IT Licentiate theses
2001-009

Efficient Symbolic State Exploration of Timed Systems: Theory and Implementation

JOHAN BENGTSSON

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

Efficient Symbolic State Exploration of Timed Systems: Theory and Implementation

BY
JOHAN BENGTSSON

May 2001

DEPARTMENT OF COMPUTER SYSTEM
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Systems
at Uppsala University 2001

Efficient Symbolic State Exploration of Timed Systems:
Theory and Implementation

Johan Bengtsson
johanb@docs.uu.se

Department of Computer System
Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Timing aspects are important for the correctness of safety-critical systems. It is crucial that they are carefully analysed in designing such systems. UPPAAL is a tool designed to automate the analysis process. In UPPAAL, a system under construction is described as a network of timed automata and the desired properties of the system can be specified using a query language. Then UPPAAL can be used to explore the state space of the system description to search for states violating (or satisfying) the properties. If such states are found, the tool provides diagnostic information, in form of executions leading to the states, to help the desginers, for example, to locate bugs in the design.

The major problem for UPPAAL and other tools for timed systems to deal with industrial-size applications is the state space explosion. This thesis studies the sources of the problem and develops techniques for real-time model checkers, such as UPPAAL, to attack the problem. As contributions, we have developed local-time semantics for timed systems to allow partial order reductions, the notion of committed locations to model atomicity and a number of implementation techniques to reduce time and space consumption in state space exploration. The techniques are studied and compared by case studies. Our experiments demonstrate significant improvements on the performance of UPPAAL.

Acknowledgements

First I want to thank my supervisor Wang Yi, for guiding me towards the completion of this thesis. I have learnt a lot during the years we have been working together. I also would like to thank all current and former colleagues in the "UPPAAL-group" in Uppsala, *i.e.* Tobias Amnell, Alexandre David, Elena Fersman, Fredrik Larsson, Justin Pearson and Paul Petterson. It has been a pleasure working together with you, discussing implementation issues and other problems. Further I want to thank Kim G. Larsen and the rest of the Aalborg part of the UPPAAL project for stimulating collaboration. Without your participation UPPAAL would not have been what it is today. I am also grateful to all my other co authors, *i.e.* David Griffioen, Bengt Johnsson and Johan Lilius, for fruitful discussions. It has been fun working with you.

Last but not least I want to thank my family. Without their love, support and encouragement this thesis would never have been possible.

This thesis is based on four different publications, written between 1995 and 2001.

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

Paper B: 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, Lecture Notes in Computer Science, Springer Verlag, 1998

Paper C: Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. Automated Analsysis of an Audio-Control Protocol using UPPAAL. In Proceedings, Ninth International Conference on Computer Aided Verification, volume 1102, Lecture Notes in Computer Science, Springer Verlag, 1996

Paper D: Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In Proceedings, Hybrid Systems III: Verification and Control, volume 1066, Lecture Notes in Computer Science, Springer Verlag, 1995

Comments on My Participation

Paper A: I discussed the content with Wang Yi. I implemented everything and wrote the report.

Paper B: I participated in discussions and wrote part of the paper. I made a prototype implementation but it is not described in the paper.

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

Paper D: I implemented UPPAAL together with Fredrik Larsson.

Contents

Introduction

1 Background

During the last decades, computers have become one of the most important tools in our society. They are no longer used only for word processing, banking or scientific computations. Nowadays computers are widely used in our daily life to control e.g. stereos, micro wave ovens, cars, medical equipments, aeroplanes *etc.* This change became possible due to the development of powerful micro processors that can be integrated in so called *embedded systems*. A large class of embedded systems is *real-time systems* where it is important that the control computer should deliver not only correct output, but also in time. As an example, consider an industrial robot that picks boxes from conveyor belt.

Because of the wide spread safety-critical applications of real time systems, their correct functioning has been an issue of vital importance in the system development process. Various new techniques have been developed to check the correctness of such systems. Among others, formal verification has been a promising technique to complement the traditional method by testing. The basic idea of formal verification is to describe the system under development in a formal systems and then apply rigorous methods to prove that the system meets its requirements (*e.g.* [Hoa69, Dij75, Hoa78, Rei85, Mil89, Hol91, AD90, AD94, BD91, Yi91, RR88, ACD90, CES86, AH94, HNSY92, SS95]).

In this thesis we focus on *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 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 sys-

tems. These tools have all been successfully applied to industrial-size case studies,*e.g.* [HLP98, JMMS98, SD95a, MMS97, LPY98, HSLL97, TY98, HWT96].

Even though the existing techniques and tools have become increasingly efficient and success stories are reported frequently, they currently do not scale up to the size of most industrial systems. The bottleneck is the state space explosion problem. The main source of the problem is that the state space of a system of parallel processes can grow exponentially with the number of components. This is due to the fact that parallelism is modelled by interleaving of steps from the processes. In the literature, various techniques have been proposed to attack the state space explosion problems e.g. partial order reductions [God90, Val90, Pel93], symmetry reductions [HJJJ84, ID96, ES97], loop reduction [LLPY97], sweep-line and state space caching methods to keep only part of the state space in main memory [Hol85, GHP95, SD96, CKM01], and probabilistic methods [Hol91, Hol98, WL93, SD95].

2 The State Explosion Problem for Timed Systems

During the past few years, a number of verification tools have been developed for real-time systems in the framework of timed automata (e.g. KRONOS and UPPAAL [DOTY95, LPY97, BLL+98]). One of the major problems in applying these tools to industrial-size systems is the huge memory-usage (e.g. [BGK+96]) for the exploration of the state-space of a network (or product) of timed automata. The main reason is that the model-checkers must store a large number of symbolic states each of which contains information not only on the control structure of the automata but also the clock values specified by clock constraints or time zone. A well-known technique to represent time zone in the existing model checkers for timed systems is *Difference Bounds Matrices* [Dil89] (DBM). In DBM, the memory requirement for each state is quadratic in the number of clocks. For example, in a system with 10 clocks and 10 processes and each with 10 states, the time zone for each state requires 484 bytes in addition to 10 bytes for the control part.

It is crucial for performance to reduce the memory requirement to represent time zone. In the literature there are a few techniques that address this problem. In [DY96] live-range analysis is used to reduce the number of clocks in a model. By analysing the control structure of the model it is possible to compute, for each control location, the set of active (live) clocks. Then, for each state, only timing information regarding the active clocks need to be stored. Another approach is taken in [LLPY97]. This work is based on the observation that the DBM representation of a time zone often contain a lot of redundant information, *i.e.* the solution

set can be represented using much less constraints. The paper presents a three step procedure for computing a minimal set of constraints with the same solution set as a given DBM. One step further is presented in [BLP+99]. This paper introduce CDDs, a BDD-like structure for representation of time zones. The key feature of this structure is the possibility to store non-convex unions of zones. A CDD is a directed acyclic graph with two types of nodes: terminal nodes labelled true and false, and inner nodes labelled with pairs of clocks. The edges in this graph correspond to bounds on the difference between the clocks in source node.

3 Contributions

The main contribution of this thesis is in the implementation of UPPAAL. In particular, we have developed a number of efficient techniques to minimise memory usage in model checking timed systems. We study the memory consumption problem in two fronts: the data structures to store and manipulate each symbolic state and the whole state space. We present two different methods that can be used 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. 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 keeping 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. These techniques have been implemented in the UPPAAL tool, evaluated and compared by real-life examples; their strengths and weaknesses are described.

In addition, we have developed 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.

Another contribution is the notion of *committed locations*. This notion allows accurate modelling of atomic behaviours, such as atomic broadcast. More importantly committed location are utilised to guide the state-space exploration of the model checker to avoid exploring unnecessary interleavings of independent transitions. In the thesis 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 of the modified model checking algorithm.

4 Conclusions and Future Work

The bottleneck for model checking technology to scale up is the state state explosion problem. For timed systems, the problem is even more critical because the model checker must keep track on not only the state space of the control structure of a system, but also timing constraints over the clock variables of the system.

This thesis summarises the implementation decisions and techniques adopted in UPPAAL. We have presented a collection of techniques to improve the performance of UPPAAL, in particular, to reduce the huge memory consumption in state space explosion. We believe that these techniques are general and applicable to other model checkers for timed systems. We would like to point out that the development of UPPAAL demonstrates it is possible today to build a model checker for timed systems that is both easy to use and capable of handling realistic, industrial size, systems.

The work presented in this thesis can be extended in several directions. As future work, we will study and develop techniques to reduce the memory requirements even further without losing performance, *e.g.* by finding better packing methods or developing hash functions for zones that still allow inclusion checking. Another challenge is to further investigate partial order reduction can be efficiently applied to timed automata. The local-time semantics is just a step on the way. Further investigations are needed to develop an efficient implementation of the technique. Hierarchical extensions to timed automata is a direction that is currently being pursued. A new challenge is how to take advantage of the hierarchical structures

to reduce the time and space consumption of the state space exploration process.

Bibliography

[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.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[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.

[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.

[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*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

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

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(3):244–263, 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.

[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.

[GHP95] Patrice Godefroid, Gerard J. Holzmann, and Didier Pirottin. State-space caching revisited. *Journal of Formal Methods in System Design*, 7(3):227–241, 1995.

[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.

[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.

[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.

[Hol85] Gerhard J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(10), 1985.

[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.

[Hol98] Gerard J. Holzmann. An analysis of bitstate hashing. *Journal of Formal Methods in System Design*, November 1998.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[SD96] Ulrich Stern and David L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme: 4. GI/ITG/GME Workshop Proceedings*, 1996.

[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.

[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.

[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.

Paper A:

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

Johan Bengtsson. Technical Report, 2001-009, Department of Information Technology, Uppsala University, 2001.

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

Johan Bengtsson

Abstract. One of the major problems to scale up model checking techniques to the size of industrial systems is the large memory consumption. This report address the problem in the context of verifiers for timed systems and present a number of techniques that reduce the amount memory used for state space exploration in such a tool. The methods are evaluated and compared by real-life examples and their strengths and weaknesses are described. In particular we adress the memory consumption problem on two fronts, first by reducing the size of each symbolic state by means of compression and second by reducing the size of the stored state space by early inclusion checking and probabilistic methods.

1 Introduction

During the last ten years timed automata [AD90, AD94] has 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. The two best known of these tools are UPPAAL [LPY97, ABB+01] and KRONOS [DOTY95, Yov97]. 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 does not only suffer from the large number of states that needs to be explored, but also from the large size of each state.

In the literature a number of approaches studying these problems have been developed, for both timed and untimed systems: Some methods do not only aim at the memory consumption but also reduce the reachable state space of a system, such as partial order reduction methods [God90, Val90, Pel93] and symmetry reductions [HJJJ84, ID93]. Other methods aim at reducing the number of states stored in the passed list. In the global reduction technique from [LLPY97] all states that are not possible loop entry points are never entered in the passed list. A related method is the sweep-line technique described in [CKM01]. Here a notion of progress is used to throw away parts of the passed list that can never be revis-

ited. The work presented in [SD98] describe how the passed list can be pushed down in the memory hierarchy, from main memory to disk, with a very small penalty in terms of increased runtime. On the border between reducing the size of the state space and reducing the size of each state we find the work presented in [DY96]. In this work, methods similar to live-range analysis in compilers are used to reduce the number of clocks in timed automata. Another method that fits into this group is the CDD technique described in [BLP+99]. Here a BDD-like structure, that can represent unions of zones efficiently, is used to store the timing information for each state.

In probabilistic methods, such as the supertrace method [Hol91] and the hash compaction method [WL93, SD95], the demand for exact verification is relaxed in order to save space. In these methods there is a probability that a part of the state space will never be visited during verification.

In this report we address both the problem of large state spaces and the problem of large states.

The problem with large symbolic states is addressed by means of compaction. We present two different methods that can be used 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.

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 keeping 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 report is organised as follows: In section 2 we introduce timed automata as a model for timed systems and give a brief introduction on how to check properties for systems modelled as timed automata.

In section 3 we present three ways 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 state space of a model. We describe how the wait and past lists are handled in UPPAAL. We also describe an approximation method of the past list that can be used when the complete state space of a model is too big to be stored in memory.

Finally, section 5 wraps up the report by summarising the most important results and suggests some directions for future work.

A description of the platform and examples used to evaluate the presented techniques is given in Appendix A.

2 Timed Automata and Reachability Analysis

In this section we briefly review the notation. A more extensive description can be found in *e.g.* [AD94, Pet99]. For clarity, we start by describing a version of timed automata which is somewhat simplified compared to the model used in UPPAAL and then extend it to the full UPPAAL model in a less formal manner.

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. 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 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, $\rightarrow \in N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$ is the set of edges and $I : N \longrightarrow \mathcal{B}(C)$ assign invariants to locations. As a simplification we will use $l \xrightarrow{g,a,r} l'$ to denote $\langle l, g, a, r, l'\rangle \in \rightarrow$.*

An example automaton is shown in Figure 1. The automaton is a model of a time dependent light switch. Initially the the light is off and if the switch is pressed a dim light is switched on. If the switch is pressed one more time within 10 seconds the light gets brighter but if the second press is later than 10 seconds after the first, the light is switched off. If the switch is pressed when the light is bright, the light is switched off.

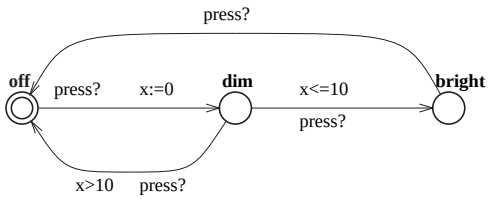


Figure 1: A timed automaton modelling a light switch.

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 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 $\langle l, D \rangle$, where $D \in \mathcal{B}(C)$ [HNSY92, YPD94]. The symbolic counterpart of the transitions are given by:

- $\langle l, D \rangle \rightsquigarrow \langle l, D^{\uparrow} \wedge I(l) \rangle$
- $\langle l, D \rangle \rightsquigarrow \langle l', r(D \wedge g) \rangle$ 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 $\langle l, u \rangle \rightarrow \langle l', u' \rangle$. As an example, the symbolic semantics of the lamp switch is shown in Figure 2.

