Compositional Backwards Reachability for Timed Automata

Ulrik Larsen
ulrikl@cs.auc.dk

Masters Thesis

Department of Computer Science
Aalborg University

June 20th 2002

Aalborg University

Department of Computer Science

Title:
Compositional Backwards Reachability for Timed Automata

Semester:
Spring 2002, DAT6

Project Period:
February 1st 2002 to
June 20th 2002

Author:
Ulrik Larsen

Supervisor:
Kim G. Larsen

Number of Pages:
89

Keywords:
Reachability, Compositional Backwards Reachability, Timed Automata, Verification, State-space explosion.

Abstract:

This report deals with the development of a general framework for Compositional Backwards Reachability (CBR) and with the verification of reachability properties on Timed Automata Networks (TAN). The CBR method is developed on the basis of a series of finer and finer partitionings of the state-space. Two different CBR algorithms are presented and proven correct. The domain of TAN, which is a real-time model, is described. The symbolic DBM-based analysis of Timed Automata used in existing verification tools, like Uppaal is explained. The second of the CBR algorithms is applied to the domain of TAN. Several extensions to the domain are discussed, and a test implementation of the basic method is developed. This implementation is used to obtain some experimental results. Finally future work is discussed and a conclusion is drawn.

Preface

This Masters Thesis, was written at the research unit of Distributed Systems and Semantics, Department of Computer Science, Aalborg University. It is a further development of the Dat-5 report "Compositional Backwards Reachability for Simple Timed Automata" [Lar02]. Two of the chapters from the previous report have been included without substantial changes.

Acknowledgements

First and foremost, I would like to thank my supervisor Professor Kim G. Larsen for his good advice and many suggestions throughout the project period. Secondly, I would like to thank Gerd Behrmann for introducing me to, and answering question about, the UPPAAL source code.

Ulrik Larsen

Contents

1 Introduction

This chapter first motivates the work by explaining the problem at hand. After this the work is put into the context of related work. Finally the relation to the previous report [Lar02] is described together with an outline of the report.

1.1 Motivation

When trying to verify properties of parallel compositions of several components, the main problem is the fact that the state-space grows exponentially in the number of components, known as state-space explosion. When extending the models from discrete to timed models the state-space increases even further in size. This report focuses on yet another way to handle the state-space explosion problem in the presence of time. The underlying patented method, called Compositional Backwards Reachability (CBR), was first presented in [LNAB+98] where it was developed for a discrete model called state/event systems. The goal of this report is to continue the work from the previous report of extending the CBR method to a new domain of problems, namely verification of real-time models. When modeling continuous real-time the state-space becomes not just larger but in fact infinite. However, the infinite state-space can be reduced to a finite one by using symbolic techniques to represent and manipulate certain relevant subsets of the state-space. These subsets are known as Zones. This technique is well known, and implemented in the verification tool Uppaal [LPY97] that can verify safety and reachability properties of real-time models described as Timed Automata (TA). In the newest version Uppaal one can also verify certain general liveness properties. In this report we combine the notion of Zones with the CBR method to develop CBR for TA. Uppaal is developed in cooperation between Aalborg University and Uppsala university. The source code for Uppaal has been used as a basis in the development of a test implementation of the method described in this report.

1.2 Related Work

This section contains a discussion of related work. All the work mentioned in this section deals in some way with handling the state-space explosion problem. First we describe some techniques developed for verification of discrete systems. After this we discuss how these methods can or has been extended to apply to real-time systems. Some of the citations in this related work section has been found in [Kat98] by Joost-Pieter Katoen.

The main inspiration for this project is the article [LNAB+98], in which the CBR method is developed and applied to a discrete model. This, later patented, method was developed specifically for the industrial verification tool visualState™, which is used in the development of embedded systems. In this tool a number of predefined checks is performed on the model entered by the user. CBR outperformed not only the traditional forward analysis that was implemented in the tool, but also the current state of the art symbolic BDD-based methods. Models that could not be verified earlier because of the state-space explosion, can be verified using CBR. The strength of CBR is it's compositionality, which is closely linked to the fact that it performs backwards verification. This means that in many cases a much smaller part of the state-space has to be checked before a solution is found.

