Clock Matrix Diagrams

Bachelor's Thesis

Daniel Fass

daniel@react.cs.uni-sb.de

Reactive Systems Group
Department of Computer Science
Universität des Saarlandes
Saarbrücken, Juni 2009

Supervisor
Prof. Dr. Bernd Finkbeiner

Advisor
Dipl. Inf. Hans-Jörg Peter

Reviewers
Prof. Dr. Bernd Finkbeiner
Prof. Dr. Reinhard Wilhelm

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, June 2009

Abstract

The success of the timed model checker Uppaal heavily relies on the use of Difference Bound Matrices (DBM) as the main data structure for storing reachability information symbolically. However, since DBM are only capable of representing convex sets, they are inappropriate for applications such as game solving where non-convex sets may arise.

There are currently two independent approaches to tackle this problem: Clock Federations and Clock Difference Diagrams. While Clock Federations are based on linear lists of DBM, Clock Difference Diagrams are BDD-like tree structures where each node refers to a single difference constraint. Despite the space-efficiency of difference diagrams, concerning running time they are often outperformed by federations which in turn benefit from the convex efficiency of DBM.

In this thesis, we will present Clock Matrix Diagrams, a novel data structure that combines the space-efficiency of decision diagrams with the convex efficiency of DBM. We show that our data structure is a generalisation of DBM, Clock Federations, and Clock Difference Diagrams, which makes it universally usable.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Today, the complexity of the design of modern systems have increased drastically. They consist of distributed components which communicate with each other. The control structures of the models grow exponentially with their size. The timing components of the models are defined in the dense time domain.

A promising approach for automatically establishing the correctness of real-time systems is Model Checking [2], where during an exhaustive state space exploration the satisfaction of a given specification is ensured. The dense time domain (hence the term real-time) in conjunction with a distributed system description implies an inherent state-space explosion which in turn calls for efficient symbolic data structures. In this context, a desirable data structure should be both time- and space-efficient. Popular applications for model checkers are, e.g., the verification of embedded controllers and network protocols.

In this thesis, we present a novel data structure for a symbolic real-time state space representation that combines the strengths of the already established data structures, Clock Difference Diagrams and Difference Bound Matrices.

The rest of the thesis is structured as follows. Chapter 2 describes Timed Automata and a first approach for a symbolic representation of the infinite state space. Then in Chapter 3 Difference Bound Matrices, a data structure for representing clock zones, are described. Then we present our novel data structure Clock Matrix Diagrams. In the end, we show experimental results that show the efficiency of our data structure and give an outlook.

1.2 Related work and contribution

Various ways of adding time constraints to system models have been proposed, e.g., Timed Petri Nets [15], Timed Transition Systems [13, 9], timed I/O automata [12], and Modecharts [10]. But the most successful model that adds timing constraints is the model of Timed Automata [3] proposed by Alur and Dill in 1994. In contrast to the other models, in Timed Automata the time that is needed to traverse a whole path through an automaton can be expressed while in the other models, only the time between the successive transitions can be expressed. Timed Automata are used for modeling real-time systems in the state of the art model checker Uppaal [4].

The efficiency of a timed model checker strongly depends on the way how the timing parameters are stored. An early approach was to create a region graph [3] from the timing constraints of the model. This is a finite representation of the infinite state space of timed automata by creating suitable equivalence relations that satisfy the same clock valuations. The state space is infinite because timed automata operate on real valued clocks. Still the finite representation in region graphs is too fine-grained for a practical application.

Another symbolic representation of clock evaluations are clock zones [1], which are convex sets of clock regions. A very common and efficient data structure for representing clock zones are Difference Bound Matrices (DBM) [1]. They are described in detail in Chapter 3.

In the analysis of Timed Automata, one constructs so called (finite) zone graphs that represent the reachable states symbolically. Here, each node of the graph consists of a location, and a clock zone. Real-time model checkers such as Uppaal [4] represent the zone graph as a mapping from locations to Clock Federations which are lists of DBMs.

But Clock federations suffer from considerable drawbacks. Checking if a certain clock valuation is contained in a Clock Federation takes linear time in the number of contained DBMs. Furthermore Clock Federations suffer from fragmentation. This means that an actual convex set might be distributed in many DBMs of the Federation.

A data structure called Clock Difference Diagrams (CDD) [8] tries to address these issues. CDDs are a BDD-like [5] structure that stores clock valuations as a rooted, directed, and acyclic graph. Their advantage over Clock Federations is their space-efficiency. They are described in detail in Chapter 4.

CDDs can't exploit the advantage of convex sets. Diagrams carrying large parts of convex clock valuations become unnecessarily large. So, on the one hand we have Clock Federations which are suitable for mostly convex sets but inefficient for non convex sets and CDDs whose structure will become unnecessarily large if the set is mostly convex. None of those structures seems to be suitable for both kinds of sets.

In this thesis, we will present a novel approach for storing arbitrary sets of clock evaluations efficiently, no matter if most parts are convex or not. This data structure is called Clock Matrix Diagrams (CMD). CMDs are similar to CDDs, but in the nodes of the graph, not only single clock differences are stored, but a set of clock evaluations, represented by DBMs, and thus they combine the space-efficiency of CDDs with the convex efficiency of DBMs. CMDs are described in Chapter 5.

Chapter 2

Timed Automata

For modeling real time systems[1] the transition systems used for pure discrete model checking are not sufficient because there is no way to model timing behaviour.

Timed Automata [3, 1] are finite-state machines extended with a finite set of real-valued clocks. Locations can be labelled with invariants. Transitions are labelled with guards, a synchronous action and a set of clocks to be reset. Invariants and guards are described by clock constraints. Time elapses uniformly for all clocks. At the start of the execution, all clocks have the value zero. The automaton can stay in a location as long as its invariant is fulfilled, transitions can only be taken if the guards of the transitions are fulfilled and new locations can only be entered if their invariants are satisfied. When taking a transition, then the values of the clocks, which are in the set of clocks to be reset, are set to zero, all other clocks keep their values. A state of a Timed Automaton is a tuple of a location and clock evaluations. An action is either a switching of locations or the elapsing of time while staying in a location.

Definition 1. *[1] A Timed Automaton is a tuple* $\langle L, L^0, \Sigma, X, I, E \rangle$ *where*

- L *is a finite set of locations*
- $L^0 \subseteq L$ *is a set of initial locations*
- Σ *is a finite set of labels*
- X *is the set of clocks, with* $\Phi(X)$ *denoting constraints of the clocks in* X
- I *is a mapping that labels each location* s *with some clock constraints* $\Phi(X)$
- $E \subseteq L \times \Sigma \times 2^X \times \Phi(X) \times L$ *is a set of switches*

[1] (i.e., under the assumption of a dense time domain)

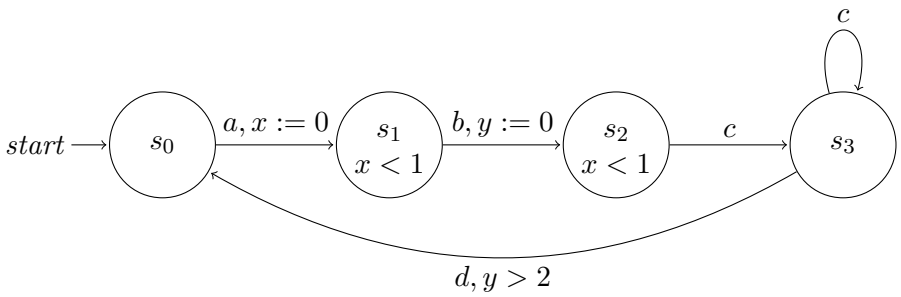

Figure 2.1: Example Timed Automaton

The switches are edges from one location to another on a symbol. A switch $< s, a, \varphi, \lambda, s' >$ is an edge from s to s' with the clock constraint φ and a set of clocks λ that are reset when taking the switch. The clock variables in X have real-time values. However the clock constraints can only compare to integers. Clock constraints have the form

$$\bigwedge(n \prec x \prec m),$$

where $n, m \in \mathbb{Z}, x \in X$ and $\prec \in \{<, \leq\}$. Guards and invariants are conjunction of clock constraints.

Example. An example of a Timed Automaton is shown in Figure 2.1. The automaton starts at the location s_0 with both clocks x and y set to zero, and it can stay an arbitrary amount of time in s_0. There is only one transition out of s_0, namely to s_1. When action a is triggered this transition is taken and the value of clock x is reset to zero. Then the automaton can stay in location s_1 only as long as the value of clock x is less than one time unit. Action b triggers the transition from s_1 to s_2 as long as the value of x is less than one time unit and then clock y is reset. s_2 has to be left before the value clock x exceeds 1 time unit. The transition from s_3 to s_4 can only be taken when enough time has passed so that the value of clock y is greater than two time units.

2.1 Symbolic State Space Representation

Due to the fact that the clock variables are defined over the real numbers, the semantics of Timed Automata imply an infinite state space[1].

A first approach for a finite representation of the the infinite state space are clock regions and the region graph. We define an equivalence on states, where each equivalence class is called a clock region. Two states of the infinite transition system of a Timed Automata are equivalent, if they agree

[1] to be precise, one obtains uncountable many states

on the integral parts and on the ordering of the fractional parts of all clock values and if they have the same location.

