Symbolic Model Checking of Timed Automata using LTSmin

Sybe van Hijum

July 26, 2016

Contents

1 Introduction

Timed automata [2] is a widely used modelling formalism. A recent usage of this formalism is the modelling of biological signalling pathways [22]. ANIMO is a tool that generates these timed automata from biological signalling pathways models. 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) [1, 9] and variations like LDDs (List Decision Diagrams) [8] and MDDs (Multi-valued Decision Diagrams) [23] 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) [16], CMDs (Constraint Matrix Diagrams) [14], CRDs (Clock Restriction Diagrams) [25] and DDDs (Difference Decision Diagrams) [18, 20]. All of these methods show some extra difficulties or limitations over BDDs. Also after their introduction they have not been developed further.

LTSmin [7, 15] is a language independent on the fly model checker with several algorithmic back-ends. Its symbolic back-end 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 UPPAAL [3] through the Opaal [11] lattice model checker. Through this module UPPAAL models can be loaded into LTSmin. For this language currently, only the multi-core back-end can be used [10]. This multi-core approach showed efficient enough to compete with the latest version of the UPPAAL model checker. It showed significant speedups on multi-core machines, at the cost of some memory increase however. To tackle the memory increase a combination of the Opaal front-end and the symbolic back-end could be a solution.

The symbolic back-end 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 [24]. Using this together with the UPPAAL language front-end 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 Preliminaries

We will first define timed automata and zones, a method used to represent time in timed automata. Also a subsumption check over zones will be defined.

2.1 Timed Automata

Timed automata is a formalism that extends labelled transition systems with one ore more clocks. Guards over these clocks, denoted as $G(C)$ can be used for transitions. Also reset actions for clock can be defined for transitions. All clocks in the system will increase at the same rate. As our work continues on [10] we use the same definition of timed automata.

Definition 1 (Timed Automata)**.** *An extended timed automaton is a 6-tuple* $A = \langle L, C, Act, s_0, \rightarrow, I_c \rangle$ *where*

- L *is a finite set of locations, typically denoted by* l
- C *is a finite set of clocks, typically denoted by* c
- Act *is a finite set of actions*
- $l_0 \in L$ *is the initial location*
- $\rightarrow \subseteq L \times G(C) \times Act \times 2^C \times L$ *is the (non-deterministic) transition relation. We normally 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, which is a parallel composition, to define larger systems.

Definition 2 (Network of timed automata [10])**.** *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* $A_i = \langle L_i, C, Act, S_0^i, \rightarrow_i, I_C^i \rangle$ *for all* $1 \leq i \leq n$*, where* $n \in \mathbb{N}$*, is a network of timed automata, denoted* $A = A_1||A_2||..||A_n$.

A network of timed automata is a parallel composition that synchronizes on a set of channels $Chan$ [3]. $ch!$ and $ch?$ represent the output and input action on the channel $ch \in Chan$.

$$\begin{array}{c} \\ \mathbf{O} \\ c_1 \\ c_2 \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{O} & c_1 & c_2 \end{array} \\ \begin{pmatrix} (0,\leq) & (0,\leq) & (0,\leq) \\ (5,\leq) & (0,\leq) & (\infty,\leq) \\ (4,\leq) & (\infty,\leq) & (0,\leq) \end{pmatrix} \end{array}$$

Figure 1: DBM

2.2 Zones

For basic transition systems the state space can grow exponentially for the size of the system. The state space of Timed automata is by definition infinite, as clocks have real values. If a state is defined between two points in time, an infinite amount of moments in time can happen during that state. Even when some granularity is used, that defines that clocks will only increase with certain step size the automata can still have infinite state space. 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) [4, 12].

