Symbolic Model checking of Timed Automata using LTSmin

Sybe van Hijum

December 3, 2015

Contents

1 Introduction

Timed Automata [1] are a widely used modelling formalism. A recent usage of this formalism is the modelling of biological signalling pathways [2]. This leads however to large state spaces, and sometimes to models that are too large to handle by conventional methods. Therefore the ANIMO [2] tool at this time uses simulation of models and not complete state space generation and property checking.

BDDs(Binary Decision Diagrams) [**?**] and variations like LDDs(List Decision Diagrams) [3] and MDDs(Multi-valued Decision Diagrams) [4] have proven their value in model checking algorithms. Due to advances in this field models with much larger state spaces can be explored on the same machine. This progress has not translated directly to more efficient methods for Timed Automata. Several methods have been proposed, like CDDs(Clock Difference Diagrams) [5], CMDs(Constraint Matrix Diagrams [6], CRDs(Clock

Restriction Diagrams) [7] and DDDs(Difference Decision Diagrams) [8]. All of these methods show some extra difficulties or time complexity over BDDs or some limitations.

LTSmin [9, 10] is a language independent on the fly model checker with several algorithmic backends. Its symbolic backend uses LDDs to both represent the state space and the transition relations of models. LTSmin has a language module for the UPPAAL [11] through the opaal [12] lattice model checker. For this language at this time, only the multicore backend can be used [13]. This multicore approach showed efficient enough to compete with the latest version of the UPPAAL model checker. It showed significant speedups on multicore machines, at the cost of some memory increase however.

The symbolic backend of LTSmin provides both a memory reduction by using LDDs and a speedup by using multi-threaded search algorithms and the multi-threaded LDD package Sylvan [14]. Using this together with the UPPAAL language frontend will hopefully result in a model checker that can compete both on time and memory consumption with the UPPAAL model checker. We propose a method that uses LDDs to represent both the discrete states as the clocks and that uses DBMs [15, 16] only in the next state generation. This way we can combine existing techniques to build a complete model checker. It will also remain possible to use the other techniques that LTSmin has, such as transition caching, variable reordering and LTL checking. This will result in a complete model checker for Timed Automata.

We will propose a method that will use the best of both worlds. We will use the DBMs in the state exploration such that we can find a canonical representation of the clock zone of a newly explored state quite easily. For the symbolic representation of the state space, including the clock zones, we will use normal LDDs. Therefore both the efficient algorithms and the memory efficient representation can be used. A downside to this approach is zones that are contained in larger zones might be explored again. Further we will focus on efficient orderings of the LDDs, as both clock zones and states are contained in a single structure. A final point of research will be the achievable efficiency by partitioning the next state function, or leaving it intact as a single function. This will of course be combined with the different orderings.

2 Related Work

Already several model checkers for Timed Automata exist such as UPPAAL [11], KRONOS [17] and RED [7]. We focus mainly on the UPPAAL tool as we use the same input format. Opaal uses the XML format that is created by the UPPAAL tools. This way we can use the UPPAAL user interface to create and adapt models. We also use the UPPAAL DBM library to represent zones. Several methods exist to represent the clock variables in a timed model. The most used methods are digitization and zones. Digitization approximates the continuous values of clocks by using discrete values. This approach is however very sensitive to the granularity of the values used and the upper bound of the clock values. An advantage of this approach is that basic model checking approaches can be used and no extra complexity due to zone calculations is added. In [18] a similar approach is proposed by using clock tick actions and removing clock variables altogether.

The most established method to represent clock zones are DBMs [15,16]. DBMs use a matrix structure that gives an lower and upper bound to each clock and to the difference between each pair of clocks. By this approach convex zones of clocks can be created. By using graph algorithms a normal form can be found quite efficiently. The downside of this approach is that only these convex zones can be represented, when a state has multiple zones that are not a convex combination multiple DBMs are needed and thus increasing the memory usage.

Several methods based on BDDs have been developed to represent zones. All of these are based on DMBs in the sense that the use upper and lower bounds of clocks and of the difference between a pair of clocks. CDDs [5] use single nodes for each value with a larger fanout with disjoint intervals on each edge. DDDs [8] use a constraint on each node that can either be true or false. This requires a fixed ordering based on the variables, values and operators. CRDs [7] differ mainly from CDDs by using not disjoint intervals but possibly overlapping upperbounds on their edges. They also use different normal forms for the diagrams which results in different performances. It is also shown that CRDs can be combined with BDDs into a single structure to represent state space. CMDs [6] combine CDDs, CRDs and DMBs into a single structure. This diagram type differs from the others by having multiple constraint per edge, resulting in a diagram with few nodes. CMDs do not have a normal form so only reduced forms are proposed. In [19, 20] a method is proposed purely based on BDDs by translating the constraints into BDD nodes. This results in a unified structure for both the discrete variables and the clock constraints. The method has not been implemented