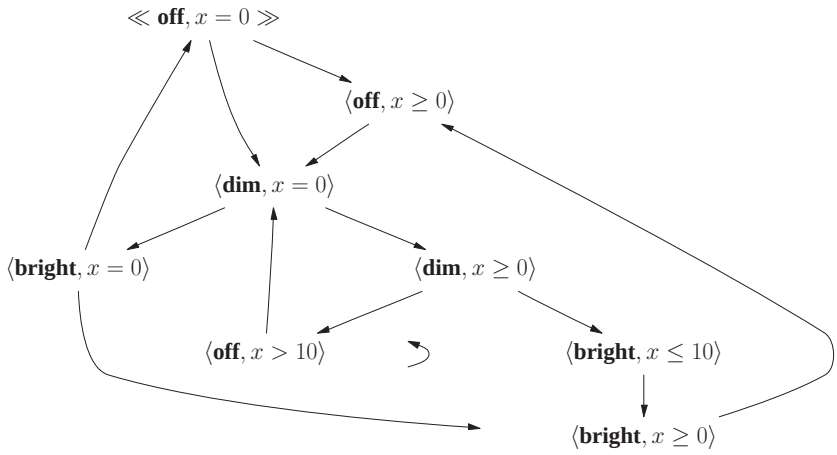


Figure 2: Symbolic semantics of lamp switch automaton

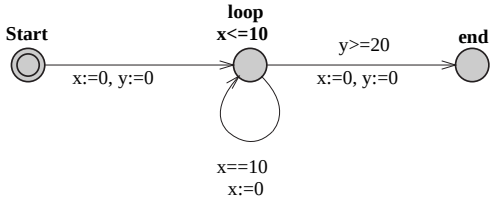


Figure 3: Timed automaton with an infinite symbolic semantics

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. As an example, consider the automaton in Figure 3. The symbolic semantics of this simple automaton is shown in Figure 4. The symbolic state space is infinite because a clock drifts away unboundedly. The solution problem is to introduce a k-*normalised* version of the infinite symbolic semantics. The idea is to utilise the maximum constant appearing in clock constraints in the automaton, to render a finite symbolic semantics. For details we refer the reader to [Pet99, Rok93] but the main fact and the intuition behind it will be described here.

In order to do this we first have to introduce the notion of closed constraint sys-

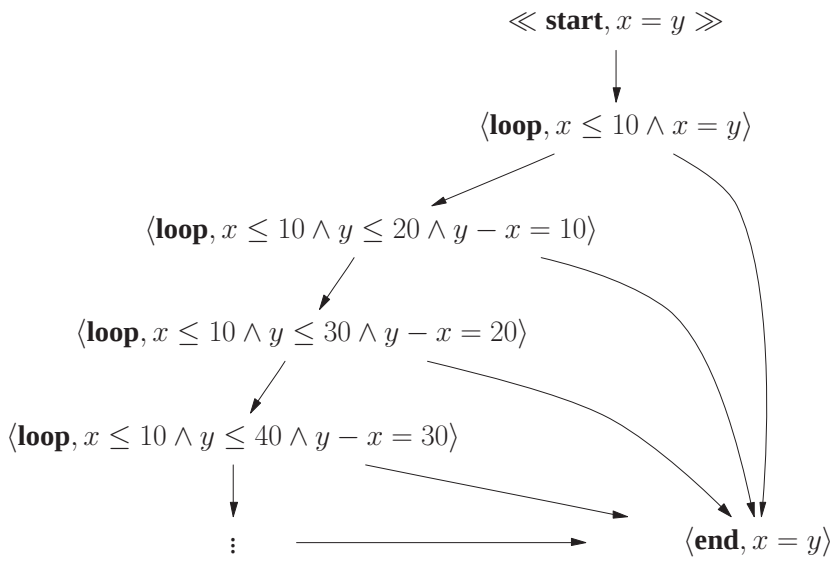


Figure 4: Symbolic semantics of timed automaton in Figure 3

tems. 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 natural number k, the k-normalisation of D, denoted $\mathsf{norm}_k(D)$, is computed from the closed representation of D by (a) removing all constraints of the form $x < m$, $x \leq m$, $x - y < m$ and $x - y \leq m$ where $m > k$, (b) replacing all constraints of the form $x > m$, $x \geq m$, $x - y > m$ and $x - y \geq m$ where $m > k$ with $x > k$ and $x - y > k$ respectively. This can then be used to define a notion of k-normalised symbolic transitions ($\rightsquigarrow_k$) by modifying the transitions of the standard symbolic semantics to preserve k-normalisation. The discrete action transition already preserves this so there is no need to modify it, but the delay transition should be modified to $\langle l, D\rangle \rightsquigarrow_k \left\langle l, \mathsf{norm}_k(D^{\uparrow} \wedge I(l))\right\rangle$.

Proposition 2 *Assume a timed automaton A with symbolic initial-state $\langle l_0, D_0\rangle$ and let k be the largest constant appearing in any constraint 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 k-normalised transitions $\langle l_0, D'_0\rangle \rightsquigarrow_k^* \langle l, D'_n\rangle$ such that $D \cap D'_n \neq \emptyset$.*

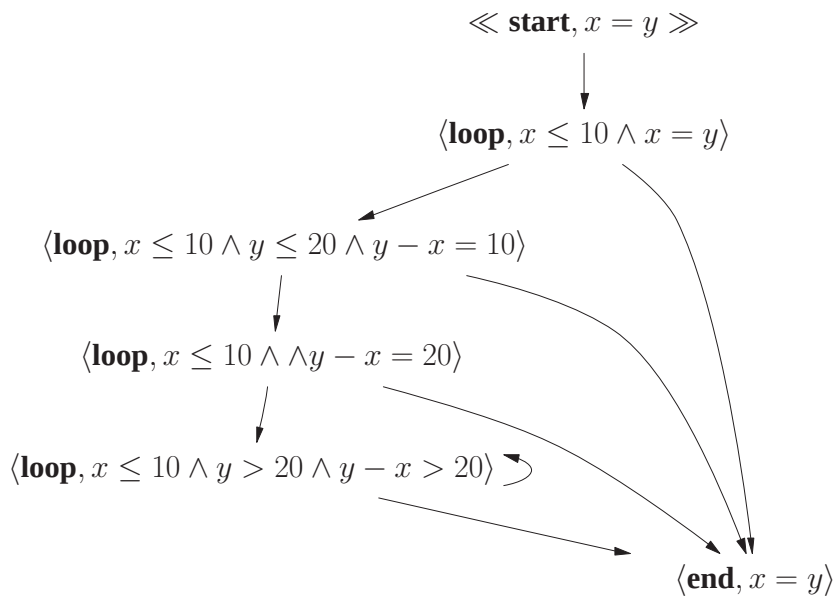


Figure 5: 20-normalised symbolic semantics for automaton in Figure 3

The intuition behind this construction is that when the value of a clock is larger than the maximum constant appearing in A it is not possible for any edge in A to distinguish the actual value of the clock, only that the value is above the maximum constant. Therefore it is not needed to keep track of the actual value of the clock when it is larger than the maximum constant. As an example we apply this to the automaton from Figure 3. For this automaton, the maximum constant is 20 (from the guard on y on the edge from **loop** to **end**). Thus we need to compute the 20-normalised symbolic semantic of the automaton to preserve the reachability properties. This semantics can be found in Figure 5.

Using this we will get a finite symbolic state-space where we can apply a standard reachability algorithm for graphs, such as the one in Algorithm 1. The algorithm uses two important data structures: WAIT and PASSED. WAIT is a list of states waiting to be explored. By controlling how new states are added to WAIT the exploration order can be altered. If WAIT is organised as a queue the exploration will be breadth first, and if WAIT is organised as a stack the exploration will be depth first. When the exploration is started the initial state is placed in WAIT. PASSED is a table of states explored so far. Initially PASSED is empty. Due to the size of the state space, these structures may consume a considerable amount of main memory.

Algorithm 1 Symbolic 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 $\langle l, D \rangle \rightsquigarrow_k \langle l', D' \rangle$ **do**
      add $\langle l', D' \rangle$ to WAIT
    end for
  end if
end while
return "NO"

2.3 UPPAAL Extensions

In UPPAAL there are some extensions to the model described above. The most important of these are networks of timed automata (to model parallel tasks), shared integer variables, urgent channels and committed locations.

Networks of Timed Automata

To model concurrent systems, UPPAAL has a notion of a *network of timed automata*, *i.e.* several timed automata (called processes) are combined into a single system. Synchronisation between the processes are performed either using the edge labels or, as will be described later, using shared integer variables.

To control synchronisation we partition the set of labels into *local labels* and *synchronising labels*. The synchronising labels are then paired two by two In UPPAAL each such pair of synchronising labels is called a channel and they share a common prefix (or channel name) followed by either ! or ?. A label test! will be called an *output* on channel test and test? will be called an *input* on the same channel.

As an example, consider the network in Figure 6. Here the lamp switch model described earlier have been combined with a model of a user. This particular user have two main interests, reading and watching TV, which he may start doing at any time. If the user wants to watch TV he press the switch once to get a cosy dim light and starts to watch. If, on the other hand, the user wants to read he press the switch twice within five seconds to get a light bright enough for reading and starts to read.

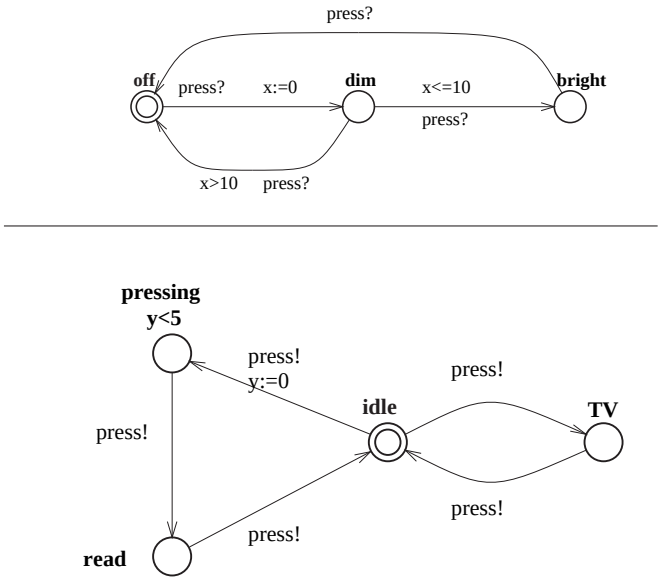


Figure 6: A network of timed automata modelling a light switch and its user.

The semantics of a network is similar to the semantics of a single timed automaton; the difference is that we have to define how processes interact. A state is a pair $\langle \bar{l}, u \rangle$, the difference to the timed automata semantics is that the location of a network is a vector of control locations, one for each process in the system. In a network there are three types of transitions, delay transitions, local action transitions and synchronising action transitions. The delay transitions works exactly like delay transitions for single timed automata with the exception that for a network the invariant on the current location of all processes have to be taken into account. The discrete action transition type of the single timed automata is split into two types in the network case. A discrete action transition of a network is either a local action transition, where one of the processes makes a move entirely on its own, or a synchronising action transition, where two processes synchronise on a channel and move simultaneously. The transition rules are as follows:

- $\langle \bar{l}, u \rangle \xrightarrow{\epsilon(t)} \langle \bar{l}, u + t \rangle$ if $u \in I(\bar{l})$ and $(u + d) \in I(\bar{l})$, where $I(\bar{l}) = \bigwedge I(l_i)$
- $\langle \bar{l}, u \rangle \xrightarrow{a} \langle \bar{l}[l'_i/l_i], u' \rangle$ if $l_i \xrightarrow{g,a,r} l'_i, u \in g, u' = [r \mapsto 0]u$, $u' \in I(\bar{l}[l'_i/l_i])$
- $\langle \bar{l}, u \rangle \xrightarrow{\tau} \langle \bar{l}[l'_i/l_i][l'_j/l_j], u' \rangle$ if $l_i \xrightarrow{g_i,a?,r_i} l'_i, l_j \xrightarrow{g_j,a!,r_j} l'_j, i \neq j, u \in g_i \wedge g_j, u' = [r_i \cup r_j \mapsto 0]u$ and $u' \in I(\bar{l}[l'_i/l_i][l'_j/l_j])$.

This concrete semantics are then easily extended to a normalised symbolic semantics that can serve as a base for model checking procedures.

Shared Integer Variables

As a convenience, the timed automata model of UPPAAL has a notion of *shared integer variables*. Each network of timed automata $A_1|\cdots|A_n$ is augmented with a set, $\mathcal{D}$, of integer variables each with a bounded domain and an initial value. Predicates over the integer variables can be used as guards on the edges in any process and their values can be updated on by any process, as a part of the reset. In the current versions of UPPAAL an *integer guard* have the form $E_1 \sim E_2$ where E_1, E_2 are integer expressions over $\mathcal{D}$, as defined by the grammar below, and $\sim \in \{<, \leq, =, \neq, \geq, >\}$. An *integer reset* has the form $v := E$ where $v \in \mathcal{D}$ and E is an integer expression over $\mathcal{D}$. The integer expressions are generated by the following syntax:

$$E ::= E + E \mid E - E \mid E * E \mid E/E \mid n \mid v$$

Where $n \in \mathbb{Z}$ is an integer value and $v \in \mathcal{D}$ is an integer variable. This expression can easily be extended to handle function calls and procedures but it is currently not implemented.

The semantics needs to be extended to deal with the integer variables. A state of a network is now a triple $\langle \bar{l}, d, u \rangle$ where $\bar{l}$ and u are as before and d is a function mapping each variable in $\mathcal{D}$ to a value in its domain. Functions such as d will be called *integer assignments*. Similar to clock assignments we use $d \in g_i$ to denote that the integer assignment d satisfies the integer guard g_i.

Since delay does not affect the integer variables the delay transitions are the same as for networks without integer variables. The action transitions are extended in the natural way, *i.e.* for an action transition to be enabled the integer assignment must satisfy all integer guards on the corresponding edges and when a transition is taken the integer assignment is updated according to the integer resets.

However, there is one problem with this extension that needs to be considered. How should we handle the case where one of the processes participating in a synchronising transition updates a variable that is at the same time used by the other.[1] There are three possible solutions to this problem, first it is possible to prevent this by demanding that the integer resets of the edges in a synchronising transition may not update a variable that is used or updated by the other. This property can

[1] Used in the sense that the variable is either updated itself or its value is needed to compute the new value of another variable.

be checked, either statically or during verification, and an error can be reported to the user if the property does not hold. Second, the situation can be handled by introducing a non-determinism where either of the resets are performed before the other. However, this feature would probably be of little use to user and the extra non-determinism introduced may give a significantly larger state space. The third solution is to give a defined order between the resets on the edges. In UPPAAL resets on the edge with an output-label is performed before the resets on the edge with an input-label. The drawback is that this solution destroys symmetry properties that could have been used to optimise the state space search, but on the other hand it can be utilised to create more efficient models.

