Symbolic Model Checking of Timed Automata using LTSmin

Sybe van Hijum

August 8, 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 [28]. 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) [1,10] and variations like LDDs (List Decision Diagrams) [9] and MDDs (Multi-valued Decision Diagrams) [29] 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) [19], CMDs (Constraint Matrix Diagrams) [15], CRDs (Clock Restriction Diagrams) [31] and DDDs (Difference Decision Diagrams) [23, 26]. All of these methods show some extra difficulties or limitations over BDDs. Also after their introduction they have not been developed further.

LTSmin [8, 17] 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 [4] through the Opaal [12] 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 [11]. 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 [30]. 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 [11] 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 a finite number of timed automata to a network of timed automata, which is a parallel composition, to define larger systems.

Definition 2 (Network of timed automata [11])**.** *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 $Chan$ [4]. $ch!$ and $ch?$ represent the output and input action on the channel $ch \in Chan$.

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

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 if a clock is 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 (DBMs) [5, 13].

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.

A number of operations on DBMs has been defined. We will introduce the operations we use. The same notation as [11] is used.

- $D \uparrow$ is also called the delay operator. This lets time pass unlimitedly from the zone in D.
- $D \cap D'$ adds additional constraints from D' to D. This is used for transitions that have clock constraints. These constraints can be represented as a DBM.

- $D[r]$ with $r \subseteq C$, resets all clocks in r.
- D/B does a maximal bounds extrapolation. In section 5.2 we will go into more detail about this extrapolation.

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 [11] 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 [4], KRONOS [32], RABBIT [7] and RED [31]. We focus mainly on the Uppaal tool as we use the same input format. Opaal [12], 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 [6]. 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 [27] 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 [19] 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 [23,26] use a upper-bound constraint on each node that can either

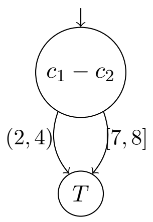


Figure 2: CDD representation

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.

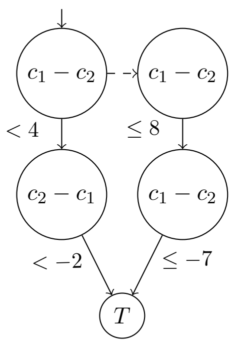


Figure 3: DDD representation

CRDs [31] 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 [15] 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.

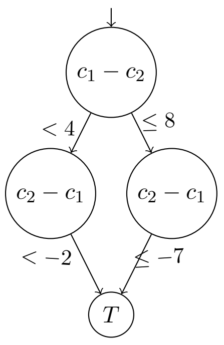


Figure 4: CRD representation

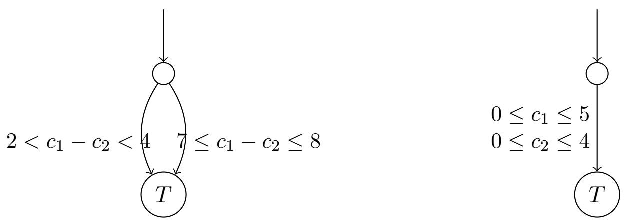


Figure 5: CMD representation

In [14, 33] 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

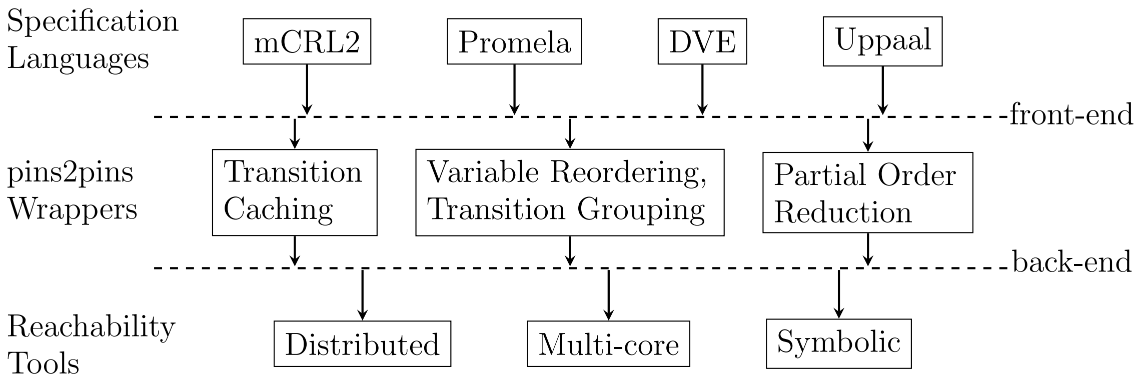


Figure 6: Modular structure of LTSmin

3.2 LTSmin

LTSmin [8, 17] 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 [21]. 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 [26])**.** *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.

Now we have the definition of the structure. We also give the semantics of this structure.

Definition 4 (DDD semantics)**.** *The semantics of a vertex is defined recursively by the function $\mathcal{V} : V \rightarrow$ **Exp** :*

- $\mathcal{V}[[0]] \overset{\text{def}}{=} \mathit{false}$,
- $\mathcal{V}[[1]] \overset{\text{def}}{=} \mathit{true}$,

- $\mathcal{V}[[v]] \overset{\text{def}}{=} \begin{cases} (pos(v) - neg(v) < const(v)) \rightarrow \mathcal{V}[[high(v)]], \mathcal{V}[[low(v)]] \mathit{if}\ op(v) =' <' \\ (pos(v) - neg(v) \leq const(v)) \rightarrow \mathcal{V}[[high(v)]], \mathcal{V}[[low(v)]] \mathit{if}\ op(v) =' \leq' \end{cases}$

In the semantics we only take the information on the high edges. The implicit information on the low edge is not used. A node can thus only represent an upper-bound which is either true or false, it can not implicitly represent a lower-bound on the same variable pair. This representation also makes it easier to work with the state-vectors of LTSmin.

In [26] 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 5 (Ordered DDD [26])**.** *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 6 (Locally Reduced DDD [26])**.** *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.

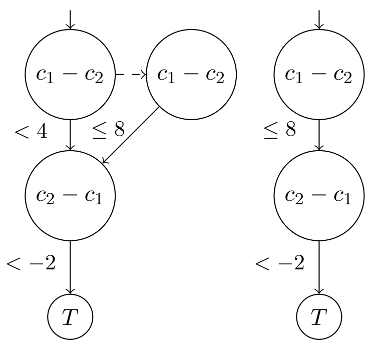


Figure 7: Local reduction

For BDDs these reductions would be enough to have a fully canonical structure. For DDDs this is not the case. Due to dependencies between the bounds. In figure 8 we give an example for this by giving two different locally reduced DDDs representing the same zone. The resulting zone of both these DDDs is drawn in figure 9 we show the result of this zone, which is the square in which both clock c_1 and c_2 are between 0 and 5.

The $R_L DDD$ is clearly not canonical. We first define a path in a DDD as the bound on all high edges that are traversed in a single walk from the top node to the true node. A path will only have one bound for each variable pair.

Definition 7 (Path-reduced DDD [26]). *A path-reduced DDD ($R_P DDD$) is a locally reduced DDD where all paths are feasible.*

This definition ensures that all paths in a DDD actually represent a zone, and that there are no redundant paths in the DDD that just represent an empty set. This usage of paths is compatible to the state vectors used in LTSmin. An $R_P DDD$ is still not canonical. We need to define tightness, saturation and disjunctive vertices. To define tightness we first need to define dominating constraints.

