From now on, we shall use (l^0, D^0) to denote the initial symbolic global time state for networks, where $D^0 = (\{u^0\})^{\uparrow} \wedge I(l^0)$. We write $(l, D) \Rightarrow (l', D')$ if $(l', D') = sp_t(\alpha)(l, D)$ for some action α. It can be shown (e.g. [YPD94]) that the symbolic semantics characterises the concrete semantics given earlier in the following sense:

Theorem 1 *A state (l, u) of a network is reachable if and only if $(l^0, D^0)(\Rightarrow)^* (l, D)$ for some D such that $u \models D$.*

The above theorem can be used to construct a symbolic algorithm for reachability analysis. In order to keep the presentation simple, we will in the rest of the paper only consider a special form of *local* reachability, defined as follows. Given a control node l_k of some automaton A_k, check if there is a reachable state (l, u) such that $l[k] = l_k$. It is straight-forward to extend our results to more general reachability problems. The symbolic algorithm for checking local

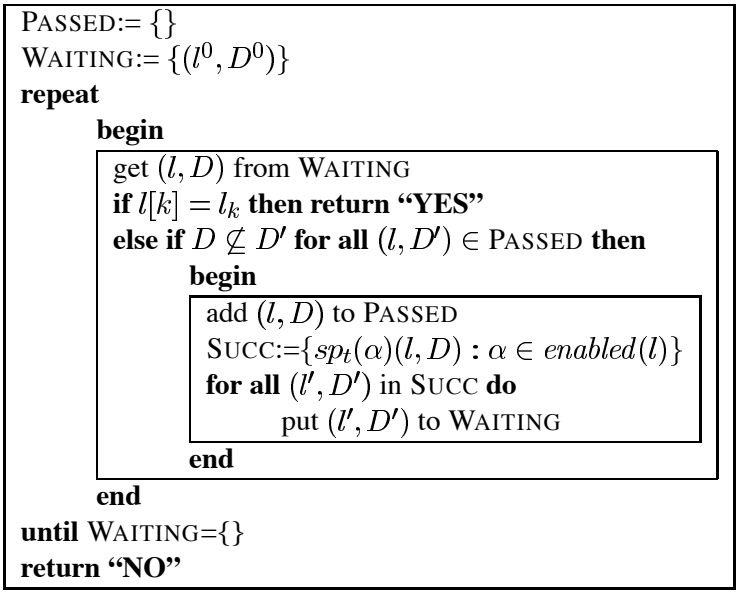


Figure 2: An Algorithm for Symbolic Reachability Analysis.

reachability is shown in Figure 2 for a network of timed automata. Here, the set $enabled(l)$ denotes the set of all actions whose source node(s) are in the control vector l i.e., a local action $l_i \xrightarrow{g,a,r} l'_i$ is enabled at l if $l[i] = l_i$, and a synchronising action $l_i \xrightarrow{g_i,a,r_i} l'_i | l_j \xrightarrow{g_j,\overline{a},r_j} l'_j$ is enabled at l if $l[i] = l_i$ and $l[j] = l_j$.

3 Partial Order Reduction and Local-Time Semantics

The purpose of partial-order techniques is to avoid exploring several interleavings of independent transitions, i.e., transitions whose order of execution is irrelevant, e.g., because they are performed by different processes and do not affect each other. Assume for instance that for some control vector l, the set $enabled(l)$ consists of the local action α_i of automaton A_i and the local action α_j of automaton A_j. Since executions of local actions do not affect each other, we might want to explore only the action α_i, and defer the exploration of α_j until later. The justification for deferring to explore α_j would be that any symbolic state which is reached by first exploring α_j and thereafter α_i can also be reached by exploring these actions in reverse order, i.e., first α_i and thereafter α_j.

Let τ_1 and τ_2 be two predicate transformers. We say that τ_1 and τ_2 are *independent* if $(\tau_1;\tau_2)((l,D)) = (\tau_2;\tau_1)((l,D))$ for any symbolic state (l,D). In the absence of time, local actions of different processes are independent, in the sense that $sp(\alpha_i)$ and $sp(\alpha_j)$ are independent. However, in the presence of time, we do not have independence. That is, $sp_t(\alpha_i)$ and $sp_t(\alpha_j)$ are in general not independent, as illustrated e.g., by the example in Figure 1.

If timed predicate transformers commute only to a rather limited extent, then partial order reduction is less likely to be successful for timed systems than for untimed systems. In this paper, we present a method for symbolic state-space exploration of timed systems, in which predicate transformers commute to the same extent as they do in untimed systems. The main obstacle for commutativity of timed predicate transformers is that timed advancement is modelled by globally synchronous transitions, which implicitly synchronise all local clocks, and hence all processes. In our approach, we propose to replace the global time-advancement steps by local-time advancement. In other words, we remove the constraint that all clocks advance at the same speed and let clocks of each automaton advance totally independently of each other. We thus replace one global time scale by a local-time scale for each automaton. When exploring local actions, the corresponding predicate transformer affects only the clocks of that automaton in its local-time scale; the clocks of other automata are unaffected. In this way, we have removed any relation between local-time scales. However, in order to explore pairs of synchronising actions we must also be able to "resynchronise" the local-time scales of the participating automata, and for this purpose we add a local *reference clock* to each automaton. The reference clock of automaton A_i represents how far the local-time of A_i has advanced, measured in a global time scale. In a totally unsynchronised state, the reference clocks of different automata can be quite different. Before a synchronisation between A_i and A_j, we must add the condition that the reference clocks of A_i and A_j are equal.

To formalise the above ideas further, we present a local-time semantics for networks of timed automata, which allows local clocks to advance independently and resynchronising them only at synchronisation points.

Consider a network $A_1|\cdots|A_n$. We add to the set C_i of clocks of each A_i a reference clock, denoted c_i. Let us denote by $u +_i d$ the time assignment which maps each clock x in C_i (including c_i) to the value $u(x) + d$ and each clock x in $C \setminus C_i$ to the value $u(x)$. In the rest of the paper, we shall assume that the set of clocks of a network include the reference clocks and the initial state is

(l^0, u^0) where the reference clock values are 0, in both the global and local time semantics.

Local Time Semantics.

The following rules define that networks may change their state locally and globally by performing three types of transitions:

- Local Delay Transition: $(l,u) \mapsto (l, u +_i d)$ if $I_i(l_i)(u +_i d)$
- Local Discrete Transition: $(l,u) \mapsto (l[l'_i/l_i], u')$ if there exists a local action $l_i \xrightarrow{g,a,r} l'_i$ such that $u \models g$ and $u' = [r \mapsto 0]u$
- Synchronising Transition: $(l,u) \mapsto (l[l'_i/l_i][l'_j/l_j], u')$ if there exists a synchronising action $l_i \xrightarrow{g_i,a,r_i} l'_i | l_j \xrightarrow{g_j,\overline{a},r_j} l'_j$ such that $u \models g_i$, $u \models g_j$, and $u' = [r_i \mapsto 0][r_j \mapsto 0]u$, and $u(c_i) = u(c_j)$

Intuitively, the first rule says that a component may advance its local clocks (or execute) as long as the local invariant holds. The second rule is the standard interleaving rule for discrete transitions. When two components need to synchronise, it must be checked if they have executed for the same amount of time. This is specified by the last condition of the third rule which states that the local reference clocks must agree, i.e. $u(c_i) = u(c_j)$.

We call (l,u) a local time state. Obviously, according to the above rules, a network may reach a large number of local time states where the reference clocks take different values. To an external observer, the interesting states of a network will be those where all the reference clocks take the same value.

Definition 1 *A local time state (l,u) with reference clocks $c_1 \cdots c_n$ is synchronised if $u(c_1) = \cdots = u(c_n)$.*

Now we claim that the local-time semantics simulates the standard global time semantics in which local clocks advance concurrently, in the sense that they can generate precisely the same set of reachable states of a timed system.

Theorem 2 *For all networks, $(l_0,u_0)(\longrightarrow)^*(l,u)$ iff for all synchronised local time states (l,u) $(l_0,u_0)(\mapsto)^*(l,u)$.*

3.1 Symbolic Local-Time Semantics

We can now define a local-time analogue of the symbolic semantics given in Section 2.2 to develop a symbolic reachability algorithm with partial order

reduction. We need to represent local time states by constraints. Let us first assume that the constraints we need for denote symbolic local time states are different from standard clock constraints, and use $\widehat{D}, \widehat{D}'$ etc to denote such constraints. Later, we will show that such constraints can be expressed as a clock constraint.

We use $\widehat{D}^{\uparrow_i}$ to denote the clock constraint such that for all $d \in \mathbf{R}$ we have $u +_i d \models \widehat{D}^{\uparrow_i}$ iff $u \models \widehat{D}$. For local-time advance, we define a *local-time predicate transformer*, denoted $\widehat{sp_t}(\delta_i)$, which allows only the local clocks C_i including the reference clock c_i to advance as follows:

- $\widehat{sp_t}(\delta_i)(l, \widehat{D}) \stackrel{def}{=} \left(l, \widehat{D}^{\uparrow_i} \wedge I(l)\right)$

For each local and synchronising action α, we define a local-time predicate transformer, denoted $\widehat{sp_t}(\alpha)$, as follows:

- If α is a local action $l_i \stackrel{g,a,r}{\longrightarrow} l_i'$, then $\widehat{sp_t}(\alpha) \stackrel{def}{=} sp(\alpha); \widehat{sp_t}(\delta_i)$
- If α is a synchronising action $l_i \stackrel{g_i,a,r}{\longrightarrow} l_i' | l_j \stackrel{g_j,\overline{a},r_j}{\longrightarrow} l_j'$, then

  $\widehat{sp_t}(\alpha) \stackrel{def}{=} \{c_i = c_j\}; sp(\alpha); \widehat{sp_t}(\delta_i); \widehat{sp_t}(\delta_j)$

Note that in the last definition, we treat a clock constraint like $c_i = c_j$ as a predicate transformer, defined in the natural way by $\{c_i = c_j\}(l, \widehat{D}) \stackrel{def}{=} (l, \widehat{D} \wedge (c_i = c_j))$.

We use $(l^0, \widehat{D}^0)$ to denote the initial symbolic local time state of networks where $\widehat{D}^0 = \widehat{sp_t}(\delta_1); \cdots; \widehat{sp_t}(\delta_n)(\{u^0\})$. We shall write $(l, \widehat{D}) \models\!\!\Rightarrow (l', \widehat{D}')$ if $(l', \widehat{D}') = \widehat{sp_t}(\alpha)(l, \widehat{D})$ for some action α.

Then we have the following characterisation theorem.

