Symbolic Model checking of Timed Automata using LTSmin

Sybe van Hijum

January 15, 2016

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 better model checking techniques for timed automata, that can handle larger state spaces are needed. We look into symbolic algorithms for timed automata.

BDDs(Binary Decision Diagrams) [3, 4] and variations like LDDs(List Decision Diagrams) [5] and MDDs(Multi-valued Decision Diagrams) [6] 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 been translated directly to more efficient methods for Timed Automata. Several methods have been proposed, like CDDs(Clock Difference Diagrams) [7], CMDs(Constraint Matrix Diagrams) [8], CRDs(Clock Restriction Diagrams) [9] and DDDs(Difference Decision Diagrams) [10,11]. All of these methods show some extra difficulties or limitations over BDDs. Also after their introduction they have not been developed further.

LTSmin [12,13] is a language independent on the fly model checker with several algorithmic backends. Its symbolic backend uses BDDs to both represent the state space and the transition relations of models. These BDDs are generated on the fly by the search algorithms. LTSmin has a language module for the UPPAAL [14] through the opaal [15] lattice model checker. For this language at this time, only the multicore backend can be used [16]. 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. To tackle the memory increase a combination of the opaal frontend and the symbolic backend could be a solution.

The symbolic backend of LTSmin provides both a memory reduction by using BDDs and a speedup by using multi-threaded search algorithms and the multi-threaded BDD package Sylvan [17]. 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 will propose a symbolic reachability for timed automata that is capable of handling the models that are generated by the ANIMO tool.

2 Related Work

2.1 Methods

Already several model checkers for Timed Automata exist such as UPPAAL [14], KRONOS [18], RABBIT [19] and RED [9]. We focus mainly on the UPPAAL tool as we use the same input format. Opaal [15] 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 [20]. 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. The method however only works for closed timed automata, meaning that no strict comparisons on clocks can be made in the model. In [21] 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 [22,23]. 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 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 DBMs in the sense that they use upper and lower bounds of clocks and of the difference between a pair of clocks. They use a BDD like structure to represent the zones more efficiently. CDDs [7] use single nodes for each variable and have disjoint intervals for that variable on the edges. This results in a node with a larger fanout and the upper and lower bound in a single node. DDDs [10, 11] use a constraint on each node that can either be true or false, when a constraint is false a next node will have another constraint on the same variable. This requires a fixed ordering based on the variables, values and operators. CRDs [9] differ mainly from CDDs by using not disjoint intervals but possibly overlapping upper bounds for a variable pair on their edges. This diagram will have a larger fanout, like CDDs. They also use several 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 [8] combine CDDs, CRDs and DBMs into a single structure. This diagram type differs from the others by having multiple constraints per edge, resulting in a diagram with few nodes. CMDs do not have a normal form so only reduced forms are proposed. In [24, 25] a method is proposed purely based on BDDs by translating the constraints directly into BDD nodes. This results in a unified structure for both the discrete variables and the clock constraints. The method is only a proof of concept and has not been implemented in a model checker and no performance results are known. In table 1 we compare the different types of diagrams. In figure 1

3

we have an example of all four BDD like structures representing the zone $2 < c_1 - c_2 < 4 \vee 7 \leq c_1 - c_2 \leq 8$.

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 on the variables and the ordering of the values is fixed. 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 with these techniques.

LTSmin [12, 13] is a language independent model checker. It is build in a modular way 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 2. 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 [26]. 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, and all of these algorithms can not be used.

2.2 Timed Automata

Timed Automata is a formalism that extends labelled transition systems with one ore more, but finite, clocks. As our work continues on [16] 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*

Table 1: Comparing Diagrams

Type	Pro	Con
DBM	Canonical form for convex zones Existing library Inclusion check	Concave zones need multiple DBMs Not memory efficient
DDD	Structure like 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 Expensive normal form computation Only time performance tested Only reduction algorithms
CDD	Structure like MDD Inclusion check (intersection of complement)	No algorithm to get normal form Only high level algorithms given Methods don't maintain disjointness Expensive normal form computation No implementation results available Disjointness memory inefficient
CRD	Combination with BDD possible Variable reordering shows advantage Library available Some benchmarks exp better than CDD Extensive benchmarks Good performance backwards reach	3 possible canonical forms No algorithms in paper Some benchmarks linear worse than CDD
CMD	Benchmarks against RED and UPPAAL	Results differ per case Needs translation from vector to edges Two reduced forms
BDD discrete	Using existing BDD packages Good performance for small clock values	Performance decreases fast for large values Not possible with current opaal PINS Introducing additional '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 implementation results

- 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 [16]). *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 $A = A_1||A_2||..||A_n$.*

2.3 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 and in some cases become unbounded by the addition of clocks. 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(DBMs) [22, 23].