Definition 8 (Dominating constraint [26]). *A constraint $x_i - x_j \precsim c$ is dominating in a path p if all other constraints $x_i - x_j \precsim' c'$ on the same pair of variables in p are less restrictive.*

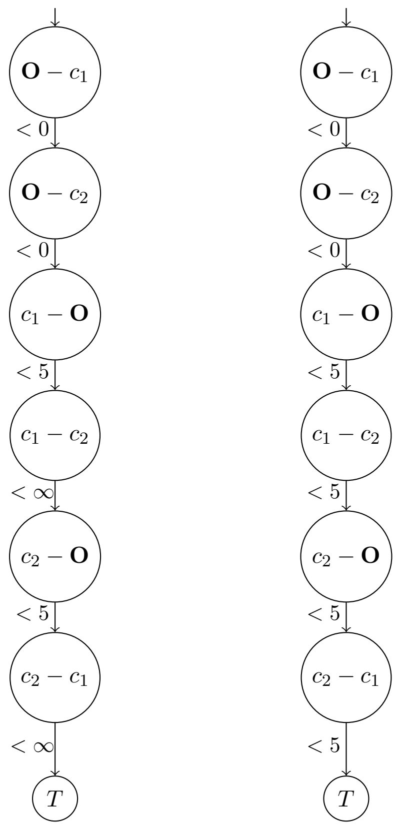


Figure 8: Two DDDs representing the same zone

Definition 9 (Tightness [26]). *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 10 (Saturation [26]). *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 11 (Disjunctive vertex [26]). *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 $[p] \wedge (\alpha \to h, l)$ and $[p] \wedge (h \vee l)$ have the same set of solutions.*

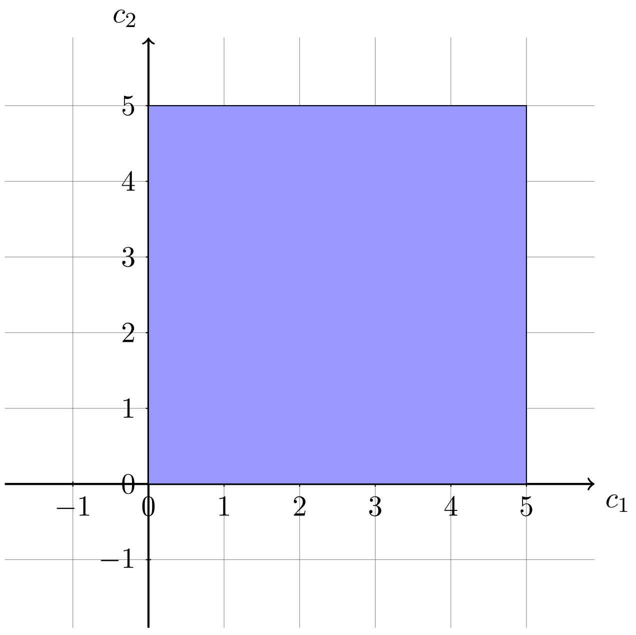


Figure 9: Resulting zone of DDDs in figure 8

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

Definition 12 (Fully reduced DDD [26])**.** *An R_pDDD u is a fully-reduced DDD (R_FDDD) if it is tight, saturated and has no disjunctive vertices.*

We assume that this fully-reduced DDD is canonical and work from that. It is not ensured that this is actually the case, there is no proof for it.

Conjecture 1 (Canonical DDD [26])**.** *If u and v are R_FDDDs with the same set of solutions then $u = v$.*

DDDs can also be 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. We will connect the DDD to an LDD to represent discrete variables to limit the number of nodes.

So far we only found the results of two benchmark tests of DDDs, Milner's scheduler and Fischer's protocol [24]. 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 or number of nodes needed.

3.4 List Decision Diagram

The DDD nodes are connected to LDD nodes to represent the discrete variables. We will introduce the LDD structure here. An LDD is used to represent variables with integer values, not only binary values. In contrast to MDDs this is done for one value per node. Resulting in nodes with equal size. We will first define the LDD structure.

Definition 13 (List Decision Diagram)**.** *A list decision diagram (LDD) 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*
var(v)	***Var***	*Variable x*
const(v)	$\mathbb{Z}$	*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.*

The definition is almost equal to DDDs, definition 3. The difference is the operator that is not in LDDs. LDDs can be seen as a DDD with not a $<$ or $\leq$ as operator, but a $=$.

4 Implementation

This section will go more into detail about the implementation we made and the design choices that were needed.

4.1 Flattening DBM

In the LTSmin implementation that we already have the state vector exists of all discrete variables and an 64 bit pointer to a C++ class containing a DBM [11]. For a symbolic solution this pointer has no meaning, thus we take the actual values from the DBM and put these into the state vector. This increases the length of a state vector, but does not need to increase the memory footprint, as the DBM was already stored. In the DBM library we use a DBM is represented by a one dimensional array of 32-bit integers. In the integers the complete bound is stored, so both the operator and the constant value. We flattened the DBMs to work with a symbolic solution. 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 $(0, \leq)$ internally. This reduces the number of state variables in the state vector by one for each clock. This flattening of DBMs results into a language module that can be connected to all LTSmin algorithmic back-ends for state-space generation.

4.2 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. For even better results, also the must-write matrix is needed. This needs effort when analysing the code, this can be done, but is left out for this thesis. 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, in such cases the variables that were found in the parsing of these functions are added to the read and may-write variables of the transition. 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 for this extrapolation, 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. In which only an other transition can take place. The correct filling of the matrices is only for the discrete parts of the states. For the zone part, to optimizations have been created. The matrices for these parts will always be filled. The problem is that changing only one clock can have a much larger impact on a DBM when a normal form is used. The flattened DBMs and the sparser dependency matrices together enable the reordering algorithms in the symbolic back-end of LTSmin to be used.

4.3 DBM reduction

We work towards a fully reduced DDD solution. This is already started at the language module size. The next-state function will only return tight and saturated paths. In DBM terms this is a minimal constraint system [5]. As the length of a state-vector can not be changed on the fly, all removed constraints are set to $(<, \infty)$. This means that there is no upper-bound on the variable pair of that position. In algoorithm 2 which uses algorithm 1 we show the algorithm that determines all bounds that are not needed an can be set to $(<, \infty)$. The DBM library can not use these minimal constraint systems. In the next state function the incoming DBM is tightened, then all needed operations for the successor generation are conducted and if a successor is returned, its DBM is again turned into a minimal constraint system. This will give algorithmic overhead for each next-state call. The advantage of this procedure is that many bounds will be redundant and turned into $(<, \infty)$. In the symbolic back-end these bounds which are the same can be shared in a single node. Thus taking more time in the successor generator, it can also reduce the number of nodes in the algorithmic back-end. This reduction is used in the successor generator for the symbolic back-end, and will also be used for the DDD solution.