Theorem 3 *For all networks, a synchronised state* (l, u), $(l^0, u^0) \longrightarrow^* (l, u)$ *if and only if* $(l^0, \widehat{D}^0)(\models\!\!\Rightarrow)^*(l, \widehat{D})$ *for a symbolic local time state* $(l, \widehat{D})$ *such that* $u \models \widehat{D}$.

The above theorem shows that the symbolic local time semantics fully characterises the global time semantics in terms of reachable states. Thus we can perform reachability analysis in terms of the symbolic local time semantics. However, it requires to find a symbolic local time state that is *synchronised* in the sense that it contains synchronised states. The searching for such a synchronised symbolic state may be time and space-consuming. Now, we relax the condition for a class of networks, namely those containing no local time-stop.

Definition 2 *A network is local time-stop free if for all* (l,u), $(l^0,u^0)(\mapsto)^*(l,u)$ *implies* $(l,u)(\mapsto)^*(l',u')$ *for some synchronised state* (l',u').

The local time-stop freeness can be easily guaranteed by syntactical restriction on component automata of networks. For example, we may require that at each control node of an automaton there should be an edge with a local label and a guard weaker than the local invariant. This is precisely the way of modelling time-out handling at each node when the invariant is becoming false and therefore it is a natural restriction.

The following theorem allows us to perform reachability analysis in terms of symbolic local time semantics for local time-stop free networks without searching for synchronised symbolic states.

Theorem 4 *Assume a local time-stop free network* A *and a local control node* l_k *of* A_k. *Then* $(l^0,D^0)(\Rightarrow)^*(l,D)$ *for some* (l,D) *such that* $l[k]=l_k$ *if and only if* $(l^0,\widehat{D}^0)(\models\!\!\Rightarrow)^*(l',\widehat{D}')$ *for some* $(l',\widehat{D}')$ *such that* $l'[k]=l_k$.

We now state that the version of the timed predicate transformers based on local time semantics enjoy the commutativity properties that were missing in the global time approach.

Theorem 5 *Let* α_1 *and* α_2 *be two actions of a network* A *of timed automata. If the sets of component automata of* A *involved in* α_1 *and* α_2 *are disjoint, then* $\widehat{sp_l}(\alpha_1)$ *and* $\widehat{sp_l}(\alpha_2)$ *are independent*.

3.2 Finiteness of the Symbolic Local Time Semantics

We shall use the symbolic local time semantics as the basis to develop a partial order search algorithm in the following section. To guarantee termination of the algorithm, we need to establish the finiteness of our local time semantics, i.e. that the number of *equivalent* symbolic states is finite. Observe that the number of symbolic local time states is in general infinite. However, we can show that there is finite partitioning of the state space. We take the same approach as for standard timed automata, that is, we construct a finite graph based on a notion of regions.

We first extend the standard region equivalence to synchronised states. In the following we shall use C_r to denote the set of reference clocks.

Definition 3 *Two synchronised local time states (with the same control vector)* (l, u) *and* (l, u') *are synchronised-equivalent if* $([C_r \mapsto 0]u) \sim ([C_r \mapsto 0]u')$ *where* $\sim$ *is the standard region equivalence for timed automata.*

Note that $([C_r \mapsto 0]u) \sim ([C_r \mapsto 0]u')$ means that only the non-reference clock values in (l, u) and (l, u') are region-equivalent. We call the equivalence classes w.r.t. the above equivalence relation *synchronised regions*. Now we extend this relation to cope with local time states that are not synchronised. Intuitively, we want two non-synchronised states, (l, u) and (l', u') to be classified as equivalent if they can reach sets of equivalent synchronised states just by letting the automata that have lower reference clock values advance to catch up with the automaton with the highest reference clock value.

Definition 4 *A local delay transition* $(l, u) \mapsto (l, u')$ *of a network is a catch-up transition if* $max(u(C_r)) \leq max(u'(C_r))$.

Intuitively a catch-up transition corresponds to running one of the automata that lags behind, and thus making the system more synchronised in time.

Definition 5 *Let* (l, u) *be a local time state of a network of timed automata. We use* $R((l, u))$ *to denote the set of synchronised regions reachable from* (l, u) *only by discrete transitions or catch-up transitions.*

We now define an equivalence relation between local time states.

Definition 6 *Two local time states* (l, u) *and* (l', u') *are catch-up equivalent denoted* $(l, u) \sim_c (l', u')$ *if* $R((l, u) = R((l', u'))$. *We shall use* $|(l, u)|_{\sim_c}$ *to denote the equivalence class of local time states w.r.t.* $\sim_c$.

Intuitively two catch-up equivalent local time states can reach the same set of synchronised states i.e. states where all the automata of the network have been synchronised in time.

Note that the number of synchronised regions is finite. This implies that the number of catch-up classes is also finite. On the other hand, there is no way to put an upper bound on the reference clocks c_i, since that would imply that for every process there is a point in time where it stops evolving which is generally not the case. This leads to the conclusion that there must be a periodicity in the region graph, perhaps after some initial steps. Nevertheless, we have a finiteness theorem.

Theorem 6 *For any network of timed automata, the number of catch-up equivalence classes* $|(l, u)|_{\sim_c}$ *for each vector of control nodes is bounded by a func-*

tion of the number of regions in the standard region graph construction for timed automata.

As the number of vectors of control nodes for each network of automata is finite, the above theorem demonstrates the finiteness of our symbolic local time semantics.

4 Partial Order Reduction in Reachability Analysis

The preceding sections have developed the necessary machinery for presenting a method for partial-order reduction in a symbolic reachability algorithm. Such an algorithm can be obtained from the algorithm in Figure 2 by replacing the initial symbolic global time state (l^0, D^0) by the initial symbolic local time state $(l^0, \widehat{D}^0)$ (as defined in Theorem 4), and by replacing the statement

$$\text{SUCC}:=\{sp_t(\alpha)(l, D) : \alpha \in \mathit{enabled}(l)\}$$

by $\text{SUCC}:=\{\widehat{sp_t}(\alpha)(l, D) : \alpha \in \mathit{ample}(l)\}$ where $\mathit{ample}(l) \subseteq \mathit{enabled}(l)$ is a subset of the actions that are enabled at l. Hopefully the set $\mathit{ample}(l)$ can be made significantly smaller than $\mathit{enabled}(l)$, leading to a reduction in the explored symbolic state-space.

In the literature on partial order reduction, there are several criteria for choosing the set $\mathit{ample}(l)$ so that the reachability analysis is still complete. We note that our setup would work with any criterion which is based on the notion of "independent actions" or "independent predicate transformers". A natural criterion which seems to fit our framework was first formulated by Overman [Ove81]; we use its formulation by Godefroid [God96].

The idea in this reduction is that for each control vector l we choose a subset $\mathcal{A}$ of the automata $A_1, \ldots, A_n$, and let $\mathit{ample}(l)$ be all enabled actions in which the automata in $\mathcal{A}$ participate. The choice of $\mathcal{A}$ may depend on the control node l_k that we are searching for. The set $\mathcal{A}$ must satisfy the criteria below. Note that the conditions are formulated only in terms of the control structure of the automata. Note also that in an implementation, these conditions will be replaced by conditions that are easier to check (e.g. [God96]).

C0 $\mathit{ample}(l) = \emptyset$ if and only if $\mathit{enabled}(l) = \emptyset$.

C1 If the automaton $A_i \in \mathcal{A}$ from its current node $l[i]$ can possibly synchronise with another process A_j, then $A_j \in \mathcal{A}$, regardless of whether such a synchronisation is enabled or not.

C2 From l, the network cannot reach a control vector l' with $l'[k] = l_k$ without performing an action in which some process in $\mathcal{A}$ participates.

Criteria **C0** and **C2** are obviously necessary to preserve correctness. Criterion **C1** can be intuitively motivated as follows: If automaton A_i can possibly synchronise with another automaton A_j, then we must explore actions by A_j to allow it to "catch up" to a possible synchronisation with A_i. Otherwise we may miss to explore the part of the state-space that can be reached after the synchronisation between A_i and A_j.

A final necessary criterion for correctness is *fairness*, i.e., that we must not indefinitely neglect actions of some automaton. Otherwise we may get stuck exploring a cyclic behaviour of a subset of the automata. This criterion can be formulated in terms of the *global control graph* of the network. Intuitively, this graph has control vectors as nodes, which are connected by symbolic transitions where the clock constraints are ignored. The criterion of fairness then requires that

C3 In each cycle of the global control graph, there must be at least one control vector at which $ample(l) = enabled(l)$.

In the following theorem, we state correctness of our criteria.

Theorem 7 *A partial order reduction of the symbolic reachability in Figure 2, obtained by replacing*

1. *the initial symbolic global time state* (l^0, D^0) *with the initial symbolic local time state* $(l^0, \widehat{D}^0)$ *(as defined in theorem 4)*
2. *the statement* SUCC:=$\{sp_t(\alpha)(l, D) : \alpha \in enabled(l)\}$ *with the statement* SUCC:=$\{\widehat{sp_t}(\alpha)(l, D) : \alpha \in ample(l)\}$ *where the function* $ample(\cdot)$ *satisfies the criteria* **C0** - **C3**,
3. *and finally the inclusion checking i.e.* $D \not\subseteq D'$ *between constraints with an inclusion checking that also takes* $\sim_c$ *into account*[1].

is a correct and complete decision procedure for determining whether a local state l_k *in* A_k *is reachable in a local time-stop free network* A.

The proof of the above theorem follows similar lines as other standard proofs of correctness for partial order algorithms. See e.g., [God96].

[1] This last change is only to guarantee the termination but not the soundness of the algorithm. Note that in this paper, we have only shown that there exists a finite partition of the local time state space according to $\sim_c$, but not how the partitioning should be done. This is our future work.

4.1 Operations on Constraint Systems

Finally, to develop an efficient implementation of the search algorithm presented above, it is important to design efficient data structures and algorithms for the representation and manipulation of symbolic distributed states i.e. constraints over local clocks including the reference clocks.

In the standard approach to verification of timed systems, one such well-known data structure is the Difference Bound Matrix (DBM), due to Bellman [Bel57], which offers a canonical representation for *clock constraints*. Various efficient algorithms to manipulate (and analyse) DBM's have been developed (see e.g [LLPY97]).

However when we introduce operations of the form $\widehat{sp_t}(\delta_i)$, the standard clock constraints are no longer adequate for describing possible sets of clock assignments, because it is not possible to let only a subset of the clocks grow. This problem can be circumvented by the following. Instead of considering values of clocks x as the basic entity in a clock constraint, we work in terms of the relative offset of a clock from the local reference clock. For a clock $x_i^l \in C_i$, this offset is represented by the difference $x_i^l - c_i$. By analogy, we must introduce the constant offset $0 - c_i$. An *offset constraint* is then a conjunction of inequalities of form $x_i \sim n$ or $(x_i^l - c_i) - (x_j^k - c_j) \sim n$ for $x_i^l \in C_i, x_j^k \in C_j$, where $\sim \in \{\leq, \geq\}$. Note that an inequality of the form $x_i^l \sim n$ is also an offset, since it is the same as $(x_i^l - c_i) - (0 - c_i) \sim n$. It is important to notice, that given an offset constraint $(x_i^l - c_i) - (x_j^k - c_j) \sim n$ we can always recover the absolute constraint by setting $c_i = c_j$.