Definition 2. *[1] For any $\delta \in \mathbb{R}, fr(\delta)$ denotes the fractional part of δ, and $\lfloor \delta \rfloor$ denotes the integral part of δ. $\delta = \lfloor \delta \rfloor + fr(\delta)$. For each clock $x \in X$, let c_x be the largest integer c such that x is compared with c in some clock constraint appearing in an invariant or a guard. The equivalence relation $\cong$, called the region graph, is defined over the set of all clock interpretations for X. For two clock interpretations v and v', $v \cong v'$ iff all of the following conditions hold:*

- *For all $x \in X$, either $\lfloor v(x) \rfloor$ and $\lfloor v'(x) \rfloor$ are the same, or both $v(x)$ and $v'(x)$ exceed c_x.*
- *For all x, y with $v(x) \leq c_x$ and $v(y) \leq c_y, fr(v(x)) \leq fr(v(y))$ iff $fr(v'(x)) \leq fr(v'(y))$.*
- *For all $x \in X$ with $v(x) \leq c_x, fr(v(x)) = 0$ iff $fr(v'(x)) = 0$.*

Region graph. If we apply definition 2 to Timed Automata, we get a finite graph representing the state space. Let s and s' be locations and v and v' be clock valuations, then

$$(s, v) \cong (s', v') \text{ iff } s = s' \text{ and } v \cong v'.$$

The quotient $[TA]_{\cong}$ of a Timed Automaton is called the region graph.

While clock regions are used for proving decidability, they are inappropriate for practical applications. When building the region graph of a Timed Automaton, the graph grows exponentially with the number of clocks and with the number of constants. A more coarse symbolic representation are clock zones which are a convex subsumption of regions. In the following, we only consider clock zones as the basic symbolic representation of clock evaluations.

Chapter 3

Clock zones and Federations

The first structure we will look at for storing clock evaluations are Difference Bound Matrices (DBM) which are well suited for storing clock zones. Every clock zone can be represented by a DBM and every DBM represents a clock zone.

3.1 Clock zones

Definition 3. *A clock zone is a conjunction of inequalities that compare either a clock value to an integer or the difference between two clock values to an integer.*

A clock zone consisting of k clocks is a convex set in the k-dimensional Euclidean space. Inequalities of the following types are allowed in a clock zone:

$$x \prec c, c \prec x \text{ and } x - y \prec c,$$

where $\prec \in \{<, \leq\}$, $c \in \mathbb{N}_0$ and x and y are clocks. We introduce a special clock called x_0 whose value is always 0. This leads to a more uniform notation for clock constraints. With the help of x_0, we only need one form of clock constraints. The general form of a clock zone is:

$$x_0 = 0 \wedge \bigwedge_{0 \leq i \neq j \leq n} x_i - x_j \prec c_{i,j}$$

Operations on clock zones are described in 3.3.

3.2 Difference Bound Matrices

Every clock zone can be represented by a Difference Bound Matrix (DBM) [1, 6].

Definition 4. *A DBM is a matrix $D_{i,j}$ with the dimensions $|X| \times |X|$, where X is the set of clocks including a special clock called x_0 whose value is always 0. An entry is described by $d_{i,j} = (c, \prec)$ where $c \in \mathbb{N}$ and $\prec \in \{<, \leq\}$. Each entry denotes an inequality of the form $x_i - x_j < c$ or $x_i - x_j \leq c$ where $x_i - x_j$ denotes a difference between two clocks, where x_i and x_j are clocks and $c \in \mathbb{N}_0$.*

We can express constraints of the form $x_i - x_j \succ c$ by transforming them to $x_j - x_i \prec -c$. With the help of the clock x_0 we can express constraints of the form $x_i < c$ as $x_i - x_0 < c$.

Example. Consider the following clock zone:

$$x_1 - x_2 < 2 \wedge 0 < x_2 \leq 2 \wedge 1 \leq x_1$$

The corresponding DBM is given as:

	0	1	2
0	$(0, \leq)$	$(-1, \leq)$	$(0, <)$
1	$(\infty, <)$	$(0, \leq)$	$(2, <)$
2	$(2, \leq)$	$(\infty, <)$	$(0, \leq)$

3.3 Convex Operations

The following operations are closed, e.g., the results of these operations are convex and can thus be stored again as a DBM.

Intersection. The intersection of two DBMs is computed entry-wise. Given two DBMs D_1 and D_2, and $D' = D_1 \wedge D_2$, then

$$d'_{i,j} = min\{d^1_{i,j}, d^2_{i,j}\},$$

where $(c, <)$ is smaller than $(c, \leq)$.

Containment check. Since each clock zone is a set of clock constraints, we can easily check if one clock zone is a subset of another. Let Z_1 and Z_2 be DBMs. If we want to check if $Z_1 \subseteq Z_2$, we test if:

$$Z_1 \wedge Z_2 = Z_1$$

Time Elapse. Elapsing time means that the upper bounds of the clocks are set to infinity, i.e., after that operation $\forall x \in X : x - x_0 < \infty$ holds. Let $D' = D^{\Uparrow}$, then:

$$d'_{i,j} = \begin{cases} (\infty, <) & \text{for any } i \neq 0 \text{ and } j = 0 \\ d_{i,j} & \text{if } i = 0 \text{ or } j \neq 0 \end{cases}$$

Reset. With the reset operation, the values of clocks can be set to zero. Let λ be the set of clocks that should be reset. $[\lambda := 0]$ denotes that all clocks in λ are set to zero. $\forall c \in \lambda : c - c_0 \leq 0$, where the value of c_0 is always 0. Let $D' = D[\lambda := 0]$ and $\lambda \subseteq X$, then:

$$d'_{x,y} = \begin{cases} (0, \leq) & \text{if } x_i \in \lambda, x_j \in \lambda \\ d_{0,j} & \text{if } x_i \in \lambda, x_j \notin \lambda \\ d_{i,0} & \text{if } x_i \notin \lambda, x_j \in \lambda \\ d_{i,j} & \text{if } x_i \notin \lambda, x_j \notin \lambda \end{cases}$$

3.3.1 Implementation

An often used operation on the entries of DBMs is the compare operation which is needed,e.g., for intersection. So it would be desirable to compare the entries of DBMs with a single fast operation. To achieve that we must find a way to represent a DBM entry $(n, \prec)$ with $n \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$ with a well comparable structure. The most basic and fast comparable structure we could think of is an integer.

To profit from the easy comparability of integers we have to transform the DBM entries. How this can be done is shown in the following table. The first line shows the original DBM entries and the second line the corresponding integer representations.

...	(-4,<)	(-4,≤)	(-3,<)	(-3,≤)	(-2,<)	(-2,≤)	(-1,<)	(-1,≤)	(0,<)
...	-9	-8	-7	-6	-5	-4	-3	-2	-1

(0,≤)	(1,<)	(1,≤)	(2,<)	(2,≤)	(3,<)	(3,≤)	(4,<)	(4,≤)	...
0	1	2	3	4	5	6	7	8	...

The clock variables in X have real time values. However the clock constraints can only compare to integers. Clock constraints have the form $(n \prec x \in X \prec m)$, where $n, m \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$.

More formally we transform an entry $(n, \prec)$ in the following way:

$$(n, \prec) \Rightarrow \begin{cases} 2n - 1 & \text{if } \prec \text{ is } < \\ 2n & \text{if } \prec \text{ is } \leq \end{cases}$$

To get the corresponding tuple out of the integer x we use the following computation:

$$x \Rightarrow \begin{cases} (\frac{x}{2}, \leq) & \text{if x mod } 2 = 0 \\ (\frac{x+1}{2}, <) & \text{otherwise} \end{cases}$$

The comparison of the integer representations gives the same result as comparing the original tuples.

The DBM itself can then be efficiently implemented as an array of integers.

3.4 Clock Federations

Operations on DBMs might result in a non convex set which is not representable by a DBM. The simplest way of dealing with such sets is to store the several convex parts of the result in a linear list of DBMs. We call those lists Clock Federations.

We now define the operations on DBMs that result in non convex sets.

3.4.1 Non-convex operations

These operations do not lead necessarily to a convex polyhedron. So they can't be represented as DBMs. Structures that can handle non-convex sets are discussed later in detail.

Negation. We only define a negation operator for a constraint and not for the whole DBM. The negation of a constraint is computed by switching the relational operator as follows:

$$(n, <) \text{ is switched to } (n, \geq) \text{ and } (n, \leq) \text{ to } (n, >).$$

Union. To unify DBMs we just add them to the list of the Clock Federation.

Zone subtraction. We can describe the subtraction of two clock zones with the help of the intersection and the negation:

$$Z_1 - Z_2 = Z_1 \cap \overline{Z_2}$$

We haven't defined a negation operator for DBMs yet. A DBM is defined by the conjunction of its constraints $C_1 ... C_n$, so it holds that:

$$Z = \bigcap_{i=1}^{n} C_i$$

So we can thus express the negation of Z as:

$$\overline{\bigcap_{i=1}^{n} C_i} = \bigcup_{i=1}^{n} \overline{C_i}$$

So $Z_1 - Z_2$ can be expressed as:

$$Z_1 \cap \bigcup_{i=1}^{n} \overline{C_i} = \bigcup_{i=1}^{n} Z_1 \cap \overline{C_i}$$

Algorithm 3.4.1 exploits this transformation and computes the difference of two zones [11]. For this algorithm, we require clock zones to be reduced.