Algorithm 1 Reduce

```
1: procedure REDUCE($dbm, dim$)
2:     for $i \in dim$ do
3:         for $j \in dim$ do
4:             for $k \in dim$ do
5:                 if !($dbm[i,k] \vee dbm[k,j] \vee dbm[i,j]$ on diagonal) then
6:                     if $dbm[i,k] + dbm[k,j] \leq dbm[i,j]$ then
7:                         $dbm[i,j] := \infty$
```

Algorithm 2 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 $\infty$ rest 0
7:     for $i \in dim$ do
8:         if $placed[i] = 0$ then
9:             for $j \in$ 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 $i \in cl$ do
16:        for $j \in dim$ do
17:            if $eq[i, j] = 1$ then
18:                $repr[i] := j$
19:                break
20:    $clg[cl, cl]$
21:    for $i \in cl$ do
22:        for $j \in cl$ do
23:            $clg[i, j] := dbm[repr[i], repr[j]]$
24:    Reduce($clg, cl$)
25:    for $i \in cl$ do
26:        for $j \in dim$ do
27:            if $eq[i, j] = 1$ then
28:                for $k \in dim$ do
29:                    if $eq[i, k]$ then
30:                        $newDBM[j, k] = dbm[j, k]$
31:        for $j \in cl$ do
32:            $newDBM[repr[i], repr[j]] := clg[i, j]$
33:    return newDBM
```

4.4 Connecting LDD and DDD

To represent the discrete variables in states LDD nodes are used. The structure of these nodes is quite similar to DDD nodes. We decided to not mix the nodes, but to first have all the LDD nodes and then all DDD nodes in the tree. In the state vector the first part exists of all discrete variables, the last part are the DBM variables. The top of the diagram can be seen as a MTLDD(Multi-Terminal List Decision Diagram) with not values on the leaf nodes, but pointers to DDD nodes. The DDD part is not influenced by the LDD part, as a node is only influenced by the nodes below it, it has no information about the nodes above it in the diagram. This strict separation between LDD and DDD nodes makes that the reordering algorithms can not be used. The lack of reordering makes it also possible to reconstruct the DBMs on the DDD side. This is used for the minus function which we discuss later.

4.5 DDD nodes

We used the basis of the LDD package in Sylvan to create our DDD nodes. The nodes are the same as the LDD nodes, only two previously unused bits are now used to store the operator and the type of the node. DDD nodes are stored in 128 bits, represented as a struct of two 64 bit integers. The hashtable that is already used by Sylvan is specifically for 128 bit entries, so the DDD nodes can use the same hashtable. A node is in C code represented as follows:

```
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. These values are not specifically named in the struct, all values are stored in the two integers a and b. Figure 10 shows how this is coded in memory. The type, operator and flag bit are stored in the black areas. We do not show them explicitly due to the scale. 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

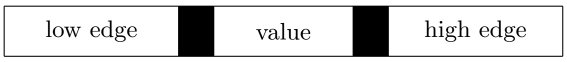


Figure 10: In memory representation of DDD node

by $< \infty$, which is true for every upper bound. For such nodes the false edge will always directly lead to the false end node.

4.6 Creating Nodes

To create a node a special MK function is used. This function will ensure that a DDD is always locally reduced. This MK function is shown in algorithm 3. This function ensures the correct total structure and puts newly created nodes in the hashtable. The actual creation of a node is done in the MakeNode function that is called inside the MK function. The code for the MakeNode function is not shown here as it is only technical coding, putting all the information in the struct.

Algorithm 3 MK

```
1: procedure MK($value, h, l, type, op$)
2:     if $h = 0 \wedge type = LDD$ then
3:         return $l$
4:     if $h = 1 \wedge l = 1$ then
5:         return 1
6:     if $h = 0 \wedge l = 0$ then
7:         return 0
8:     if $h = 0 \wedge l \neq 0$ then
9:         return 1
10:    if $h = high(l)$ then
11:        return $l$
12:    $node$ = MAKENODE($value, h, l, type, op$)
13:    if $node \notin table$ then
14:        PUT($node$)
15:    return $node$
```

4.7 Apply

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 [26] a general definition of the algorithm is given. We turned this more mathematical definition into an algorithm, we give pseudo-code in Algorithm 4. The algorithm will search down to the leaf nodes and use the operator on that level. We can optimize this a bit for cases where we see two equal nodes, or only one leaf node. In Algorithm 5 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 false leaf, or two equal nodes. The apply operator does not ensure path-reducedness, even when both inputs are path reduced.

Algorithm 4 Apply

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

4.8 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.

Algorithm 5 Union

```
procedure UNION($v1, v2$)
    if $v1 = v2$ then return $v1$
    else if $v1 =$ false then return $v2$
    else if $v2 =$ false then return $v1$
    else if $var(v1) \prec var(v2)$ then
        $high \leftarrow$ UNION($high(v1), v2$)
        $low \leftarrow$ UNION($low(v1), v2$)
        $result \leftarrow$ MK($cstr(v1), high, low$)
    else if $var(v2) \prec var(v1)$ then
        $high \leftarrow$ UNION($high(v2), v1$)
        $low \leftarrow$ UNION($low(v2), v1$)
        $result \leftarrow$ MK($cstr(v2), high, low$)
    else if $v1 \prec v2$ then
        $high \leftarrow$ UNION($high(v1), high(v2)$)
        $low \leftarrow$ UNION($low(v1), v2$)
        $result \leftarrow$ MK($cstr(v1), high, low$)
    else if $v2 \prec v1$ then
        $high \leftarrow$ UNION($high(v1), high(v2)$)
        $low \leftarrow$ UNION($v1, low(v2)$)
        $result \leftarrow$ MK($cstr(v2), high, low$)
    else if $v1 = v2$ then
        $high(v1) \leftarrow$ UNION($high(v1), high(v2)$)
        $low(v1) \leftarrow$ UNION($low(v1), low(v2)$)
        $result \leftarrow$ MK($cstr(v1), high, low$)
    return $result$
```

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 11 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, as shown in figure 12. 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, figure 13. 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 11. 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 and tightened. 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 made into a single structure using the union function.

4.9 Relation

The transition relation we use is stored in an LDD structure. Both bound values and operators are implicitly encoded in a single value, like in the DBM library. When creating new nodes, the nodes are matched against the state space. By checking the type of the node on the current level it can be checked if the relation node should be treated as a normal LDD node with a discrete variable, or as an LDD node which implicitly stores an upper-bound. The choice to not use the DDD type nodes in the relation has been made to have better support for possible future reordering options. If reorderings are used, it would need explicit information for which relation levels contain zone variables, with matching against the states this extra information is not needed.

4.10 BFS

The DBM minus function we use is quite expensive. As it is imported from a library we do not know the exact complexity. 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 6 we show the standard BFS algorithm, this will be the first algorithm we use. Algorithm 7 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 been changed. This check is basically the same, the first checks if now new states are found, the second checks that the total state-space has not been changed. This change now shows that the minus is not necessary any more, as shown in algorithm 8. This version uses the same check as the previous one, but now the minus of the current and the visited set has been removed. 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. We have implemented these two versions in the bfs-prev algorithm [21]. This is the default search algorithm that is used in LTSmin. In the results section we will show the outcome of both BFS algorithms.