The nice feature of these constraints is that they can be represented by DBM's, by changing the interpretation of a clock from being its value to being its local offset. Thus given a set of offset constraints D over a C, we construct a DBM M as follows. We number the clocks in C_i by $x_i^0, \ldots, x^{|C_i|-2}, c_i$. An offset of the form $x_i^l - c_i$ we denote by $\hat{x}_i^l$ and a constant offset $0 - c_i$ by $\hat{c}_i$. The index set of the matrix is then the set of offsets $\hat{x}_i^l$ and $\hat{c}_i$ for $x_i^l, c_i \in C_i$ for all $C_i \in C$, while an entry in M is defined by $M(\hat{x}, \hat{y}) = n$ if $\hat{x} - \hat{y} \leq n \in D$ and $M(\hat{x}, \hat{y}) = \infty$ otherwise. We say that a clock assignment u is a solution of a DBM M, $u \models M$, iff $\forall x, y \in C : \quad u(\hat{x}) - u(\hat{y}) \leq M(\hat{x}, \hat{y})$, where $u(\hat{x}) = u(x) - u(c_i)$ with c_i the reference clock of x.

The operation $D^{\uparrow i}$ now corresponds to the deletion of all constraints of the form $\hat{c}_i \geq \hat{x} + n$. The intuition behind this is that when we let the clocks in i grow, we are keeping the relative offsets $\hat{x}_i^k$ constant, and only the clock $\hat{c}_i$ will decrease,

because this offset is taken from 0. $D^{\uparrow_i}$ can be defined as an operation on the corresponding DBM M: $M^{\uparrow_i}(\hat{x},\hat{y}) = \infty$ if $\hat{y} = \hat{c}_i$ and $M^{\uparrow_i}(\hat{x},\hat{y}) = M(\hat{x},\hat{y})$ otherwise. It then easy to see that $u \models M$ iff $u +_i d \models M^{\uparrow_i}$.

Resetting of a clock x_i^k corresponds to the deletion of all constraints regarding $\hat{x}_i^k$ and then setting $\hat{x}_i^k - \hat{c}_i = 0$. This can be done by an operation $[x_i^k \rightarrow 0](M)(\hat{x},\hat{y}) = 0$ if $\hat{x} = \hat{x}_i^k$ and $\hat{y} = \hat{c}_i$ or $\hat{x} = \hat{c}_i$ and $\hat{y} = \hat{x}_i^k$, ∞ if $\hat{x} = \hat{x}_i^k$ and $\hat{y} \neq \hat{c}_i$ or $\hat{x} \neq \hat{c}_i$ and $\hat{y} = \hat{x}_i^k$, and $M(\hat{x},\hat{y})$ otherwise. Again it is easy to see, that $[x_i^k \rightarrow 0]u \models [x_i^k \rightarrow 0](M)$ iff $u \models M$.

5 Conclusion and Related Work

In this paper, we have presented a partial-order reduction method for timed systems, based on a *local-time* semantics for networks of timed automata. We have developed a symbolic version of this new (local time) semantics in terms of predicate transformers, in analogy with the ordinary symbolic semantics for timed automata which is used in current tools for reachability analysis. This symbolic semantics enjoys the desired property that two predicate transformers are independent if they correspond to disjoint transitions in different processes. This allows us to apply standard partial order reduction techniques to the problem of checking reachability for timed systems, without disturbance from implicit synchronisation of clocks. The advantage of our approach is that we can avoid exploration of unnecessary interleavings of independent transitions. The price is that we must introduce extra machinery to perform the resynchronisation operations on local clocks. On the way, we have established a theorem about finite partitioning of the state space, analogous to the region graph for ordinary timed automata. For efficient implementation of our method, we have also presented a variant of DBM representation of symbolic states in the local time semantics. We should point out that the results of this paper can be easily extended to deal with shared variables by modifying the predicate transformer in the form $c_i = c_j$) for clock resynchronisation to the form $c_i \leq c_j$ properly for the reading and writing operations. Future work naturally include an implementation of the method, and experiments with case studies to investigate the practical significance of the approach.

Related Work

Currently we have found in the literature only two other proposals for partial order reduction for real time systems: The approach by Pagani in [Pag96] for

timed automata (timed graphs), and the approach of Yoneda et al. in [YSSC93, YS97] for time Petri nets.

In the approach by Pagani a notion of independence between transitions is defined based on the global-time semantics of timed automata. Intuitively two transitions are independent iff we can fire them in any order and the resulting states have the same control vectors and clock assignments. When this idea is lifted to the symbolic semantics, it means that two transitions can be independent only if they can happen in the same global time interval. Thus there is a clear difference to our approach: Pagani's notion of independence requires the comparison of clocks, while ours doesn't.

Yoneda et al. present a partial order technique for model checking a timed LTL logic on time Petri nets [BD91]. The symbolic semantics consists of constraints on the differences on the possible firing times of enabled transitions instead of clock values. Although the authors do not give an explicit definition of independence (like our Theorem 5) their notion of independence is structural like ours, because the persistent sets, ready sets, are calculated using the structure of the net. The difference to our approach lies in the calculation of the next state in the state-space generation algorithm. Yoneda et al. store the relative firing order of enabled transitions in the clock constraints, so that a state implicitly remembers the history of the system. This leads to branching in the state space, a thing which we have avoided. A second source of branching in the state space is synchronisation. Since a state only contains information on the relative differences of firing times of transitions it is not possible to synchronise clocks.

Acknowledgement: We would like to thank Paul Gastin, Florence Pagani and Stavros Tripakis for their valuable comments and discussions.

References

[AD90] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of of International Colloquium on Algorithms, Languages and Programming*, volume 443 of *LNCS*, pages 322–335. Springer Verlag, 1990.

[BD91] B. Berthomieu and M. Diaz. Modelling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

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

[BGK+96] J. Bengtsson, D. Griffioen, K. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In *Proc. of 9th Int. Conf. on Computer Aided Verification*, volume 1102 of *LNCS*, pages 244–256. Springer Verlag, 1996.

[BLL+96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Proc. of the 2nd Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 431–434. Springer Verlag, 1996.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, volume 1066 of *LNCS*, pages 208–219. Springer Verlag, 1995.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer Verlag, 1996.

[GW90] P. Godefroid and P. Wolper. Using partial orders to improve automatic verification methods. In *Proc. of Workshop on Computer Aided Verification*, 1990.

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

[HP94] G. J. Holzmann and D. A. Peled. An improvement in formal verification. In *Proc. of the 7th International Conference on Formal Description Techniques*, pages 197–211, 1994.

[LLPY97] F. Larsson, K. G. Larsen, P. Pettersson, and W. 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, December 1997.

[LPY95] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 76–87, December 1995.

[Ove81] W. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, UCLA, Aug. 1981.

[Pag96] F. Pagani. Partial orders and verification of real-time systems. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 327–346. Springer Verlag, 1996.

[Pel93] D. Peled. All from one, one for all, on model-checking using representatives. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *LNCS*, pages 409–423. Springer Verlag, 1993.

[Val90] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *LNCS*, pages 491–515. Springer Verlag, 1990.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *LNCS*, pages 59–70, 1993.

[YPD94] W. Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7th International Conference on Formal Description Techniques*, 1994.

[YS97] T. Yoneda and H. Schlingloff. Efficient verification of parallel real-time systems. *Journal of Formal Methods in System Design*, 11(2):187–215, 1997.

[YSSC93] T. Yoneda, A. Shibayama, B.-H. Schlingloff, and E. M. Clarke. Efficient verification of parallel real-time systems. In *Proc. of 5th Int. Conf. on Computer Aided Verification*, volume 697 of *LNCS*, pages 321–332. Springer Verlag, 1993.

Paper E:

Automated Verification of an Audio-Control Protocol using UPPAAL

Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi.

Automated Verification of an Audio-Control Protocol using UPPAAL

Johan Bengtsson[1] *W. O. David Griffioen*[2] *Kåre J. Kristoffersen*[3] *Kim G. Larsen*[3] *Fredrik Larsson*[1] *Paul Pettersson*[1] *Wang Yi*[1]

[1] Department of Computer Systems, Uppsala University, Sweden.
Email: {johanb,fredrikl,paupet,yi}@docs.uu.se.
[2] CWI, Amsterdam, The Netherlands. Email: griffoe@cwi.nl.
[3] BRICS, Aalborg University, Denmark. Email: {jelling,kgl}@cs.auc.dk.

Abstract. In this paper we present a case-study in which the tool UPPAAL is extended and applied to verify an Audio-Control Protocol developed by Philips. The size of the protocol studied in this paper is significantly larger than case studies, including various abstract versions of the same protocol without bus-collision handling, reported previously in the community of real-time verification. We have checked that the protocol will function correctly if the timing error of its components is bound to $\pm 5\%$, and incorrectly if the error is $\pm 6\%$. In addition, using UPPAAL's ability of generating diagnostic traces, we have studied an erroneous version of the protocol actually implemented by Philips, and constructed a possible execution sequence explaining the error.

During the case-study, UPPAAL was extended with the notion of *committed locations*. It allows for accurate modelling of atomic behaviours, and more importantly, it is utilised to guide the state-space exploration of the model checker to avoid exploring unnecessary interleavings of independent transitions. Our experimental results demonstrate considerable time and space-savings of the modified model checking algorithm. In fact, due to the huge time and memory-requirement, it was impossible to check a simple reachability property of the protocol before the introduction of committed locations, and now it takes only seconds.

1 Introduction

In the past decade a number of tools for automatic verification of hybrid and real-time systems have emerged, e.g. HYTECH [HHWT97], KRONOS [Yov97], PMC [ST01], RT-Cospan [AK95] and UPPAAL [LPY97a]. These tools have by now reached a state, where they are mature enough for industrial applications.

In this paper, we substantiate the claim by reporting on an industry-size case study where the tool UPPAAL is applied.