Definition 5. *Given a DBM D, then D is reduced if it has the minimal number of bounded elements.*

Algorithm 1 Zone subtraction

```
Z_intersection ← Z_1 ∩ Z_2
Reduce(Z_intersection)
Z_remain ← Z_1
for all bound Z_intersect(i, j), i ≠ j do
  Z_tmp ← Z_remain ∩ Z_intersect(i, j);
  if (Z_tmp ≠ ∅) then
    Z ← Z ∪ {Z_tmp}
    Z_remain ← Z_remain ∩ Z_intersect(i, j)
  end if
end for
return Z
```

Zone merging. For Algorithm 3.4.1 only the first execution guarantees a minimal set of clock zones. This is not ensured for consecutive executions, i.e., subtracting several zones from a Clock Federation.

But it is preferable to keep the number of zones in a Federation minimal. For this reason there are algorithms for merging a set of clock zones to the minimal number of Zones possible. A merging algorithm is presented in [11]. More algorithms which are implemented in the Uppaal model checker are presented in [7].

3.5 Discussion

DBMs are an efficient data structure as long as the set of clock evaluations is convex. But even simple set-theoretic operations such as union or subtraction result in non-convex sets.

Although the linear lists profit from the DBM structure for convex sets, this advantage gets less important the longer the list grows. Inserting new

zones is easy, a new DBM just has to be added to the list. But the time for checking if a certain time point is part of the set depends on the length of the list and it might take undesired long time for very large lists.

Another problem is the fragmentation of the set. This means that a large part of the set is actually convex but it is distributed over many several DBMs. Recognizing convex subsets is not a trivial problem. This leads to the problem, that sets represented by Clock Federations usually consume too much space.

Chapter 4

Clock Difference Diagrams

So far we have seen one way to deal with non convex sets: the representation of clock federations by linear lists. In this Chapter, we describe Clock Difference Diagrams (CDD), a BDD-like structure.

Binary Difference Diagrams (BDDs). A BDD [5] represents a Boolean formula as a rooted directed acyclic graph. The inner nodes of the graph represent the variables of the formula. There are exactly two terminal nodes: The true node and the false node. All paths through the graph end in either one of those nodes. There are two kinds of edges, one (the solid lines) representing that the variable of the node where the edge starts is evaluated to true and another one (the dashed lines) representing that the variable evaluates to false. An example of a BDD is shown is Figure 4.1.

Interval Decision Diagrams (IDDs). IDDs [16] are an extension to BDDs. While BDDs only allow two kinds of edges, IDDs not only operate on the two truth values true and false but on intervals of real numbers.

The idea of IDDs are further extended to CDDs. Instead of intervals, edges in CDDs are labeled with differences of variables. With this structure CDDs exactly represent the union of clock zones.

4.1 Syntax

A CDD is a rooted directed acyclic graph. It consists of nodes N and edges E. The nodes are labeled with clock constraints $x_i - x_j$ and the edges are labeled with intervals. A CDD has exactly two terminal nodes, namely *True* and *False*. All other nodes are inner nodes. They have a certain type $t \in \mathcal{T}$ and a finite set of successors where:

- $\mathcal{T} = \{(x_i, x_j) \mid x_i, x_j \in X, i \neq j\} \cup \{x_i, 0) \mid x_i \in X\} \cup \{\mathit{True}, \mathit{False}\}$.

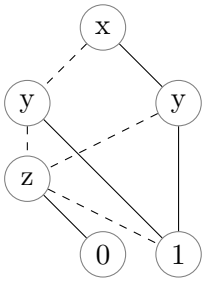

Figure 4.1: BDD representing the Boolean formula: $Y \vee (\neg X \wedge \neg Y \wedge \neg Z)$

The types of inner nodes denote clock differences and X is the set of clocks.

- $succ(n) = \{(I_1, n_1), ...(I_k, n_k)\}$, where $(I_i, n_i) \in \mathcal{I} \times V$. $\mathcal{I}$ is the set of all intervals and V is the set of nodes.

For example if a node of a CDD has the type $(2, 1)$ and the interval on the edge to its successor would be $I = (3, 5)$, then this would describe the clock constraint $3 < x_2 - x_1 < 5$. Figure 4.2 shows an example of a CDD. The top node has the type $X - 0$, where 0 is a special clock that always has the value 0. The interval on the edge from X to Y means that the clock constraint of the top node lies in that interval for the convex sub-part following this path.

We will now define important properties for the diagram. $n \xrightarrow{I} m$ is a shortcut for $(I, m) \in succ(n)$. For each inner node n, the following properties must hold:

- The successors are disjoint: for $(I, m), (I', m') \in succ(n), I \cap I' = \emptyset$ for $I \neq I'$.
- The successor set is an $\mathbb{R}$-cover: $\bigcup\{I \mid \exists m.n \xrightarrow{I} m\} = \mathbb{R}$. The Intervals that are not explicitly shown in the diagram point implicitly to the *False* node.
- The nodes are ordered according to their types. For $n \xrightarrow{I} m$ holds that $type(m) \sqsubseteq type(n)$.

We define a CDD $(V, type, succ)$ by its source node. Every sub-graph of a CDD is again a CDD.

Reduced Form. To decide equality and set inclusion we require a kind of normal form. There exists a *reduced form* that is easier to compute as a normal form and that is sufficient for this purpose. However this reduced

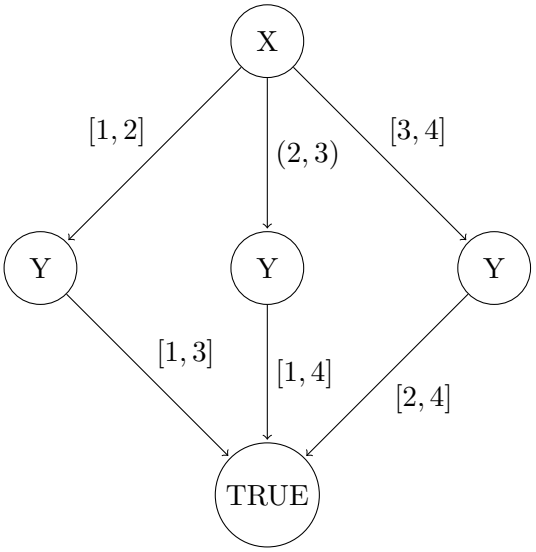

Figure 4.2: CDD example

form is not unique. Computing the reduced form is faster as computing the normal form. A CDD is in normal form if the following three properties are satisfied:

- It has maximal sharing. This means that there are no equivalent subgraphs.
- There are no trivial edges: If $n \xrightarrow{I} m$ then $I \neq \mathbb{R}$.
- All intervals are maximal: Whenever $n \xrightarrow{I_1} m, n \xrightarrow{I_2}$ m,then $I_1 = I_2$ or $I_1 \cup I_2 \notin \mathcal{I}$.

4.2 Denotational Semantics

Let $\mathcal{V}$ denote the set of clock evaluations. We define the semantics of a CDD recursively:

- $[\![False]\!] = \emptyset$, $[\![True]\!] = \mathcal{V}$
- $[\![n]\!] := \{v \in \mathcal{V} \mid n \xrightarrow{I} m, v \in [\![m]\!]$ and $v(x_i) - v(x_j) \in I$, where $type(n) = (x_i, x_j)\}$

If we traverse a path $n_0 \xrightarrow{I_1} ... \xrightarrow{I_k} n_k$ through a CDD, we find the representation of the conjunction of clock constraints described by that path. The conjunction of these constraints is a clock zone.

4.3 Operational Semantics

A CDD stores a set of possible clock evaluations. We can apply all the standard set-theoretic operations on CDDs.

4.3.1 Complement

CDDs are in contrast to DBMs closed under the complement. It is built by swapping the *True* and the *False* node. All the implicit paths that have led to *False* before the operation, point to *True* afterwards.

4.3.2 Union and Intersection

For CDDs we can define every binary set-theoretic operation recursively, e.g., Union and Intersection. The procedure is described in Algorithm 1. We apply the algorithm on two nodes, each node defining a whole CDD. At first, we have to define the base case for the desired operation. In the algorithm, this is called `baseOp`. This operation is only executed, if each of both nodes is either *True* or *False*. For all other cases, the algorithm operates recursively as shown in Algorithm 1 [8].