Algorithm 6 BFS

```
1: procedure BFS($initial$)
2:     $vis := cur := initial$
3:     while $cur \neq \emptyset$ do
4:         $cur := next(cur)$
5:         $vis := vis \cup cur$
6:         $cur := cur \setminus vis$
```

Algorithm 7 BFS

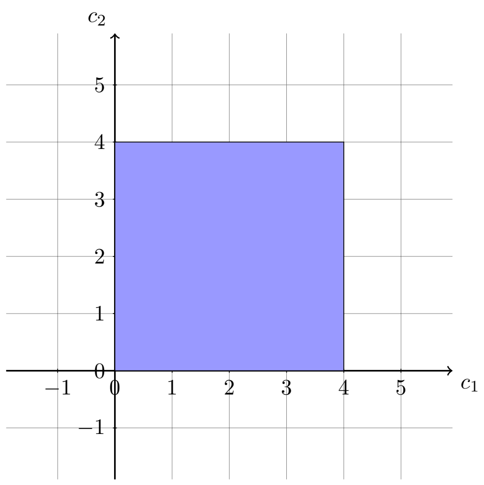
```
1: procedure BFS($initial$)
2:     $vis := cur := initial$
3:     $vis_{prev} := \emptyset$
4:     while $vis \neq vis_{prev}$ do
5:         $vis_{prev} := vis$
6:         $cur := next(cur)$
7:         $vis := vis \cup cur$
8:         $cur := cur \setminus vis$
```

Algorithm 8 BFS

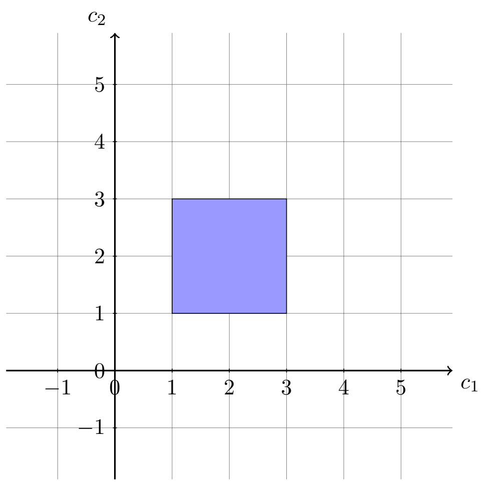
```
1: procedure BFS($initial$)
2:     $vis := cur := initial$
3:     $vis_{prev} := \emptyset$
4:     while $vis \neq vis_{prev}$ do
5:         $vis_{prev} := vis$
6:         $cur := next(cur)$
7:         $vis := vis \cup cur$
```

4.11 State-space count

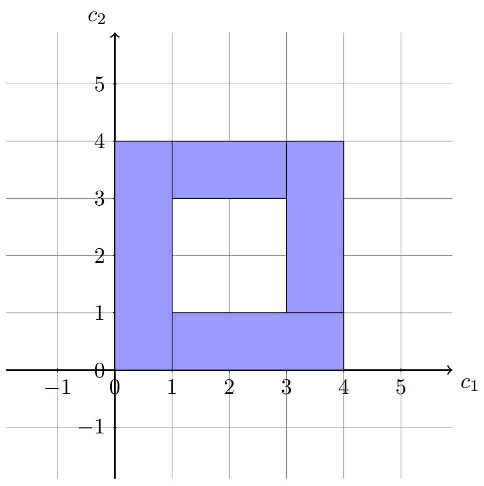
One of the basic outputs that LTSmin gives when calculating a state-space, is the number of states. For timed automata this is not trivial, as a state is not well defined. Systems with digitization will have other states than systems which use zones for representing time. Even for zones no clear definition of a state exists, as DBMs give no canonical representation of zones, when they are not convex. Now our representation with DDDs will again give another result. We decided to take as the state count only the number of discrete states. This number should be equal for each method for analysing timed automata.



(a) Minuend



(b) Subtrahend



(c) Difference

Figure 11: Minus complexity example

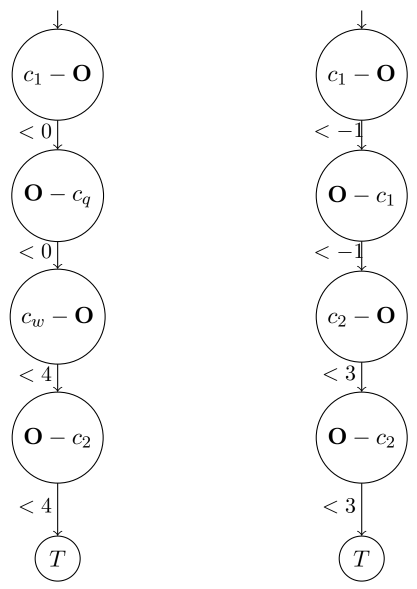


Figure 12: DDD representation of the minuend and subtrahend of figure 11

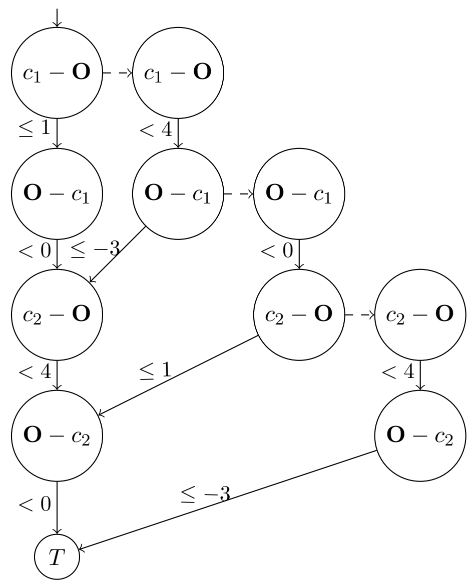


Figure 13: DDD representation of the difference of figure 11

5 Notes

5.1 Successor Generator

The language module uses the opaal successor generator for Uppaal models. This generator is written in Python and reads Uppaal XML files. A C++ file is generated from this. These files are compiled to object files which can be dynamically linked to LTSmin. The structure of the next-state function is slightly different from [11]. The new structure can be found in algorithm 9. Al line 6, the function iterates over all outgoing transitions from the current location. If it is an internal transition the successor will be generated on lines 9-18. If it is a sending transition, receivers will be searched for on lines 20-32. In the generated C++ code the loops on lines 5 and 21 are unrolled. The algorithm contains several empty checks, on lines 8, 13, 23 and 27. After each addition of constraints the DBM can possibly be empty. If the DBM is at one of these points empty, no point in time exists where the new state can exist, so further exploration of the transition is not needed. After the empty checks on lines 13 and 27 the extrapolation and the reduction are done. These operations can not empty the DBM, the extrapolation can make the zone larger, not smaller. The reduction will not change the zone at all, only its representation. If the DBM is not empty before these operations it can safely be put into the output.

5.2 Time Extrapolation

In the successor generator step a time extrapolation is used. This extrapolation step reduces the number of DBMs created and makes sure that this number is finite. The most coarse abstraction as described in [3] is used. This extrapolation reduces the number of zones that are explored significantly. It also makes that less improvements can be made on the representation of the zones, for some models all states are extrapolated to the same zone, so nothing interesting happens at the timed side of the model any more. In opaal this algorithm is implemented in such a way that all Uppaal locations are always read. The maximum extrapolation is based on the values of these locations. Only if there is no difference between all values for a certain location, it is not needed to read this. This results into an densely populated dependency matrix.