Urgent Channels

In the standard network model processes are always allowed to delay up to the time specified by the location invariants even if there are enabled synchronisation actions. Sometime this is the wanted behaviour but on other occasions the preferred behaviour would be that the synchronisation occurred as soon as it gets enabled. To allow the second type of behaviours as well as the first, UPPAAL has a notion of urgent channels. An urgent channel works much like an ordinary channel, but with the exception that if a synchronisation on an urgent channel is possible the system may not delay. Interleaving with other enabled action transitions, however, is still allowed. In order to keep the time regions representable using one normal zone we forbid clock guards on edges synchronising on urgent channels. To illustrate why this restriction is necessary we use an example.

Consider the network presented in Figure 7. Both processes may independently go from their first state to their second state. In the second state the processes must delay for at least 10 time units before they are allowed to wait for synchronisation on the urgent channel u. As soon as both processes have spent 10 time units in their second state they should synchronise and move to their third state. The problem with this network arise in [**S1,T1**]. As you see in Figure 8, the timing region for this state is not representable using a zone. Since the synchronisation occurs as soon as the transition is enabled the region won't even be convex.

For this simple example the problem can be solved by splitting the timing region into two zones (marked by a dashed line in the figure), but for a more complicated example the number of zones needed to represent one timing region may be much larger.

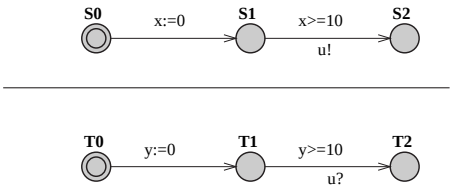


Figure 7: An example of a network with non convex timing regions.

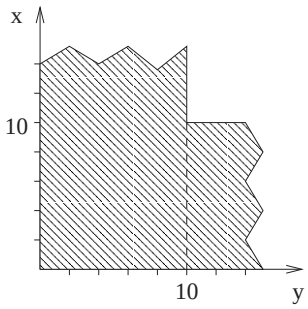


Figure 8: Timing region for state [**S1,T1**] in the network from Figure 7.

Committed Locations

For some some models it is necessary to have a notion of atomic sequences of actions, *e.g.* to model atomic broadcast or multicast. In UPPAAL this notion is supported using so called *committed locations*. A committed location is a control location where no delay is allowed and if any process is in such a location then only transitions starting in a committed location are enabled. Thus, processes that are in committed locations may not be interleaved with processes that aren't. However, a process that is in a committed location may be interleaved with other processes that are in committed locations.

A little more formally, each process A_i in a network of timed automata has a set $N_i^C \subseteq N_i$ of committed locations. For a location vector $\bar{l}$ of the network, we use $C(\bar{l})$ to denote the subset of the locations in $\bar{l}$ that are committed. In the semantics the states have the same form as for networks without committed locations, but the transitions are somewhat different. First, delay must be forbidden if any process is in a committed location. Second, if there are processes that are in a committed location one of them must take part in the next transition. The transition rules are described in terms of the transitions for a network without committed location. In the description $\rightarrow_c$ denote transitions for a network with committed locations and

$\rightarrow$ denote transitions for a network without.

- $\langle \bar{l}, u \rangle \xrightarrow{\epsilon(t)}_c \langle \bar{l}, u+t \rangle$ if $\langle \bar{l}, u \rangle \xrightarrow{\epsilon(t)} \langle \bar{l}, u+t \rangle$ and $C(\bar{l}) = \emptyset$
- $\langle \bar{l}, u \rangle \rightarrow_c \langle \bar{l}[l'_i/l_i], u' \rangle$ if $\langle \bar{l}, u \rangle \rightarrow \langle \bar{l}[l'_i/l_i], u' \rangle$ and either $l_i \in C(\bar{l})$ or $C(\bar{l}) = \emptyset$
- $\langle \bar{l}, u \rangle \xrightarrow{\tau}_c \langle \bar{l}[l'_i/l_i][l'_j/l_j], u' \rangle$ if $\langle \bar{l}, u \rangle \xrightarrow{\tau} \langle \bar{l}[l'_i/l_i][l'_j/l_j], u' \rangle$ and either $l_i \in C(\bar{l})$, $l_j \in C(\bar{l})$ or $C(\bar{l}) = \emptyset$

3 Representing Symbolic States

The 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.

In this section we will present different ways to represent symbolic states, explain their strengths and weaknesses and give hints on when to use them. But we start one level above, between the logical level with location vectors integer assignments and clock zones and the physical representation that is the focus of this section.

The encoding of the location vector and the integer assignment is, at this level, straight forward. For the location vector, we start by numbering the locations in each process. Then instead of a vector of locations, we get a vector of location numbers. For the integer assignment we number all the integer variables in the system and represent the assignment as a vector of integers, where the i:th element is the value of the variable with number i.

Representing the clock zone is a little 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 DBM:s and operations on them can be found in [Dil89].

Now we zoom in on the representation of a single bound in the DBM. Since we want a finite symbolic state space, we only consider normalised clock zones. Then, given a maximum constant k, the maximum number of significant values for any clock bound is $2 \cdot (2k + 2)$. (Each bound is either in the interval $[-k, k]$ or infinite, and it can be strict or non-strict[2].)

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. However, due to an early design decision, this is not the case in the UPPAAL implementation. In UPPAAL the symbolic state is split into three different objects: a state object containing a location vector and pointers to an integer assignment and a clock zone, an integer assignment object representing the integer values and an object representing the clock zone. A sketch of the connection between the objects is shown in Figure 9

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 will fit in much less 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.

[2]For the infinite bound only strict is needed but non-strict is included for simplicity.

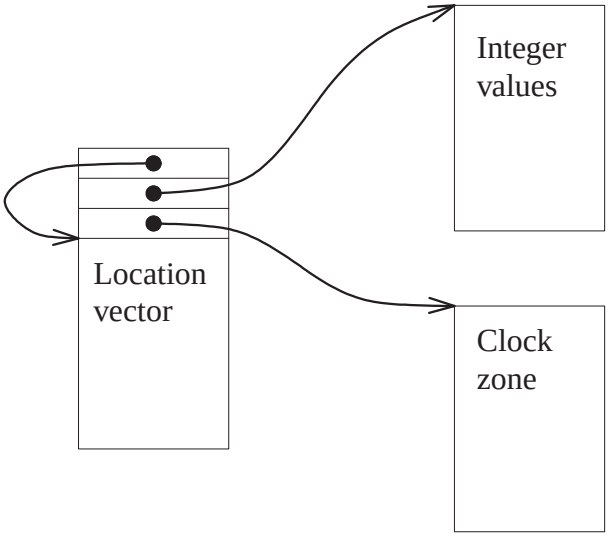


Figure 9: The symbolic state representation used in UPPAAL.

3.2 Packed States

The second representation is on the opposite side of the spectra compared to the previous one and it can be used both 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:

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

Note that in this context $\langle l, D \rangle$ is a sequence of numbers. The encoding $E(\langle l, D \rangle)$ is often too large to fit in a machine word and it have to be in fixed precision; some kind of bignums are needed. In our prototype implementation we used the GMP bignum package [Gra00].

Proposition 3 *The representation of states using bit string encoding is canonical and minimal in terms of space usage.*

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.

In order to test the performance of this representation, it is implemented in the PASSED structure in UPPAAL. In the implementation the packing algorithm has been re-shuffled in order to eliminate the use of temporary bignum variables. The implemented algorithm is listed as Algorithm 2.

Algorithm 2 Algorithm used to pack states

Parameters:

$\bar{l}$	–	Location vector
d	–	Integer assignment vector
D	–	DBM representation of zone

$E \Leftarrow \bar{l}[0]$
for $i = 1$ **to** $\#proc$ **do**
  $E \Leftarrow E * \text{\#states}(A_i) + \bar{l}[i]$
end for
for $i = 1$ **to** $\#var$ **do**
  $E \Leftarrow E * \text{domain}(d[i]) + d[i]$
end for
for $i = 0$ **to** $\#clock$ **do**
  for $j = 0$ **to** $\#clock$ **do**
    if $i \neq j$ **then**
      $E \Leftarrow E * 2k + D[i, j] + k$
    end if
  end for
end for

In the experiment two different algorithms to check for visited states are used, one version where only equality checking is implemented and the other where inclusion checking is used for the clock zone part of the states. The version with only equality is straight forward. The version with inclusion checking for the clock

zone is a little more complicated. Now we have to unpack (at least) the zone part of the state in order to compare the DBMs bound by bound. This is described as Algorithm 3

Algorithm 3 Algorithm used to compare packed states.

Parameters:
E_1 – First state in comparison
E_2 – Second state in comparison
inclusion$(E_1, E_2) \Leftarrow$ tt, inclusion$(E_2, E_1) \Leftarrow$ tt
cmp$_1 \Leftarrow E_1$, cmp$_2 \Leftarrow E_2$
for $i = \#clock$ **downto** 0 **do**
  for $j = \#clock$ **downto** 0 **do**
    if $i \neq j$ **then**
      inclusion$(E_1, E_2) \Leftarrow$ inclusion$(E_1, E_2) \wedge$ (cmp$_1 \mod k <=$ cmp$_2 \mod k$)
      inclusion$(E_2, E_1) \Leftarrow$ inclusion$(E_2, E_1) \wedge$ (cmp$_2 \mod k <=$ cmp$_1 \mod k$)
      cmp$_1 \Leftarrow$ cmp$_1/k$
      cmp$_2 \Leftarrow$ cmp$_2/k$
    end if
  end for
end for
if cmp$_1 \neq$ cmp$_2$ **then**
  return 'E_1 is not related to E_2'
end if
case
  $\square$ inclusion$(E_1, E_2) \wedge$ inclusion$(E_2, E_1) \Rightarrow$ **return** '$E_1 = E_2$'
  $\square$ inclusion$(E_1, E_2) \wedge \neg$inclusion$(E_2, E_1) \Rightarrow$ **return** '$E_1 \subseteq E_2$'
  $\square \neg$inclusion$(E_1, E_2) \wedge$ inclusion$(E_2, E_1) \Rightarrow$ **return** '$E_1 \supseteq E_2$'
  $\square \neg$inclusion$(E_1, E_2) \wedge \neg$inclusion$(E_2, E_1) \Rightarrow$ **return** 'E_1 is not related to E_2'
end case

In Table 1 we see the performance for the packed representation with only equality checking for the clock zone. In the table the results are given both by measured numbers and relative to the current UPPAAL implementation. First, note that for the Field Bus example the verification does not terminate normally. (Denoted by $\perp$ in the table.) The reason for this is that when only equality checking is used large parts of the state space are revisited, which makes WAIT expand until the verification process run out of memory. For the other examples the representation is very good, with space savings of up to 67% for Fischers protocol and 57% for

Example	Time		Space	
	real (Sec)	relative	real (MB)	relative
Field Bus (Faulty 1)	$\perp$	$\perp$	$\perp$	$\perp$
Field Bus (Faulty 2)	$\perp$	$\perp$	$\perp$	$\perp$
Field Bus (Faulty 3)	$\perp$	$\perp$	$\perp$	$\perp$
Field Bus (Fixed)	$\perp$	$\perp$	$\perp$	$\perp$
B&O	20.42	1.09	10.48	0.48
DACAPO (big)	326.91	1.14	43.14	0.47
DACAPO (small)	18.50	1.30	6.60	0.78
Fischer 5	14.53	0.90	4.45	0.47
Fischer 6	733.91	0.54	48.34	0.33

Table 1: Performance for packed states without inclusion checking

a more realistic example[3] at a very moderate slowdown (or even a speedup for Fischers protocol).

The result of the experiment with packed states where zone inclusion checking was implemented using division and modulo is shown in Table 2 (as absolute figures and in relation to the current PASSED implementation in UPPAAL). We note that using this representation we are able to verify all the examples and that the space performance is very good. However the time performance is very poor, for one instance 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

[3]Fischers protocol behaves, as mentioned in Appendix A, different from all other examples, with respect to verification.

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 2: Performance for packed states with expensive inclusion checking

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 all bounds 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 setting 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 where all test bits are set to one. 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 $D \supset D'$.

Example 1 *Consider a system with two clocks x, y and the maximum constant 2. In order to cut away some unnecessary detail, we don't consider strictness of bounds. The number of bits needed to store all possible values of one clock bound in this system is 3. Let $D = \{x - y \leq 1, y - x \leq 1\}$ and $D' = \{x - y \leq 1, y - x \leq 2\}$ be two zones that arise from a verification of this system.*

$$E(D) = \left\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 0 & 001 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 0 & 001 \\ \hline \end{array} \right\|$$

$$E(D') = \left\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 0 & 001 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 0 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 0 & 010 \\ \hline \end{array} \right\|$$

To check if $D \subseteq D'$, start by setting all the test bits in D' to one, e.g. *by doing a bitwise or with the precomputed mask M. The extra bits set to one will serve two purposes. As mentioned above they will indicate the result of the comparison, but they will also serve as borrow bits and prevent interference between the packed bounds.*

$$E(D') \mid M = \left\| \begin{array}{|c|c|} \hline 1 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 1 & 011 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 1 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 1 & 001 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 1 & 011 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 1 & 010 \\ \hline \end{array} \right\|$$

The actual comparison is then made by a subtraction. All the packed bounds will be subtracted by one subtraction operation, and since the test bits are set in the first term and unset in the second term the bounds will not interfere with each other.

$$E(D') \mid M - E(D) = \left\| \begin{array}{|c|c|} \hline 1 & 000 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 1 & 000 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 1 & 000 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 1 & 000 \\ \hline \end{array} \left\| \begin{array}{|c|c|} \hline 1 & 000 \\ \hline \end{array} \right\| \begin{array}{|c|c|} \hline 1 & 001 \\ \hline \end{array} \right\|$$

Now the result of the inclusion check is read out of the test bits. Since all the test bits are one we conclude that $D \subseteq D'$.

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, such as GMP, 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 padded with zero bits.

Example 2 *Assuming a nine-bit machine, the packed zone $E(D')$ from Example 1 is represented as follows:*

100110010	110011000	000001100

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 4. In the description we use $E(D)[i]$ to denote the i:th limb of $E(D)$ and $-_w$ to denote a binary subtraction of machine word size.