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 $x \in \mathbb{Z}$. For the constraints over the single clocks an extra clock $\mathbf{O}$ with a constant value 0 is added. This way the upper and lower bound of a clock c_i can be given by $c_i - \mathbf{O} \preceq x$ and $\mathbf{O} - c_i \preceq y$. The addition of this $\mathbf{O}$ clock will give the matrix of a timed automaton always size $(|C| + 1)^2$. 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. In figure 1 we give an example of a DBM with two clocks: c_1 and c_2, representing the zone $0 \leq c_1 \leq 5 \wedge 0 \leq c_2 \leq 4$. The diagonal only contains $(0, \leq)$ values as these elements give the difference between a clock and itself, which is clearly always 0.

2.3 Zone subsumption

In model checking an important function is to check if a certain state has been visited already earlier. For normal automata this can be done by comparing the newly found state to all states that have already been visited, and check if one of those states is equal to that new state. This is often done by more efficient methods, like hash functions, but the equality check remains. For states with zones this equality check does not satisfy. Two

zones do not need to be equal, but the newly discovered zone can also be a subset of the earlier discovered zones. In LTSmin this is done by a subsumption check [10] that is performed over the DBMs. This check is delegated to the UPPAAL DBM library. The function checks if a new zone is a subset of the zone represented by a DBM.

3 Related Work

In this related work section we will discuss a number of methods used for model checking timed automata. We will choose a method to extend our work on, and go more into detail on that method.

3.1 Methods

Already several model checkers for timed automata exist such as UPPAAL [3], KRONOS [26], RABBIT [6] and RED [25]. We focus mainly on the UPPAAL tool as we use the same input format. Opaal [11], the language module for LTSmin, 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 [5]. The method however only works for closed timed automata, meaning that no strict comparisons on clocks can be made in the model and that clocks only can be compared to integers. This approach is very sensitive to the granularity of the values used and the upper bound of the clock values. When fine granularity or large upper bounds are used, the memory usage will increase too much. An advantage of this approach is that basic model checking approaches can be used and no extra complexity due to zone calculations is added. This method results in a transition system with only discrete variables, so a normal BDD package can be used. In [21] a similar approach is proposed by using clock tick actions to represent time progress and removing clock variables altogether.

The most established method to represent clock zones are DBMs. We gave an introduction to this structure in the preliminaries section. Several methods based on BDDs have been developed to represent zones. All of these are similar to DBMs in the sense that they use clock constraints to represent the zones. These structures use a BDD-like structure to represent the zones more efficiently. Below we shortly describe four methods. For each method we give an example, all examples represent $2 < c_1 - c_2 < 4 \vee 7 \leq c_1 - c_2 \leq 8$.

CDDs [16] use single nodes for each variable and have multiple disjoint intervals for that variable on the edges. This results in a node with a larger fanout. The upper and lower bound for each pear of clocks are represented in a single node, as the edges represent intervals. Requiring the disjointness of intervals can lead to a memory inefficient representation, as intervals need to be cut in more smaller parts. All algorithms on CDDs do not maintain disjointness, after every step it needs to be re-established. In figure 2 we have an example of a CDD.

DDDs [18,20] use a upper-bound constraint on each node that can either

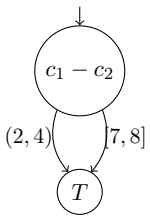

Figure 2: CDD representation

be true or false. Each node thus has a fixed fanout of two. 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. In figure 3 an example of a DDD is shown.

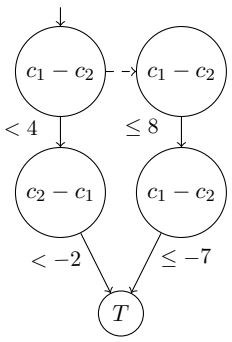

Figure 3: DDD representation

CRDs [25] differ mainly from CDDs by not using disjoint intervals but possibly overlapping upper bounds, for a pair of variables on their edges. This diagram will have a larger fanout per node, like CDDs. Several normal forms for this diagram are proposed, with different performance results. It is also shown that CRDs can be combined with BDDs into a single structure to fully symbolic represent state space. In figure 4 we give an example of a CRD.