Algorithm 2 $op(n_1, n_2, \texttt{baseOp})$

if n_1 and n_2 are terminal nodes **then**
  return node equal to n_1 `baseOp` n_2.
end if
if $type(n_1) \sqsubseteq type(n_2) \wedge type(n_1) \neq type(n_2)$ **then**
  return the node $(type(n_1), \{(I_1, op(n_1', n_2, \texttt{baseOp})) \mid n_1 \xrightarrow{I_1} n_1'\}$
end if
if $type(n_2) \sqsubseteq type(n_1) \wedge type(n_1) \neq type(n_2)$ **then**
  return the node $(type(n_2), \{(I_2, op(n_1, n_2', \texttt{baseOp})) \mid n_2 \xrightarrow{I_2} n_2'\}$
end if
if $type(n_1) = type(n_2)$ **then**
  return the node $(type(n_1), \{(I_1 \cap I'2, op(n_1', n_2', \texttt{baseOp})) \mid n_1 \xrightarrow{I_1} n_1', n_2 \xrightarrow{I_2} n_2', I_1 \cap I_2 \neq \emptyset\}$
end if

4.3.3 Set inclusion

CDD $\subseteq$ CDD. If we want to check if a CDD is completely part of another CDD, we can use the following set-theoretic equivalence:

$$A \subseteq B \Leftrightarrow A \cap \neg B = \emptyset$$

For checking emptiness we would need a canonical form, but we have only defined a reduced form which is not unique. We can decide emptiness anyway by looking at the satisfiability of all paths leading to *True*. If there is at least one path ending in *True* the CDD can't be empty. A more formal description is given by Algorithm 2 [8].

Algorithm 3 Test for emptiness of a CDD

```
let P be the set of paths starting in n leading to True
while P ≠ ∅ do
  extract and remove a path p from P
  test p for satisfiability
  if there is a valuation that satisfies p then
    return false
  end if
end while
return true
```

Zone ⊆ CDD. Deciding if a zone lies completely in a CDD is a special case of the CDD ⊆ CDD case. Every CDD representing one convex clock zone has only one path without any branches. Algorithm 3 describes a function to test, if a clock zone, represented by a CDD, lies completely in an arbitrary CDD [8].

Algorithm 4 zone ⊆ CDD

```
subset(D,n)
if D = False or n = True then
  return True
else if n = False then
  return False
else
  return ⋀_{n -I-> m} subset(D ∧ I(type(n)), m)
end if
```

4.4 Discussion

We have seen that CDDs are able to store non-convex sets of clock evaluations efficiently regarding the space-consumption, because every clock constraint is only stored once. Also all basic set-theoretic operations can be performed on CDDs time-efficiently.

The main problem of CDDs is that they can't exploit the convex efficiency of DBMs. Large parts of the diagram may be convex, but we still need one node per clock constraint which affects the runtime of all algorithms.

Chapter 5

Clock Matrix Diagrams

In the previous Chapters two data structures for handling non-convex sets of difference-constraints have been described. We have seen that both of them have their individual strengths but they both suffer from certain disadvantages. None of these structures is superior to the other one in general. It strongly depends on the field of application which is the best choice between these two data structures. Due to the structure of the application, a non-convex set might change over time, a supposed good choice for a convenient data structure at the beginning might turn out as a bad decision afterwards. In this Chapter a novel data structure called Clock Matrix Diagrams which generalises the concepts of Clock Federations and Clock Difference Diagrams is presented. The new data structure will exploit the advantages of both of those structures by recognizing convex sub-parts of a set and storing them as DBMs.

5.1 Syntax

A Clock Matrix Diagram (CMD) is a rooted directed acyclic graph. It consists of a special root node and inner nodes. Inner nodes of a CMD are defined by a difference bound matrix and a set of successor nodes. The graph is rooted with a special node that carries an unrestricted clock zone and that has no predecessor nodes. There are only two leave nodes, the *True* node ($\top$) which has an unrestricted clock zone and the *False* node ($\bot$) whose zone is the empty zone. All paths through the graph end either at the *True* node or at the *False* node. All implicit paths that are not shown in the graph evaluate to false.

For all direct successors of a node we require that the clock zones of these nodes are disjoint.

Definition 6. *A CMD is a tuple* $C = \langle Q, q_0, X, \delta, Z \rangle$*, where*

- Q *is the set of nodes with* $\{\top, \bot\} \subseteq Q$*,*
- $q_0 \in Q$ *is the root node,*
- X *is the set of clocks,*
- δ *is the transition relation,* $\delta \subseteq Q \times Q$*,*
- Z *is a labeling function which assigns clock zones to nodes:* $Z : Q \to \mathcal{Z}$*, where* $\mathcal{Z}$ *is the set of all clock zones.*

Nodes $n \in Q$ *are tuples* (m, s) *where* m *is a difference bound matrix and* s *is the set of successor nodes.*

5.2 Denotational Semantics

If we take a path from the root node to a leaf node, the resulting zone is the intersection of the clock zones of all nodes that lie on that path. The zones that are described by the several branches are unified.

Let X be the set of clocks and $\mathcal{V}$ the set of all possible clock evaluations, $\mathcal{V} = \mathbb{R}^X_{\geq 0}$. Any convex polyhedron of clock differences that is described by a clock zone is a subset of $\mathcal{V}$. Each node in the CMD stores a clock zone.

The semantics of a CMD C is given as

$$
\begin{aligned}
[\![C]\!] &:= [\![q_0]\!], \text{ where for any } q \in Q: \\
[\![q]\!]_C &:= \begin{cases} \mathcal{V} & \text{if } q = \top \\ \emptyset & \text{if } q = \bot \\ [\![Z(q)]\!] \cap (\bigcup_{(q,q') \in \delta} [\![q']\!]) & \text{for any } q \in Q \end{cases}
\end{aligned}
$$

Definition 7. *Determinism criterion: We call a CMD deterministic if and only if the zones of all direct successors of a node to be disjoint from each other.*

$$\forall (q, q'), q \neq q' \wedge (p, q) \in \delta \wedge (p, q') \in \delta.\ Z(q) \uplus Z(q')$$

Example. An example for the denotational semantics is shown in Figure 5.1. The valid time points of C are calculated according to the semantics:

$$[\![C]\!] := r \cap ((a \cap (c \cup d)) \cup (b \cap (e \cup f)))$$

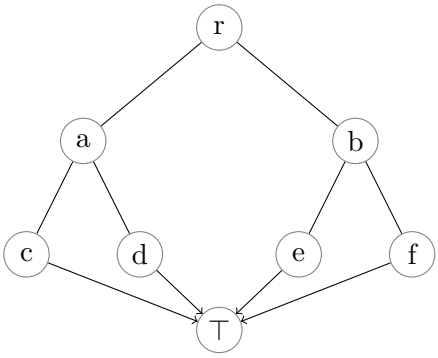

Figure 5.1: CMD denotational semantics illustrated

5.3 Operational Semantics

5.3.1 Intersection

Intersection of a CMD with a clock zone depicted by the operation $op_\cap$ is computed by the node wise intersection of the clock zone that is to be added and the clock zone of the existing node. The intersection operation does not destroy the disjointness criterion.

Let $C = \langle Q, q_0, X, \delta, Z \rangle$ be a CMD and z a clock zone.

$$\text{If } C' = C \ op_\cap \ z, \text{ then } C' = \langle Q', q_0', X', \delta', Z' \rangle, \text{ where :}$$

- $Q' = Q$
- $q_0' = q_0$
- $X' = X$
- $\delta' = \delta$
- $\langle Z, \emptyset \rangle \ op_\cap \ z = \langle Z, \emptyset \rangle$
- $\langle Z', \emptyset \rangle = \langle Z, \{q_0\} \rangle \ op_\cap \ z$, where
- $\langle Z, \hat{q} \rangle \ op_\cap \ z = \big\langle Z[q \mapsto Z(q) \wedge z \mid q \in \hat{q}], \{q' \in Q \setminus \{\top, \bot\} \mid \exists q \in \hat{q}.(q, q') \in \delta\} \ op_\cap \ z \big\rangle$

Lemma 1.

Let $C = \langle Q, q_0, X, \delta, Z \rangle, C' = \langle Q', q_0', X', \delta', Z' \rangle$ *and* $C' = C \ op_\cap \ z$, *then*

$$[\![C']\!] = [\![C]\!] \cap [\![z]\!]$$

Proof. In the following, we use this abbreviation:

$$[\![q \rightarrow]\!]_C \text{ stands for } \bigcup_{(q,q') \in \delta} [\![q']\!]_C$$

This denotes the union of the evaluation of all successor nodes of the node q of the CMD C.