Algorithm 4 Inclusion check for packed zones

```
b ⇐ 0
for i = 1 to #limbs do
  cmp ⇐ (M[i] | E(D')[i]) −_w (E(D)[i] + b)
  if cmp ≠ M[i] then return "false"
  if (M[i] | E(D')[i]) < (E(D)[i] + b) then
    b ⇐ 1
  else
    b ⇐ 0
  end if
end for
return "true"
```

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 3, both as absolute figures and compared to the standard state representation. We note that using this method the space usage is typically reduced with around 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[4].

4 Representing Symbolic State-Space

The two key data structures in a model checker is, as mentioned before, WAIT, that keeps track of states not yet explored, and PASSED, that keeps track of states already visited. Both theses data structures tend to be large, and how to represent

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

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 3: Performance for packed states with cheap zone coding

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 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.

In the following presentation we say that a symbolic state $\langle \bar{l}, d, D \rangle$ is included in a another symbolic state $\langle \bar{l}', d', D' \rangle$ if they have the same discrete part (*i.e.* $\bar{l} = \bar{l}'$ and $d = d'$) and $D \subseteq D'$. For simplicity we will not separate the discrete parts from each other in the presentation, and from now on we will write $\langle l, D \rangle$ for $\langle \bar{l}, d, D \rangle$.

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

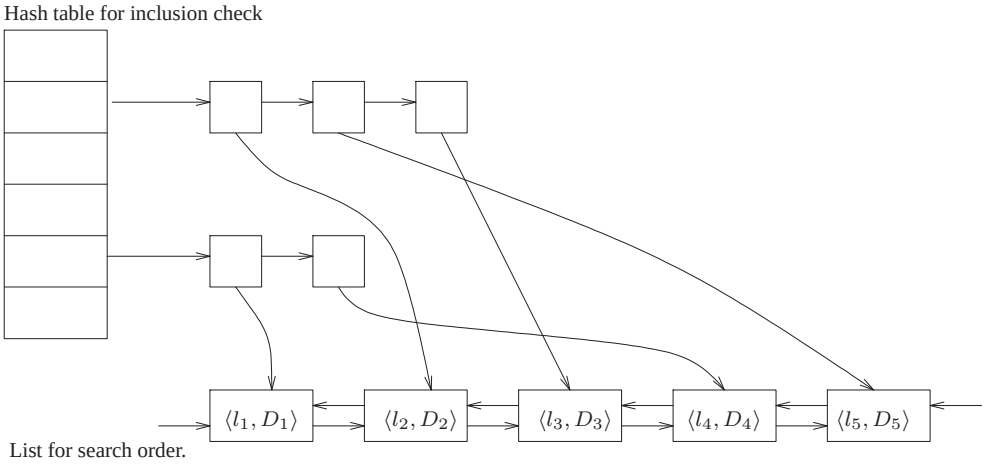


Figure 10: Structure of WAIT

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 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 10.

To test the performance of this solution we compared the space and time needed to explore the state space of the examples mentioned in section A, for one version of UPPAAL without inclusion checking on WAIT and one version with the combined scheme. The result is shown in Table 4. It is worth noting that the version with inclusion checking is both significantly faster and less memory consuming than

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 4: Performance impact of inclusion check on WAIT

the version without inclusion checking, for all examples except Fischers protocol which is, as mentioned earlier, 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]. The layout of the standard PASSED structure is illustrated in Figure 11.

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 the 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

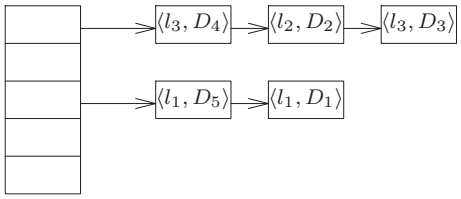


Figure 11: Structure of PASSED.

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 a 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 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 effect 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:

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

To simplify the algorithm and to eliminate the need for bignum integers, we push the modulo operation as far as possible. The resulting operation is shown, as

pseudo-code, in Algorithm 5. 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.

Algorithm 5 Hash function used in supertrace algorithm

Parameters:

$\bar{l}$	–	Location vector
d	–	Integer assignment vector
D	–	DBM representation of zone
k	–	The size of the hash table

$E \Leftarrow \bar{l}[0] \mod k$
for $i = 1$ **to** $\#proc$ **do**
  $E \Leftarrow (E * \text{\#states}(A_i) + \bar{l}[i]) \mod k$
end for
for $i = 1$ **to** $\#var$ **do**
  $E \Leftarrow (E * \text{domain}(d[i]) + d[i]) \mod k$
end for
for $i = 0$ **to** $\#clock$ **do**
  for $j = 0$ **to** $\#clock$ **do**
    if $i \neq j$ **then**
      $E \Leftarrow (E * 2k + D[i, j] + k) \mod k$
    end if
  end for
end for

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 number of explored states are increased. This leads to longer verification times and harder pressure 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 5. 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 on that the hash function is far from optimal on this example.

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.)

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

To see how the supertrace algorithm behave time-wise we made an experiment where the verification time was measured. 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. To compare the verification speed we used two different versions of the classic PASSED implementation as reference, one version where inclusion checking between states were switched off and the standard version where inclusion checking between states is used to optimise the search. The version without inclusion checking is included since the number of states explored when it is used is in the same order of magnitude as for the supertrace PASSED. The standard version is included to compare the performance for the supertrace PASSED to the PASSED implementation that is normally used when working with the tool. The result of this experiment is presented in Table 6. As we see in the table the times for the supertrace is in the same order of magnitude as the standard PASSED implementation in UPPAAL.

Example	Supertrace			Classic	
	16MB	32MB	64MB	exact	inclusion
Philips (Correct)	2.39	2.58	2.97	2.18	1.91
Philips (Erroneous)	98.83	98.18	102.20	270.91	22.22
B&O	15.59	16.20	16.30	17.37	16.57
DACAPO (big)	268.32	268.04	269.67	294.77	254.01
DACAPO (small)	15.32	15.47	15.90	15.56	12.45
Fischer 5	11.57	13.01	12.20	18.53	14.36
Fischer 6	688.28	681.67	686.75	1675.04	1217.88

Table 6: Time (in seconds) to explore the entire state space for three different supertrace PASSED lists and two versions of the standard PASSED list

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. A sketch of this is shown in Figure 12. 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

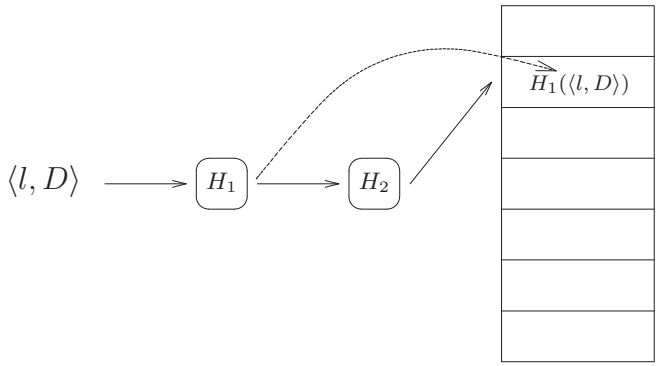


Figure 12: Hash compaction with hash table key based on signature

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 13.

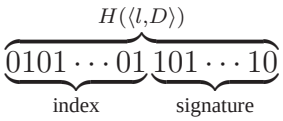


Figure 13: The table index and the signature as one hash value.

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 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 lays in what we store in the hash table. In the first implementation we store signatures of entire symbolic states. This solution gives very a 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 report.

As an introductory experiment all the examples are run with 47-bit signatures[5] 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 since the first PASSED implementation only can detect equal states and not inclusion between states. In contrast to

[5]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.

this example, the large instance of the DACAPO example terminates for all sizes using the first PASSED implementation, while it fail to do so for 16MB and 32MB using the second. This is due to that, 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 big 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.[6] The result of this experiment can be seen in Table 7. 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.

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

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

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 8. We note from the table that the combined scheme (signatures of the discrete part + packed zone) is somewhat 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

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

needed to compute the signatures and partly due to that the hash compaction implementation within UPPAAL is partly a prototype.

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

The measured memory use for the different examples is listed in Table 9. 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 report 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 with up to 70% compared to the current representation used in UPPAAL, at the price of 3–13 times slowdown due

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

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 report describes how to improve performance by checking for already visited states not only on PASSED, but also on WAIT. For the evaluated examples this optimisation reduces the verification time with 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‰ and 0.04‰ 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 omissions are less than $1/10$:th of the omissions in the supertrace PASSED representation.

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

Acknowledgement: We would like to thank Wang Yi for valuable comments and discussions on the content of this report.

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*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Pro-*

ceedings, 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.

[ID93] C. Norris Ip and David L. Dill. Better verification through symmetry. In *Proceedings, Eleventh International Conference on Computer Hardware Description Languages and their Applications*, volume 32 of *IFIP Transactions A: Computer Science and Technology*, pages 97–112. North-Holland, 1993.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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

To evaluate the the ideas presented in this report we have made a number of experiments. In this section we describe the environment in which the experiments are performed. The results themselves are presented together with the ideas behind them.

The experiments are run on a Sun Ultra Enterprise 450 with four[7] 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 sim-

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

ple 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 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 B:

Partial Order Reductions for Timed Systems

Johan Bengtsson, Bengt Jonsson, Johan Lilius and Wang Yi. In Proceedings, Ninth International Conference on Concurrency Theory, volume 1466, Lecture Notes in Computer Science, Springer Verlag, 1998

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 synchronization 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 *resynchronize* 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 resynchronization 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 interleavings 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 synchronized. 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 synchronization between the processes or their clocks.

A simple illustration of this problem is given in Fig. 1. In (1) of Fig. 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 synchronization state (m, s) where the automata may synchronize on label a (we use the synchronization 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 synchronization on a is possible. A partial order reduction technique will explore only one of these two interleavings, after having analyzed that the initial transitions of the two automata are independent.

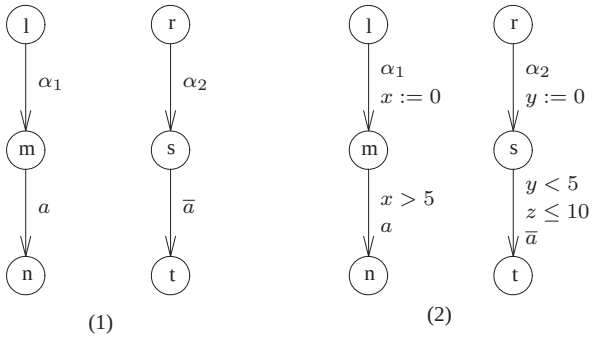


Figure 1: Illustration of Partial Order Reduction

Let us now introduce timing constraints in terms of clocks into the example, to obtain the system in (2) of Fig. 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 initialization of the system. We note that the initial transitions of the two automata are logically independent of each other. However, if we naively analyze 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 synchronization 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 synchronization 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 synchronization between processes by letting clocks advance independently of each other. In other words, we *desynchronize* 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 Fig. 1. Suppose that in state (l, r) all clocks are initialized 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 "desynchronized" 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 desynchronization 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 desynchronized 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 synchronization between local time scales completely, we also lose timing information required for synchronizaton 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 synchronization on a should be within 10 time units from system initialization. Implicitly, this then becomes a requirement on the left automaton. A naive desynchronization of local clocks including z will allow the left process to wait for more than 10 time units, in its local time scale, before synchronizing. Therefore, before exploring the effect of a transition in which two automata synchronize, we must explicitly "resynchronize" 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 synchronization 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 initialization. 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 synchronize 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 synchronization 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 synchronization of clocks.

The paper is organized 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, LPY95a] 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 *synchronizing*. If a is a synchronizing label, then it has a *complement*, denoted $\overline{a}$, which is also a synchronizing 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 *synchronizing 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 synchronizing 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 conjuction $I_1(l[1]) \wedge \cdots \wedge I_n(l[n])$. The initial state of A is (l^0, u^0) where l^0 is the control vector such that $l[i] = l^0_i$ 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 $l_i \stackrel{g,a,r}{\longrightarrow} l'_i$ such that $u \models g$ and $u' = [r \mapsto 0]u$.
- Synchronizing Transition: $(l, u) \longrightarrow (l[l'_i/l_i][l'_j/l_j], u')$ if there exists a synchronizing action $l_i \stackrel{g_i,a,r_i}{\longrightarrow} l'_i | l_j \stackrel{g_j,\bar{a},r_j}{\longrightarrow} l'_j$ 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 synchronizing 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 synchronizing) 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 characterizes 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 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 $l_i \stackrel{g,a,r}{\longrightarrow} l_i'$ is enabled at l if $l[i] = l_i$, and a synchronizing action $l_i \stackrel{g_i,a,r_i}{\longrightarrow} l_i' | l_j \stackrel{g_j,\overline{a},r_j}{\longrightarrow} l_j'$ 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,

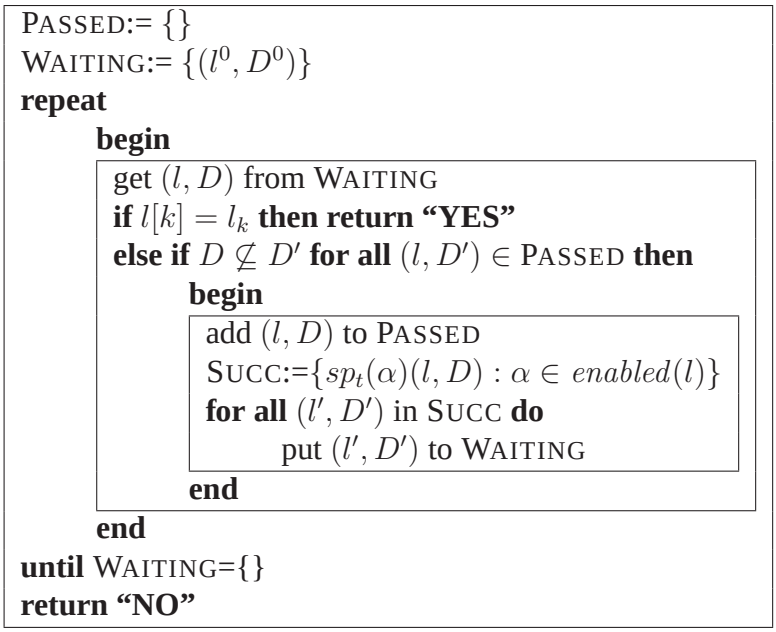


Figure 2: An Algorithm for Symbolic Reachability Analysis.