CMDs [14] 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. Upper- and lower-bounds of multiple clock pairs can be on a single edge. CMDs do not have a canonical form so only some reductions are proposed. An example of a CMD is given in figure 5. This figure contains two examples, the first is a diagram of the constraint we use in this section. To show the difference with other diagrams we also give a diagram representing the same zone as the DBM in figure 1.

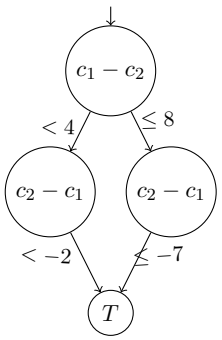

Figure 4: CRD representation

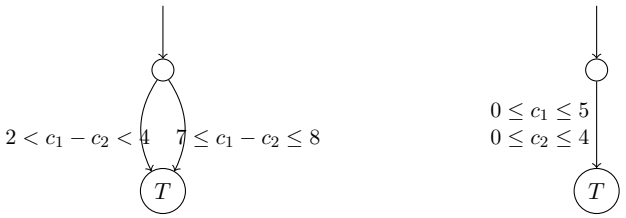

Figure 5: CMD representation

In [13, 27] a method is proposed purely based on BDDs by translating the constraints directly into BDD nodes. We call this method BDD zones. 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. Subsumption for this method may be difficult. On BDDs only equalities can be checked, and no inequalities. This way inclusion is not trivial to check by normal BDD algorithms.

A known difficulty in BDDs is the variable ordering. A bad ordering can lead to a BDD of exponential size, where a good ordering can sometimes lead to a significantly smaller diagram. Of the zone diagrams named above, only for CRDs experiments with different orderings have been conducted, 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. In Table 1 we compare the different types of diagrams we discussed above.

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

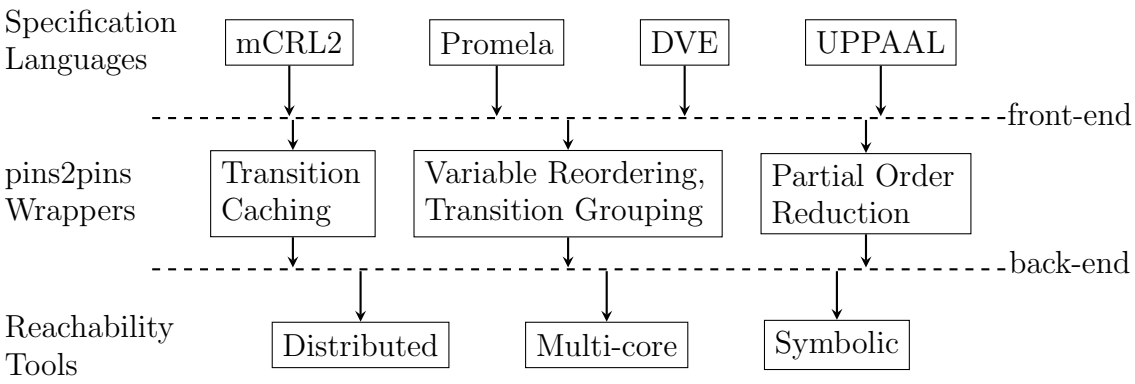

Figure 6: Modular structure of LTSmin

3.2 LTSmin

LTSmin [7, 15] is a language independent model checker. It is built in a modular way such that new languages can be added by a PINS (Partitioned Next-State Interface) interface without too much effort, and new algorithms can be added 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 6. The PINS is the core of LTSmin. This interface abstracts as much as possible from the model without losing the structure. It represents states as fixed length integer arrays. 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 [17]. With these matrices, more efficient symbolic algorithms can be used, the number of next-state calls can be reduced, efficient variable re-orderings can be used, 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 none of these algorithms can be used. Also the DBM variable is only represented by a pointer, which is not a meaningful value for the transition system. LTSmin uses the pointer to a DBM to do the subsumption check as described in section 2.3.