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. By switching the two variables also a lower bound can be given to each pair. This way convex zones of clock variables can be represented. Each matrix can however only contain a single convex zone. Concave zones and multiple convex zones need multiple matrices to be represented. As a solution often a list of DBMs is used.

2.4 Difference Decision Diagrams

We have discussed several symbolic approaches for representing zones. All of these approaches have benefits and downsides over each other. We chose to develop one of these approaches in LTSmin. The DDD approach has been chosen for this. It is a diagram form that is closely related to LDDs, for which we already have a library, and it is also quite compatible to the current PINS structure and its' next state function. The method still has some loose ends that need research, mostly on the algorithms and efficiently creating a canonical form. No results on the memory usage are available, which is normally the greatest benefit of a symbolic approach. So DDDs are a diagram type that seems to fit well in the current structure we have, but there is still room for some more research.

Definition 3 (Difference Decision Diagram [10]). *A difference decision diagram (DDD) is a directed acyclic graph (V, E). The vertex set V contains two terminals 0 and 1 with out-degree zero, and a set of non-terminal vertices with out-degree two and the following attributes.*

Attribute	*Type*	*Description*
pos(v), neg(v)	***Var***	*Positive variable x_i, and negative variable x_j.*
op(v)	$\{<, \leq\}$	*Operator $<$ or $\leq$.*
const(v)	$\mathbb{D}$	*Constant c.*
high(v), low(v)	*V*	*High-branch h, and low-branch l.*

The set E contains the edges $(v, low(v))$ and $(v, high(v))$, where $v \in V$ is a non-terminal vertex.

In [10] a canonical form for DDDs is discussed, also called a fully reduced DDD. Only definitions are given here, no algorithms to reach this form. It is stated that it is difficult to reach this fully reduced form. It is not clear if they managed to make their apply function in such a way that it maintains canonicity. To reach canonicity, local reductions and ordering are a first step, but it is not enough due to dependencies among the constraints. For BDDs the local reductions and ordering are sufficient to reach a canonical form. First we give some notational shorthands and then we define an ordering and local reductions on DDDs.

$$\begin{aligned} var(v) &= (pos(v), neg(v)) \\ bound(v) &= (const(v), op(v)) \\ cstr(v) &= (var(v), bound(v)) \end{aligned}$$

Definition 4 (Ordered DDD [10]). *An ordered DDD (ODDD) is a DDD where each non-terminal vertex v satisfies:*

1. *$neg(v) \prec pos(v)$,*

2. *$var(v) \prec var(high(v))$,*

3. *$var(v) \prec var(low(v))$ or*
   $var(v) = var(low(v))$ and $bound(v) \prec bound(low(v))$.

After ordering a DDD some local reductions can be defined to reduce the size of a DDD.

Definition 5 (Locally Reduced DDD [10]). *A locally reduced DDD ($R_L DDD$) is an ODDD satisfying, for all non-terminals u and v:*

1. *$\mathbb{D} = \mathbb{Z}$ implies $op(v) =' \leq'$,*

2. *$(cstr(u), high(u), low(u)) = (cstr(v), high(v), low(v))$ implies $u = v$,*

3. *$low(v) \neq high(v)$,*

4. *$var(v) = var(low(v))$ implies $high(v) \neq high(low(v))$.*

These reductions are not enough to reach a canonical form. Here we define the other reductions and methods needed to reach a canonical form.

Definition 6 (Path-reduced DDD [10]). *A path-reduced DDD ($R_P DDD$) is a locally reduced DDD where all paths are feasible.*

Definition 7 (Tightness [10]). *A dominating constraint $\alpha = x_i - x_j \lesssim c$ is tight in a feasible path $[p] = [p_1] \wedge \alpha \wedge [p_2]$ if for all tighter constraints $(c', \lesssim') < (c, \lesssim)$, the systems $[p_1] \wedge (x_i - x_j \lesssim' c') \wedge [p_2]$ and $[p]$ have different solutions. A path p is tight if it is feasible and all dominating constraints on it are tight. An $R_L DDD u$ is tight if all paths from u are tight.*