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 modeled by globally

synchronous transitions, which implicitly synchronize 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 synchronizing actions we must also be able to "resynchronize" 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 unsynchronized state, the reference clocks of different automata can be quite different. Before a synchronization between A_i and A_j, we must add the condition that the reference clocks of A_i and A_j are equal.

To formalize the above ideas further, we present a local-time semantics for networks of timed automata, which allows local clocks to advance independently and resynchronizing them only at synchronization 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 $u' = [r \mapsto 0]u$
- Synchronizing Transition: $(l,u)\mapsto(l[l'_i/l_i][l'_j/l_j],u')$ if there exists a synchronizing 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 synchronize, 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 synchronized 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 synchronized 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 synchronizing 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 synchronizing 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 characterization theorem.

Theorem 3 *For all networks, a synchronized 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 characterizes 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 *synchronized* in the sense that it constains synchronized states. The searching for such a synchronized 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 synchronized 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 synchronized 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 $\widehat{sp_t}(\alpha_1)$ and $\widehat{sp_t}(\alpha_2)$ 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 synchronized states. In the following we shall use C_r to denote the set of reference clocks.

Definition 3 *Two synchronized local time states (with the same control vector) (l, u) and (l, u') are synchronized-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 *synchronized regions*. Now we extend this relation to cope with local time states that are not synchronized. Intuitively, we want two non-synchronized states, (l, u) and (l', u') to be classified as equivalent if they can reach sets of equivalent synchronized 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 synchronized 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 synchronized 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 synchronized states i.e. states where all the automata of the network have been synchronized in time.

Note that the number of synchronized 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 function 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

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

by $\text{SUCC}:=\{\widehat{sp_t}(\alpha)(l, D) : \alpha \in ample(l)\}$ where $ample(l) \subseteq enabled(l)$ is a subset of the actions that are enabled at l. Hopefully the set $ample(l)$ can be made significantly smaller than $enabled(l)$, 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 $ample(l)$ 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 $ample(l)$ 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 $ample(l) = \emptyset$ if and only if $enabled(l) = \emptyset$.

C1 If the automaton $A_i \in \mathcal{A}$ from its current node $l[i]$ can possibly synchronize with another process A_j, then $A_j \in \mathcal{A}$, regardless of whether such a synchronization 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 synchronize with another automaton A_j, then we must explore actions by A_j to allow it to "catch up" to a possible synchronization with A_i. Otherwise we may miss to explore the part of the state-space that can be reached after the synchronization between A_i and A_j.

A final necessary criterion for correctness is *fairness*, i.e., that we must not indefinitly neglect actions of some automaton. Otherwise we may get stuck exploring a cyclic behavior 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].

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 analyze) DBM's have been developed (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

[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.

$(x_i^l - c_i) - (x_j^k - c_j) \sim n$ 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 synchronization 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 resynchronization 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 resynchronization 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 Thm. 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 synchronization. Since a state only contains information on the relative differences of firing times of transitions it is not possible to synchronize 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 C:

Automated Analsysis 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. In Proceedings, Ninth International Conference on Computer Aided Verification, volume 1102, Lecture Notes in Computer Science, Springer Verlag, 1996

Automated Analsysis 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 where 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 verified 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 in their audio products, and constructed a possible execution sequence explaining a known 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 truly 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

During the last few years a number of tools for automatic verification of hybrid and real-time systems have emerged, e.g. HYTECH [HHWT95], KRONOS [DY95], Polka [HRP94], RT-Cospan [AK95] and UPPAAL [BLL+95]. These tools have by now reached a state, where they are mature enough for industrial applications. We

hope to 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 [LPY95b] and KRONOS [DY95]. 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 had 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 node-space is 10^3 times larger than the case without bus collision handling and the number of clocks, variables and channels 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 be-

haviours 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 to guide the state-space exploration of the model checker to avoid exploring unnecessary interleavings of independent transitions. Our experimental results demonstrate truly 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.

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, we in this paper present an application of the current version of UPPAAL, also supporting committed location, installed on an ordinary Pentium 150 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 Philip 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 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 behaviours such as 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.

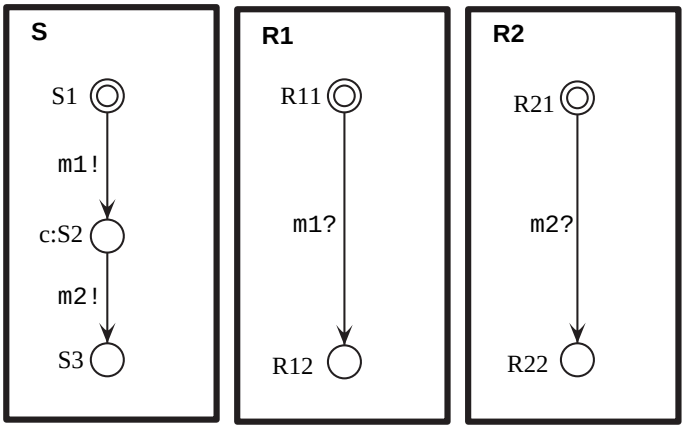


Figure 1: Broadcasting Communication and Committed Locations.

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 $m_1!m_2!$ 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 name(a) to denote the action name of a, defined by name$(\alpha?) =$ 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* l of a network $\overline{A}$ is a vector of locations where l_i is a location of A_i. We write $l[l'_i/l_i]$ to denote the vector where the ith element l_i of l is replaced by l'_i. Furthermore, we shall write $C(l)$ whenever $C(l_i)$ for some i.

A *state* of a network $\overline{A}$ is a configuration (l, u) where l is a control vector of $\overline{A}$ and u is a variable assignment. The initial state of $\overline{A}$ is (l^0, u^0) where l^0 is the initial control vector whose elements are the initial locations l^0_i 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:

- $(l, u) \rightsquigarrow (l[l'_i/l_i], r_i[u])$ 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 $C(l_i)$.

- $(l, u) \rightsquigarrow (l[l'_i/l_i, l'_j/l_j], (r_j \cup r_i)[u])$ 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 $C(i)$ or $C(j)$.

- $(l, u) \rightsquigarrow (l, u \oplus d)$ if $I(l)(u)$, $I(l)(u \oplus d)$, $\neg C(l)$ 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 $I(l) = \bigwedge_i I(l_i)$.

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 components operating in committed location are required to participate in the next transition, which must be an action transition. Furthermore, transition sequences marked as committed are *instantaneous* in the sense that they happen without duration, and *non-interleaved* (or indivisible) as they are never interfered by other components.

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. It combines constraint-solving 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 (l, D), where D is a constraint system (i.e. a conjunction of clock constraints and non-clock constraints). It checks if a symbolic state (l^f, D^f) is reachable from the initial symbolic state (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 (l, D) a state instead of symbolic state.

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 (l^0, D^0). The algorithm then repeats the following steps:

S1. Pick a state (l, D) from the WAITING buffer.

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

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

b. Otherwise, save (l, D) in the PASSED buffer.

S4. Find all successor states (l_s, D_s) reachable from (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.

The use of committed location to model atomic behaviours render possible two potential reductions of the PASSED buffer size. 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 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:

S3′. a. If $C(l)$ go directly to step S4.

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

c. If neither of the above steps are applicable, save (l, D) in the PASSED buffer.

So, for a given state (l, D), if 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 l is committed. In fact, the next transition must be an action transition and it must involve all l_i which are committed in l (according to the transition rules in the previous section). This reduces the time spent on generating successor states of (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 l is *not* committed as it involves a search through the PASSED buffer.

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 parameterizable sce-

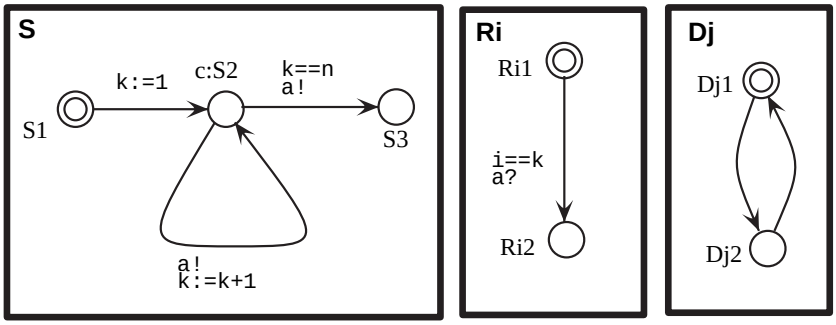


Figure 2: Broadcasting Using Committed Locations.

nario, 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 synchronizing 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 test sequences, with S_2 declared as respectively not committed and committed. 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.

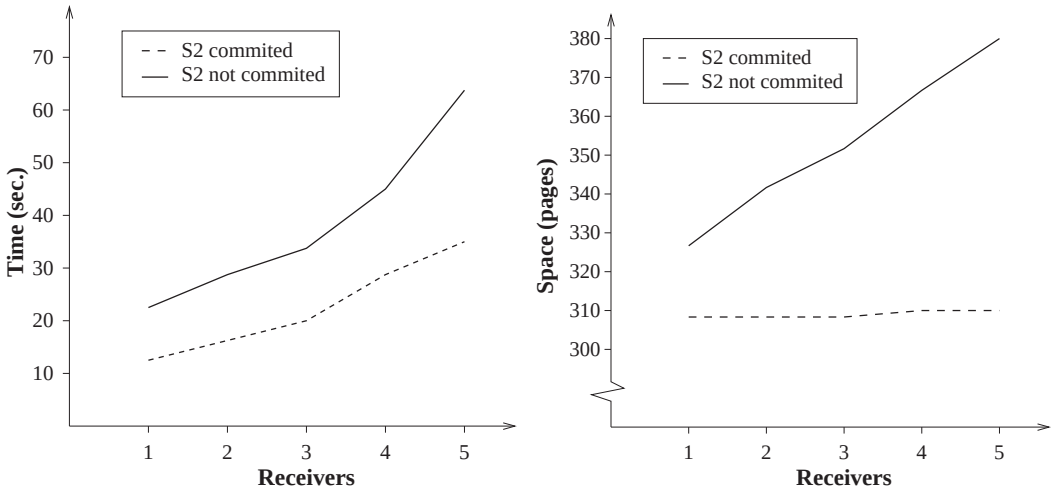


Figure 3: Time and Space Consumption.

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 changes 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 nice 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 to decode a Manchester encoded message by only looking to the up-going messages (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 behavior 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

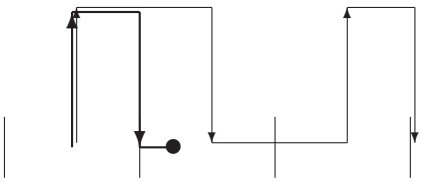


Figure 4: An Example.

detect a bus collision. This collision detection happens at certain points in time. Just before each up-going transition, and at one and three 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 retransmit 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 an other 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 is hard to distinguish from one line, 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 does a down, 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 this up-going edge must be at most 20 microseconds. The timing uncertainty on the collision detection on one and three quarters after the generation of a down-going edge is ± 22 microseconds. Also the receiver has a timing uncertainty of $\pm 5\%$. And, 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).

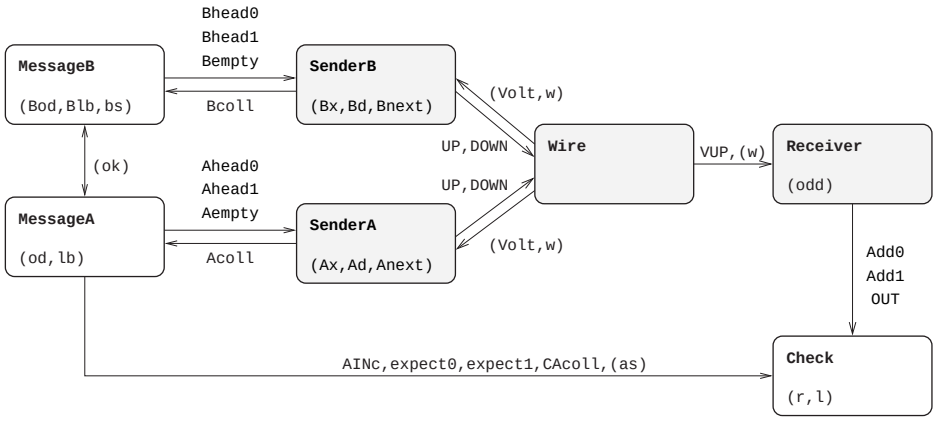


Figure 5: Philips Audio-Control Protocol with Bus Collision.

5 A Formal Model of the Protocol

To analyse the behavior 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 and an output checker, is used to model assumptions about the environment of the protocol and for testing the behavior 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 (enclosed within parenthesis).

The general idea of the model is as follows. The two automata MessageA and MessageB generate messages for the both senders, 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. 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 this symmetry, it is enough to check that the messages transmitted by sender A are received correctly. 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 A.

The Senders

SenderA is depicted in Figure 9. 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 10.

The Receiver

Receiver, shown in Figure 8, decodes the bit sequence using the up-going (modeled as VUP?) changes of the wire. Decoded bits are signaled to the environment using output actions Add0!, Add1! and OUT! (where OUT! is used for signaling 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 modeling the decoding. This decoding happens at once, therefore the intermediate locations are modeled 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 to 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 messages of odd length for sender A and B respectively. Furthermore, the messages generated for sender A are communicated to the checker. The start of a message is signaled 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.

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. 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 protocol formally described in the previous section. We will use $A.l$ to denote the (implicit) proposition at(A, 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:

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

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:

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

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 comes down 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 counterexample trace to Property C.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 makes that the distance between the second UP's is quite 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 the ideal 6Q. And 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". 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. This is prevented by, among others, the following: It is not likely that two senders do start at (roughly) the same time. The timing uncertainty is at most 2% instead of 5%. And the "average" timing uncertainty is even less. And finally, the source address is in the beginning of the messages, this makes the senders detect the collision. See [Gri94] for more details.