3.3 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. We wanted a diagram that can store both discrete states and zones, this can either be done in the diagram, or in a combination of the diagram and BDD or LDD nodes. Also a subsumption check on the diagram should be possible. We chose from the four zone representing diagrams discussed earlier. The CDD approach was not chosen due to the memory inefficient disjoint intervals and their algorithms not maintaining these disjointness. The CMD approach is too similar to DBMs, on which we already have an approach. The choice between CRD and DDD was between two quite similar diagrams. We have decided to continue on the DDD. It is a diagram form that is closely related to LDDs, for which we already have a library, so we can reuse parts of the LDD 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 also on the results side we can extend the current research.

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. First we give the definition of a DDD.

Definition 3 (Difference Decision Diagram [20])**.** *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 [20] 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, as the function for BDDs does. 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.

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

To order DDD nodes we use the operator $\prec$. This orders variables and variable pairs in a predefined order. It orders bounds by increasing constants, and the $\leq$ operator before the $\leq$ operator. So a node v with $bound(v) = (0, <)$ comes before $bound(u) = (0, \leq)$ which comes before $bound(w) = (1, <)$.

Definition 4 (Ordered DDD [20])**.** *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 [20])**.** *A locally reduced DDD ($R_L DDD$) is an ODDD satisfying, for all non-terminals u and v:*

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

We give an example of the last point in figure 7. Here both diagrams represent the same zone: $2 < c_1 - c_2 \leq 8$. The node with < 4 on the high edge is redundant in this example and can thus be removed. 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 [20])**.** *A path-reduced DDD ($R_P DDD$) is a locally reduced DDD where all paths are feasible.*

To define tightness we first need to define dominating constraints.

Definition 7 (Dominating constraint [20])**.** *A constraint $x_i - x_j \lesssim c$ is dominating in a path p if all other constraints $x_i - x_j \lesssim' c'$ on the same pair of variables in p are less restrictive.*

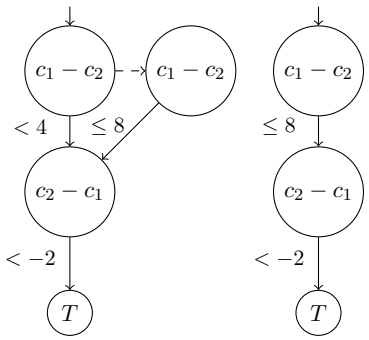

Figure 7: Local reduction

Definition 8 (Tightness [20]). *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 $R_L DDDu$ is tight if all paths from u are tight.*

Definition 9 (Saturation [20]). *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 10 (Disjunctive vertex [20]). *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 together lead to the following definition of a fully reduced DDD.

Definition 11 (Fully reduced DDD [20]). *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 [19]. 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.

4 Plan

We will first implement a method that will use the best of both worlds, the efficient algorithms from the DBMs and the memory efficiency of a symbolic approach. 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, and the transition relations, we will use normal LDDs. The DBMs will be flattened and put directly into the state vector and can then be handled by the symbolic BDD back-end. 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, as only equalities and no inequalities can be checked on BDDs, 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 explicit-state multi-core tool, such that we can still use the subsumption check that is implemented in LTSmin. 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 aim at a complete symbolic solution with more operations on the DDD, such as the progress of time, we can then aim at a language module which does not use DBM's, but only gives bounds on clocks which can be calculated in the DDDs. We will compare the different approaches we implement extensively to each other. All of these approaches will be implemented in the LTSmin toolset. This way we can really compare the methods and not just the tools. We will use the same language front-end and architecture, only the algorithmic back-end will really change. In the language front-end only smaller changes will be needed.

Alongside this we will also have to make the Opaal PINS work with the UPPAAL models generated by ANIMO. The current versions do not work together because global variables are used in the system declaration in the generated model, and this is a feature that Opaal does not support. 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.