We analyse an audio control protocol developed by Philips for the physical layer of an interface bus connecting the various devices e.g. CD-players, amplifier etc. in audio equipments. It uses Manchester encoding to transmit bit sequences of arbitrary length between the components, whose timing errors are bound. A simplified version of the protocol is studied by Bosscher et.al. [BPV94]. It is showed that the protocol is incorrect if the timing error of the components is $\pm\frac{1}{17}$ or greater. The proof is carried out without tool support. The first automatic analysis of the protocol is reported in [HWT95] where HYTECH is applied to check an abstract version of the protocol and automatically synthesise the upper bound on the timing error. Similar versions of the protocol have been analysed by other tools, e.g. UPPAAL [LPY97a] and KRONOS [Yov97]. However, all the proofs are based on a simplification on the protocol, introduced by Bosscher *et.al.* in 1994, that only one sender is transmitting on the bus so that no bus collisions can occur. In many applications the bus will have more than one sender, and the full version of the protocol by Philips therefore handles bus collisions. The protocol with bus collision handling was manually verified in [Gri94] without tool support. Since 1994, it has been a challenge for the verification tool developers to automate the analysis on the full version of the protocol.

The first automated proof of the protocol with bus collision handling was presented in 1996 in the conference version of this paper [BGK+96]. It was the largest case study, reported in the literature on verification of timed systems, which has been considered as a primary example in the area (see [CW96, LSW97]). The size of the protocol studied is significantly larger than various simplified versions of the same protocol studied previously in the community, e.g. the discrete part of the state space (the node-space) is 10^3 times larger than in the case without bus collision handling and the number of clocks, variables and channels in the model is also increased considerably.

The major problem in applying automatic verification tools to industrial-size systems is the huge time and memory-usage needed to explore the state-space of a network (or product) of timed automata, since the verification tools must keep information not only on the control structure of the automata but also on the clock values specified by clock constraints. It is known as the state–space explosion problem. We experienced the problem right on the first attempt in checking a simple reachability property of the protocol using UPPAAL, which did not terminate in hours though it was installed on a super computer with giga-

bytes of main memory. We observed that in addition to the size and complexity of the problem itself, one of the main causes to the explosion was the inaccurate modelling of atomic behaviours and inefficient search of the unnecessary interleavings of atomic behaviours by the tool. As a simple solution, during the case-study, UPPAAL was extended with the notion of *committed locations*. It allows for accurate modelling of atomic behaviours, and more importantly, it is utilised in the state-space exploration of the model checker to avoid exploring unnecessary interleavings of independent transitions. Our experimental results demonstrate that the modified model-checking algorithm consume less time and space than the original algorithm. In fact, due to the huge time and memory-requirement, it was impossible to check certain properties of the protocol before the introduction of committed locations, and now it takes only seconds.

The automated analysis was originally carried out using an UPPAAL version extended with the notion of committed location installed on a super computer, a SGI ONYX machine [BGK$^+$96]. To make a comparison, in this paper we present an application of the current version (version 3.2) of UPPAAL, also supporting committed location, installed on an ordinary Pentium II 375 MHz PC machine, to the protocol. We have checked that the protocol will function correctly if the timing error of its components is bound to $\pm 5\%$, and incorrectly if the error is $\pm 6\%$. In addition, using UPPAAL's ability of generating diagnostic traces, we have studied an erroneous version of the protocol actually implemented by Philips in their audio products, and constructed a possible execution sequence explaining a known error.

The paper is organised as follows: In the next two sections we present the UPPAAL model with committed location and describe its implementation in the tool. In section 4 and 5 the Philips Audio-Control Protocol with Bus Collision is informally and formally described. The analysis of the protocol is presented in section 6 where we also compare the performance of the current UPPAAL version with the one used in [BGK$^+$96]. Section 7 concludes the paper. Finally, formal descriptions of the protocol components are enclosed in the appendix.

2 Committed Locations

The basis of the UPPAAL model for real-time systems is networks of timed automata extended with data variables [AD90, HNSY94, YPD94]. However, to meet requirements arising from various case-studies, the UPPAAL model has

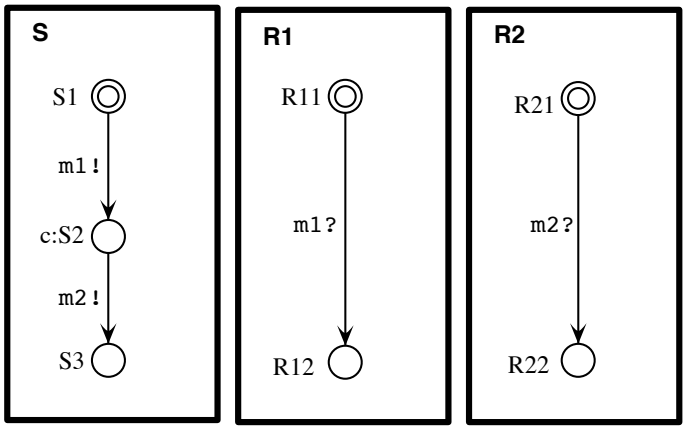


Figure 1: Broadcasting Communication and Committed Locations.

been extended with various new features such as urgent transitions [BLL+95] etc. The present case-study indicates that we need to further extend the UPPAAL model with *committed locations* to model atomic behaviours such as multi-way synchronisations and atomic broadcasting in real-time systems. Our experiences with UPPAAL show that the notion of committed locations introduced in UPPAAL is not only useful in modelling but also yields significant improvements in performance.

We assume that a real-time system consists of a fixed number of sequential processes communicating with each other via channels. We further assume that each communication synchronises two processes as in CCS [Mil89]. Broadcasting communication can be implemented in such systems by repeatedly sending the same message to all the receivers. To ensure atomicity of such "broadcast" sequences we mark the intermediate locations of the sender, which are to be executed immediately, as so-called *committed locations*.

2.1 An Example

To introduce the notion of committed locations in timed automata, consider the scenario shown in Figure 1. A sender S is to broadcast a message m to two receivers R_1 and R_2. As this requires synchronisation between *three* processes this can not directly be expressed in the UPPAAL model, where synchronisation is between two processes with complementary actions. As an initial attempt we may model the broadcast as a sequence of two two-process synchronisations,

where first S synchronises with R_1 on m_1 and then with R_2 on m_2. However, this is not an accurate model as the intended atomicity of the broadcast is not preserved (i.e. other processes may interfere during the broadcast sequence). To ensure atomicity, we mark the intermediate location S_2 of the sender S as a *committed location* (indicated by the c:-prefix). The atomicity of the action sequence $\mathsf{m}_1!\mathsf{m}_2!$ is now achieved by insisting that a committed sequence must be left immediately! This behaviour is similar to what has been called "urgent transitions" [HHWT95, DY95, BLL+95], which insists that the next transition taken must be an action (and not a delay), but the essential difference is that no other actions should be performed in between such an atomic sequence. The precise semantics of committed locations will be formalised in the transition rules for networks of timed automata with data variables in Section 2.3.

2.2 Syntax

We assume a finite set of clock variables $\mathcal{C}$ ranged over by x, y, z and a finite set of data variables $\mathcal{D}$ ranged over by i, j. We use $\mathcal{B}(\mathcal{C})$ to stand for the set of *clock constraints* that are the conjunctive formulas of simple constraints in the form of $x \prec n$ or $x - y \prec n$, where $\prec \in \{<, \leq, =, \geq, >\}$ and n is a natural number. Similarly, we use $\mathcal{B}(\mathcal{D})$ to stand for the set of *non-clock constraints* that are conjunctive formulas of $i \sim j$ or $i \sim k$, where $\sim \in \{<, \leq, =, \neq, \geq, >\}$ and k is an integer number. We use $\mathcal{B}(\mathcal{C},\mathcal{D})$ ranged over by g to denote the set of formulas that are conjunctions of clock constraints and a non-clock constraints. The elements of $\mathcal{B}(\mathcal{C},\mathcal{D})$ are called *constraints* or *guards*.

To manipulate clock and data variables, we use reset-sets which are finite sets of reset-operations. A reset-operation on a clock variable should be in the form $x := n$ where n is a natural number and a reset-operation on an data variable should be in the form: $i := k * j + k'$ where k, k' are integers. A reset-set is a *proper* reset-set when the variables are assigned a value at most once, we use $\mathcal{R}$ to denote the set of all proper reset-sets.

We assume that processes synchronise with each other via complementary actions. Let $\mathcal{A}$ be a set of action names with a subset $\mathcal{U}$ of urgent actions on which processes should synchronise whenever possible. We use $\mathcal{A}ct = \{\ \alpha? \mid \alpha \in \mathcal{A}\ \} \cup \{\ \alpha! \mid \alpha \in \mathcal{A}\ \} \cup \{\ \tau\ \}$ to denote the set of actions that processes can perform to synchronise with each other, where τ is a distinct symbol representing internal actions. We use $\text{name}(a)$ to denote the action name of a, defined by $\text{name}(\alpha?) = \text{name}(\alpha!) = \alpha$.

An automaton A over actions $\mathcal{A}ct$, clock variables $\mathcal{C}$ and data variables $\mathcal{D}$ is a tuple $\langle N, l_0, \longrightarrow, I, N_C \rangle$ where N is a finite set of locations (control-locations) with a subset $N_C \subseteq N$ being the set of committed locations, l_0 is the initial location, $\longrightarrow \subseteq N \times \mathcal{B}(\mathcal{C}, \mathcal{D}) \times \mathcal{A}ct \times \mathcal{R} \times N$ corresponds to the set of edges, and $I : N \mapsto \mathcal{B}(\mathcal{C})$ is the invariant assignment function. To model urgency, we require that the guard of an edge with an urgent action is a non-clock constraint, i.e. if name$(a) \in \mathcal{U}$ and $\langle l, g, a, r, l' \rangle \in \longrightarrow$ then $g \in \mathcal{B}(\mathcal{D})$.

In the case, $\langle l, g, a, r, l' \rangle \in \longrightarrow$ we shall write $l \xrightarrow{g\,a\,r} l'$ which represents a transition from the location l to the location l' with guard g, action a to be performed, and a sequence of reset-operations r to update the variables. Furthermore, we shall write $C(l)$ whenever $l \in N_C$.

To model networks of processes, we introduce a CCS-like parallel composition operator for automata. Assume that $A_1, ..., A_n$ are automata. We use $\overline{A}$ to denote their parallel composition. The intuitive meaning of $\overline{A}$ is similar to the CCS parallel composition of $A_1, ..., A_n$ with *all* actions being restricted, that is, $\overline{A} = (A_1|...|A_n)\backslash\mathcal{A}ct$. Thus only synchronisation between the components A_i is possible. We call $\overline{A}$ a *network of automata*. We simply view $\overline{A}$ as a vector and use A_i to denote its ith component.

2.3 Semantics

Informally, a process modelled by an automaton starts at location l_0 with all its variables initialised to 0. The values of the clocks may increase synchronously with time at location l as long as the invariant condition $I(l)$ is satisfied. At any time, the process can change location by following an edge $l \xrightarrow{g\,a\,r} l'$ provided the current values of the variables satisfy the enabling condition g. With this transition, the variables are updated by r.