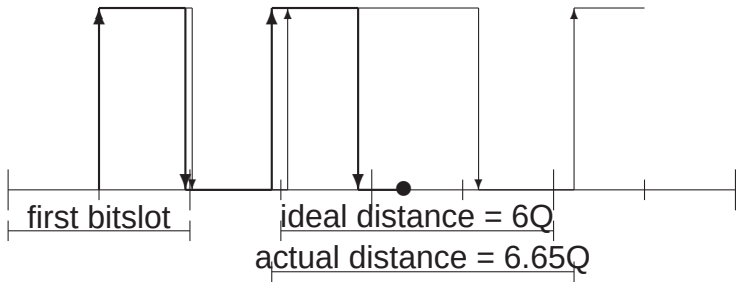


Figure 6: Error execution of the incorrect protocol.

Although this problem was know 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 C.1 and C.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 C.1 and C.2 was performed in 12.75 sec using 2.1 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 4.5 sec using 1.8 MB of memory. Also, attempts to verify Property C.1 for the full protocol with an error tolerance of 6% on the timing failed. 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 2.17 [LPY97a, BLL+98] that implements the verification algorithm for handling committed locations described in Section 3. It was installed on a Pentium 150 MHz MMX running Red Hat Linux 5.0. In the conference version of this paper [BGK+96] we reported that the same protocol was verified using UPPAAL version 0.96[1] 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 have been reduced

[1]The two UPPAAL versions 0.96 and 2.17 are dated Nov 1995 and March 1997 respectively.

with over 99%. These improvements of the UPPAAL verifier are due to a number of developments in the last two years that will not be discussed further here, but we refer the reader to [LPY97b, BLL+98].

7 Conclusions

In this paper we have presented a case-study where the verification tool UPPAAL is applied to analyse a realistic audio-control protocol by Philips with bus collision handling. The protocol has received a lot of attention in the formal methods research community (see e.g. [LSW97, 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. To our knowledge, the full protocol considered in this paper has never before been automatically analysed.

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 accurate modelling of atomic behaviours. More importantly, it is also utilised to guide the state-space exploration of the model checker to avoid exploring unnecessary interleavings of independent transitions. Our experimental results demonstrate truly 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

[AD90] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of ICALP'90*, LNCS 443, 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 LNCS, Springer Verlag, pages 220–231. Springer–Verlag, October 1995.

[BGK+96] Johan Bengtsson, David Griffioen, Kåre Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. Accepted for presentation at the 8*th* Int. Conf. on Computer Aided Verification, 1996.

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

[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 the 4*th *DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995. To appear in LNCS, 1996.

[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.J.B. Bosscher, I. Polak, and F.W. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of FTRTFT'94*, LNCS 863, pages 170–192, 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, December 1995.

[Gri94] W.O.D. 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, December 1995.

[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.

[HRP94] N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of linear hybrid systems by means of convex approximations. In *Static Analysis Symposium*, LNCS 864, pages 223–237, 1994.

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

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of the 4*th *DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995. To appear in LNCS, 1996.

[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. 9*th *Int. Conf. on Computer Aided Verification*, number 1254 in LNCS, Springer Verlag, 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.

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

A The System Description

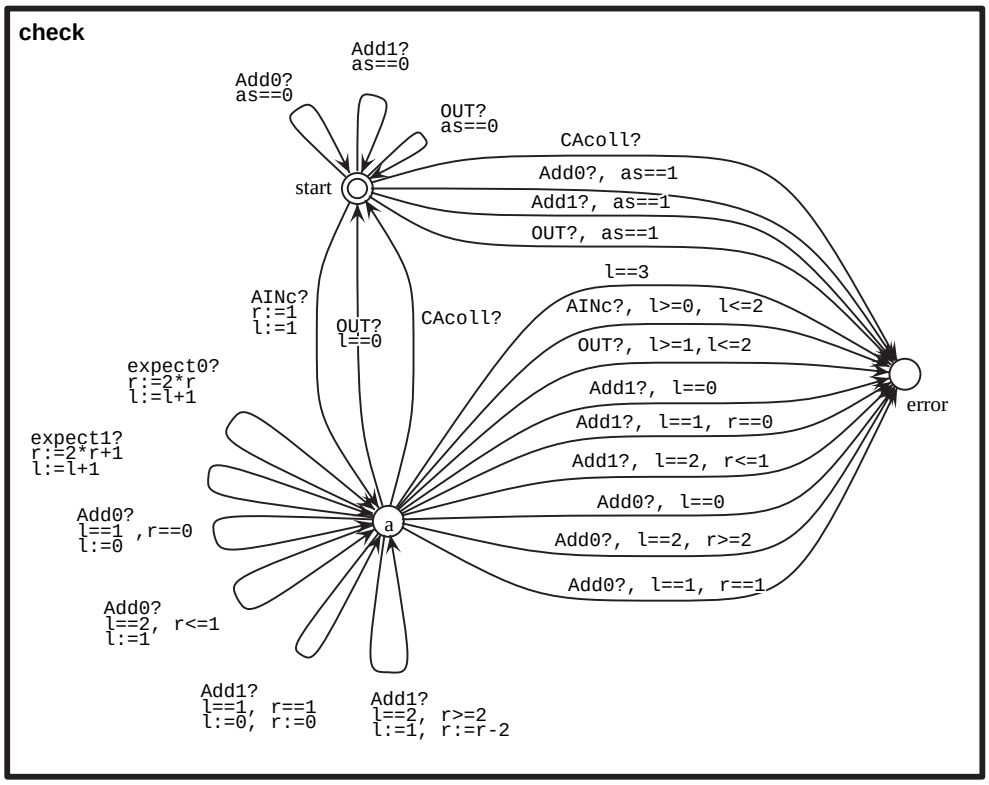


Figure 7: The Check Automaton.

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.

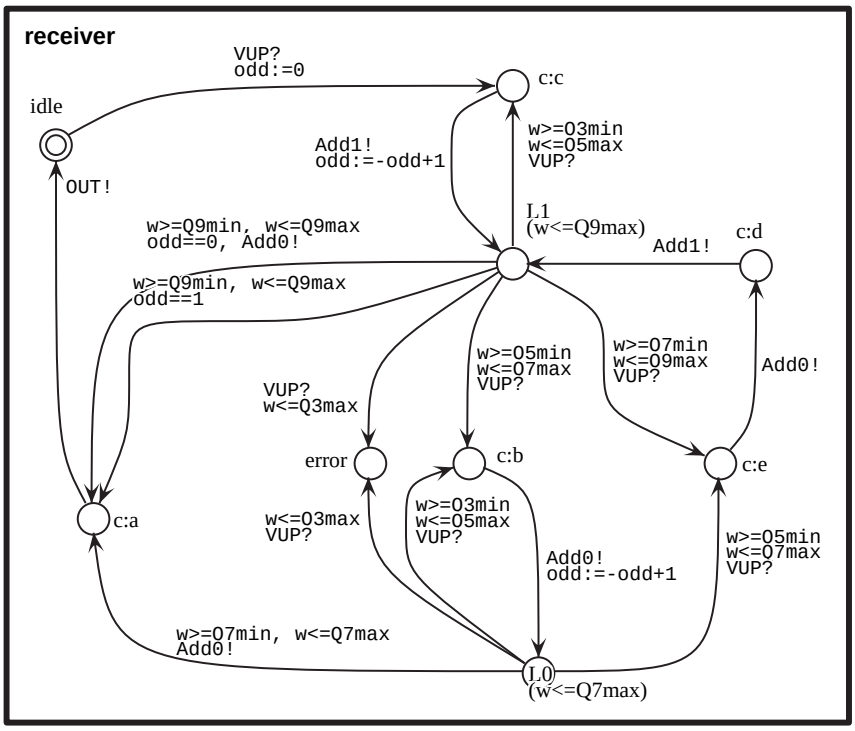


Figure 8: The Receiver Automaton.

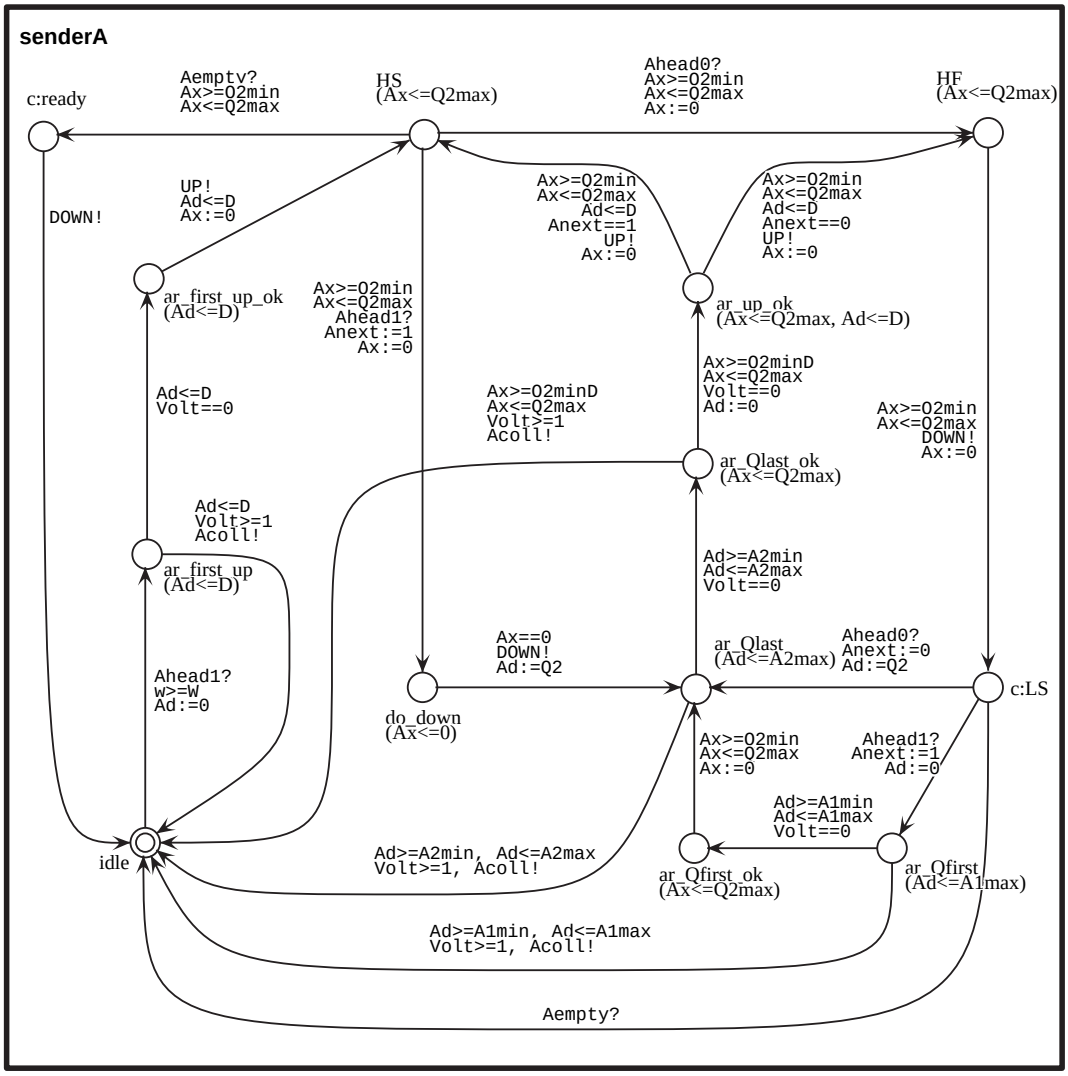


Figure 9: The SenderA Automaton.

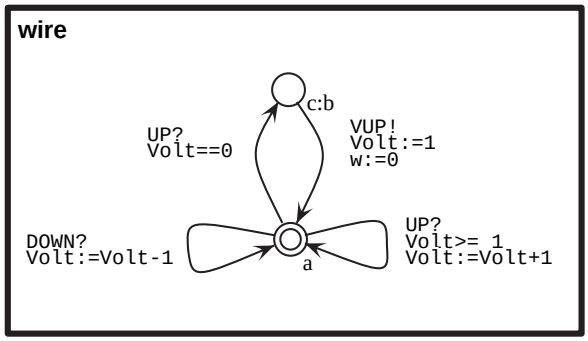
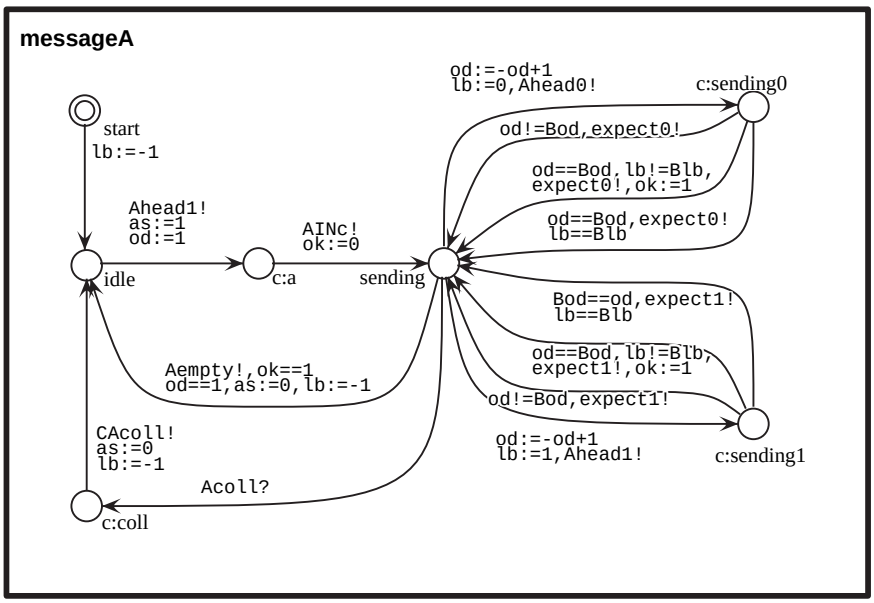


Figure 10: The Wire Automaton.