4.1 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, with flattened DBMs, also usable for the explicit state multi-core approach with subsumption?
- Can the BDD approach be generalized towards a method using DDDs?
- Is a fully symbolic reachability analysis using DDDs more efficient than the combination of DDDs and DBMs, both on memory and speed?

4.2 Algorithms

To create a DDD library we will implement a number of functions over DDDs. We will limit the functions to the ones needed for this purpose. Therefore it will not become a complete DDD package. One of the core operations on DDDs is the apply operation. This operation takes two DDDs and a binary operator and combines the two DDDs according to the operator. The apply function for DDDs is a generalisation of the function for BDDs. In [20] a general definition of the algorithm is given. We turned this more mathematical definition into an algorithm, we give pseudo-code in Algorithm 1. In Algorithm 2 we give the pseudo-code for the apply function with the or operator, or the union function, this way we can increase performance by not going down the entire diagram if we already found a terminal. All functions rely on a Mk function which checks if the node needed already exists, and otherwise creates a new node. The Mk function will ensure that a DDD is locally reduced as described in definition 5. The subsumption check, which we lost in the BDD approach, will be possible again with DDDs. This will be the same check as a state membership in an LDD. The only difference is that no equality, but upper bounds will be checked. Pseudo-code for this algorithm is given in Algorithm 3. If we combine DDDs with LDDs, only the correct check has to be adapted, checking for equalities, not inequalities, the algorithm will remain the same.

4.3 Planning

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

Algorithm 1 Apply

```
1: procedure APPLY(v1, v2, op)
2:     if v1 ∈ {0, 1} ∧ v2 ∈ {0, 1} then
3:         result ← (v1 op v2)
4:     else if var(v1) ≺ var(v2) then
5:         high ← APPLY(high(v1), v2, op)
6:         low ← APPLY(low(v1), v2, op)
7:         result ← MK(cstr(v1), high, low)
8:     else if var(v2) ≺ var(v1) then
9:         high ← APPLY(high(v2), v1, op)
10:        low ← APPLY(low(v2), v1, op)
11:        result ← MK(cstr(v2), high, low)
12:    else if v1 ≺ v2 then
13:        high ← APPLY(high(v1), high(v2), op)
14:        low ← APPLY(low(v1), v2, op)
15:        result ← MK(cstr(v1), high, low)
16:    else if v2 ≺ v1 then
17:        high ← APPLY(high(v1), high(v2), op)
18:        low ← APPLY(v1, low(v2), op)
19:        result ← MK(cstr(v2), high, low)
20:    else if v1 = v2 then
21:        high(v1) ← APPLY(high(v1), high(v2), op)
22:        low(v1) ← APPLY(low(v1), low(v2), op)
23:        result ← MK(cstr(v1), high, low)
24:    return result
```

Algorithm 2 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) ≺ var(v2) then
6:         high ← UNION(high(v1), v2)
7:         low ← UNION(low(v1), v2)
8:         result ← MK(cstr(v1), high, low)
9:     else if var(v2) ≺ var(v1) then
10:        high ← UNION(high(v2), v1)
11:        low ← UNION(low(v2), v1)
12:        result ← MK(cstr(v2), high, low)
13:    else if v1 ≺ v2 then
14:        high ← UNION(high(v1), high(v2))
15:        low ← UNION(low(v1), v2)
16:        result ← MK(cstr(v1), high, low)
17:    else if v2 ≺ v1 then
18:        high ← UNION(high(v1), high(v2))
19:        low ← UNION(v1, low(v2))
20:        result ← MK(cstr(v2), high, low)
21:    else if v1 = v2 then
22:        high(v1) ← UNION(high(v1), high(v2))
23:        low(v1) ← UNION(low(v1), low(v2))
24:        result ← MK(cstr(v1), high, low)
25:    return result
```

Algorithm 3 Zone containment for DDDs