To formalise the semantics we shall use variable assignments. A *variable assignment* is a mapping which maps clock variables $\mathcal{C}$ to the non-negative reals and data variables $\mathcal{D}$ to integers. For a variable assignment u and a delay d, $u \oplus d$ denotes the variable assignment such that $(u \oplus d)(x) = u(x) + d$ for a clock variable x and $(u \oplus d)(i) = u(i)$ for any data variable i. This definition of $\oplus$ reflects that all clocks proceed at the same speed and that data variables are time-insensitive.

For a reset-set r (a proper set of reset-operations), we use $r[u]$ to denote the variable assignment u' with $u'(w) = \mathit{Value}(e)_u$ whenever $(w := e) \in r$ and $u'(w') = u(w')$ otherwise, where $\mathit{Value}(e)_u$ denotes the value of e in u. Given

a constraint $g \in \mathcal{B}(\mathcal{C}, \mathcal{D})$ and a variable assignment u, $g(u)$ is a boolean value describing whether g is satisfied by u or not.

A *control vector* $\bar{l}$ of a network $\overline{A}$ is a vector of locations where l_i is a location of A_i. We write $\bar{l}[l'_i/l_i]$ to denote the vector where the ith element l_i of $\bar{l}$ is replaced by l'_i. Furthermore, we shall write $C(\bar{l})$ whenever $C(l_i)$ for some i.

A *state* of a network $\overline{A}$ is a configuration $(\bar{l}, u)$ where $\bar{l}$ is a control vector of $\overline{A}$ and u is a variable assignment. The initial state of $\overline{A}$ is $(\bar{l}^0, u^0)$ where $\bar{l}^0$ is the initial control vector whose elements are the initial locations l_i^0 of A_i's and u^0 is the initial variable assignment that maps all variables to 0.

The *semantics of a network* of automata $\overline{A}$ is given in terms of a transition system with the set of states being the configurations. The transition relation is defined by the following three rules, which are standard except that each rule has been augmented with conditions handling control-vectors with committed locations:

- $(\bar{l}, u) \rightsquigarrow (\bar{l}[l'_i/l_i], r_i[u])$ if $l_i \xrightarrow{g_i\, \tau\, r_i} l'_i$ and $g_i(u)$ for some l_i, g_i, r_i, and for all k if $C(l_k)$ then $C(l_i)$.

- $(\bar{l}, u) \rightsquigarrow (\bar{l}[l'_i/l_i, l'_j/l_j], (r_j \cup r_i)[u])$ if $l_i \xrightarrow{g_i\, \alpha!\, r_i} l'_i$, $l_j \xrightarrow{g_j\, \alpha?\, r_j} l'_j$, $g_i(u)$, $g_j(u)$, and $i \neq j$, for some l_i, l_j, g_i, g_j, α, r_i, r_j, and for all k if $C(l_k)$ then $C(\bar{l}_i)$ or $C(\bar{l}_j)$.

- $(\bar{l}, u) \rightsquigarrow (\bar{l}, u \oplus d)$ if $I(\bar{l})(u)$, $I(\bar{l})(u \oplus d)$, $\neg C(\bar{l})$ and no $l_i \xrightarrow{g_i \alpha? r_i}$, $l_j \xrightarrow{g_j \alpha! r_j}$ such that $g_i(u)$, $g_j(u)$, $\alpha \in \mathcal{U}$, $i \neq j$, l_i, l_j, r_i and r_j.

where $I(\bar{l}) = \bigwedge_i I(l_i)$.

Intuitively, the first rule describes a local internal action transition in a component, and possibly the resetting of variables. An internal transition can occur if the current variable assignment satisfies the transition guard and if the control-location of any component is committed, only components in committed locations may take local transitions. Thus, only internal transitions of components in committed location may interrupt other components operating in committed locations.

The second rule describes synchronisation transitions that synchronise two components. If the control-location of any of the components is committed it is required that at least one of the synchronising components starts in a committed location. This requirement prevents transitions starting in non-committed

locations from interfering with atomic (i.e. committed) transition sequences. However, two independent committed sequences may interfere with each other.

The third rule describes delay transitions, i.e. when all clocks increase synchronously with time. Delay transitions are permitted only while the location invariants of all components are satisfied. Delays are not permitted if the control-location of a component in the network is committed, or if an urgent transition (i.e. a synchronisation transition with urgent action) is possible. Note that the guards on urgent transitions are non-clock constraints whose truth-values are not affected by delays.

Finally, we note that the three rules give a semantics where transition sequences marked as committed are *instantaneous* in the sense that they happen without duration, and without interference from components operating in non-committed locations.

3 Committed Locations in UPPAAL

In this section we present a modified version of the model-checking algorithm of UPPAAL for networks of automata with committed locations.

3.1 The Model-Checking Algorithm

The model-checking algorithm performs reachability analysis to check for invariance properties $\forall\Box\beta$, and reachability properties $\exists\Diamond\beta$, with respect to a local property β of the control locations and the values of the clock and data variables[1]. It combines symbolic techniques with on-the-fly generation of the state-space in order to avoid explicit construction of the product automaton and the immediately caused memory problems. The algorithm is based on a partitioning of the (otherwise infinite) state-space into finitely many symbolic states of the form $(\bar{l}, D)$, where D is a constraint system (i.e. a conjunction of clock constraints and non-clock constraints). It checks if a any part of a symbolic state $(\bar{l}^f, D^f)$ (i.e. a state (l_f, u_f) with $u_f \subseteq D_f$) is reachable from the initial symbolic state $(\bar{l}^0, D^0)$, where D^0 expresses that all clock and data variables are initialised to 0 [YPD94]. Throughout the rest of this paper we shall simply call $(\bar{l}, D)$ a state instead of symbolic state.

[1] From version 3.2 released in 2001, the model-checking algorithm in UPPAAL also supports liveness properties of the kind $\forall\Diamond\beta$ and $\exists\Box\beta$.

The algorithm essentially performs a forwards search of the state-space. The search is guided and pruned by two buffers: WAITING, holding states waiting to be explored and PASSED holding states already explored. Initially, PASSED is empty and WAITING holds the single state $(\bar{l}^0, D^0)$. The algorithm then repeats the following steps:

S1. Pick a state $(\bar{l}, D)$ from the WAITING buffer.

S2. If $\bar{l} = l^f$ and $D \wedge D^f \neq \emptyset$ return the answer *yes*.

S3. a. If $\bar{l} = \bar{l}'$ and $D \subseteq D'$, for some $(\bar{l}', D')$ in the PASSED buffer, drop $(\bar{l}, D)$ and go to step S1.

b. Otherwise, save $(\bar{l}, D)$ in the PASSED buffer.

S4. Find all successor states $(\bar{l}_s, D_s)$ reachable from $(\bar{l}, D)$ in one step and store them in the WAITING buffer.

S5. If the WAITING buffer is not empty then go to step S1, otherwise return the answer *no*.

We will not treat the algorithm in detail here, but refer the reader to [YPD94, BL96].

Note that in step S3.b all explored states are stored in the PASSED buffer to ensure termination of the algorithm. In many cases, it will store the whole state-space of the analysed system which grows exponentially both in the number clocks and components [YPD94]. The algorithm is therefore bound to run into space problems for large systems. The key question is how to reduce the growth of the PASSED buffer.

When committed locations are used to model atomic behaviours there are two potential possibilities to reduce the size of the PASSED buffer. First, as atomic sequences in general restrict the amount of interleaving that is allowed in a system [Hol91], the state-space of the system is reduced, and consequently also the number of states stored in the PASSED buffer. Secondly, as a sequence of committed locations semantically is instantaneous and non-interleaved with other components, it suffices to save only the (non-committed) control-location at the beginning of the sequence in the PASSED buffer to ensure termination. Hence, our proposed solution is simply *not* to save states in the PASSED buffer which involve *committed* locations. We modify step S3 of the algorithm in the following way:

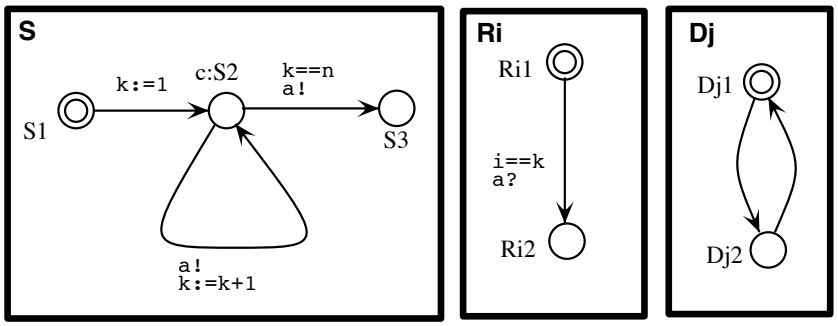


Figure 2: Broadcasting Using Committed Locations.

S3′. a. If $C(\bar{l})$ go directly to step S4.

b. If $\bar{l} = \bar{l}'$ and $D \subseteq D'$, for some $(\bar{l}', D')$ in the PASSED buffer, drop $(\bar{l}, D)$ and go to step S1.

c. If neither of the above steps are applicable, save $(\bar{l}, D)$ in the PASSED buffer.

So, for a given state $(\bar{l}, D)$, if $\bar{l}$ is committed the algorithm proceeds directly from step S3′.a to step S4, thereby omitting the time-consuming step S3′.b and the space-consuming step S3′.c. Clearly, this will reduce the growth of the PASSED buffer and the total amount of time spent on step S3′. In the following step S4 more reductions are made as interleavings are not allowed when $\bar{l}$ is committed. In fact, the next transition must be an action transition and it must involve a l_i which is committed in $\bar{l}$ (according to the transition rules in the previous section). This reduces the time spent on generating successor states of $(\bar{l}, D)$ in S4 as well as the total number of states in the system. Finally, we note that reducing the PASSED buffer size also yields potential time-savings in step S3′.b when $\bar{l}$ is *not* committed as it involves a search through the PASSED buffer.

It should be noticed that the algorithm presented in this section is not guaranteed to terminate if the notion of committed locations is used in an unintended way[2]. For the modified algorithm to terminate, it is assumed in the that committed locations are used to model atomic behaviours. In particular this means that any sequence of committed control-locations must be of finite length.

[2]In the current implementation of UPPAAL, the algorithm uses a technique presented in [LLPY97] to identify and store at least one so-called *covering state* in each *dynamic loop* to guarantee termination for all input models.