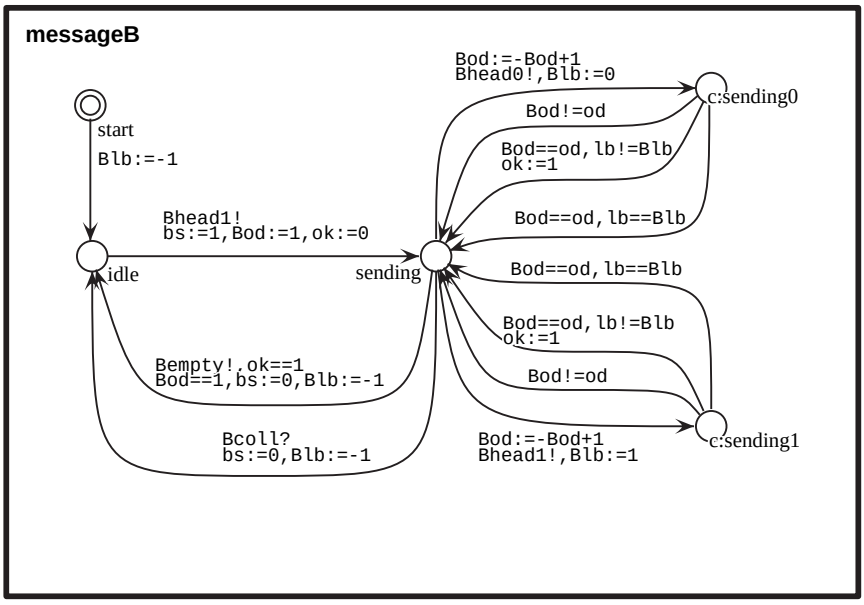


Figure 11: The Message Automata.

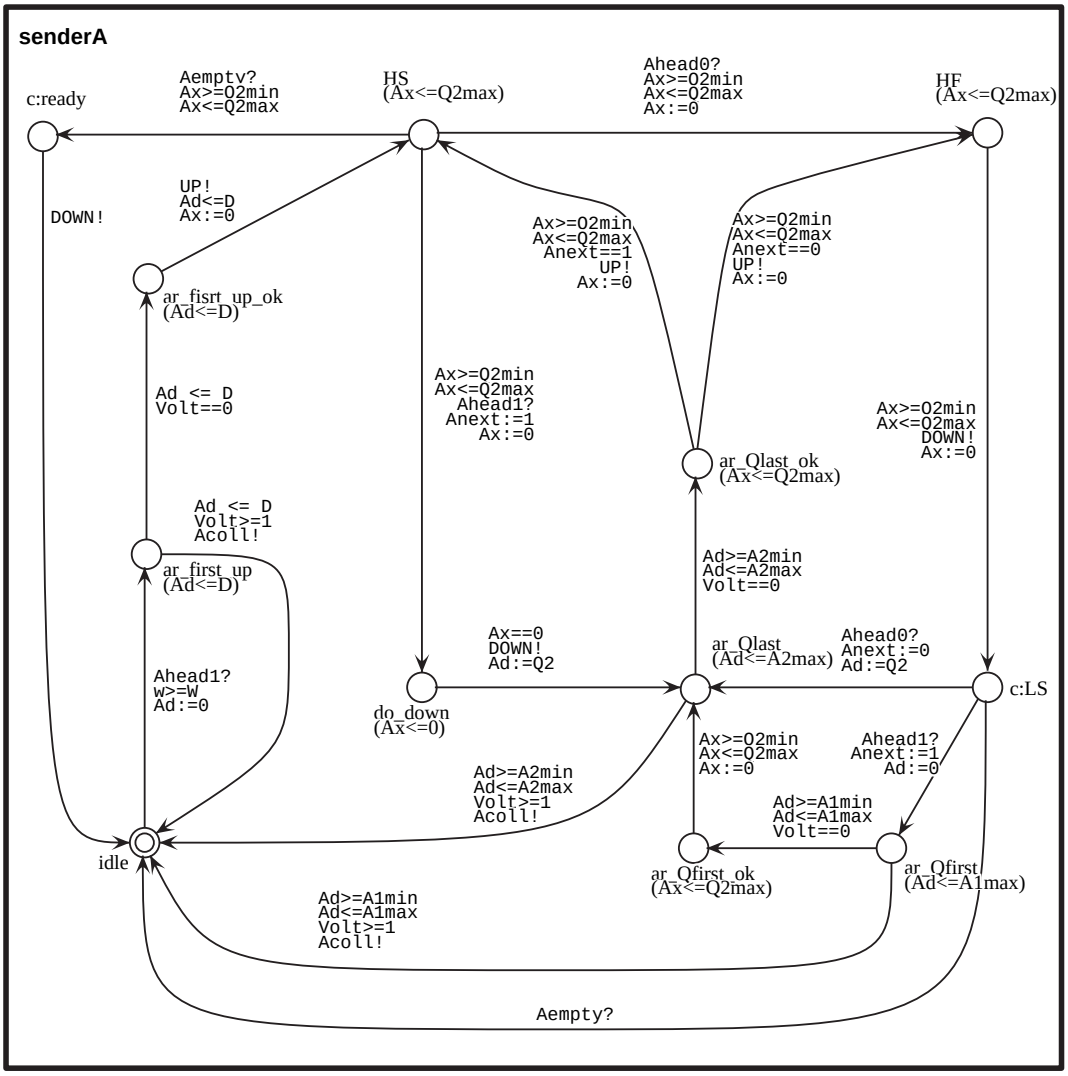


Figure 12: The Incorrect SenderA Automaton.

Paper D:

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

Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. In Proceedings, Hybrid Systems III: Verification and Control, volume 1066, Lecture Notes in Computer Science, Springer Verlag, 1995

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

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

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

Abstract. UPPAAL is a tool suite for automatic verification of safety and bounded liveness properties of real-time systems modeled as networks of timed automata. It includes: a *graphical interface* that supports graphical and textual representations of networks of timed automata, and automatic transformation from graphical representations to textual format, a *compiler* that transforms a certain class of linear hybrid systems to networks of timed automata, and a *model-checker* which is implemented based on constraint–solving techniques. UPPAAL also supports diagnostic model-checking providing diagnostic information in case verification of a particular real-time systems fails.

The current version of UPPAAL is available on the World Wide Web via the UPPAAL home page `http://www.docs.uu.se/docs/rtmv/uppaal`.

1 Introduction

UPPAAL is a new tool suite for automatic verification of safety and bounded liveness properties of networks of timed automata [YPD94, LPY95c, LPY95a]. The tool was developed during the spring of 1995 as the result of intense research collaboration between BRICS at Aalborg University and Department of Computing Systems at Uppsala University. The two main design critea for UPPAAL has been *efficiency* and *ease of usage*.

The current version of UPPAAL, as well as its future extensions, is implemented in C++. Model–checking is often hampered by various state–explosion problems.

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

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

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

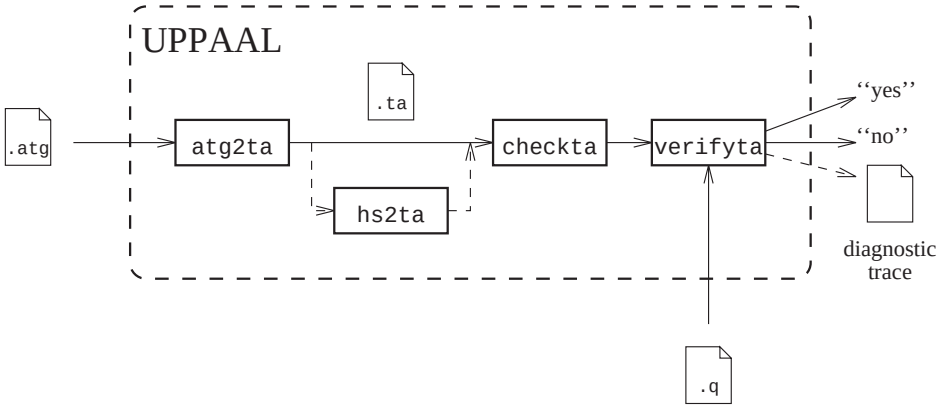


Figure 1: Overview of UPPAAL

In UPPAAL thes problems are dealt with by a combination of on–the–fly verification together with a new and coarser symbolic technique reducing the verification problem to that of solving simple linear constraint systems. The features and tools of UPPAAL includes:

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

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

2 An Overview of UPPAAL

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

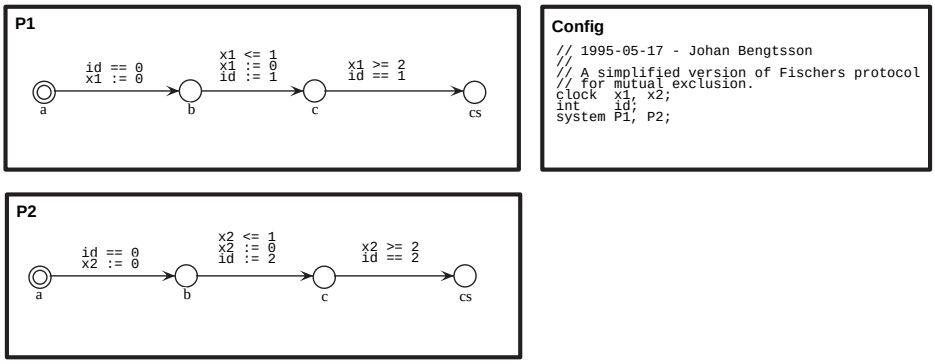


Figure 2: Graphical Description of Fischers Mutual Exclusion Protocol

2.1 Graphical Description of Networks of Timed Automata

It is possible to draw networks of timed automata using Autograph, given that certain syntactical rules are followed, e.g. the different automata in the network must be enclosed in boxes with the name of the process in the structural label, there must be a textual box describing the system configuration, i.e. declaration of clocks, channels and auxiliary integer variables. To be able to import system descriptions, drawn with help of Autograph, into UPPAAL the system must be saved in the Autograph `.atg`-format. In Figure 2 the Autograph version of Fischers Protocol [AL93, Sha93] is shown.

2.2 Textual Description of Networks of Timed Automata

In addition, UPPAAL allows networks of timed automata to be described using a textual format (called `.ta`) providing a basic *programming language for timed automata*. In certain cases we found this textual format more convenient (and faster) to work with than the graphical interface. The compiler `atg2ta` automatically transforms system description in the graphical `.atg`–format into the textual `.ta`–format, thus supporting the important principle WYSIWYV[1]. Figure 3 shows the resulting `.ta`–format for Fischers Protocol from Figure 2.

2.3 Linear Hybrid Systems

Under certain conditions, the model of timed automata may be generalized to allow clocks with rates varying between a lower and an upper bound, and to allow

[1]What You See Is What You Verify.

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

//
// Processes
//
process P1 {
   state a, b, c, cs;
   init  a;
   trans a -> b {
           guard  id == 0;
           assign x1 := 0;
         },
         b -> c {
           guard  x1 <= 1;
           assign x1 := 0, id := 1;
         },
         c -> cs {
           guard  x1 >= 2, id == 1;
         };
}

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

```
process P2 {
   state cs,c,b,a;
   init  a;
   trans c -> cs {
           guard  x2 >= 2, id == 2;
         },
         b -> c {
           guard  x2 <= 1;
           as-
sign x2 := 0, id := 2;
         },
         a -> b {
           guard  id == 0;
           assign x2 := 0;
         };
}
```

Figure 3: Textual Description of Fischers Mutual Exclusion Protocol

clock rates to change between different control-nodes (vertices) [OSY94]. This extension of timed automata is useful for modelling of hybrid systems where the behaviour of the system variables can be described or approximated using lower and upper bounds on their rates. Using abstraction techniques, this class of linear hybrid system can be transformed into timed automata and thus be verified using the techniques available for timed automata, implemented in UPPAAL. UPPAAL allows linear hybrid automata where the speed of clocks is given by an interval. Hybrid automata of this form may be transformed into ordinary timed automata using the translator `hs2ta`. Philips Audio-Control Protocol of [BPV93] is one such linear hybrid system and for its Autograph version is shown in Figure 5.

2.4 Syntactical Checks

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

2.5 Model–Checking

In the current version UPPAAL is able to check for reachability properties, in particular whether certain combinations of control-nodes and constraints on clocks and integer variables are reachable from an initial configuration. The desired mutual exclusion property of Fischers protocol (Figure 2 and Figure 3) falls into this class. Bounded liveness properties can be obtained by reasoning about the system in the context of testing automata. The model-checking is performed by the module `verifyta` which takes as input a network of timed automata in the `.ta`-format and a formula. `verifyta` can also be used interactively. In case verification of a particular real-time system fails (which happens more often than not), a *diagnostic trace* is automatically reported by `verifyta` [LPY95b]. Such a trace may be considered as diagnostic information of the error, useful during the subsequent debugging of the system. This principle could be called WYDVYAE[2].

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

3 The UPPAAL Model

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

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

3.1 Syntax

Alur and Dill developed the theory of timed automata [AD90], as an extension of classical finite–state automata with clock variables. To have a more expressive model and to ease the modelling task, we further extend timed automata with more general types of data variables such as boolean and integer variables. Our final goal is to develop a modelling (or design) language which is as close as possible to a high–level real–time programming language. Clearly this will create problems for decidability. However, we can always require that the value domains of the data variables should be finite in order to guarantee the termination of a verification procedure. The current implementation of UPPAAL allows integer variables in addition to clock variables.

In a finite–state automaton, a transition takes the form $s \overset{\alpha}{\rightsquigarrow} s'$ meaning that the process modelled by the automaton will perform an α–transition in state s and reach state s' in doing so. Note that there is no condition on the transition. Alur and Dill [AD90] extend the untimed transition to the timed version: $s \xrightarrow{g,a,\phi} s'$ where g is a simple linear constraint over the clock variables and ϕ is a set of clocks to be reset to zero. Intuitively, $s \xrightarrow{g,a,\phi} s'$ means that a process in control node s may perform the α-transition instantaneously when g is true of the current clock values and then reach control node s' with the clocks in ϕ being reset. The constraint g is called a *guard*. In UPPAAL, we allow a more general form of guard that can also be a constraint over data variables, and extend the reset–operation on clocks in timed automata to data variables.

Now assume a finite set of clock variables $\mathcal{C}$ ranged over by x, y, z etc and a finite set of data variables $\mathcal{D}$ ranged over by i, j, k etc.