```
1: procedure CONTAINS(v, z)
2:     if v ∈ 0, 1 then return v
3:     else if z[var(v)] correct in v then
4:         return CONTAINS(high(v), z)
5:     else return CONTAINS(low(v), z)
```

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. (1)

To make symbolic variable reordering possible we will need to partition the next-state function. 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. (2)

Once the next-state function is partitioned, also a sparse dependency matrix is needed. This will need to be created according to the transition groups. After this step variable reordering in LTSmin should work. (3)

To combine the new language module, with flattened DBMs, with the multi-core LTSmin back-end 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. Then it can recreate the DBM from the correct variables. (4)

For the combination with the multi-core back-end 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. (5)

To use a DDD approach, the first step will be to create a minimal DDD library that has the functionality to save a state space. This library will probably miss some features for more advanced model checking techniques. (6)

Once we have a DDD library, we need to combine this with nodes to represent discrete state variables. We will use LDD nodes for this, as they share the same structure with DDDs. (7)

For the combination of DDD and LDD nodes, 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. (8)

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. In this section we describe all things that need to be implemented to make model checking with a certain diagram possible. (9)

Once we have a working DDD representation the first set of benchmark tests can be conducted. This will be the big set of tests for the BDD, DDD and multi-core approach, we can compare these to the old multi-core approach and to the newest version of UPPAAL. (10)

To continue towards a fully symbolic approach, we will need to extend the DDD libary with some functions. We need for example a function to let

time progress, and to set invariants over the states. (11)

For the fully symbolic approach again the language module will need to be adapted. We will no longer need the DBM in the language module. The module will only need to change the discrete state variables, the zone variables will be adapted in the diagram. (12)

If we have this fully symbolic approach this will also need to be tested. We will conduct the same tests as we did for the other approaches, such that we can compare this diagram to all earlier approaches. (13)

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	Date
Start		01-02-2016
Flatten the DBMs (1)	Symbolic tool	15-02-2016
Partition the next-state function (2)	Symbolic tool	01-03-2016
Create dependency matrices (3)	Symbolic tool variable reordering	15-03-2016
Subsumption check for multi-core back end (4)		01-04-2016
Other adaptions multi-core approach (5)	Multi-core tool	01-04-2016
Create minimal DDD library (6)		15-04-2016
Combine DDD with LDD (7)		01-04-2016
Language module for DDD approach (8, 9)	Symbolic tool	08-04-2016
Benchmark tests for multiple approaches (10)		15-04-2016
Test with ANIMO models		15-04-2016
DDD library for fully symbolic approach (11)		15-05-2016
Language module for fully symbolic DDDs (12)	Symbolic tool	01-06-2016
Benchmark Testing fully symbolic approach (13)		15-06-2016
Writing Report		01-07-2016

5 Notes

5.1 Flattening DBM

We flattened the DBM, to make the opaal language module work with any algorithmic backend in LTSmin. We only did this on the edges of the successor function. So this function reads a flattened DBM as input and returns it as successor states, internally the original DBM representation is still used. This way the code had to be adapted the least. In this flattening we removed the diagonal elements of each DBM. By the way DBMs are constructed this will always represent the difference between a clock and itself. This difference is by definition always 0, so it can be removed, and hard coded be set to 0 internally. This reduces the number of state variables in the state vector by one for each clock.

5.2 Dependency Matrices

To get the best possible result of the regrouping algorithms, the dependency matrices had to be made as sparse as possible. This has been done for both the read matrix and may-write matrix. We had to use the may-write, instead of the must write matrix, as it is not always clear if a variable will really be written, or that in may stay untouched. First of all, all C-like code is parsed. Here it is stored per function which variables are read and written, and which other functions are called. Next all transitions are parsed, here some variables are read and written directly. Transitions can also call functions, then from the earlier parsed functions the read and written variables are used. In the third step we need to look at the time extrapolation. This extrapolation is based on the value of the location variable, so it results in a read dependency. In some cases, there is no difference between all possible location values, so a location does not need to be read. A final step is that a location variable that can be urgent or committed always has to be read. If this location is in an urgent state, than no other transitions can happen, so all other transitions have to check that they are not in an urgent state.

5.3 Animo Models

We started the project with ANIMO models that were not compatible with opaal. As opaal does only support a subset of all options of UPPAAL. First of all we changed the model, such that it does not use global variables in in the system declaration. Also some smaller changes to the use of structs had to be made. This resulted in a basic ANIMO model that is compatible. Larger models are still not compatible due to clock guards on input synchronization channels. This is a feature only recently implemented by UPPAAL(version 4.1.3). Opaal does not support this feature, and its' semantics are not completely clear, as it is not described in the manual.

Adding this to opaal can be done, but is not trivial. This improvement of the language module is out of scope of this thesis.

5.4 DDD nodes

We used the basis of the LDD package in Sylvan to create our DDD nodes. DDD nodes are stored in 128 bits, represented as a struct of two 64 bit integers.