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

Algorithm 9 Next-State

```
1: procedure NEXT-STATE($s_{in} = \{l_1, ...l_n, l_{n+1}, ..., l_m\}$)
2:     $out\_states := \emptyset$
3:     $D$ :=CREATEDBM($\{l_{n+1}, ..., l_m\}$)
4:     TIGHTENDBM($D$)
5:     for $l_i \in l_1, ..., l_n$ do
6:         for all $l_i \xrightarrow{g,a,r} l'_i$ do
7:             $D' := D \cap g$
8:             if $D' \neq \emptyset$ then
9:                 if $a = \tau$ then
10:                    $D' := D'[r]$
11:                    $D' := D' \uparrow$
12:                    $D' := D' \cap I^i_C(l'_i) \cap \bigcap_{k \neq i} I^k_C(l_k)$
13:                    if $D' \neq \emptyset$ then
14:                        $D' := D'/B(l_1, ..., l'_i, ..., l_n)$
15:                        REDUCEZERO($D'$)
16:                        $\{l'_{n+1}, ..., l'_m\}$ :=FLATTENDBM($D'$)
17:                        $s_{out} := \{l_1, ..., l'_i, ..., l_n, l'_{n+1}, ..., l'_m\}$
18:                        $out\_states := out\_states \cup s_{out}$
19:                else
20:                    if $a = ch!$ then
21:                        for $l_j \in l_1, ..., l_n, j \neq i$ do
22:                            for all $l_j \xrightarrow{g_j, ch?, r_j} l'_j$ do
23:                                if $D''" = D' \cap g_j \neq \emptyset$ then
24:                                    $D'' := D''[r][r_j]$
25:                                    $D'' := D'' \uparrow$
26:                                    $D'' := D'' \cap I^i_C(l'_i) \cap I^j_C(l'_j) \cap \bigcap_{k \neq \{i,j\}} I^k_C(l_k)$
27:                                    if $D'' \neq \emptyset$ then
28:                                        $D'' := D''/B(l_1, ..., l'_i, ..., l'_j, ..., l_n)$
29:                                        REDUCEZERO($D''$)
30:                                        $\{l'_{n+1}, ..., l'_m\}$ :=FLATTENDBM($D'$)
31:                                        $s_{out}$ :=
           $\{l_1, ..., l'_i, ..., l'_j, ..., l_n, l'_{n+1}, ..., l'_m\}$
32:                                        $out\_states := out\_states \cup s_{out}$
33:     return $out\_states$
```

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 Correctness

The DDD state space generator needs to be checked for correctness to say anything about the results. We only checked for partial correctness by comparing discrete states. Counting the discrete state-space can be done by counting the number of paths until the first DDD level in the diagram. These numbers were compared to the discrete state space in the LDD solution without reordering, here the discrete state-space can also be determined by counting paths until the first level representing zones. We can not directly compare state-spaces to Uppaal, different representations of the timing part of the state-space can give different numbers.

6 Benchmarks

Below we describe the different models we used to run the benchmarks. We tried to find models that scale up for a number of nodes or processes, so that we can also check the behaviour of our approaches for different sizes of the same model. In this section we use the terminology 'locations' and other 'discrete variables'. The definition of timed automata does not have this difference, but we use it to describe models, because the time extrapolation is dependent on locations, and not on the other discrete variables. This dependency fills a large part of the dependency matrices. In this section, a location is a state in the Uppaal transition system editor, the other discrete variables are declared in the C-like syntax that Uppaal uses.

6.1 Viking

The set of Viking tests, models the classical Viking and bridge problem. It models 4 Vikings at a dark bridge, they only carry one torch. The torch is only strong enough to give light for 2 Vikings. All Vikings have different walking speeds, a faster Viking will have to adapt to a slower one, when crossing the bridge together. The walking speed of the Vikings is modelled by time constraints on the action of letting go of the torch. The model has a low number of discrete variables, one per Viking, one for the torch and an indicator for the side of the bridge on which the torch is. It has a global clock and a clock per Viking. The standard version of this problem has 4 Vikings. This can however be generalized to n Vikings.

The model results in a densely filled dependency matrix. The torch and all Viking variables are always read for the time extrapolation. Only the side indicator is not read always. The write matrix is sparser.

The difference between the LDD and DDD representation is quite small for this model. In the extrapolation step all clock zones are set to $[0..\infty]$ for all states, so in both diagrams the zones are represented by a single path. So the interesting things are only happening in the discrete parts.

6.2 Fischer

Fischers mutual exclusion protocol [18] is modelled for a number of processes. There is no synchronization between processes, only blocking of actions can occur. This model has a slightly higher number of discrete variables compared to the Viking tests. Each process has a location and an id. The model also has 2 global discrete variables. Each process has a local clock, no global clock is used.

The dependency matrix of this model has some sparse rows, as each model has an id, which is a constant and can only be read. Again all the location variables are always read due to the time extrapolation.

6.3 CSMA-CD

The Carrier Sense Multiple Access/Collision Detection [32] is modelled for a number of senders. The model has a few discrete variables. Each sender only has a location and only one global counter is used. Each sender and the bus have a local clock. The model uses a lot of synchronizations between the senders and the bus.

6.4 Animo

We could not use the ANIMO models, only the smallest model with no synchronizations was possible. As we started the project to work on ANIMO models, we still included that single model in the benchmark set. It is a model with only one node, so only one location variable. The model has two clocks, a global clock and a clock for the node. Further it does have quite a large number of discrete variables. Both the global declaration and the node have a portion of c-like code with a number of global variables.

This results in a model with a quite sparse dependency matrix, as only the single location is used for the time extrapolation. We expected this model to have good performance for the LDD method with variable reordering.

6.5 Lynch-Shavit

The Lynch-Shavit mutual exclusion protcol [20] is modelled for different number of processes. The structure of the model is quite like the Fischer model. It only uses one global variable more than Fischer.

6.6 Milner

Milners scheduler [22] is modelled for a number of nodes. The structure is like that of the CSMA-CD model, except that it does not use a bus. The model has a lot of synchronizations between the nodes, an between the node and a global process. Each node has two clocks, so the zone representation blows up quickly.

6.7 Other models

We also used some models that we could not scale up enough due to memory/time limitations, or that could not scale up due to the nature of the model. We will not describe those models into detail. These models were the critRegion, Critical, bocdp(-fixed) [16], bando and timelock model.

6.8 Benchmark Runs

We ran benchmarks with the different solutions we described to compare them to each other. The DDD solution has been ran with the two BFS algorithms as explained in section 4.10. For the LDD solution we only used the original BFS algorithm. We ran this without reordering and with some of the reordering algorithms that LTSmin provides. We used the options gsa, rb4w, cw, rs,rn, rs,ru. These results are compared to the explicit-state multi-core LTSmin and the original Uppaal. All solutions are ran with one thread. The LDD and explicit-state multi-core solutions can be ran with multiple treads. The DDD solution does not support this, so for comparison reasons all methods are used in single-core mode.

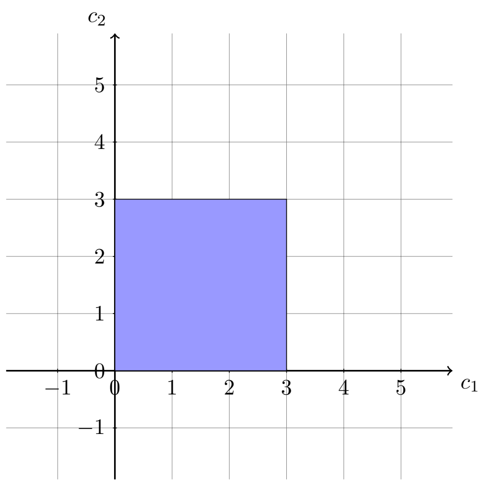
7 Results

There is a difference between the number of nodes for the normal BFS and the BFS without minus. This is possible because we do not use a canonical form of DDDs. Most results show a higher number of nodes for the runs with the minus. In figure 14 we show an example of how this can happen. We assume all zones in the figures belong to the same set of locations. In figure 14a we have the zone that is already visited. Now a new state with the zone in figure 14b is discovered. If the minus is not used, successors of this state are directly generated from the set of locations and this zone. If the minus is used the first zone will first be subtracted before successors are generated. The result of the subtraction is shown in figure 14c. This is not a convex zone, so a DDD with multiple paths is needed. From this state also other successors can be generated, possibly needing more nodes to be represented. If the newly generated states are then unioned with the visited set the result can again have more nodes than the version without minus. The less fractionated zones in the current set can also have implications on the time results, as less work in the next-state function is needed. On the other hand the next-state function can also need extra time, as some states would otherwise have completely been removed from the current set, and no work for that states would need to be done.