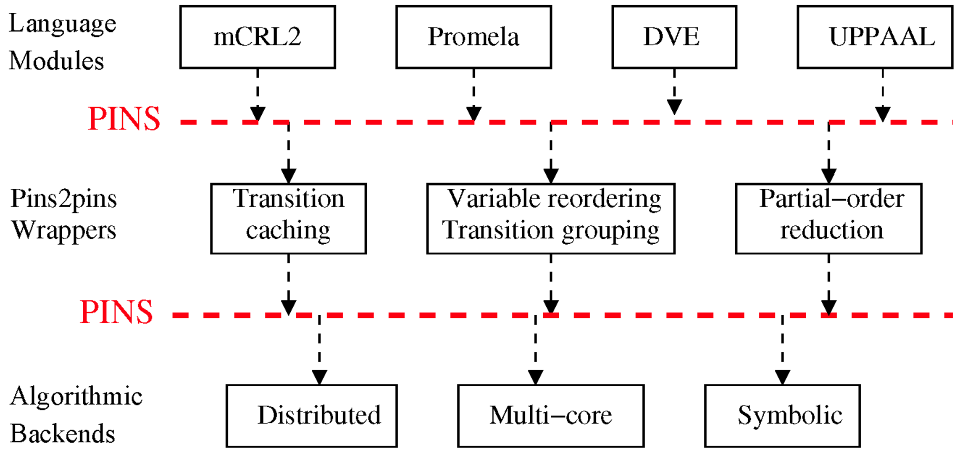

Figure 1: Modular structure of LTSmin

in a model checker and no performance results are known, only the concept is known.

A known difficulty in BDDs is the variable ordering. A bad ordering can lead to a BDD exponential in size where a good ordering can sometimes lead to a significant smaller diagram. Of the diagrams named above only CRDs have experimented with different orderings, the other researches assume a given ordering. The CRD case shows that full interleaving and having related variables close to each other in the ordering is preferable and gives the best results, both on speed and memory. This is the same result as expected with BDDs, this suggests that similar orderings should be used.

LTSmin [9, 10] is a language independent model checker. It is build modular such that new languages can be added by an PINS interface without too much effort and new algorithms can be added more easily. LTSmin offers four different algorithmic back ends for model analysis: symbolic, multi-core, sequential and distributed. All of these back ends support different types of reduction and model checking. Several language modules have already been built for LTSmin such as mCRL2, Promela, DVE and UPPAAL. The modular structure of LTSmin is shown in figure 1. States are stored in fixed length integer arrays. The PINS(Partitioned Next-State Interface) is the core of LTSmin. This interface abstracts as much as possible from the model without losing the structure. The main function of the interface is a (partitioned) next state function which returns the successor states. With these functions a state space can be generated on the fly. With the use of dependency matrices event locality can be determined statically [21]. Using these matrices more efficient symbolic algorithms can be used, the number of next-state calls can be reduced and transition caching can be used. In the current UPPAAL PINS the next-state function is not partitioned and therefore no meaningful dependency matrix is created.

2.1 Timed Automata

Timed Automata is a formalism that extends labelled transition systems with one ore more, but finite, clocks. As our work continues on [13] we use the same definition of timed automata.

Definition 1 (Timed Automata)**.** *An extended timed automaton is a 7-tuple* $A = < L, C, Act, s_0, \rightarrow, I_c >$ *where*

- L *is a finite set of locations, typically denoted by* l
- C *is ia finite set of clocks, typically denoted by* c
- Act *is a finite set of actions*
- $s_0 \in L$ *is the initial location*
- $\rightarrow \subseteq L \times G(C) \times Act \times 2^C \times L$ *is the (non-deterministic) transition relation. We noramlly write* $l \xrightarrow{g,a,r}$ *l' for a transition., where* l *is the source location,* g *is the guard over the clocks,* a *is the action, and* r *is the set of clocks reset.*
- $I_C : L \rightarrow G(C)$ *is a function mapping locations to downwards closed clock invariants.*

With this definition we can combine timed automata to a network of timed automata to define larger systems.