We prove the lemma by showing $[\![q]\!]_{C'} = [\![q]\!]_C \cap [\![z]\!]$ by induction over δ.

$$\begin{array}{ll} (i) & \text{if } q = \bot : \quad [\![\bot]\!]_{C'} = [\![\bot]\!]_C \\ (ii) & \text{if } q = \top : \quad [\![\top]\!]_{C'} = [\![\top]\!]_C \\ (iii) & \text{if } q \text{ is an inner node:} \end{array}$$

$$\begin{array}{lll} [\![q]\!]_{C'} & \overset{\text{Def. } [\![q]\!]}{=} & [\![Z'(q)]\!]_{C'} \cap [\![q \rightarrow]\!]_{C'} \\ & \overset{\text{Def. op}_\cap}{=} & [\![Z(q) \wedge z]\!]_C \cap [\![q \rightarrow]\!]_{(C \ op_\cap \ z)} \\ & \overset{\text{Def. } \wedge, [\![z]\!]}{=} & [\![Z(q)]\!]_C \cap [\![q \rightarrow]\!]_{(C \ op_\cap \ z)} \\ & \overset{\text{I.H.}}{=} & [\![Z(q)]\!]_C \cap [\![z]\!] \cap [\![q \rightarrow]\!]_C \cap [\![z]\!]) \\ & \overset{\text{distrib.} \cap}{=} & \big([\![Z(q)]\!]_C \cap [\![q \rightarrow]\!]_C\big) \cap [\![z]\!] \\ & \overset{\text{Def. } [\![q]\!]}{=} & [\![q]\!]_C \cap [\![z]\!] \end{array}$$

□

5.3.2 Union

The union operation adds a new branch to the diagram. This might destroy the consistency criterion that the zones of the direct successor nodes have to be disjoint from each other. We require $Q_1 \cap Q_2 = \emptyset$ and $Z_1(q_0^1) \equiv Z_2(q_0^2)$. Another operation is required to restore this invariant. This operation is called *determize* and will be described in Section 5.3.5. If we unify a CMD C_1 with a CMD C_2 we throw away the root node of C_2 and add all children of C_2 as new children of C_1.

Let $C_1 = \langle Q_1, q_0^1, X_1, \delta_1, Z_1 \rangle$ and $C_2 = \langle Q_2, q_0^2, X_2, \delta_2, Z_2 \rangle$be two CMDs.

$$\text{If } C' = C_1 \ op_\cup \ C_2, \text{ then } C' = \langle Q', q_0', X', \delta', Z' \rangle, \text{ where :}$$

- $Q' = Q_1 \cup (Q_2 \setminus q_0^2)$
- $q_0' = q_0^1$

- $X' = X$
- $\delta' = \delta_1 \cup \{(q, q') \in \delta_2 \mid q \neq q_0^2\} \cup \{(q_0^1, q') \mid (q_0^2, q') \in \delta_2\}$
- For all $q \in Q'$, $Z'(q) = \begin{cases} Z_1(q) & \text{, if } q \in Q_1 \\ Z_2(q) & \text{, if } q \in Q_2 \end{cases}$

Lemma 2. *Let* $C_1 = \langle Q_1, q_0^1, X_1, \delta_1, Z_1 \rangle$, $C_2 = \langle Q_2, q_0^2, X_2, \delta_2, Z_2 \rangle$, $C' = \langle Q', q_0', X', \delta', Z' \rangle$ *and* $C' = C_1 \; op_\cup \; C_2$, *then*

$$[\![C']\!] = [\![C_1]\!] \cup [\![C_2]\!].$$

Proof.

$$\begin{array}{rcl}
[\![C_1]\!] \cup [\![C_2]\!] & \overset{\text{Def } [\![\text{C}]\!]}{=} & [\![q_0^1]\!] \cup [\![q_0^2]\!] \\
& \overset{\text{Def } [\![\text{q}]\!]}{=} & \big([\![Z_1(q_0^1)]\!]_{C_1} \cap [\![q_0^1 \rightarrow]\!]_{C_1}\big) \cup \big([\![Z_2(q_0^2)]\!]_{C_2} \cap [\![q_0^2 \rightarrow]\!]_{C_2}\big) \\
& \overset{\text{Def } \delta'}{=} & \big([\![Z'(q_0^1)]\!]_{C'} \cap [\![q_0^1 \rightarrow]\!]_{C'}\big) \cup \big([\![Z'(q_0^2)]\!]_{C'} \cap [\![q_0^2 \rightarrow]\!]_{C'}\big) \\
& \overset{\text{Def Z}'}{=} & \big([\![Z'(q_0')]\!]_{C'} \cap [\![q_0^1 \rightarrow]\!]_{C'}\big) \cup \big([\![Z'(q_0')]\!]_{C'} \cap [\![q_0^2 \rightarrow]\!]_{C'}\big) \\
& = & [\![Z'(q_0')]\!]_{C'} \cap \big([\![q_0^1 \rightarrow]\!]_{C'} \cup [\![q_0^2 \rightarrow]\!]_{C'}\big) \\
& \overset{\text{Def } \delta'}{=} & [\![Z'(q_0')]\!]_{C'} \cap [\![q_0^1 \rightarrow]\!]_{C'} \\
& \overset{\text{Def } [\![\text{q}]\!]}{=} & [\![q_0']\!]_{C'} \\
& \overset{\text{Def } [\![\text{C}]\!]}{=} & [\![C']\!]
\end{array}$$

□

5.3.3 Clock Reset

Let $C = \langle Q, q_0, X, \delta, Z \rangle$ be a CMD.

$$\text{If } C' = op_{[\lambda:=0]}, \text{ then } C' = \langle Q', q_0', X', \delta', Z' \rangle, \text{ where:}$$

- $Q' = Q$
- $q_0' = q_0$
- $X' = X$
- $\delta' = \delta$
- $op_{[\lambda:=0]}\big(\langle Z, \emptyset \rangle\big) = \langle Z, \emptyset \rangle$
- $\langle Z', \emptyset \rangle = op_{[\lambda:=0]}\big(\langle Z, \{q_0\} \rangle\big)$, where

- $op_{[\lambda:=0]}\big(\langle Z, \hat{q}\rangle\big) = \big\langle Z[q \mapsto Z(q)[\lambda := 0] \mid q \in \hat{q}], op_{[\lambda:=0]}\big(\{q' \in Q \setminus \{\top, \bot\} \mid \exists q \in \hat{q}.(q, q') \in \delta\}\big)\big\rangle$

Lemma 3.

Let $C = \langle Q, q_0, X, \delta, Z\rangle, C' = \langle Q', q_0', X', \delta', Z'\rangle$ *and* $C' = op_{[\lambda:=0]}(C)$, *then*

$$[\![C']\!] = [\![C]\!][\lambda := 0]$$

Proof. We prove the lemma by showing $[\![q]\!]_{C'} = [\![q]\!]_C[\lambda := 0]$ by induction over δ.

(i)	if $q = \bot$:	$[\![\bot]\!]_{C'} = [\![\bot]\!]_C$
(ii)	if $q = \top$:	$[\![\top]\!]_{C'} = [\![\top]\!]_C$
(iii)	if q is an inner node:	

$$\begin{aligned}
[\![q]\!]_{C'} &\overset{\text{Def } [\![\text{q}]\!]}{=} [\![Z'(q)]\!]_{C'} \cap [\![q \rightarrow]\!]_{C'} \\
&\overset{\text{Def op}_{[\lambda:=0]}}{=} [\![Z(q)[\lambda := 0]]\!]_C \cap [\![q \rightarrow]\!]_{(op_{[\lambda:=0]}(C))} \\
&= [\![Z(q)]\!]_C[\lambda := 0] \cap [\![q \rightarrow]\!]_{(op_{[\lambda:=0]}(C))} \\
&\overset{\text{I.H.}}{=} [\![Z(q)]\!]_C[\lambda := 0] \cap [\![q \rightarrow]\!]_C[\lambda := 0] \\
&\overset{\text{distrib.}[\lambda:=0]}{=} \big([\![Z(q)]\!]_C \cap [\![q \rightarrow]\!]_C\big)[\lambda := 0] \\
&\overset{\text{Def } [\![\text{q}]\!]}{=} [\![q]\!]_C[\lambda := 0]
\end{aligned}$$

□

5.3.4 Time Elapse

Let $C = \langle Q, q_0, X, \delta, Z\rangle$ be a CMD.

$$\text{If } C' = op^{\Uparrow}, \text{ then } C' = \langle Q', q_0', X', \delta', Z'\rangle, \text{ where :}$$

- $Q' = Q$
- $q_0' = q_0$
- $X' = X$
- $\delta' = \delta$
- $op^{\Uparrow}\big(\langle Z, \emptyset\rangle\big) = \langle Z, \emptyset\rangle$

- $\langle Z', \emptyset \rangle = op^{\Uparrow}\big(\langle Z, \{q_0\}\rangle\big)$, where
- $op^{\Uparrow}\big(\langle Z, \hat{q}\rangle\big) = \big\langle Z[q \mapsto Z(q)^{\Uparrow} \mid q \in \hat{q}], op^{\Uparrow}\big(\{q' \in Q \setminus \{\top, \bot\} \mid \exists q \in \hat{q}.(q, q') \in \delta\}\big)\big\rangle$

Lemma 4.

Let $C = \langle Q, q_0, X, \delta, Z\rangle, C' = \langle Q', q_0', X', \delta', Z'\rangle$ *and* $C' = op^{\Uparrow}(C)$, *then*

$$[\![C']\!] = [\![C]\!] \Uparrow$$

Proof. We prove the lemma by showing $[\![q]\!]_{C'} = [\![q]\!]_C \Uparrow$ by induction over δ.