Definition 8 (Saturation [10]). *A tight path p from an $R_P DDD$ is saturated if for all constraints α not on p, if α is added to p either (1) α is not dominating and tight, or (2) the constraint system $[p_1] \wedge \neg \alpha$ is infeasible when $[p]$ is written $[p] = [p_1] \wedge [p_2]$ with all constraints on p_1 smaller than α with respect to $\prec$ and all constraints on p_2 larger than α. An $R_P DDD$ u is saturated if all paths from u are saturated.*

Definition 9 (Disjunctive vertex [10]). *Let p be a path leading to the vertex u in a DDD, and assume $\alpha = cstr(u), h = high(u)$, and $l = low(u)$. Then u is disjunctive in p if $[p] \wedge (\alpha \rightarrow h, l)$ and $[p] \wedge (h \vee l)$ have the same set of solutions.*

All of these definitions lead to the following definition of a fully reduced DDD.

Definition 10 (Fully reduced DDD [10])**.** *An $R_p DDD$ u is a fully-reduced DDD ($R_F DDD$) if it is tight, saturated and has no disjunctive vertices.*

DDDs are also used to represent the discrete variables in automata. This is done by translating the variable into a difference constraint. For example $x_1 = 3$ will be translated into $x_1 - 0 \leq 3 \wedge 0 - x_1 \leq -3$, thus resulting into a DDD with two nodes.

So far we only found the results of two benchmark tests of DDDs, Milner's scheduler and Fischer's protocol [27]. Here the DDD approach has been compared with KRONOS and UPPAAL which were both slower than the DDD implementation. The results of these benchmarks show no memory usage.

3 Plan

3.1 Questions

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

- Is the combination of BDDs and flattened DBMs an efficient method for symbolic reachability analysis of timed automata? Both on memory usage and speed.

- Can improvements be achieved by using different orderings? Both by changing the order of only the clock variables and by mixing the clock and state variables.

- Is the new language module needed for the symbolic approach also usable for the multicore approach with subsumption?

- Can the BDD approach be generalized towards a method using DDDs?

- Can we create a fully symbolic reachability analysis for timed automata using DDDs, without using DBMs?

3.2 Zone Containment

In BDDs it is simple to test for containment of a state. This can be done in linear time over the number of variables. For zones in DDDs this can be a harder question, as a zone can be contained in some larger zone in the

DDD. For this we need an extension to the normal LDD algorithm, that can handle these zones. The main difference is that on some nodes both the high and the low edge can lead to a satisfying path. Algorithm 1 tests if a zone z is contained in the DDD rooted at v.

Algorithm 1 Zone containment for DDDs

1: **procedure** CONTAINS(v, z)
2:     **if** $v \in 0, 1$ **then return** v
3:     **if** z[$var(v)$] correct in v **then**
4:         **if** CONTAINS($high(v), z$) **then return true**
5:         **else return** CONTAINS($low(v), z$)
6:     **else return** CONTAINS($low(v), z$)

Where the correct in check will evaluate if the upper-bound in the zone is lower than the upper bound in the node v. As we will use maximal sharing of subtrees, some nodes might be revisited. Once we revisit a node, we know that none of it subtrees will lead to a correct evaluation. Therefore we could use a visited flag, to save computation time. The worst case running time of the algorithm will be $\mathcal{O}(|v|)$, where $|v|$ is the number of nodes in the DDD. This can increase the running time of the state space generation significantly, as this algorithm will need to be ran every time a new state is found. It will however save memory, as no zones will be added that are already in the DDD. As the main goal of a symbolic approach is the reduction of memory, we think this is the better solution, over a less time consuming approach which would lead to more memory consumption. The algorithm will not work correctly when the discrete variables are also translated into DDD nodes, as the value in those cases needs to be correct, and an higher upper bound will not work. For the mix of DDD and LDD nodes we will need to change the algorithm to algorithm 2. This algorithm checks if a node is of type DDD. If it is, the old algorithm will be executed normally, if not, then it is an LDD node and in case of a correct evaluation only the high edge will be explored. The low edge will in that case never lead to a correct evaluation of the state.