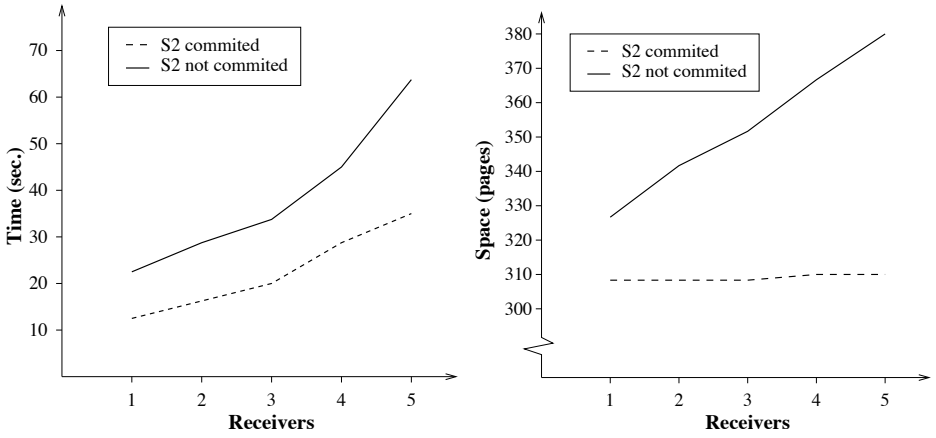


Figure 3: Time and Space Consumption.

3.2 Space and Time Performance Improvements

To investigate the practical benefits from the usage of committed locations and its implementation in UPPAAL we perform an experiment with a parameterisable scenario, where a sender S wants to broadcast a message to n receivers $\mathsf{R}_1, \ldots, \mathsf{R}_n$. The sender S simply performs n a!-transitions and then terminates, whereas the receivers are all willing to perform a single a?-transition hereby synchronising with the sender. The data variable k ensures that the ith receiver participates in the ith handshake. Additionally, there are m auxiliary automata $\mathsf{D}_1, \ldots, \mathsf{D}_m$ simply oscillating between two states. Consider Figure 2, where the control node S_2 is committed (indicated by the c:-prefix).

We may now use UPPAAL to verify that the sender succeeds in broadcasting the message, i.e. it forces all the receivers to terminate. More precisely we verify that $\mathsf{SYS}_{n,m} = (\ \mathsf{S}_n \mid \mathsf{R}_1 \mid \ldots \mid \mathsf{R}_n \mid \mathsf{D}_1 \mid \ldots \mid \mathsf{D}_m)$ satisfies the formula $\exists\Diamond(\mathsf{at(S,S}_3) \ \wedge_{i=1}^{n} \mathsf{at(R}_i\mathsf{,R}_{i2}))$, where we assume that the proposition at(A,l) is implicitly assigned to each location l of the automaton A, meaning that the component A is operating in location l. We perform two verifications, one with S_2 declared as committed, and one with S_2 being non-committed but with a location invariant $\mathsf{x} \leq 0$, where x is a clock which is reset on the transition from S1 to S2, preventing the automaton from delaying in location S_2. The result is shown in Figure 3. In both test sequences the number of disturbing automata was fixed to eight. Time is measured in seconds and space is measured in pages (4KB). The general observation is that use of committed locations in broadcasting saves time as well as space. The most important observation is

that in the committed scenario the space consumption behaves as a constant function in the number of receivers.

4 The Audio Control Protocol with Bus Collision

In this section an informal introduction to the audio protocol with bus collision is given. The audio control protocol is a bus protocol, all messages are received by all components on the bus. If a component receives a message not addressed to it, the message is just ignored. Philips allows up to 10 components.

Messages are transmitted using Manchester encoding. Time is divided into bit-slots of equal length, a bit "1" is transmitted by an up-going edge halfway a bit-slot, a bit "0" by a down-going edge halfway a bit-slot. If the same bit is transmitted twice in a row the voltage must of course change at the end of the first bit-slot. Note that only a single wire is used to connect the components, no extra clock wire is needed. This is one of the properties that makes it a useful protocol.

The protocol has to cope with some problems: (a) The sender and the receiver must agree on the beginning of the first bit-slot, (b) the length of the message is not known in advance by the receiver, (c) the down-going edges are not detected by the receiver. To resolve these problems the following is required: Messages must start with a bit "1" and messages must end with a down-going edge. This ensures that the voltage on the wire is low between messages. Furthermore the senders must respect a so-called "radio silence" between the end of a message and the beginning of the next one. The radio silence marks the end of a message and the receiver knows that the next up-going edge is the first edge of a new message. It is almost possible, and actually mandated in the Philips documentation, to decode a Manchester encoded message by only looking to the up-going edges (problem c) only the last zero bit of a message can not be detected (consider messages "10" and "1"). To resolve this, it is required that all messages are of odd length.

It is possible that two or more components start transmitting at the same time. The behaviour of the electric circuit is such that the voltage on the wire will be high as long as one of the senders pulls it high. In other words: The wire implements the or-function. This makes it possible for a sender to notice that someone else is also transmitting. If the wire is high while it is transmitting a low, a sender can detect a bus collision. This collision detection happens at certain points in time: Just before each up-going transition, and at one and three

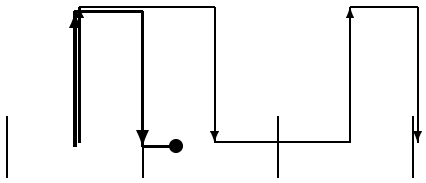


Figure 4: An Example.

quarters of a bit-slot after a down going edge (if it is still transmitting a low). When a sender detects a collision it will stop transmitting and will try to re-transmit its message later.

If two messages are transmitted at the same time and one is a prefix of the other, the receiver will not notice the prefix message. To ensure collision detection it is not allowed that a message is a prefix of another message in transit. In the Philips environment this restriction is met by embedding the source address in each message (and assigning each component a unique source address).

In Figure 4 an example is depicted. Assume two senders, named A and B, that start transmitting at exactly the same time. Because two lines on top of each other are hard to distinguish from one line, in the picture they are shifted slightly. The sender A (depicted with thick lines) starts transmitting "11..." and sender B (depicted with thin lines) "101...". At the end of the first bit-slot sender A changes from high to low voltage, to prepare for the next up-going edge. But one quarter after this down it detects a collision and stops transmitting. Sender B did not notice the other sender and continues transmitting. Note that the receiver will decode the message of the sender B correctly.

The protocol has to cope with one more thing: timing uncertainty. Because the protocol is implemented on a processor that also has to execute a number of other time critical tasks, a quite large timing uncertainty is allowed. A bit-slot is 888 microseconds, so the ideal time between two edges is 888 or 444 microseconds. On the generation of edges a timing uncertainty of $\pm 5\%$ is allowed. That is, between 844 and 932 for one bit-slot and between 422 and 466 for half a bit-slot. The collision detection just before an up-going edge and the actual generation of the same up-going edge should be separated by at most 20 microseconds (according to the protocol specification). The timing uncertainty on the collision detection appearing at the first and third quarters after a down-going edge is ± 22 microseconds. Also the receiver has a timing uncertainty of $\pm 5\%$. To complete the timing information, the distance between the end of one message and the beginning of the next must be at least 8000 microseconds (8 milliseconds).

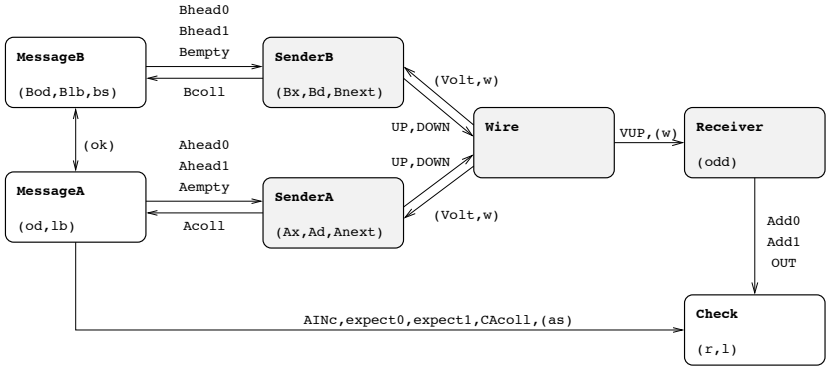


Figure 5: Philips Audio-Control Protocol with Bus Collision.

5 A Formal Model of the Protocol

To analyse the behaviour of the protocol we model the system as a network of seven timed automata. The network consists of two parts: a *core part* and a *testing environment*. The core part models the components of the protocol to be implemented: two senders, a wire and a receiver. The testing environment, consisting of two message generators (one for each sender) and an output checker, is used to model assumptions about the environment of the protocol and for testing the behaviour of the core part. Figure 5 shows a flow-graph of the network where nodes represent timed automata and edges represent synchronisation channels or shared variables, the latter enclosed within parentheses.

The general idea of the model is as follows. The two automata MessageA and MessageB are designed to non-deterministically generate possible valid messages for the both senders (as described in section 4), in addition MessageA informs the Check-automaton on the bits it generated for SenderA. The senders transmit the messages via the wire to the receiver. We have chosen to model the wire as an automaton to separate its behaviour from the two senders and the receiver. The receiver communicates the bits it decoded to the checker. Thus the Check automaton is able to compare the bits generated by MessageA and the bits received by Receiver. If this matches the protocol is correct.

The senders A and B are, modulo renaming (all A's in identifiers to B's), exactly the same. Because of the symmetry, it is enough to check that the messages transmitted by sender A are received correctly. If a scenario exits in which a message of sender B is received incorrectly, the same scenario (modulo renaming) exists for sender A. We will proceed with a short description of each

automaton. The definition of these uses a number of constants that are declared in Table 1 in Appendix 8.

The Senders

SenderA is depicted in Figure 10. It takes input actions Ahead0?, Ahead1? and Aempty?. The output actions UP! and DOWN! will be the Manchester encoding of the message. The clock Ax is used to measure the time between UP! and DOWN! actions. The idea behind the model (taken from [DY95]) is that the sender changes location each half of a bit-slot. The locations HS (wire is High in Second half of the bit-slot) and HF (High in First half of the bit-slot) refer to this idea. Extra locations are needed because of the collision detection.

The clock Ad is used to measure the time elapsed between the detection just before UP! action and the corresponding UP! action. The system is in the locations ar_Qfirst and ar_Qlast when the next thing to do is the collision test at one or three quarters of a bit-slot. When Volt is greater than zero, at that moment, the sender detects a collision, stops transmitting and returns to the idle location. The clock w is used to ensure the radio silence between messages. This variable is checked on the transition from idle to ar_first_up.

The Wire

This small automaton keeps track of the voltage on the wire and generates VUP! actions when appropriate, that is when a UP? action is received when the voltage is low. The automaton is shown in Figure 9.