Guard over Clock and Data Variables

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

Reset–Operations

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

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

Channel, Urgent Channel and Syncronization

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

Automata with clock and data variables

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

Definition 7 *An automaton A over actions ct, clock variables $\mathcal{C}$ and data variables $\mathcal{D}$ is a tuple $\langle N, l_0, \longrightarrow \rangle$ where N is a finite set of nodes (control-nodes), l_0 is the initial node, and $\longrightarrow \subseteq N \times \mathcal{G}(\mathcal{C}, \mathcal{D}) \times ct \times 2^{\mathcal{R}} \times N$ corresponds to the set of edges. To model urgency, we require that the guard of an edge with an urgent action should always be* **tt**, *i.e. if name$(a) \in \mathcal{U}$ and $\langle l, g, a, r, l' \rangle \in \longrightarrow$ then $g \equiv$* **tt**. *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 node l to the node l' with guard g (also called the enabling condition of the edge), action a to be performed and a set of reset–operations r to update the variables.* □

Concurrency and Synchronization

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

$$(A_1 | ... | A_n) \backslash \mathcal{A}$$

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

3.2 Semantics

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

Variable Assignment

Now, we introduce the notion of a *variable assignment*. A variable assignment is a mapping which maps clock variables $\mathcal{C}$ to the non-negative reals and data variables $\mathcal{D}$ to integers. For a variable assignment u and a delay d, $v \oplus d$ denotes the variable assignement such that $(v \oplus d)(x) = v(x) + d$ for any clock variable x

and $(v \oplus d)(i) = v(i)$ for any integer variable i. This definition of $\oplus$ reflects that all clocks operate with the same speed and that data variables are time–insensitive. For a reset-operation r (a set of assignment–operations), we use $r(u)$ to denote the variable assignment u' with $u'(w) = V(e, u)$ whenever $w := e \in r$ and $u'(w') = u(w')$ otherwise, where $V(e, u)$ denotes the value of e in u. Given a guard $g \in \mathcal{G}(\mathcal{C}, \mathcal{D})$ and a variable assignment u, $g(u)$ is a boolean value describing whether g is satisfied by u or not.

Control Vector and Configuration

A *control vector* $\bar{l}$ of a network $\overline{A}$ is a vector of nodes where l_i is a node of A_i. We shall 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.

A *state* of a network $\overline{A}$ is a configuration $\langle \bar{l}, u \rangle$ where $\bar{l}$ is a control vector of $\overline{A}$ and u is a variable assignment. The initial state of $\overline{A}$ is $\langle \bar{l}_0, u_0 \rangle$ where $\bar{l}_0$ is the initial control vector whose elements are the initial nodes of A_i's and u_0 is the initial variables assignment that maps all variables to 0.

Maximal Delay

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

Definition 8 *(Maximal Delay for Automata)*

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

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

Definition 9 *(Maximal Delay for Networks of Automata)*

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

□

Transition Rules

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

Definition 10 *(Transition Rules for Networks of Automata)*

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

4 The UPPAAL Model–Checker

In the current version, UPPAAL is able to check for reachability properties, in particular whether certain combinations of control–nodes and constraints on clock and data variables are reachable from an initial configuration.

Logic

The properties that can be analysed are of the forms:

$$\varphi \ ::= \ \mathsf{INV}\beta \mid \mathsf{Poss}\beta \qquad\qquad \beta \ ::= \ a \mid \beta_1 \wedge \beta_2 \mid \neg\beta$$

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

Intuitively, for $\mathsf{INV}\beta$ to be satisfied all reachable states must satisfy β. Dually, for $\mathsf{Poss}\beta$ to be satisfied some reachable state must satisfy β. Formally let $\rightsquigarrow$ denote the transitive closure of the delay– and action–transition relations between

network configurations. Then the satisfaction relation $\models$ between network configurations and formulas are defined as follows:

$$\begin{array}{rcl}(\bar{l},v) \models \mathsf{Poss}\beta & \iff & \exists(\overline{l'},v').(\bar{l},v) \rightsquigarrow (\overline{l'},v') \wedge (\overline{l'},v') \models \beta \\ (\bar{l},v) \models \mathsf{INV}\beta & \iff & \forall(\overline{l'},v').(\bar{l},v) \rightsquigarrow (\overline{l'},v') \Rightarrow (\overline{l'},v') \models \beta\end{array}$$

Satisfaction with respect to a boolean combination β of atomic formulas is defined inductively on the structure of β (behaving as usual with respect to the boolean connectives). Satisfaction with respect to an atomic formula is given by the following definitions:

$$\begin{array}{rcl}(\bar{l},v) \models c & \Leftrightarrow & v \in c \\ (\bar{l},v) \models A_i\mathsf{at}l & \Leftrightarrow & l_i = l\end{array}$$

Our (simple and efficient) model–checking technique extends to the logic presented in [LPY95b], which also allows for bounded liveness properties to be specified. Currently, bounded liveness properties are obtained by reachability analysis of the system in the context of testing (and time–sensitive) automata. We conjecture that all bounded liveness properties of the logic in [LPY95b] can be translated into reachability problems in this manner.

Model Checking

The model–checking procedure implemented in UPPAAL is based on an interpretation using a finite–state symbolic semantics of networks. More precisely, we interpret the logic with respect to symbolic network configurations of the form $[\bar{l}, D]$, where D a constraint system (i.e. a conjunction of atomic clock and data constraints) and $\bar{l}$ a control–vector. Some of the rules defining this symbolic interpretation is given in Table 1.

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

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

An important observation is that, whenever D is a constraint system (i.e. a conjunction of atomic clock and data constraints), then so are both $r(D)$ and $D^{\uparrow}$. Moreover, due to Richard Bellman representing constraint systems as weighted directed graphs (with clock and data variables as nodes), these operations as well as

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

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

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

Table 1: Symbolic Interpretation of Reachability Logic

testing for inclusion between constraint systems may be effectively implemented in $O(n^2)$ and $O(n^3)$ using shortest path algorithms [TCR90, YL93, LPY95a].

Now, by applying the proof rules of Table 1 in a goal directed manner we obtain an algorithm (see also [YPD94]) for deciding whether a given symbolic network configuration $[\bar{l}, D]$ satisfies a property $\mathsf{Poss}\beta$. To ensure termination (and efficiency), we maintain a (past–) list $\mathcal{L}$ of the symbolic network configurations encountered. If, during the goal directed application of the proof rules of Table 1 a symbolic network configuration $[\bar{l}, D']$ is generated which is already "covered" by a configuration $[\bar{l}, D]$ in $\mathcal{L}$ (i.e. $D' \subseteq D$) then the the goal directed search fails at $[\bar{l}, D']$ and backtracking is needed. If $[\bar{l}, D']$ "covers" some configuration $[\bar{l}, D]$ in $\mathcal{L}$ (i.e. $D \subseteq D'$) then $[\bar{l}, D']$ replaces $[\bar{l}, D]$ in $\mathcal{L}$.

5 Applications and Performance

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

In [LPY95c] an experiment was performed using four existing real-time verification tools: UPPAAL, HYTECH (Cornell), Kronos (Grenoble) and Epsilon (Aalborg). In the experiment it was verified that the so-called Fischer's mutual ex-

clusion protocol [Sha93, AL93], shown in Figure 2, satisfies the mutual exclusion property $\forall\Box\neg((P_1 \text{ at } cs) \wedge (P_2 \text{ at } cs))$. With all the tools installed on the same machine[3] the standard Unix command `time` was used to measure execution time. The resulting time-performance diagram, shown in Figure 4, indicate that UPPAAL performs time- and space-wise favorably compared to the other tools in the experiment.

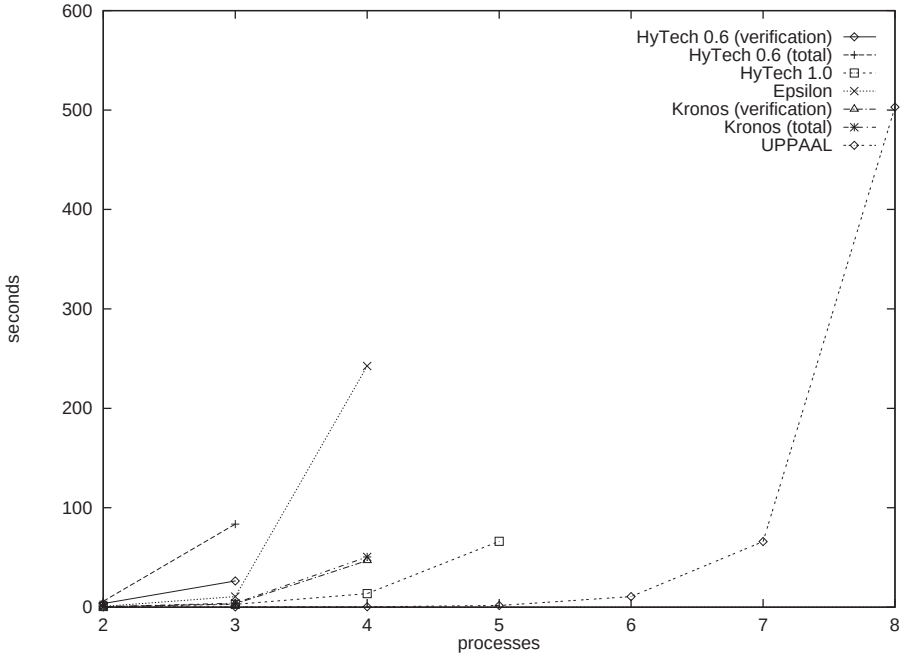


Figure 4: Execution Times for Fischer's Protocol.

In [LPY95b], in this volume, the Philips Audio-Control Protocol [BPV93, HWT95] was verified using UPPAAL. A version of the protocol is shown in Figure 5. In the verification of this protocol, we found the diagnostic model-checking feature of UPPAAL useful for detecting and correcting several errors in the description of the protocol. UPPAAL verifies that the received bit stream is guaranteed to be identical to the sent bit stream in 3.8 seconds[4].

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

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

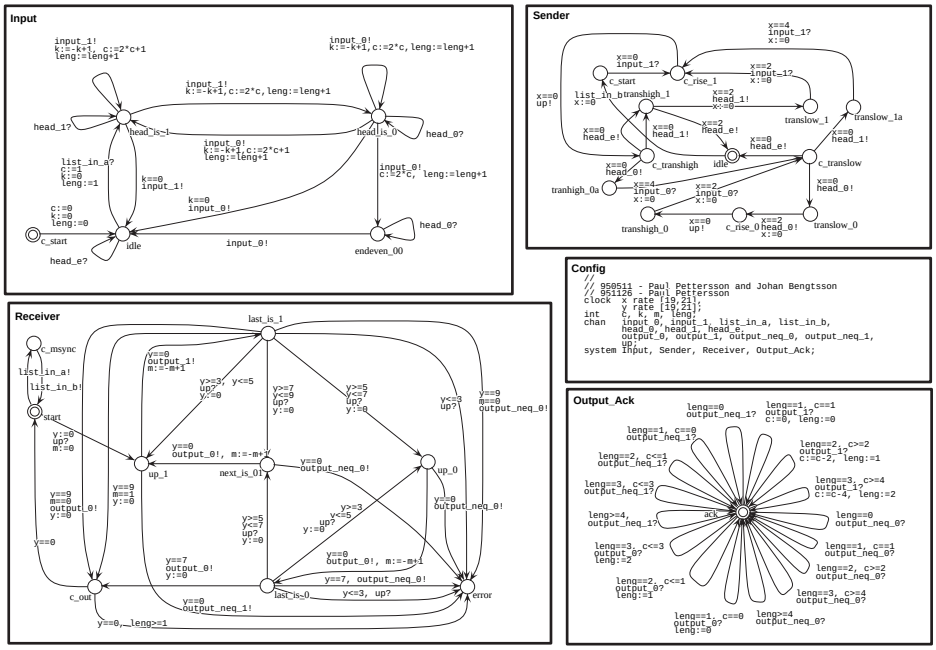


Figure 5: Philips Audio-Control Protocol.

6 Conclusion and Future Work

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

Future versions of UPPAAL will extend the current model–checker to the safety and bounded liveness logic of [LPY95b]. Also future versions of UPPAAL will integrate the newly developed compositional model–checking technique of [LPY95a], which, judged from experimental results using a CAML prototype implementation [LL95], seems to be a powerful technique in the on–going fight against explosion problems.

References

[AD90] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of ICALP'90*, volume 443, 1990.

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

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

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

[LL95] F. Laroussinie and K.G. Larsen. Compositional Model Checking of Real Time Systems. In *Proc. of CONCUR'95*, Lecture Notes in Computer Science. Springer Verlag, 1995.

[LPY95a] K.G. Larsen, P. Pettersson, and W. Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. To appear in *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, December 1995.

[LPY95b] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of the 4*th *DIMACS Workshop on Verification and Control of Hybrid Systems*, Lecture Notes in Computer Science, October 1995.

[LPY95c] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, 1995.

[OSY94] A. Olivero, J. Sifakis, and S. Yovine. Using Abstractions for the Verification of Linear Hybrids Systems. In *Proc. of CAV'94*, volume 818 of *Lecture Notes in Computer Science*, 1994.

[Sha93] N. Shankar. Verification of Real-Time Systems Using PVS. In *Proc. of CAV'93.*, volume 697, 1993.

[TCR90] C.E. Leiserson T.H. Cormen and R.L. Rives. *Introduction to ALGORITHMS*. MIT Press, McGraw-Hil, 1990.

[YL93] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real–time transition systems. In *Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 210–224, 1993.

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

Recent licentiate theses from the Department of Information Technology

2000-008 Marcus Nilsson: *Regular Model Checking*

2000-009 Jan Nyström: *A formalisation of the ITU-T Intelligent Network standard*

2000-010 Markus Lindgren: *Measurement and Simulation Based Techniques for Real-Time Analysis*

2000-011 Bharath Bhikkaji: *Model Reduction for Diffusion Systems*

2001-001 Erik Borälv: *Design and Usability in Telemedicine*

2001-002 Johan Steensland: *Domain-based partitioning for parallel SAMR applications*

2001-003 Erik K. Larsson: *On Identification of Continuous-Time Systems and Irregular Sampling*

2001-004 Bengt Eliasson: *Numerical Simulation of Kinetic Effects in Ionospheric Plasma*

2001-005 Per Carlsson: *Market and Resource Allocation Algorithms with Application to Energy Control*

2001-006 Bengt Göransson: *Usability Design: A Framework for Designing Usable Interactive Systems in Practice*

2001-007 Hans Norlander: *Parameterization of State Feedback Gains for Pole Assignment*

2001-008 Markus Bylund: *Personal Service Environments — Openness and User Control in User-Service Interaction*

2001-009 Johan Bengtsson: *Efficient Symbolic State Exploration of Timed Systems: Theory and Implementation*

UPPSALA
UNIVERSITY

Department of Information Technology, Uppsala University, Sweden