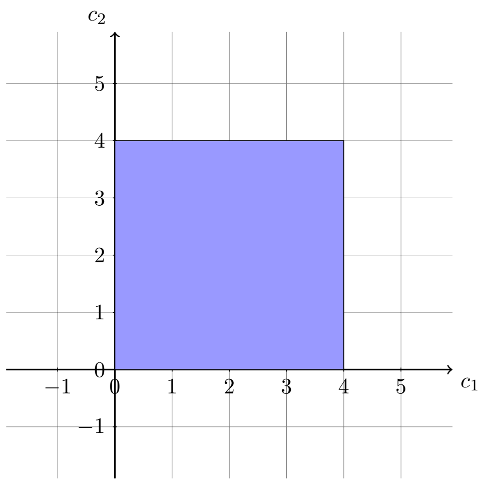
In the LDD solution the standard setup with no reordering is for most models faster and uses less nodes than with the reordering algorithms. This is probably due to the densely filled dependency matrices as described in section 4.2. If we could make these matrices sparser we expect better results from the reordering algorithms.

	DDD		LDD					
		no-min		gsa	rb4w	cw	rs,rn	rs,ru
fischer1	0.8	0.8	0.8	1.1	0.8	0.8	0.8	0.8
fischer2	0.9	0.9	0.9	1.5	0.9	0.9	0.9	0.9
fischer3	0.9	0.9	0.9	2.0	0.9	0.9	0.9	0.9
fischer4	1.2	1.2	1.2	2.9	1.3	1.2	1.2	1.2
fischer5	14.1	10.2	6.1	9.3	7.3	6.5	6.1	5.9
critRegion1	0.9	0.9	0.9	1.2	0.9	0.9	0.8	0.8
critRegion2	0.9	0.9	0.9	1.5	0.9	0.9	0.9	0.9
critRegion3	3.7	3.6	1.8	2.8	1.9	1.8	1.7	2.2
Critical_01-25-50	0.9	0.9	0.9	1.3	0.9	0.9	0.9	0.9
Critical_02-25-50	0.9	1.0	1.0	1.6	1.0	1.0	0.9	1.0
Critical_03-25-50	13.8	13.8	7.2	9.9	9.3	7.4	6.9	11.6
CSMACD_01	0.8	0.8	0.8	1.0	0.8	0.8	0.8	0.8
CSMACD_02	1.0	1.0	1.0	1.3	1.0	1.0	1.0	1.0
CSMACD_03	1.0	1.0	1.0	1.5	1.0	1.0	1.0	1.0
CSMACD_04	1.2	1.2	1.2	1.7	1.2	1.1	1.2	1.1
CSMACD_05	1.9	1.4	1.5	2.1	1.5	1.5	1.4	1.4
CSMACD_06	5.0	2.0	2.6	3.2	2.6	2.6	2.4	2.4
CSMACD_07	19.7	3.7	7.2	7.6	7.4	7.4	6.5	6.4
CSMACD_08	237.0	10.4	26.7	25.5	28.0	28.0	23.5	22.8
viking1	0.8	0.8	0.8	1.1	0.8	0.8	0.8	0.8
viking2	0.9	0.9	0.9	1.3	0.9	0.9	0.9	0.9
viking3	0.9	0.9	0.9	1.5	0.9	0.9	0.9	0.9
viking4	1.0	1.0	1.0	1.7	1.0	1.0	1.0	1.0
viking5	1.1	1.1	1.1	2.0	1.1	1.1	1.1	1.1
viking6	1.7	1.7	2.0	2.9	2.0	2.0	1.8	1.7
viking7	2.2	2.2	2.5	3.6	2.5	2.5	2.2	2.1
viking8	4.7	4.7	5.8	6.4	5.9	5.9	4.9	4.4
viking9	12.1	12.2	16.2	14.9	16.4	16.3	13.1	11.5
viking10	33.9	34.0	46.9	38.9	47.2	47.3	37.1	31.5
Lynch1-16	0.8	0.8	0.8	1.4	0.8	0.8	0.8	0.8
Lynch2-16	0.9	0.9	0.9	2.2	0.9	0.9	0.9	0.9
Lynch3-16	1.3	1.3	1.2	3.3	1.3	1.2	1.2	1.2
Lynch4-16	8.8	8.6	8.4	11.3	10.0	8.6	8.2	8.0
bocdp	1.6	1.6	1.6	14.6	1.6	1.6	1.6	1.6
bocdpFIXED	1.6	1.6	1.6	13.5	1.6	1.6	1.6	1.5
bando	1.6	1.6	1.5	13.5	1.6	1.6	1.5	1.5
timelock	0.7	0.7	0.7	0.9	0.7	0.7	0.7	0.7
Milner-2Nodes-flat	0.9	0.9	0.9	1.3	0.9	0.9	0.9	0.9
Milner-3Nodes-flat	1.0	0.9	1.0	1.6	1.0	1.0	1.0	1.0
Milner-4Nodes-flat	1.1	1.0	1.6	2.2	1.6	1.6	1.5	1.5
Milner-5Nodes-flat	1.2	1.2	2.1	2.9	2.1	2.1	2.1	2.0
Milner-6Nodes-flat	1.5	1.4	3.0	3.8	3.0	3.0	2.8	2.7
Milner-7Nodes-flat	1.8	1.7	4.2	5.2	4.3	4.3	4.0	3.8
Milner-8Nodes-flat	2.3	2.2	6.1	7.0	6.2	6.2	5.5	5.2
hddi_input_1	0.9	0.9	0.9	1.0	0.9	0.9	0.9	0.9
hddi_input_2	1.0	0.9	0.9	1.2	0.9	0.9	0.9	0.9
hddi_input_3	179.4	1.0	1.1	1.4	1.1	1.1	1.1	1.1
ANIMO_small	1.1	1.1	1.1	1.8	1.1	1.1	1.1	1.1