When verifying continuous real-time models the state-spaces to be analyzed are infinite. This can be handled by partitioning the continuous part of the state-space into so-called regions. Regions are subsets of the state-space, such that every pair of states form a region cannot be distinguished by the model. If a region is split in two, the two parts would be indistinguishable by the constraints and guards in the model. This creates a finite but very large state-space. A better solution is to represent convex unions, so-called Zones, of such regions. Theoretically there are even more Zones than regions, but a much smaller number of these will ever be considered in practice during analysis of real systems. The success of Zones depends on the efficient data-structure Difference Bounded Matrices (DBMs) used to represent the Zones and the efficient operations defined on this data-structure. This is the technology implemented in the tools Uppaal and Kronos [BDM+98].

Another technique that has significantly increased the size of discrete systems that can be verified is the Binary Decision Diagram (BDD) technique first introduced by Bryant [Bry86][K.L93]. The BDD technology has been extended to Clock Difference Diagrams (CDDs) to apply to the verification of real-time systems [BLP+99]. It gives reduction in the size of the state-space representation but not in the time used for verification.

Another technique to limit the state-space explosion problem is partial order reduction [God96]. Many different but equivalent interleavings are considered

at once, by only unfolding one of the interleavings, hereby reducing the state-space explosion. Partial order reduction has also been attempted applied to verification of real-time systems but without great success [Min99].
The first goal of this report is to generalize the CBR method such that it can be applied to many possible domains. The second goal of this report is to show the feasibility of creating a verification tool for Timed Automata (TA) based on the compositional backwards method for reachability analysis. This is done by combining the well known DBM technology with the CBR method in a test implementation. This implementation is then used to obtain some experimental data.

1.3 Relation to Previous Report and Outline

This section describes how this report is related to the previous report and gives an outline of the following chapters. The work in this report can be seen as an extension of the previous report, in which a generalization of the CBR method was presented. After this the domain of Timed Automata (TA) was described. Finally the CBR method was applied to the domain of TA. The CBR framework developed in the former report was not general enough and had to be adjusted, when applied to the domain of TA. Some parts of the previous report has been included without substantial changes. This includes chapters 3 and 4, and parts of chapter 5. This is the sections that describe the domain of TA. The CBR framework, which is completely redesigned, is presented in chapter 2. It generalizes the CBR method and shows the correctness of two different CBR algorithms, each applicable to their own type of domains. Chapter 3 defines the model of Timed Automata (TA). The symbolic analysis of TA and the algorithm implemented in Uppaal is described in chapter 4. In chapter 5 we proceed to apply the CBR method to the model of TA. Chapter 6 describes some extension to the model of TA, and how these would effect the CBR for TA method. The development of a test implementation of the CBR for TA method is described in chapter 7. This test implementation is used to obtain some experimental results, which are discussed in chapter 8. Future work and conclusion is included in chapter 9.

2 CBR in General

In this chapter we will firstly describe what backwards reachability analysis is. After this we will describe the concept of compositional backwards reachability (CBR). Section 2.3 contains the central definitions and theorems needed for the CBR algorithm. Two versions of the CBR algorithm will be presented, and the correctness of both will be proven. Finally the differences between the original CBR method and the one developed in this report, are described in section 2.5. In this section we also describe differences from the framework developed in the previous report [Lar02].

2.1 Reachability Analysis

In this section we describe the basic concept of reachability analysis. We define reachability and reachability analysis on general transition systems, where transition system is defined in the following manner.

Definition 2.1 : Transition System

$(ST, \longrightarrow)$, where ST is a set of states (finite or infinite) and $\longrightarrow \subseteq ST \times ST$ is a transition relation.

Each transition has a source state and a target state. The source state is the first ST component and the target state is the second ST component. If we have the transition $(s,t) \in \longrightarrow$ we write $s \longrightarrow t$. When writing a sequence of transitions we write $s_1 \longrightarrow s_2 \longrightarrow s_3$ instead of $s_1 \longrightarrow s_2, s_2 \longrightarrow s_3$. We now define what it means for a state to be reachable.

Definition 2.2 : Reachability

Given a set of initial states $Init \subseteq ST$ and goal states $Goal \subseteq ST$ we define $Goal$ to be reachable if there is a transition sequence

$$s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots \longrightarrow s_n$$

with $s_0 \in Init$ and $s_n \in Goal$.

There are two fundamental ways of deciding reachability: Forward and backwards. The two methods can also be combined. We describe each of the possibilities in the following three sections.

2.1.1 Forward

In the forward reachability analysis we start with the set of states $Init$ and iteratively compute the set of reachable states in steps as illustrated in figure 2.1.