(i) if $q = \bot$: $[\![\bot]\!]_{C'} = [\![\bot]\!]_C$

(ii) if $q = \top$: $[\![\top]\!]_{C'} = [\![\top]\!]_C$

(iii) if q is an inner node:

$$\begin{aligned}
[\![q]\!]_{C'} &\overset{\text{Def. } [\![q]\!]}{=} [\![Z'(q)]\!]_{C'} \cap [\![q \rightarrow]\!]_{C'} \\
&\overset{\text{Def. op}^{\Uparrow}}{=} [\![Z(q) \Uparrow]\!]_C \cap [\![q \rightarrow]\!]_{(op^{\Uparrow}(C))} \\
&= [\![Z(q)]\!]_C \Uparrow \cap [\![q \rightarrow]\!]_{(op^{\Uparrow}(C))} \\
&\overset{\text{I.H.}}{=} [\![Z(q)]\!]_C \Uparrow \cap [\![q \rightarrow]\!]_C \Uparrow \\
&\overset{\text{distrib.}\Uparrow}{=} \big([\![Z(q)]\!]_C \cap [\![q \rightarrow]\!]_C\big) \Uparrow \\
&\overset{\text{Def. } [\![q]\!]}{=} [\![q]\!]_C \Uparrow
\end{aligned}$$

□

5.3.5 Determize

As we have seen, some operations might lead to inconsistency of the structure. So we need to restore it with a special operation called *determize*. With $\vec{q}$ we denote the set of successor nodes of q. The abbreviation $[\![q \setminus q' \rightarrow]\!]$ stands for $\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1]\!]$, meaning the union of the evaluations of all successor nodes of q except of q'. The operation of *determize* is shown in Figure 5.3.

$$
\begin{aligned}
det(q_0) &= det\left(\llbracket Z(q_0)\rrbracket \cap \left(\llbracket q'\rrbracket_{q'\in\vec{q_0}} \cup \llbracket q \setminus q' \rightarrow\rrbracket\right)\right) \\
&= \llbracket Z(q_0)\rrbracket \cap det\left(\llbracket q'\rrbracket_{q'\in\vec{q_0}}, \llbracket q \setminus q' \rightarrow\rrbracket\right)
\end{aligned}
$$

$$
\begin{aligned}
& Z(q_0) \cap det\big(\llbracket q'\rrbracket, \llbracket q \setminus q' \rightarrow\rrbracket\big) \\
= \;& \llbracket Z(q_0)\rrbracket \cap \Big(\\
& \big(\llbracket Z(q')\rrbracket \setminus \bigcup_{q_1\in\vec{q}\setminus\{q'\}} \llbracket Z(q_1)\rrbracket\big) \cap det\left(\llbracket q' \rightarrow\rrbracket\right) \\
& \cup det\big(\big(\bigcup_{q_1\in\vec{q}\setminus\{q'\}} \llbracket Z(q_1)\rrbracket\big) \setminus \llbracket Z(q')\rrbracket\big) \cap det\big(\bigcup_{q_1\in\vec{q}\setminus\{q'\}} \llbracket q_1 \rightarrow\rrbracket\big) \\
& \cup \llbracket Z(q')\rrbracket \cap \bigcup_{q_1\in\vec{q}\setminus\{q'\}} \llbracket Z(q_1)\rrbracket \cap det\big(\llbracket q' \rightarrow\rrbracket, \bigcup_{q_1\in\vec{q}\setminus\{q'\}} \llbracket q_1 \rightarrow\rrbracket\big)\Big)
\end{aligned}
$$

Lemma 5. *Let $C = \langle Q, q_0, X, \delta, Z\rangle$, $C' = \langle Q', q_0', X', \delta', Z'\rangle$ and $det(C) = C'$, then $\llbracket C\rrbracket = \llbracket C'\rrbracket$.*

Proof.

$$
\begin{aligned}
& [\![Z(q)]\!] \cap det\big([\![q']\!], [\![q \setminus q' \rightarrow]\!]\big) \\
\overset{\text{Def } [\![\mathrm{q}]\!]}{=} \quad & [\![Z(q)]\!] \cap det\big([\![Z(q')]\!] \cap [\![q' \rightarrow]\!], \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap [\![q_1 \rightarrow]\!]\big) \\
\overset{\text{Def det}}{=} \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![Z(q')]\!] \setminus \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap det\big([\![q' \rightarrow]\!]\big) \\
& \cup\, det\big(\big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \setminus [\![Z(q')]\!]\big) \cap det\big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big) \\
& \cup [\![Z(q')]\!] \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap det\big([\![q' \rightarrow]\!] \cup \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big)\Big) \\
= \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![Z(q')]\!] \cap \overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]}\big) \cap det\big([\![q' \rightarrow]\!]\big) \\
& \cup\, det\big(\big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap \overline{[\![Z(q')]\!]}\,\big) \cap det\big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big) \\
& \cup [\![Z(q')]\!] \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap det\big([\![q' \rightarrow]\!] \cup \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big)\Big) \\
\overset{\text{I.H.}}{=} \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![Z(q')]\!] \cap \overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]}\big) \cap ([\![q' \rightarrow]\!] \\
& \cup \big(\big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap \overline{[\![Z(q')]\!]}\big) \cap \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!] \\
& \cup [\![Z(q')]\!] \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap [\![q' \rightarrow]\!] \cup \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\Big) \\
= \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![q']\!] \cap \overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]}\big) \\
& \cup \big(\big([\![q \setminus q' \rightarrow]\!]\big) \cap \overline{[\![Z(q')]\!]}\big) \\
& \cup [\![Z(q')]\!] \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap [\![q' \rightarrow]\!] \cup \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\Big)
\end{aligned}
$$

$$
\begin{aligned}
= \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![q']\!] \cap \overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]}\big) \\
& \cup \big(([\![q \setminus q' \rightarrow]\!]) \cap \overline{[\![Z(q')]\!]}\big) \\
& \cup \big([\![Z(q')]\!] \cap [\![q' \rightarrow]\!]\big) \\
& \cup \big([\![Z(q')]\!] \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big)\big) \cap \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\Big) \\
= \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![q']\!] \cap \overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]}\big) \\
& \cup \big(([\![q \setminus q' \rightarrow]\!]) \cap \overline{[\![Z(q')]\!]}\big) \\
& \cup \big([\![q']\!]\big) \\
& \cup \big([\![Z(q')]\!] \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big)\big) \cap \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\Big) \\
= \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![q']\!] \cap \overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]}\big) \\
& \cup \big(([\![q \setminus q' \rightarrow]\!]) \cap \overline{[\![Z(q')]\!]}\big) \\
& \cup \big([\![q']\!] \cap \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \\
& \cup \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big) \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap [\![Z(q')]\!]\Big) \\
= \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![q']\!] \cap \overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]}\big) \\
& \cup \big(([\![q \setminus q' \rightarrow]\!]) \cap \overline{[\![Z(q')]\!]}\big) \\
& \cup \big([\![q']\!] \cap \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \\
& \cup \big([\![q \setminus q' \rightarrow]\!] \cap [\![Z(q')]\!]\big)\Big)
\end{aligned}
$$

$$
\begin{aligned}
= \quad & [\![Z(q)]\!] \cap \Big(\\
& \big([\![q']\!] \cap \big(\overline{\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]} \cup \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \\
& \cup \big([\![q \setminus q' \rightarrow]\!]\big) \cap \big(\overline{[\![Z(q')]\!]} \cup [\![Z(q')]\!]\big) \\
= \quad & [\![Z(q)]\!] \cap \big([\![q']\!] \cup [\![q \setminus q' \rightarrow]\!]\big)\Big) \\
\overset{\text{Def}[\![q]\!]}{=} \quad & [\![q]\!]
\end{aligned}
$$

□

Lemma 6. *Let $C = \langle Q, q_0, X, \delta, Z\rangle$, $C' = \langle Q', q_0', X', \delta', Z'\rangle$ and $det(C) = C'$, then C' is deterministic according to Definition 7.*

Proof. The claim follows immediately from the correctness of the following set theoretic formula. Let:

$$
\begin{aligned}
A &= [\![Z(q_0)]\!] \\
B &= \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]
\end{aligned}
$$

Then it holds that,

$$A \setminus B \uplus B \setminus A \uplus A \cap B.$$

□

Special case for usage with the Union operation. When we unify two CMD, we have the assumption that both CMDs are already deterministic and that after the union operation, only the first level of the CMD might not be deterministic, but deeper levels are. This enables us to modify the definition of determize for that special case. For the modified definition, we only have to call *det* recursively for the intersected branch.

Lemma 7. *When only the first level of a CMD is not deterministic, we can use a modified version of the definition of determize as follows:*

$$
\begin{aligned}
& Z(q_0) \cap det\big([\![q']\!], [\![q \setminus q' \rightarrow]\!]\big) \\
= \; & [\![Z(q_0)]\!] \cap \Big(\\
& \big([\![Z(q')]\!] \setminus \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap [\![q' \rightarrow]\!] \\
& \cup \big(\big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \setminus [\![Z(q')]\!]\big) \cap \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} \big([\![q_1 \rightarrow]\!]\big) \\
& \cup [\![Z(q')]\!] \cap \big(\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![Z(q_1)]\!]\big) \cap det\big([\![q' \rightarrow]\!], \bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q_1 \rightarrow]\!]\big)\Big)
\end{aligned}
$$

Proof. $[\![q' \rightarrow]\!]$ and $\bigcup_{q_1 \in \vec{q} \setminus \{q'\}} [\![q1 \rightarrow]\!]$ are already deterministic by assumption, so they don't need to be determized again. □

5.3.6 Expand

The structure of the diagram can be changed by heuristics. The efficiency is gained by the choice of a "good" heuristic. The operation *expand* examines all the zones of the direct child nodes with the help of a replaceable heuristic. The nodes are split into two partitions. Figure 5.2 shows how that splitting is done. The node containing the first partition replaces the original node and the node with the second partition is the only child of the node mentioned before having the children of the original node. It holds that:

$$Z = Z_x \wedge Z_{\overline{x}}$$

Definition 8. *An expand heuristic is a function $H : 2^X \rightarrow 2^X$ that partitions a given set of clocks into two disjoint subsets.*

The determinism criterion might be destroyed after the expansion. For that reason the *expand* operation is always followed by a *determize* operation. On the other hand *expand* is always executed before the *determize* operation because only this will lead to a meaningful result of *determize*.