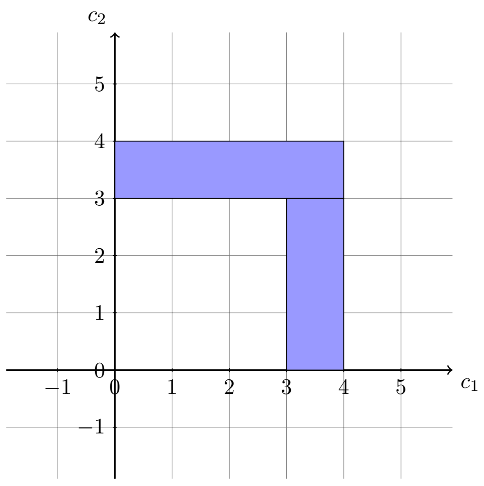
	DDD		LDD					
		no-minus		gsa	rb4w	cw	rs,rn	rs,ru
fischer1	14	14	14	13	14	13	14	14
fischer2	66	66	66	68	63	66	66	66
fischer3	509	288	409	505	433	532	409	409
fischer4	5025	1300	2541	3905	3190	4184	2541	2541
fischer5	49634	5535	17131	30665	26004	32446	17131	17131
critRegion1	24	24	24	24	20	26	24	24
critRegion2	251	190	243	358	296	242	243	243
critRegion3	4643	3825	3743	5789	5506	4627	3743	3743
Critical_01-25-50	25	25	25	24	23	29	25	25
Critical_02-25-50	313	262	316	500	427	370	316	316
Critical_03-25-50	12322	11183	17505	29297	28443	20293	17505	17505
CSMACD_01	17	17	17	17	17	17	17	17
CSMACD_02	112	108	99	101	101	101	99	99
CSMACD_03	686	458	500	553	551	551	500	500
CSMACD_04	3305	1356	2205	2528	2520	2520	2205	2205
CSMACD_05	13867	3478	8634	10154	10127	10127	8634	8634
CSMACD_06	51633	7925	30862	37022	36938	36938	30862	30862
CSMACD_07	176965	17069	102821	125264	125019	125019	102821	102821
CSMACD_08	569760	36098	324047	329844	398899	398899	324047	324047
viking1	12	15	15	15	24	24	15	15
viking2	37	37	37	37	66	66	37	37
viking3	86	86	86	91	176	176	86	86
viking4	105	105	105	111	196	196	105	105
viking5	124	124	124	131	216	216	124	124
viking6	233	233	233	240	504	504	233	233
viking7	190	190	190	199	342	342	190	190
viking8	224	224	224	235	415	415	224	224
viking9	263	263	263	276	495	495	263	263
viking10	304	304	304	317	581	581	304	304
Lynch1-16	24	24	24	22	27	21	24	24
Lynch2-16	162	162	173	185	217	187	173	173
Lynch3-16	1175	922	1277	1757	2600	1649	1277	1277
Lynch4-16	14280	8246	11113	22033	32144	17146	11113	11113
bocdp	541	541	572	329	435	517	572	572
bocdpFIXED	542	542	572	428	448	514	572	572
bando	542	542	572	428	448	514	572	572
timelock	4	4	4	4	4	4	4	4
Milner-2Nodes-flat	442	245	338	327	394	338	338	338
Milner-3Nodes-flat	2709	918	1602	1571	1702	1602	1602	1602
Milner-4Nodes-flat	4999	2968	4834	4789	4997	4834	4834	4834
Milner-5Nodes-flat	9106	5293	8653	8596	8946	8653	8653	8653
Milner-6Nodes-flat	17008	7755	14100	14018	14551	14100	14100	14100
Milner-7Nodes-flat	25493	12188	21455	21343	22098	21455	21455	21455
Milner-8Nodes-flat	39887	16324	31008	30884	31874	31008	31008	31008
hddi_input_1	221	119	130	134	134	134	130	130
hddi_input_2	2735	693	1021	1025	1090	1023	1021	1021
hddi_input_3	20485	2013	3675	3675	3971	3675	3675	3675
ANIMO_small	235	235	197	191	405	185	197	197



(a) Visited Zone



(b) Current Zone



(c) After Minus

Figure 14: Minus fragmentation

8 Different Semantics

We chose in our implementation to take no information from the low edges of nodes. A node only represents an upper-bound, a false edge does not implicitly represent a lower bound. This is a design choice we made to be able to switch efficiently from the DBM representation in the language module to the DDD representation. We could however also have used a semantics where the low edges do represent a lower-bound. We did not implement this, but this section will discuss this other semantics.

Definition 14. *The semantics of a vertex is defined recursively by the function $\mathcal{V} : V \to$ **Exp** :*

- $\mathcal{V}[[0]] \stackrel{\text{def}}{=} \mathit{false},$
- $\mathcal{V}[[1]] \stackrel{\text{def}}{=} \mathit{true},$
- $\mathcal{V}[[v]] \stackrel{\text{def}}{=} \begin{cases} (pos(v) - neg(v) < const(v)) \to \mathcal{V}[[high(v)]], \mathcal{V}[[low(v)]] \mathit{if}\ op(v) =' <' \\ (pos(v) - neg(v) \leq const(v)) \to \mathcal{V}[[high(v)]], \mathcal{V}[[low(v)]] \mathit{if}\ op(v) =' \leq' \end{cases}$

The semantics are almost equal to the one in definition 4, the difference is in the interpretation of the low edge. In this semantics the low edge does not just represent that the upper-bound is higher than the bound of the node, but the actual value of the variable is higher than the bound of the node.

8.1 DBM Translation

The translation from a single DBM to a DDD will not change. The translation from multiple DBMs will change neither, as that can be done as a union of DBMs which are individually translated to a DDD. The other way around, from a DDD back to a DBM becomes more complicated. For a DDD with a single path to true nothing will change. For paths that go down some low edges the translation will change. The falsification of an upper-bound, leading to a lower-bound, or a upper-bound of the inverse pair, can overrule the upper-bound of an other node. We give an example in figure 15. In this example all nodes that are not in the path we consider are hidden. The DDD will have more nodes to reach this representation. In figure 16 we have a DBM for both interpretations. In figure 16a we have the DBM as we use the interpretation from our implementation. In figure 16b the DBM of the other interpretation is shown. The difference between the two DBMs is on the position $c_2 - O$). The information from the low edge of the $O - c_2$ node has overruled the information of the high edge of the $c_2 - O$ node. Using a canonical form of a DDD can also overcome this problem.

To make the translation from DDD to DBM correctly the relative positions of the upper- and lower-bound of each pair of variables need to be

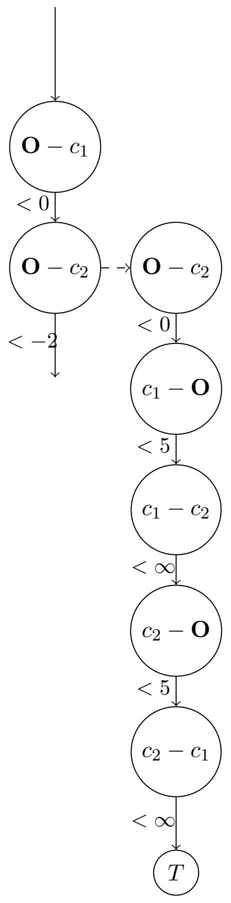


Figure 15: Implicit bound DDD

known. Also a function to determine the stronger bound of a pair needs to be created. Lastly the bounds need to be changed correctly. A $<$ sign changes into a $\leq$ and vice versa, the constant is multiplied by -1. We give an example of this change:

$$\begin{gathered} c_1 - c_2 \not< 3 \\ \Updownarrow \\ c_1 - c_2 \geq 3 \\ \Updownarrow \\ c_2 - c_1 \leq -3 \end{gathered}$$

A similar translation will have to be conducted in the relprod function. This function does not explicitly need the DBMs. The relations that are used are however created in the language module which uses DBMs. In the current implementation, a path in the state space needs to be found that

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

(a) Original semantics

$$\begin{array}{c|ccc} & \mathbf{O} & c_1 & c_2 \\ \hline \mathbf{O} & (0,\leq) & (0,<) & (0,<) \\ c_1 & (5,<) & (0,\leq) & (\infty,<) \\ c_2 & (2,\leq) & (\infty,<) & (0,\leq) \end{array}$$

(b) New semantics

Figure 16: DBM's of two different DDD interpretations

has on each level the same high edges as the relation. Which low edges are traversed on the way is not important. Now this information is taken into account some changes will have to be made. A simple path in the relation, might need some false edges in the state-space to get all the correct bounds.

8.2 Minus

Implementation of the minus function will become easier in DDDs, no coupling to the DBM library will be needed any more. First of all we will give the complement function. We give the pseudocode for this function in algorithm 10. The algorithm switches all 0 and 1 nodes. This will have a running time of $O(n)$ where n is the number of nodes in the tree. Our current implementation does not skip levels in the DDD towards a 1 node. This can happen in this complement function. This can be solved by filling the gap that is created with nodes with $(<, \infty)$ as bound. Another solution would be to allow this behaviour, this would need some extra work when creating state-vectors out of a diagram.