Definition 2 (Network of timed automata [13])**.** *Let* $Act = \{ch!, ch?|ch \in Chan\} \cup \{\tau\}$ *be a finite set of actions, and let* C *be a finite set of clocks. Then the parallel composition of extended timed automata* $A_i = (L_i, C, Act, S_0^i, \rightarrow_i, I_C^i)$ *for all* $1 \leq i \leq n$*, where* $n \in \mathbb{N}$*, is a network of timed automata, denoted* $A = A_1||A_2||..||A_n$*.*

2.2 Zones

For basic transition systems the state space can grow exponentially for the size of the system. For Timed Automata the growth can become even larger

DMB	Canonical form for convex zones Existing library Inclusion check	Concave zones need multiple DBMs Not memory efficient
DDD	Nodes with difference constraints Implement as LDD Re-ordering of variables possible Apply same efficiency as BDDs Boolean variables also in DDD	Canonicity hard to obtain No on the fly canonicity Only speed performance tested
CDD	Branching on intervals Implement as MDD Intervals need to be disjoint Inclusion check (intersection of complement)	No algorithm to get normal form Only high level algorithms Methods don't maintain disjointness Expensive normal form computation No implementation results Disjointness memory inefficient
CRD	Nodes with multiple fanout for upperbound constraints Combination with BDD possible Variable ordering with interleaving is best Modelchecker available Library available Some benchmarks better than CDD Extensive benchmarks	3 possible canonical forms Good performance backwards reach No algorithms in paper Some benchmarks linear worse than CDD
CMD	Multiple DBM constraints per edge Benchmarks against RED and UPPAAL	Results differ per case Needs translation from vector to edges
BDD discrete	Using existing BDD packages Good performance for small clock values	Performance decreases fast for large values Not possible with current opaal PINS Models created separately as C code Introducing 'tick' actions Only for closed timed automata
BDD zones	Using existing BDD packages All variable reorderings possible Only need direct translation DBM to state vector Easy to implement	Losing zone containment No research known

and in some cases become unbounded. To tackle this problem most model checkers use a notion of zones for the representation of time. A zone can be seen as a set of constraints over the clocks C of the form $c_i \sim x$ and $c_i - c_j \sim x$ where $\sim \in \{<, \leq, =, \geq, >\}$ and $x \in \mathbb{N}$. To represent these zones several data structures have been developed. One of the most common used structures are Difference Bound Matrices(DMB's) [16].
These matrices use both a column and a row for each clock, and on each position (i, j) an upper bound on the difference between the clocks c_i and c_j is given in the form $c_i - c_j \preceq x$ where $\preceq \in \{<, \leq\}$ and $x \in \mathbb{N}$. For the constraints over the single clocks an extra clock **O** with a constant value 0 is added.

3 Plan

3.1 Questions

For the research we will state a couple of research questions:

- Is the combination of LDDs and DMBs an efficient method for model checking timed automata?
- Can improvements be achieved by different orderings? Both by changing the order of only the clock variables and by mixing the clock and state variables.
- What is the effect of next state function partitioning on the efficiency?
- Can basic LDD variable ordering heuristics be used for timed automata?

3.2 Approach

The first task will be to find a suitable translation function from DBM values to integers that LTSmin uses as state vector, and vice versa. This function will be needed at the begin and end of each next-state call. Together with this the structure of the UPPAAL PINS will need to be changed. For the later experiments also changes to the ordering of variables and a partition of the next state function in the PINS will be constructed.

For the tests on variable ordering heuristics we will use the options that have already been implemented in LTSmin [21]. We will compare these against the basic ordering and the manually created orderings. For this experiment also the partitioned next state function will be needed as otherwise no meaningful dependency matrices exist and no reordering can be performed.

We will test all our approaches with multiple testcases against the state of the art UPPAAL and the latest multi core version of LTSmin. All tests will be carried out on a laptop with 4GB of memory and a 2.53GHz Intel I5 processor with 4 cores. If we achieve promising results, the tests can be expanded to a 48 core machine, on which the multi threaded algorithms can be exploited more and thus better tests on time efficiency can be performed.

3.3 To do

In this section we describe all things that need to be implemented to make model checking with a certain diagram possible.

To make most diagrams work we need to change the opaal PINS. The PINS currently uses a pointer to a DBM. For the new approach we will put the values of the DBM directly into the state vector. This will increase the size of the state vector. All other references to the types and values of the state vector entries will need to be changed.

To make symbolic reordering possible for any of the diagrams we will need to partition the next state function and create a dependency matrix. In the code the next state function is already split up per transition, but in a single transition group. Splitting this into multiple transition groups should not be too hard. In this same step also the dependency matrices need to be created and made as sparse as possible.

If we use reordering of variables a mapping function from the state vector to a DBM will be needed. Without reordering it is simply taking a part of the state vector as DBM, but now the variables might be scattered through the vector. A function will be needed to create a DBM from the vector and a function to place the DBM variables back into the state vector.

For the DDD, CDD, CRD and CMD the diagram will need to be able to identify the zone variables from the state variables in the state vector. This will need an extra function in the pins interface. This function will also need to cope with variable reordering.

To combine the new PINS with the multicore LTSmin backend the subsumption check will need to be changed. This check now relies on a pointer to a DBM, but it will now get the complete DBM, or state vector. Here the search algorithm or the subsumption check will need to know which variables are zone variables.