Let $C = \langle Q, q_0, X, \delta, Z \rangle$ a CMD.

If $C' = expand(C, H),$ then $C' = \langle Q', q_0', X', \delta', Z' \rangle,$ where :

- H is an arbitrary heuristic that splits two clock zones so that $Z(A) = Z(A_x) \wedge Z(A_{\overline{x}})$ holds.
  A_x and $A_{\overline{x}}$ have to be minimal, i.e., there are no constraints in A_x or in $A_{\overline{x}}$ which are not in A.
  $split_1(A) := A_x$ and $split_2(A) := A_{\overline{x}}$

- Q' is an arbitrary set s.t.
  (i) $Q \subseteq Q'$
  (ii) $|Q'| = |Q| + |\{(q_0, q) \in \delta\}|$
- $q'_0 = q_0$
- $X' = X$
- $\delta' = \delta \setminus \{(q'_0, q) \in \delta\} \cup \{(q_0, q') \mid q' \in Q' \wedge q' \notin Q\} \cup \{(q', q) \mid q' \in Q' \wedge q' \notin Q \wedge (q_0, q) \in \delta\}$
- $Z'(q \in Q') = \begin{cases} split_1(Z(q)) & , \text{ if } \exists (q_0, q) \in \delta' \\ split_2(Z(q)) & , \text{ if } \exists (q_0, q') \in \delta, \exists (q', q) \in \delta' \\ Z(q) & , \text{ else} \end{cases}$

Lemma 8. *Let* $C = \langle Q, q_0, X, \delta, Z \rangle$, $C' = \langle Q', q'_0, X', \delta', Z' \rangle$ *and* $expand(C) = C'$, *then* $[\![C]\!] = [\![C']\!]$.

Proof. Be q a node in a CMD. Because after the *expand* operation each clock constraint of q is either in q_x or in $q_{\overline{x}}$ according to the definition, $q = q_x \cap q_{\overline{x}}$. $q_{\overline{x}}$ is the only child node of q_x, so $[\![q_x]\!] = Z(q_x) \cap Z(q_{\overline{x}}) \cap [\![q \rightarrow]\!]$

□

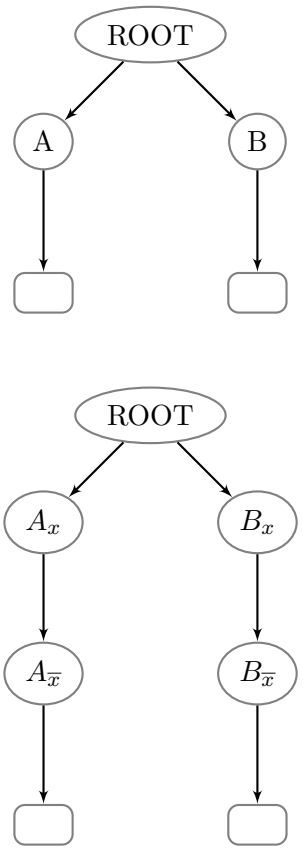

Figure 5.2: Expand

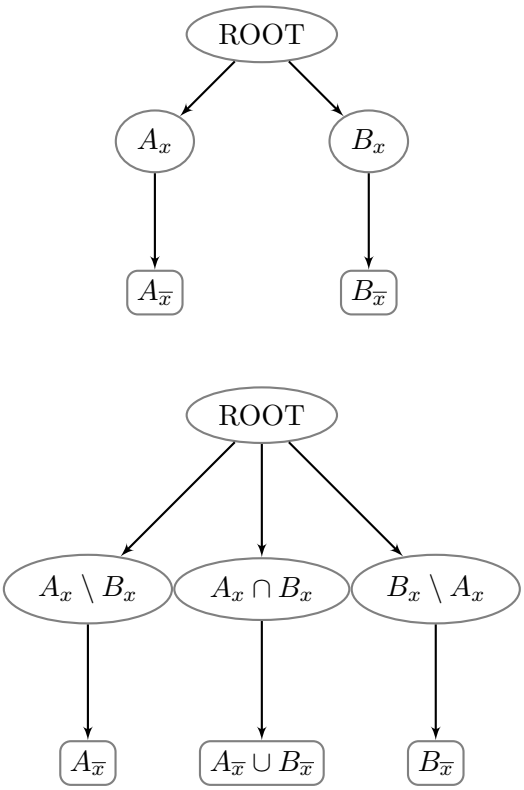

Figure 5.3: Determize

Chapter 6

Experimental Results

In this Chapter, we give information about implementation details of a prototype program that demonstrates the benefits of CMDs. We present experimental results of benchmarks which were performed with the help of the prototype by comparing the performance of several containment checks of CMDs and Clock Federations. The results are shown in 6.2.

6.1 Prototype implementation

The prototype is implemented in C++ and it it uses the clock zone library of the Uppaal model checker [14] which provides a very efficient implementation for clock zones with all necessary operations.

The prototype provides a class for dealing with CMDs and applying the standard operations as described in Chapter 5. In the following the main classes of the prototype are described. The connection of the classes are described in Figure 6.1.

6.1.1 The class Cmd

An instance of this class stores a single CMD, which consists of the root node, the top node ($\top$), the bottom node ($\bot$) and all inner nodes. Each node is an instance of the class CmdNode.

6.1.2 The class CmdNode

A CmdNode instance stores all relevant information of a CMD node: A clock zone, its minimal representation, a list of its child nodes and a pointer to the associated CMD (to access the top node and bottom node). Some important operations will be described in the following.

Union. This operation unifies the CMD with another CMD or a clock zone. For that purpose all the child nodes of the root node of the second

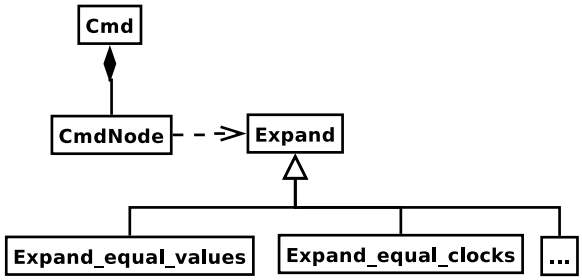

Figure 6.1: Class diagram of the prototype

CMD are added as children of the root node of the first CMD. The result might be an indeterministic CMD. For that purpose the *determize* function should be called after a union. The function *unify_det* automatically calls the *determize* function after union.

Algorithm 5 determize

```
for all childOld from oldDiagram do
  rest = child.zone;
  for all childNew from newDiagram do
    newDiagram.add(childNew.zone \ rest);
    newDiagram.add(childNew.zone ∩ rest).unify(childOld);
    rest = rest \ childNew.zone;
  end for
  newDiagram.add(rest);
end for
return newDiagram;
```

Determize. The determize function is executed after operations that lead to a non-deterministic CMD, e.g. the unify operation. At first, an operation called *split* is called to optimize the result of the *determizefunction*. The *split* function is described below. The operation of the *determize* function is described in Algorithm 4. In that algorithm we exploit the special case of *determize* as described in Section 5.3.5. With Algorithm 4, CMD is determized by sequently rebuilding a new CMD which is deterministic in every step.

Split. The split operation splits a clock zone into two new clock zone such that the intersection of the two new zones results in the original zone. Which constrains are put in which of the two new zones is determined by a heuristic, which is a parameter of the split function.

6.1.3 The class Expand

`Expand` is an abstract superclass for expand heuristics used by the *split* function. Concrete expand heuristics are derived from this class. The prototype implements the following heuristics.

`Expand_equal_values.` This heuristic searches for exact equal constraints i.e., the constraint $x_i - x_j \prec n$, for clocks $x_i, x_j \in X$ and $n \in \mathbb{N}_0$, with the same operator for $\prec$ and an identical n must be part of most clock zones. "Most" means that the constraints with the compliance in the largest number of clock zones are selected.

`Expand_equal_clocks.` With this heuristic, constraints are selected if they describe the same clock difference but are not exactly identical, e.g., all constraints of the type $x_i - x_j \prec n$, for clocks $x_i, x_j \in X$ and $n \in \mathbb{N}_0$, would be selected for any $\prec \in \{<, \leq\}$ and arbitrary n but constant i and j.

`Hybrid.` For the benchmark results of the next section, a combination of the two previously mentioned heuristics is used. If the heuristic `Expand_equal_values` cannot find any common constraints, the heuristic `Expand_equal_clocks` is used.