3.3 Approach

We will propose first 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

Algorithm 2 Zone containment for mixed diagram

1: **procedure** CONTAINS(v, z)
2:     **if** $v \in 0, 1$ **then return** v
3:     **if** z[$var(v)$] correct in v **then**
4:         **if** $type(v)$ is DDD **then**
5:             **if** CONTAINS($high(v), z$) **then return true**
6:             **else return** CONTAINS($low(v), z$)
7:         **else return** CONTAINS($high(v), z$)
8:     **else return** CONTAINS($low(v), z$)

Algorithm 3 Union

1: **procedure** UNION($v1, v2$)
2:     **if** $v1 = v2$ **then return** $v1$
3:     **else if** $v1 = $ **false then return** $v2$
4:     **else if** $v2 = $ **false then return** $v1$
5:     **else if** $var(v1) \prec var(v2)$ **then**
6:         $high \leftarrow$ UNION($high(v1), v2$)
7:         $low \leftarrow$ UNION($low(v1), v2$)
8:         $result \leftarrow$ MK($cstr(v1), high, low$)
9:     **else if** $var(v2) \prec var(v1)$ **then**
10:         $high \leftarrow$ UNION($high(v2), v1$)
11:         $low \leftarrow$ UNION($low(v2), v1$)
12:         $result \leftarrow$ MK($cstr(v2), high, low$)
13:     **else if** $v1 \prec v2$ **then**
14:         $high \leftarrow$ UNION($high(v1), high(v2)$)
15:         $low \leftarrow$ UNION($low(v1), v2$)
16:         $result \leftarrow$ MK($cstr(v1), high, low$)
17:     **else if** $v2 \prec v1$ **then**
18:         $high \leftarrow$ UNION($high(v1), high(v2)$)
19:         $low \leftarrow$ UNION($v1, low(v2)$)
20:         $result \leftarrow$ MK($cstr(v2), high, low$)
21:     **else if** $v1 = v2$ **then**
22:         $high(v1) \leftarrow$ UNION($high(v1), high(v2)$)
23:         $low(v1) \leftarrow$ UNION($low(v1), low(v2)$)
24:         $result \leftarrow$ MK($cstr(v1), high, low$)
25:     **return** $result$

the symbolic representation of the state space, including the clock zones, and the transition relations, we will use normal BDDs. The DBMs will be flattened and put directly into the state vector and can than be handled by the symbolic BDD backend. Therefore both the efficient algorithms and the memory efficient representation can be used. A downside to this approach is that a zone subsumption check is not possible anymore, resulting in revisiting of some states. Further we will focus on efficient orderings of the BDDs, as both clock zones and states are contained in a single structure. We will also use this new method with the existing multicore tool, such that we can still use the subsumption check. Afther that we will continue towards a DDD model checker. First we will use the DDDs as the state space representation and still use the language module using the DBMs. We have not been able to find any literature on the combination of these techniques. There might be a significant memory improvement possible here. Eventually we want a complete symbolic solution with more operations on the DDD, such as the progress of time, then we can have a language module which does not use the DBMs any more. We will compare this approach extensively to other approaches. All of these approaches will be implemented in the LTSmin toolset. This way we can really compare the methods and not just the tools.

Alongside this we will also have to make the opaal PINS work with the UPPAAL models generated by ANIMO. The current versions doe not work together because of global variables are used in the system declaration in the generated model. We can make this work by either changing the models generated by ANIMO or by extending the opaal PINS. At this time we do not know the best solution for this problem.

3.4   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 symbolic model checking 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 also.

To make symbolic variable reordering possible 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.

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. It will also occur that different states will have the same discrete variables, but different zone variables, these all need to be checked for the subsumption check.

For the combination with the multicore backend also the data structure will need to be adapted. The current structure stores a discrete state together with a set of pointers to DBMs. In the new situation each pair of discrete state and DBM will be stored explicitly.

For the DDD the diagram will need to be able to identify the zone variables from the discrete state variables in the state vector. This will need an extra function in the PINS interface recognizing the different types of variables.

For the DDD representation the values from the DBM will need to be translated to useful variables as ordering of the DDD is based on the values. Also to check for inequalities and set containment the meaningful values are needed. In the current DBM library both the value and the operator are saved in a single 32 bit integer. The DDD will need to know the value and the operator separately.

To make an efficient DDD representation, we need to mix these DDD nodes with BDDs or LDDs for the discrete variables. For this diagram this has not been done before, we can only look at the CRD solution. Algorithms probably need some adaptation for these multiple types of nodes. Also the possibility and results of the mixing of discrete and zone variables need to be researched. It might be that it functions better when the two types of variables remain separated, or when they are mixed.

3.5 Planning

In the table below we have put all actions that need to be done into tasks. In the second column we put which tool should work correctly for the opaal language module after the task. This will give us intermediate points on which we can test the work that has been done to that point.