For a BDD with discrete clock variables another input model will be needed. This is probably not possible with the opaal PINS. An option is to encode this as C code. This new model will need a tick action to represent the progress of time. For each clock the model will need a single state variable.

For all diagrams other than BDDs the values from the DBM will need to be translated to useful variables as ordering of values is for most diagrams needed. Also to check for inequalities and set containment the meaningful values are needed. As the values in a DBM both contain the value of an inequality as the operator some more complex translation might be needed.

References

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

[2] Stefano Schivo, Jetse Scholma, Brend Wanders, Ricardo A. Urquidi Camacho, Paul E. van der Vet, Marcel Karperien, Rom Langerak, Jaco van de Pol, and Janine N. Post. Modelling biological pathway dynamics with timed automata. In *12th IEEE International Conference on Bioinformatics & Bioengineering, BIBE 2012, Larnaca, Cyprus, November 11-13, 2012*, pages 447–453, 2012.

[3] Stefan Blom and Jaco van de Pol. Symbolic reachability for process algebras with recursive data types. In J.S. Fitzgerald, A.E. Haxthausen, and H. Yenigun, editors, *Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 81–95, Berlin, Germany, August 2008. Springer Verlag.

[4] A. Srinivasan, T. Ham, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95, Nov 1990.

[5] Kim Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock difference diagrams. *BRICS Report Series*, 5(46), 1998.

[6] R. Ehlers, D. Fass, M. Gerke, and H.-J. Peter. Fully symbolic timed model checking using constraint matrix diagrams. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 360–371, Nov 2010.

[7] Farn Wang. Efficient verification of timed automata with bdd-like datastructures. In LenoreD. Zuck, PaulC. Attie, Agostino Cortesi, and

Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 189–205. Springer Berlin Heidelberg, 2003.

[8] Jesper Mller, Jakob Lichtenberg, HenrikReif Andersen, and Henrik Hulgaard. Difference decision diagrams. In Jrg Flum and Mario Rodriguez-Artalejo, editors, *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 1999.

[9] S. C. C. Blom, J. C. van de Pol, and M. Weber. Ltsmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, Edinburgh*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359, Berlin, July 2010. Springer Verlag.

[10] A. W. Laarman, J. C. van de Pol, and M. Weber. Multi-core ltsmin: Marrying modularity and scalability. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011, Pasadena, CA, USA*, volume 6617 of *Lecture Notes in Computer Science*, pages 506–511, Berlin, July 2011. Springer Verlag.

[11] Gerd Behrmann, Alexandre David, and KimG. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[12] Andreas Engelbredt Dalsgaard, Ren Rydhof Hansen, Kenneth Yrke Jrgensen, Kim Gulstrand Larsen, Mads Chr. Olesen, Petur Olsen, and Ji Srba. opaal: A lattice model checker. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 487–493. Springer Berlin Heidelberg, 2011.

[13] A. E. Dalsgaard, A. W. Laarman, K. G. Larsen, M. C. Olesen, and J. C. van de Pol. Multi-core reachability for timed automata. In M. Jurdzinski and D. Nickovic, editors, *10th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2012, London, UK*, volume 7595 of *Lecture Notes in Computer Science*, pages 91–106, London, September 2012. Springer Verlag.

[14] Tom van Dijk and Jaco van de Pol. Sylvan: Multi-core decision diagrams. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035

of *Lecture Notes in Computer Science*, pages 677–691. Springer Berlin Heidelberg, 2015.

[15] DavidL. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin Heidelberg, 1990.

[16] Johan Bengtsson. *Clocks, DBMS and States in Timed Systems (Uppsala Dissertations from the Faculty of Science Technology, 39)*. Uppsala Universitet, 7 2002.

[17] Sergio Yovine. Kronos: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

[18] TruongKhanh Nguyen, Jun Sun, Yang Liu, JinSong Dong, and Yan Liu. Improved bdd-based discrete analysis of timed systems. In Dimitra Giannakopoulou and Dominique Mry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 326–340. Springer Berlin Heidelberg, 2012.

[19] Huiping Zhang, Junwei Du, Ling Cao, and Guixin Zhu. A full symbolic reachability analysis algorithm of timed automata based on bdd. In *Autonomous Decentralized Systems (ISADS), 2015 IEEE Twelfth International Symposium on*, pages 301–304, March 2015.

[20] Junwei Du, Huiping Zhang, Gang Yu, and Xi Wang. A full symbolic compositional reachability analysis of timed automata based on bdd. In *Advanced Computational Intelligence (ICACI), 2015 Seventh International Conference on*, pages 218–222, March 2015.

[21] Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. Read, write and copy dependencies for symbolic model checking. In Eran Yahav, editor, *Hardware and Software: Verification and Testing*, volume 8855 of *Lecture Notes in Computer Science*, pages 204–219. Springer International Publishing, 2014.