```
struct dddnode {
    uint64_t a, b;
} * dddnode_t;
```

In this struct the value(32 bits), the true edge(40 bits), the false edge(40 bits) and a type bit, operator bit and flag bit are stored. The type bit indicates if a node is a DDD or an LDD node, if it is set to 0 it should be treated as a normal LDD node. The operator bit shows if the operator is $<$ or $\leq$, this can only be used if the type bit is also set to 1(DDD). The flag bit is used in some algorithms to indicate that a certain node has already been visited. All of this is stored compactly in the two 64 bit integers. The total information is 115 bits, so there are still 17 unused bits, all unused bits are set to 0. The depth of the node is not stored, this can be calculated by going down through the structure. This implies that no level can be skipped. Other DDD algorithms and reductions show that some levels are not needed. We solved this by indication a skipped level by $< \infty$, which is true for every upper bound. For such nodes the false edge will always directly lead to the false end node.

The difference between the original LDD nodes and our DDD nodes is only in the operator and type bit. This way we can still use the same unique table.

Algorithm 4 Reduce

```
1: procedure REDUCE(dbm, dim)
2:     for i ∈ dim do
3:         for j ∈ dim do
4:             for k ∈ dim do
5:                 if !(dbm[i,k] ∨ dbm[k,j] ∨ dbm[i,j] on diagonal) then
6:                     if dbm[i,k] + dbm[k,j] ≤ dbm[i,j] then
7:                         dbm[i,j] := ∞
```

We use a path reduced version of the DDD. This reduced version is not needed for all operations. Maybe we can improve performance by use the reduce function more or less often. For the count of the number of states we will use the amount of paths in the reduced DDD. States are not well defined for timed automata, as it is a set of discrete locations together with

Algorithm 5 Reduce

```
1:  procedure ReduceZero(dbm, dim)
2:      placed[dim] all 0
3:      red[dim, dim] all 0
4:      eq[dim, dim] all 0
5:      cl := 0
6:      newDBM[dim, dim] diagonal ∞ rest 0
7:      for i ∈ dim do
8:          if placed[i] = 0 then
9:              for j ∈ dim do
10:                 if dbm[i, j] + dbm[j, i] = 0 then
11:                     placed[j] := 1
12:                     eq[cl, j] := 1
13:             cl + +
14:     repr[cl]
15:     for i ∈ cl do
16:         for j ∈ dim do
17:             if eq[i, j] = 1 then
18:                 repr[i] := j
19:                 break
20:     clg[cl, cl]
21:     for i ∈ cl do
22:         for j ∈ cl do
23:             clg[i, j] := dbm[repr[i], repr[j]]
24:     Reduce(clg, cl)
25:     for i ∈ cl do
26:         for j ∈ dim do
27:             if eq[i, j] = 1 then
28:                 for k ∈ dim do
29:                     if eq[i, k] then
30:                         newDBM[j, k] = dbm[j, k]
31:         for j ∈ cl do
32:             newDBM[repr[i], repr[j]] := clg[i, j]
33:     return newDBM
```

Algorithm 6 MK