Task	Needs to function
Flatten the DBMs	Symbolic tool
Partition the next state function	Symbolic tool
Create dependency matrices	Symbolic tool variable reordering
Change subsumption check for multicore approach	
Other adaptions multicore approach	Multicore tool
Create minimal DDD library	
Combine DDD with LDD	
Extend language module for DDD approach	Symbolic tool
Benchmark tests for multiple approaches	
Test with ANIMO models	
Extend DDD library for fully symbolic approach	
Change language module for fully symbolic DDDs	Symbolic tool
Benchmark Testing fully symbolic approach	
Writing Report	

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] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] Farn Wang. Efficient verification of timed automata with bdd-like data-structures. 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.

[10] 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.

[11] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, February 1999.

[12] 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.

[13] 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.

[14] Gerd Behrmann, Alexandre David, and KimG. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Meth-*

ods for the Design of Real-Time Systems, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[15] 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.

[16] 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.

[17] 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.

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

[19] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for BDD-based verification of real-time systems. In W. A. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003, Boulder, CO, July 8-12)*, LNCS 2725, pages 122–125. Springer-Verlag, Heidelberg, 2003.

[20] Dirk Beyer. Efficient reachability analysis and refinement checking of timed automata using BDDs. In T. Margaria and T. F. Melham, editors, *Proceedings of the 11th IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001, Livingston, September 4-7)*, LNCS 2144, pages 86–91. Springer-Verlag, Heidelberg, 2001.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] Jesper Mller, Henrik Hulgaard, and Henrik Reif Andersen. Symbolic model checking of timed guarded commands using difference decision diagrams. *The Journal of Logic and Algebraic Programming*, 5253:53 – 77, 2002.
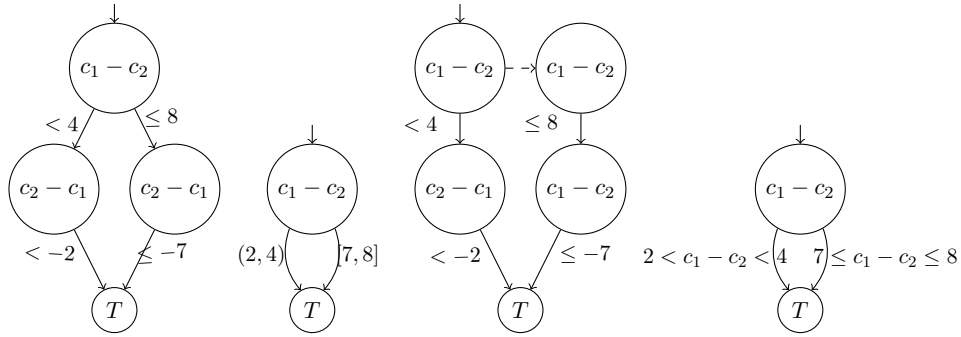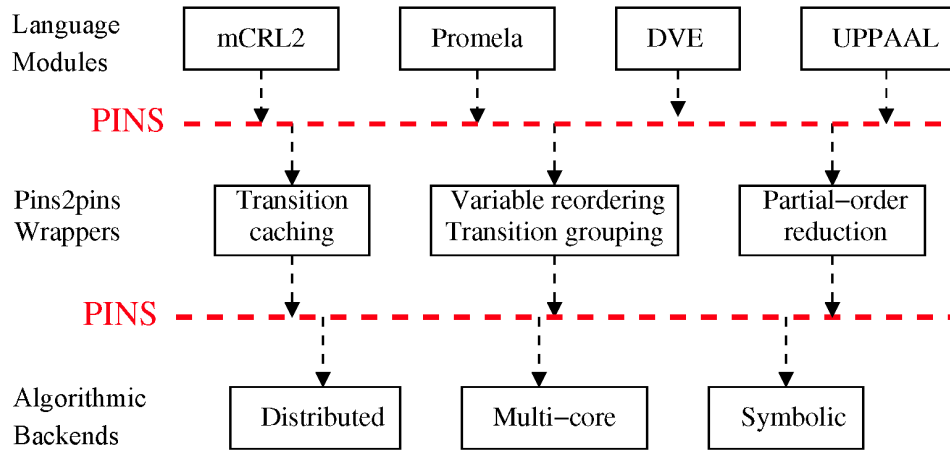
Figure 1: A CRD, CDD, DDD and CMD representation

Figure 2: Modular structure of LTSmin