The Receiver

Receiver, shown in Figure 8, decodes the bit sequence using the up-going (modelled as VUP?) changes of the wire. Decoded bits are signalled to the environment using output actions Add0!, Add1! and OUT! (where OUT! is used for signalling the end of a decoded message). The decoding algorithm of the receiver is a direct translation of the algorithm in the Philips documentation of the protocol. In the automaton each VUP? transition is followed by a transition modelling the decoding. This decoding happens at once, therefore the intermediate locations are modelled as committed locations. The automaton has two important locations, L1 and L0. When the last received bit is a bit "1" the receiver is in location L1, after receiving a bit "0" it will be in location L0. The error location is entered when a VUP? is received much too early. In the complete model the error location is not reachable, see Section 6. The receiver

keeps track of the parity of the received message using the integer variable odd. When the last received bit is a bit "1" and the message is even, a bit "0" is added to make the complete message of odd length.

The Message Generators

The message generators MessageA and MessageB, shown in Figure 11, generate valid messages (i.e. any message for which the protocol should behave correctly according to the specification) for sender A and B. In addition, the messages generated for sender A are communicated to the checker. The start of a message is signalled to the checker by AINc!, bits by expect0! and expect1!. When a collision is detected by sender A this is communicated to MessageA via Acoll?. The message generator will communicate this on his turn to the check automaton via CAcoll!.

Generating messages of odd length is quite simple. The only problem is that it is not allowed that a message for one sender is a prefix of the message for the other sender. To be more precise: If only one sender is transmitting there is no prefix restriction. Only when the two senders start transmitting at the same time, it is not allowed that one sender transmits a prefix of the message transmitted by the other. As mentioned before the reason for this restriction is that the prefix message is not received by the receiver and it is possible that the senders do not notice the collision. In other words: the prefix message can be lost. To ensure that the two generated bit-streams differ on at least on position, the generator always compare the last generated bit-values stored in the variables lb and Blb on the edge from locations sending0 or sending1 to location sending. If the bits differ, the variable ok is set to 1, which is a requirement for the message generation to end normally (on the transition from sending to idle in the two automata).

The Checker

This automaton is shown in Figure 7. It keeps track of the bits "in transit", i.e. the bits that are generated by the message generators but not yet decoded by the receiver. These bits are encoded using the two variables l, which stores the length of the bit-stream, and r that stores the actual bit-stream in transit. Whenever a bit is decoded or the end of the message is detected not conform the generated message the checker enters location error. Furthermore, when sender A detects a collision the checker returns to its initial location.

6 Verification in UPPAAL

In this section we present the results of analysing the Philips audio-control protocol formally described in the previous section. We will use $A.l$ to denote the (implicit) proposition $\mathsf{at}(A, \bar{l})$ introduced in Section 3.2. Also, note that invariance properties in UPPAAL are on the form $\forall\Box\beta$, where β is a local property.

Correctness Criteria

The main correctness criterion of the protocol is to ensure that the bit sequence received by the Receiver matches the bit sequence sent by SenderA. Moreover, the *entire* bit sequence should be received by Receiver (and communicated to Check). From the description of the Check-automaton (see the previous section) it follows that this behaviour is ensured if Check is always operating in location start or normal:

$$\forall\Box\ (\mathsf{Check.start} \vee \mathsf{Check.normal}) \tag{1}$$

When the Receiver-automaton observes changes of the wire too early it changes control to location error. If the rest of the components behave normally this should not happen. Therefore, the Receiver-automaton is required to never reach the location error:

$$\forall\Box(\neg\mathsf{Receiver.error}) \tag{2}$$

Incorrectness

Unfortunately the protocol described in this paper is not the protocol that Philips has implemented. The original sender checked less often for a bus collision. The "just before the up going edge" collision detection was only performed before the first up. In the UPPAAL model this corresponds to deleting outgoing transitions of ar_Qlast_ok and using the outgoing transitions of ar_up_ok instead. This incorrect version is shown in Figure 12. In general the problem is that if both senders are transmitting and one is slow and the other fast, the distance can cumulate to a high value that can confuse the receiver. UPPAAL generated a counter-example trace to Property 1. The trace is depicted in Figure 6. The scenario is as follows: Sender A (depicted with thick lines) tries to transmit "111..." and sender B (depicted with thin lines) "1100...". The sender A is fast and the other slow. This causes the distance between the second UP's to be very big (77 microseconds). In the third bit-slot the sender A detects the collision. The result of all this is that the time elapsed between the VUP actions is 6.65Q instead of

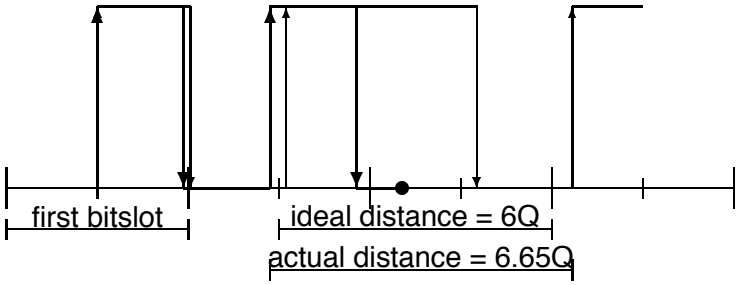


Figure 6: Error execution of the incorrect protocol.

the ideal 6Q. Because of the timing uncertainty in the receiver this can be interpreted as 7Q ($7 * 0.95 = 6.65$), and 7Q is just enough to decode "01" instead of the transmitted "0". Thus, it is possible that the sent and received message differ with this version of the protocol.

In the correct version this scenario is impossible, because if collision detection happens before *every* UP action, the distance between the UP's in the second bit-slot can not be that high (at most 20 microseconds).

It is not likely that these kind of errors happen in the actual implementation. First, it is not likely that two senders do start at sufficiently close time-points. Secondly, the timing uncertainty is at most 2% instead of 5%, and the "average" timing uncertainty is even less. For more details, see [Gri94].

Although this problem was known by Philips it is interesting to see how powerful the diagnostic traces can be. It enables us not only to find mistakes in the *model* of a protocol, but also to find design mistakes in real-life protocols.

Verification Results

UPPAAL successfully verifies the correctness properties 1 and 2 for an error tolerance of 5% on the timing. Recall that SenderA and SenderB are, modulo renaming, exactly the same, implying that the verified properties for SenderA also applies to the symmetric case for SenderB. The verification of Property 1 and 2 was performed in 0.5 sec using 2.5 MB of memory.

The analysis of the incorrect version of the protocol with less collision detection (discussed above) uses UPPAAL's ability to generate diagnostic traces whenever an invariant property is not satisfied by the system. The trace, consisting of 46 transitions, was generated in 0.4 sec using 2.5 MB of memory. Also, verification of Property 1 for the protocol with full collision detection and an error tolerance

of 6% on all the timing produces an error trace as well. The scenario is similar to the one found by Bosscher et.al. in [BPV94] for the one sender protocol.

The properties were verified using UPPAAL version 3.2 [LPY97a, BLL+98, ABB+01] that implements the verification algorithm handling committed locations described in Section 3. It was installed on a Pentium II 375 MHz PC running Debian Linux 2.2. In the conference version of this paper [BGK+96] we reported that the same protocol was verified using UPPAAL version 0.96[3] installed on a SGI ONYX machine. The verification of the two correctness properties then consumed 7.5 hrs using 527.4 MB and 1.32 hrs using 227.9 MB, whereas a diagnostic trace for the incorrect version was generated in 13.0 min using 290.4 MB of memory. Hence, both the time- and space-consumption of the verifier for this particular model have been reduced with over 99%. These improvements of the UPPAAL verifier are due to a number of developments in the last years that will not be discussed further here. It should also be noticed that the older version uses backwards analysis whereas the newer performs forwards analysis. For more information he developments of UPPAAL we refer the reader to [LPY97b, BLL+98, ABB+01].

7 Conclusions

In this paper we have presented a case-study where the verification tool UPPAAL is used to verify an industrial audio-control protocol with bus-collision handling by Philips. The protocol has received a lot of attention in the formal methods research community (see e.g. [BPV94, HWT95, CW96]) and simplified versions of the protocol without the handling of bus collisions have previously been analysed by several research teams, with and without support from automatic tools.

As verification results we have shown that the protocol behaves correctly if the error on all timing is bound to $\pm 5\%$, and incorrectly if the error is $\pm 6\%$. Furthermore, using UPPAAL's ability to generate diagnostic traces we have been able to study error scenarios in an incorrect version of the protocol actually implemented by Philips.

In this paper we have also introduced the notion of so-called committed locations which allows for more accurate modelling of atomic behaviours. More importantly, it is also utilised to guide the state-space exploration of the model

[3] The two UPPAAL versions 0.96 and 2.17 are dated Nov 1995 and March 1998 respectively.

checker to avoid exploring unnecessary interleavings of independent transitions. Our experimental results demonstrate considerable time and space-savings of the modified model checking algorithm. In fact, due to the huge time and memory-requirement, it was impossible to check certain properties of the protocol before the introduction of committed locations, and now it takes only seconds.

References

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science, pages 100–125. Springer–Verlag, 2001.

[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.

[AK95] Rajeev Alur and Robert P. Kurshan. Timing Analysis in COSPAN. 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 220–231. Springer–Verlag, October 1995.

[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 8th 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. Available as `http://www.docs.uu.se/docs/rtmv/-bl-report.pdf`.

[BLL+95] 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.

[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.

[CW96] Edmund M. Clarke and Jeanette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[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.

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

[HHWT95] 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 Hybrid Systems. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[HNSY94] Thomas. A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.

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

[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.

[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 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[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 9th Int. Conf. on Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 456–459. Springer–Verlag, June 1997.

[LSW97] Kim G. Larsen, Bernard Steffen, and Carsten Weise. Continuous modeling of real-time and hybrid systems: from concepts to tools. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):64–85, December 1997.

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

[ST01] R. F. Lutje Spelberg and W. J. Toetenel. Parametric real-time model checking using splitting trees. *Nordic Journal*, 8(1):88–120, 2001.

[Yov97] Sergio Yovine. A Verification Tool for Real Time Systems. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[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.

8 Appendix

The constants used in the formulas		
q	2220	One quarter of a bit-slot: 222 micro sec
d	200	Detection 'just before' the UP: 20 micro sec
g	220	'Around' 25% and 75% of the bit-slot: 22 micro sec
w	80000	The radio silence: 8 milli sec
t	0.05	The timing uncertainty: 5%
The constants in the automata		
W	w	80000
D	d	200
A1min	q-g	2000
A1max	q+g	2440
A2min	3*q-g	6440
A2max	3*q+g	6880
Q2	2*q	4440
Q2minD	2*q*(1-t)-d	4018
Q2min	2*q*(1-t)	4218
Q2max	2*q*(1+t)	4662
Q3min	3*q*(1-t)	6327
Q3max	3*q*(1+t)	6993
Q5min	5*q*(1-t)	10545
Q5max	5*q*(1+t)	11655
Q7min	7*q*(1-t)	14763
Q7max	7*q*(1+t)	16317
Q9min	9*q*(1-t)	18981
Q9max	9*q*(1+t)	20979

Table 1: Declaration of Constants.