```
1: procedure MK(var, bound, h, l)
2:     if bound = " < ∞" then
3:         return h
4:     if member hashtable then
5:         return lookup
6:     else if l = h then
7:         return l
8:     else if var = l.var ∧ h = l.high then
9:         return l
10:    else
11:        return insert(new Node)
```

a zone of clocks. As the zones can have different representations, this gives some strange results. We will use the DDD function Biimplication for the equal operation and we will use implication for the subsumption check.

5.5 Minus

The minus function, used for the reachability, has not been implemented as an DDD functions. This function is different to other functions, as information has to be transferred over different levels. For simple cases, an upper-bound in one of the operands of the minus, can become a lower-bound in the result, and vice-versa. A simple one dimensional example is [0..8]/[0..4), this will result in [4..8]. In this case the 4 is the upper-bound of the subtrahend. It will however become the lower-bound of the difference. As lower- and upper-bounds are saved on different levels in DDDs this makes the function different from all other functions, which only look at values on the same level.

In figure **??** we have a two-dimensional example of how the minus function can become more complex for multiple-dimensions. In this case we make a hole in a larger zone. Both the minuend and the subtrahend are represented by a DDD with a single path. For simplicity we removed the diagonals in this example, as they play no role. The difference however becomes a DDD with 4 paths and 10 nodes. Again a lot of upper- and lower-bounds are switched. Already for this example we could not find a algorithm that does this in general. For more dimensions, and DDDs with already multiple paths the problem will only get harder. That is why we returned to a DBM function for this.

The DBM function we use is defined in the UPPAAL DBM library. The minus function is defined over a federation of DBMs. This federation is a C++ class containing multiple DBMs. This federation is needed as we can do a minus over a collection of zones, multiple paths in the DDD, and

the result can contain multiple zones. As already shown in the example of figure **??**. For this function we first take the normal LDD minus function over the discrete part. At the first DDD level representing the zones, the DBM function is called. From this level all possible paths are searched, and for each path a DBM is created. All these DBMs are put in a federation, on which the library function can be called. The result is again (a possibly empty) federation. If the federation is empty, simply a DDD-false node is returned. Otherwise each DBM is turned into a DDD path and these paths are unioned together.

5.6 BFS

The DBM minus function we use is quite expensive. To overcome this problem we will use two different versions of the search algorithm. Our second version will not use the minus function. In algorithm 7 we show the standard BFS algorithm, this will be the first algorithm we use. Algorithm 8 shows how we can edit this algorithm. The constraint of the loop is changed from an empty check of the current set, to a check that the total visited set has not changed. This change now shows that the minus is not necessary any more, as shown in algorithm 9. The implication is that the current set will in some cases be larger than in the previous algorithm. This will have some negative impact on the next-state calls, which will take more time. Not using the expensive minus function might compensate for that.

Algorithm 7 BFS

```
1: procedure BFS(initial)
2:     vis := cur := initial
3:     while cur ≠ ∅ do
4:         cur := next(cur)
5:         vis := vis ∪ cur
6:         cur := cur \ vis
```

Algorithm 8 BFS

```
1: procedure BFS(initial)
2:     vis := cur := initial
3:     vis_prev := ∅
4:     while vis ≠ vis_prev do
5:         vis_prev := vis
6:         cur := next(cur)
7:         vis := vis ∪ cur
8:         cur := cur \ vis
```

Algorithm 9 BFS

```
1: procedure BFS(initial)
2:     vis := cur := initial
3:     vis_prev := ∅
4:     while vis ≠ vis_prev do
5:         vis_prev := vis
6:         cur := next(cur)
7:         vis := vis ∪ cur
```

References

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

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] Andreas Engelbredt Dalsgaard, Ren Rydhof Hansen, Kenneth Yrke Jørgensen, 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.

[12] David L. 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] Jesper Møller, 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.

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

[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 Méry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 326–340. Springer Berlin Heidelberg, 2012.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.