Algorithm 10 Complement

```
1: procedure COMPLEMENT($a$)
2:     if $a = 0$ then
3:         return 1
4:     if $a = 1$ then
5:         return 0
6:     $h :=$ COMPLEMENT($high(a)$)
7:     $l :=$ COMPLEMENT($low(a)$)
8:     return MK($bound(a), h, l$)
```

With this function we can create a minus function, as for set theory, minus can be defined as $A \backslash B = A \cap \overline{B}$. Now we can build the minus function from the complement and intersection function as shown in algorithm 11. This algorithm is probably less complex than the DBM minus we currently use. We do not know the exact complexity of the DBM minus algorithm, so we cannot call this certain.

Algorithm 11 Minus

```
procedure MINUS($a, b$)
    if $a = 0$ then
        return 0
    if $b = 0$ then
        return 1
    $notB$ =COMPLEMENT($b$)
    $result$ = INTERSECTION($a, notB$)
    return $result$
```

9 Future Work

In this section we discuss improvements that can be made for better results. In the previous section we already discussed the possibility of different semantics. This is also future work, but is written in a separate section.

9.1 Canonization

The DDD package does not use any canonical form. This means that some operations like equality and emptiness become less trivial. They can however still be done. The diagrams are ordered and locally reduced. The resulting state-vectors that the language module produces are also path-reduced. Most operators do not preserve this path-reducedness, so most diagrams will not be path-reduced.

We can implement two types of reduced DDDs. A DDD that is only path-reduced can be called semi-canonical [26]. This means that a tautology and a unsatisfiable expression can only be represented by a true or false node. This will make the checking for an empty DDD trivial, the DDD is only empty if the top node is a false node. We also defined full reducedness as a DDD that is tight and saturated, and has no disjunctive vertices. This fully reduced version is assumed to be canonical. A canonical DDD will change the equality test in a simple pointer comparison of the top nodes. Several algorithms to reach a reduced form are known [25].

The canonical forms are not needed at all times, only for some functions that need the specific form. Therefore we can choose to not have a canonical form at all times. One can choose to canonize the DDD after each operation, or to do this only before operations that actually need this form. The first option will have much canonization calls, where the second option will have less. The first option however, might have a DDD that is in all cases closer to the canonical form, so canonization might take less time. The semi-canonical form can also be used for emptiness checks, as the fully reduced diagram is not needed there. To get optimal results we need to find out what is the best option.

9.2 Reordering

The current implementation is not compatible with the reordering algorithms. All algorithms will probably have to be changed somehow. In the current implementation it is assumed that on the top there is a set of LDD nodes, and from a certain level only DDD nodes exist. With reordering this could be mixed, so algorithms can not rely on this any more. A special case will again be the minus function. It is now done by recreating DBMs from the DDD. This can be done, as the nodes are ordered in the same way as the DBM. When reorderings are used this is not trivial any more. It will need to be explicitly stored which variable is on which level. For the different semantics that we introduced in section 8, a similar problem will occur. We suggested a minus function using the complement. For zones the complement is well defined, as there is a ∞ value representing the most upper- and lower-bounds of possible values. For discrete variables this is not directly clear.

Another option for reordering, which will probably solve some of the problems with the minus function would be reordering, but keeping the discrete and the zone parts separated. The discrete part could use the normal reordering algorithms. As the matrices for the zone variables are completely filled, the reordering algorithms can not do something useful on that level. Here experiments with manual reorderings can be tried. Now the standard ordering of the DBMs is used. It might be that having both bounds on a pair of clocks together gives better results, or maybe even other orderings.

9.3 Sparser Dependency Matrix

The dependency matrices are densely filled. We already discussed the problems in section 4.2. There are some solutions that can improve this. Smaller transition groups can be created, maybe even splitting the discrete part and the timed part of a transition. This would also make that the zone part of the matrix would not need to be completely filled. The discrete transition would only need access to the zone parts on which bounds are calculated. The timed transitions would need all zone variables, but can leave discrete variables out which do not imply clock bounds. Another option that needs more work, is also filling the may-write matrices. The current code parsing that generates the matrices is not powerful enough to make a difference between may- and must-write variables. On this level also improvements can be made. The parts of the matrices for the zone variables are always filled, as the change of a single clock can have an impact on much of the DBM. We did not check however if an analysis can be done that finds fields which are not changed, or do not need to be read in a transition. A better analysis of the changes in DBMs can lead to sparser matrices on the zone variable side. The final improvement can be made for arrays. If the current implementation sees that a field from an array is read or written, then all fields in the array get a read or write dependency. It should be possible to only have dependencies for the fields that are actually read or written.

9.4 Multi-Core

The DDD library is built in the Sylvan framework which allows for multi-threaded decision diagrams. The DDD library is not suited for multi-threading however. For the most operations this will only need some small adaptations. The biggest problem is in the minus operation. This uses the DBM library. This part is not completely thread-safe. We expect this problem to be in the coupling between the DDD and the DBM library, in the DBM part no objects can be shared between threads. We expect that making the DDD part suitable for multi-threading will give much better time results.

9.5 Animo Model Compatibility

The project started to find a solution to model-check ANIMO models. This part has not succeeded. ANIMO models use a Uppaal feature that is not supported by opaal, using clock bounds on input channels. The problem why this can not be fixed directly is in the unrolling of the transitions in the next-state function. Adding the clock constraints on any of the input channels can lead to an empty DBM, in such cases the transition would not be returned. The semantics would however create the transitions, but not synchronize with the location leading to the empty DBM. To ensure that in such cases all possible transitions that can happen will be returned, a unroll of all possible combinations of synchronizing transitions would be needed. This will need a redesign of that part of the successor generator. If this functionality is added to opaal, all ANIMO models should be compatible with opaal, and thus our symbolic solution.

9.6 Subsumption

The subsumption check that is included in the multi-core explicit-state backend in LTSmin is not implemented in the DDD library. This can be implemented as a DDD operation, with the implication operator and the apply function. A check $a \subseteq b$ will result in true if $b \implies a$ returns true. If a canonical form is used as well, the result will be only a true node, or a single path of $(\infty, <)$ nodes, depending on the possibility of skipping levels. This can limit the number of states added to the current set in the state algorithm, thus reducing the number of next-state calls needed. The most obvious subsumption check would be the check that a newly discovered zone is subsumed by the already visited state-space. It can however also be turned around, check if the visited state-space is subsumed by the newly discovered zone. In such a case the zone in the state space can be replaced by this new zone, such that the union function is not needed, this will not reduce the next-state calls however.

9.7 Checking Properties

The model-checker that we have created is only suited for state-space generation. It is not suited for property checking. One extra function is needed to use the LTSmin mu-calculus checker, which can also check CTL* formulas. The DDD library needs to be extended with a relprev function, which returns the predecessors given a set of states and a relation. This will only result in a discrete model-checker. LTSmin is not suited for timing properties. Some timing properties can be checked by extending the model with an extra automaton.

10 Conclusions

Conclusions

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, Patricia Bouyer, Kim G. Larsen, and Radek Pelánek. *Lower and Upper Bounds in Zone Based Abstractions of Timed Automata*, pages 312–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[4] Gerd Behrmann, Alexandre David, and Kim G. 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 2–13, Dec 1997.

[17] 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.

[18] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, January 1987.

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

[20] N. Lynch and N. Shavit. Timing based mutual exclusion. In *Proc. of the Annual Real-Time Symposium (RTSS)*, pages 2–11, 1992.

[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.

[22] R. Milner. *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[23] 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.

[24] 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.

[25] Jesper Møller and Jakob Lichtenberg. Difference decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, aug 1998.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.