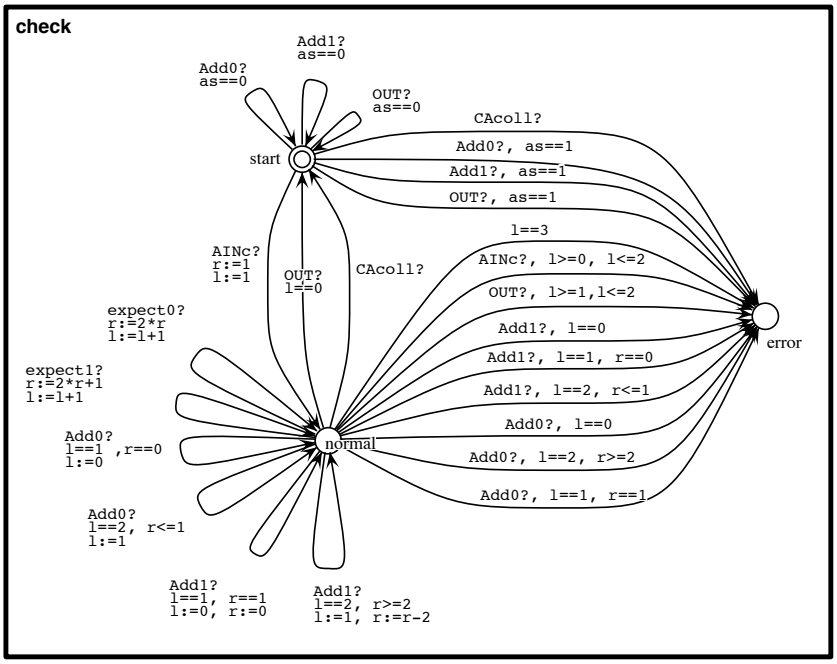


Figure 7: The Check Automaton.

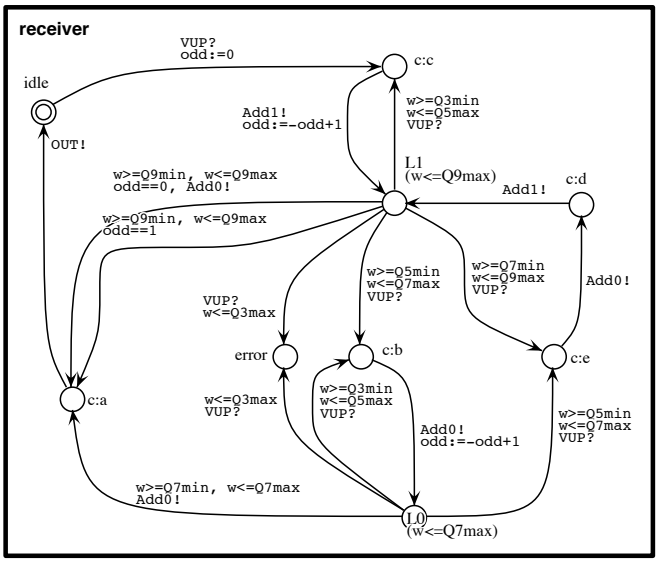


Figure 8: The Receiver Automaton.

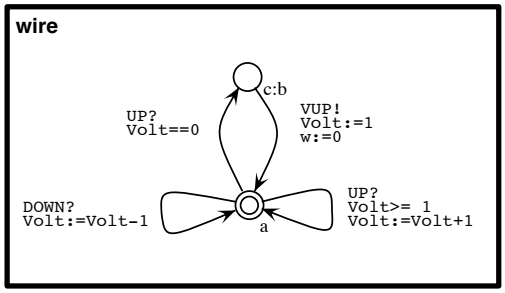


Figure 9: The Wire Automaton.

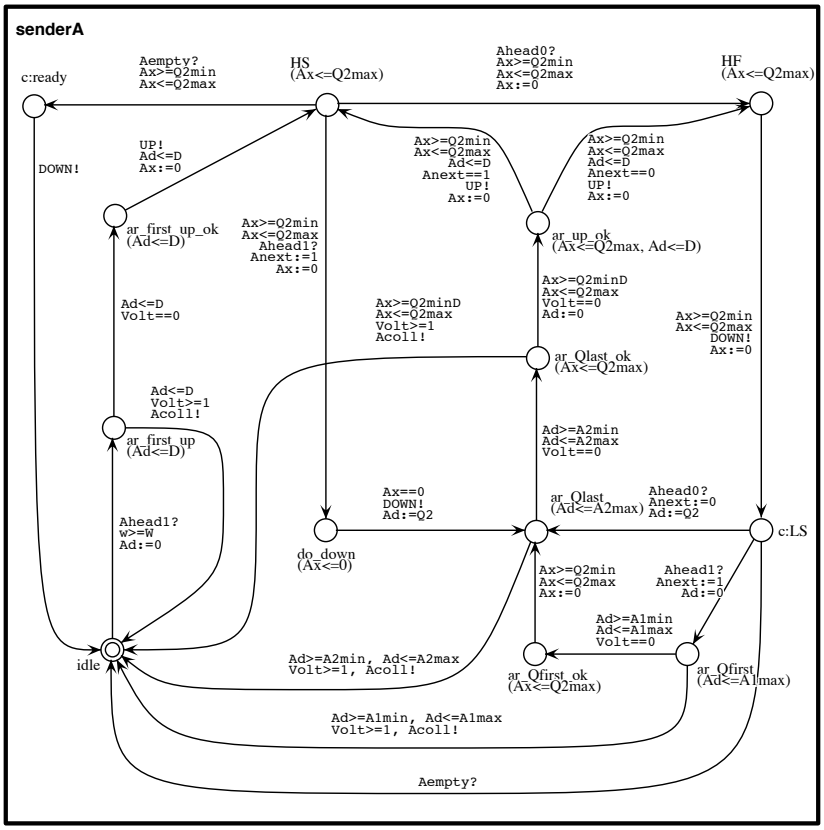
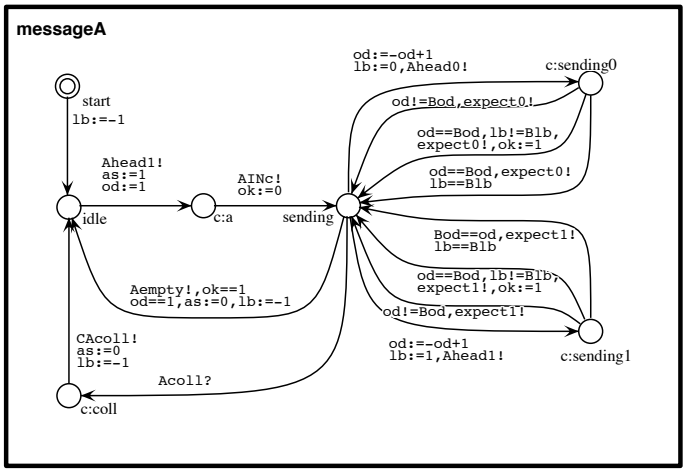


Figure 10: The SenderA Automaton.



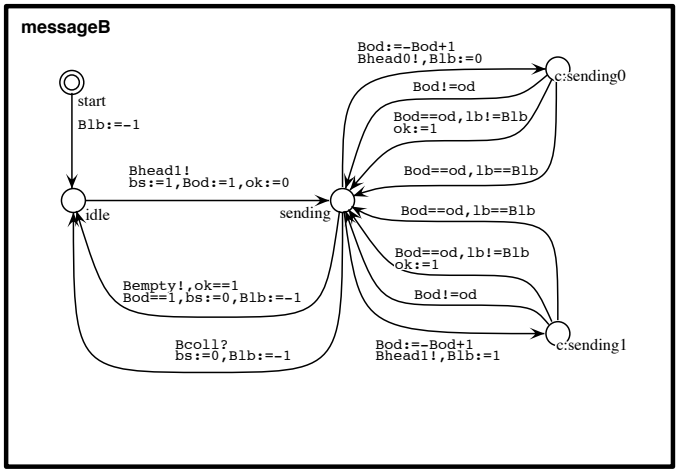


Figure 11: The Message Automata.

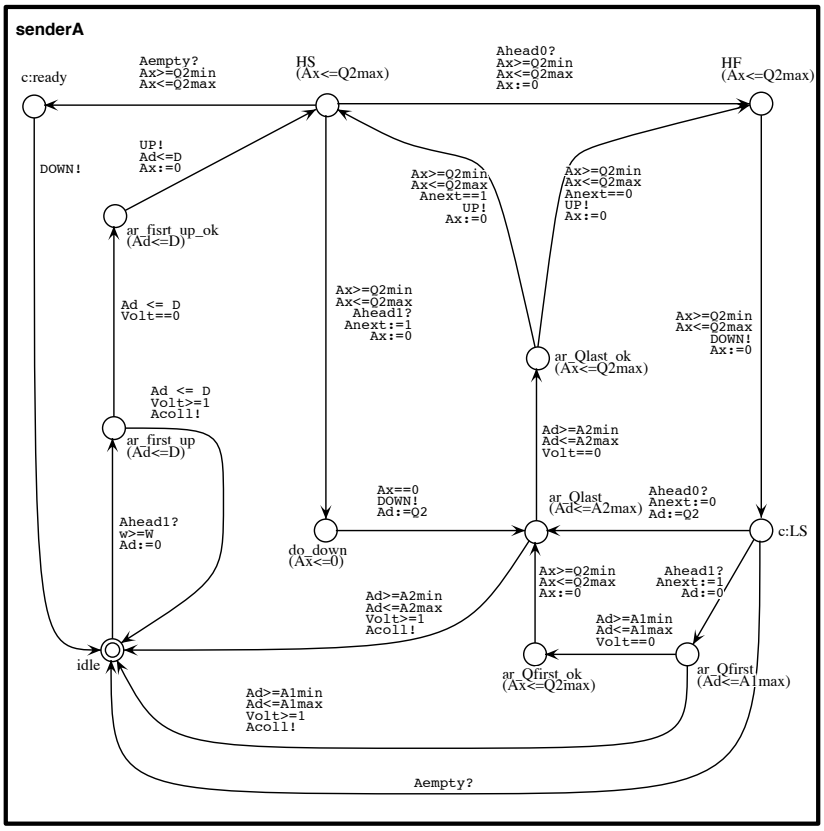


Figure 12: The Incorrect SenderA Automaton.