6.2 Benchmarks

The prototype implementation introduced in 6.1 was used to obtain experimental results about the performance of CMDs regarding the time for containment checks, i.e., testing if clock zones are contained in CMDs. All tests were measured on an AMD Opteron processor with 2.6 GHz.

In the following, the various test scenarios are described in detail and then a table with the results of the tests are presented. In every test, a large number of clock zones are unified with a CMD, which is empty at the beginning. At the same time, those clock zones are unified with a Clock Federation, which is also empty at the beginning. Then it is tested, if a certain clock zone is contained in the unified set. This is done for both, the CMD and the Clock Federation and the time for both containment checks is measured.

In all tests, every single clock zone, which is unified with the CMD and the Clock Federation, contains an identical clock constraint φ beyond other constraints.

Test scenario 1. For splitting the nodes of the CMD in the determize function, the heuristic `Hybrid` from 6.1.3 is used. Then it is tested, whether φ is contained in the unified set. Due to the functionality of the heuristic

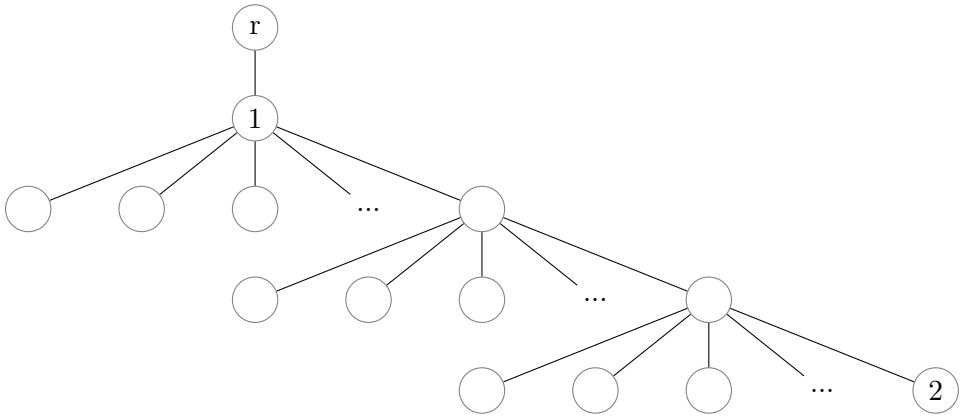

Figure 6.2: CMD used for all tests.

`Hybrid`, φ will be set in the most upper node, the only child node of the root node. The resulting CMD can be seen in Figure 6.2. Node 1 contains φ.

Test scenario 2. This test uses exactly the same setting as test 1, but the containment check differs. The tested clock constraint is not contained in the CMD. Because the checked constraint describes the same clock difference as node 1 but is disjoint to the constraint in this node, the containment check for the CMD can stop immediately at node 1.

Test scenario 3. This test is similar to test scenario 1. The only difference is that instead of the heuristic `Hybrid` the heuristic `Expand_equal_clocks` is used for building the CMD.

Test scenario 4. This test scenario is similar to test 2. Again, the zone we check is not contained in the unified set. But this time we use the heuristic `Expand_equal_clocks` instead of `Hybrid`.

Test scenario 5. Test 5 checks if a clock zone which is identical to the Zone of node 2 of the diagram is contained in the unified set. This is done using the heuristic `Hybrid`.

Test scenario 6. This test checks if a clock zone which is orthogonal to the zone of node 2 in the CMD is contained. In contrast to test 2, it should traverse the whole CMD before realizing that the zone is not contained.

	900 zones		1800 zones	
	CMD	FED	CMD	FED
Test 1	1.23	17.36	1.22	41.44
Test 2	1.63	18.02	1.63	42.96
Test 3	8.33	17.37	15.69	41.62
Test 4	20.09	17.94	41.14	43.01
Test 5	356.07	0.5	1661.04	0.51
Test 6	356.76	46.63	1659.82	102.33

Figure 6.3: Benchmark results

6.2.1 Results

The values of the benchmarks from table 6.3 are relative time units. They were obtained by executing one million iterations of each test.

The first four tests demonstrate the potential of CMDs. Whenever a containment check can be decided on an upper level of a CMD, Federations are highly outperformed because they have to be traversed from the beginning to the end in the case that the tested zone is not contained in the Federation. These four tests also show the importance of an efficient heuristic. The test scenarios where the heuristic `Expand_equal_clocks` is used perfom much worse compared to the heuristic `Hybrid`. With the heuristic `Expand_equal_clocks` the CMD still performs better than the Federation.

In the test scenarios five and six, the CMD is clearly outperformed. This bad result comes from the fact that the benchmarks were made with an early prototype implementation. With a better heuristic and a faster implementation, a CMD should not take any more time for a complete traversal than a Federation because CMDs usually don't traverse whole paths but they can exclude paths if the tested zone is not contained in any node of the path. The Federation in these tests perform so well because new DBMs are inserted at the beginning of the Federation, so the containment check in test scenario 5 gets its result when testing the first DBM in the list because node 2 was the last inserted node.

Chapter 7

Conclusion and Outlook

In this chapter we summarize the several ways of representing the timing aspect of the infinite state-space of Timed Automata with the help of symbolic representations. Finally we give some ideas for future research in this area.

7.1 Conclusion

The modeling of parallel systems lead to an explosion of the state space. When adding timing properties in the dense time domain, the state space even gets infinite. Thus we need a symbolic representation of the state space.

We have seen several data structures which can represent the timing aspect of the state space of Timed Automata. Starting with clock regions which are not appropriate for a practical application, because the region graph grows exponentially with the number of clocks and with the number of constants. Then we described Difference Bound Matrices, which are a data structure representing clock zones. Clock zones are convex sets of clock constraints. We illustrated the main operations on DBMs and presented how they can be implemented efficiently. However, there are applications where a data structure for non-convex clock constraints is needed (e.g. timed game solving). But also in the standard reachability analysis that is used for model checking, a data structure for non-convex sets can be advantageous since it provides a more compact state space representation. The first approach for representing convex sets we discussed are Clock Federations which are linear lists of DBMs. They are only appropriate for very small lists, but they suffer from certain drawbacks, namely fragmentation and the fact that containment checks need linear time. Then we reviewed Clock Difference Diagrams, another approach for storing non-convex sets of clock constraints. They constitute a more space-efficient representation, but they can't exploit the convex efficiency of DBMs. To take advantage of both, the convex-efficiency of DBMs and space-efficiency of CDDs, we introduced CMDs, a

novel data structure where DBMs are represented as nodes in a decision diagram. They are a generalisation of DBMs, Clock Federations and CDDs and combine the efficiency of all those structures. Our benchmark results show that CMDs outperform Clock Federations regarding the time.

7.2 Outlook

Our first benchmarks show the potential of CMDs. A more efficient implementation could unfold the power of CMDs even more. Several optimizations can be applied on the implementation of the *union* and the *determize* operation. Dependent on the application, an on-the-fly re-ordering of the nodes of the CMD might increase its performance in some cases. We could also reduce the number of nodes by identifying sibling nodes whose union represent a convex set, so that we can unify those nodes to a single node.

For a true generalization of Clock Federations and CDDs, we could allow additional properties for sub-graphs of CMDs, namely that sub-graphs would not have to fulfill the deterministic criterion and the enforcement of an order of the nodes to represent the according property of CDDs.

The development of a fully symbolic data structure for Timed Automata state spaces is an active field of research. We believe that CMDs, as presented in this thesis, are a good basis for such a data structure. In future work, we will investigate how to extend CMDs such that also the discrete part of the location-based control structure can be incorporated.

Our main objective is to develop a data structure for a full symbolic representation of the state space of Timed Automata. With CMDs we have a good structure for the timing part. For future work we are searching for a structure which can also represent the discrete part of the state space, namely the locations of Timed Automata which uses CMDs for the timing properties.

Bibliography

[1] Rajeev Alur. Timed automata. *Theoretical Computer Science*, 126:183–235, 1999.

[2] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.

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

[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: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.

[7] Alexandre David. Merging DBMs efficiently. In *17th Nordic Workshop on Programming Theory*, pages 54–56. DIKU, University of Copenhagen, October 2005.

[8] Clock Difference Diagrams, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Nordic journal of computing, 1999.

[9] Tom Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for real-time systems. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 353–366, New York, NY, USA, 1991. ACM.

[10] F. Jahanian and A. K.-L. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Trans. Comput.*, 36(8):961–975, 1987.

[11] Shang-Wei Lin, Pao-Ann Hsiung, Chun-Hsian Huang, and Yean-Ru Chen. Model checking prioritized timed automata. In *ATVA*, pages 370–384, 2005.

[12] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 265–280, New York, NY, USA, 1990. ACM.

[13] J. Ostro. Temporal logic of real-time systems. *Research Studies Press*, 1990.

[14] Paul Pettersson. Modelling and verification of real-time systems using timed automata: theory and practice. *PhD thesis, Uppsala University*, 1999.

[15] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical report, Cambridge, MA, USA, 1974.

[16] Karsten Strehl and Lothar Thiele. Symbolic model checking using interval diagram techniques, 1998.