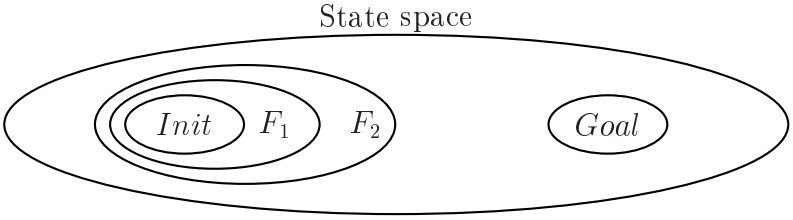

Figure 2.1: Forward Reachability Analysis

We use the following formulas to compute each new step:

$$\begin{aligned} F_0 &= Init \\ F_n &= F_{n-1} \cup \textsc{Post}(F_{n-1}) \text{ for } n > 0 \end{aligned}$$

where for $S \subseteq ST$, $\textsc{Post}(S) = \{s' \in ST \mid \exists s \in S.s \longrightarrow s'\}$.
If at any point $F_n \cap Goal \neq \emptyset$ we know that we have a sequence of transitions that can bring us from a state in $Init$ to a state in $Goal$. Hence we terminate with a positive answer. If we have $F_n = F_{n+1}$ we have reached a fix point and we know that no transitions can take us out of the set F_n. This leads us to terminate the algorithm with the answer that, there is no sequence of transitions that can bring us from $Init$ to $Goal$.

2.1.2 Backwards

The main difference between forward and backwards reachability analysis is the set of states that we start with. We again want to know if there is a sequence of transitions that can bring us from a state in $Init$ to a state in $Goal$, but this time we start with $Goal$ and compute the set of states that can

reach *Goal*. First the states that can reach *Goal* by taking one transition, then two transitions, and so on. This is illustrated in figure 2.2.

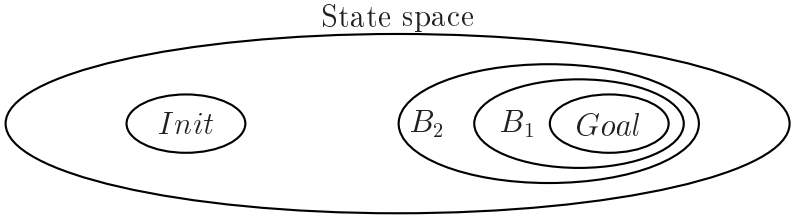

Figure 2.2: Backwards Reachability Analysis

In forward analysis, we know that all the states we explore are reachable states. In backwards analysis we can have both reachable and unreachable states in our set of states. We use the following formulas to calculate the steps:

$$\begin{aligned} B_0 &= Goal \\ B_n &= B_{n-1} \cup \textsc{Pre}(B_{n-1}) \end{aligned}$$

where for $S \subseteq ST$, $\textsc{Pre}(S) = \{s \in ST \mid \exists s' \in S.s \longrightarrow s'\}$.
Again we have two termination conditions. The algorithm terminates with a positive answer if, at any point, B_n and $Init$ intersect. The algorithm terminates with a negative answer if we reach a fix-point, $B_n = B_{n-1}$.

2.1.3 Combined

The two previously described methods can be combined, by doing forward and backwards reachability analysis in parallel. For each step we check for intersection between the two sets F_n and B_n. If these two sets intersect in a state s we know that we have a sequence of transitions leading from $Init$ to s and a sequence of transitions leading from s to *Goal*, and hence *Goal* is reachable from $Init$. This method may give a faster positive answer but has the same negative termination conditions as the other two methods. One of the sets has to reach a fix-point in order for us to conclude that *Goal* cannot be reached from $Init$.

2.2 The CBR Concept

In this section we will first give a short description of how CBR works. After this we give some intuition about why the method was developed.

Compositional Backwards Reachability (CBR) is based on traditional backwards reachability as presented in the previous section. It consists of a number of steps, each resembling one run of the conventional backwards reachability analysis. The result of each step is an under-approximations of the set of states that can reach *Goal*. After each step we check for intersection between the current under-approximation and *Init*. If the two sets intersect we have found a path leading from some state in *Init* to some state in *Goal*, and the algorithm terminates with a positive answer. If there is no intersection we have to calculate a new and larger under-approximation. This process is continued until the two sets intersect or the under-approximation no longer is an under-approximation, but the full set of states that can reach *Goal*. If the full set does not intersect with *Init* we know that no path exists from *Init* to *Goal* and the algorithm terminates with a negative answer.
An important factor in the efficiency of the method is the fact that the end result of one step can be used as the starting point of the next step. Figure 2.3 illustrates two such calculations of under-approximations. The end result of the first step, shown in the top part of the figure, is used as the starting point of the second step, shown in the bottom part of the figure.

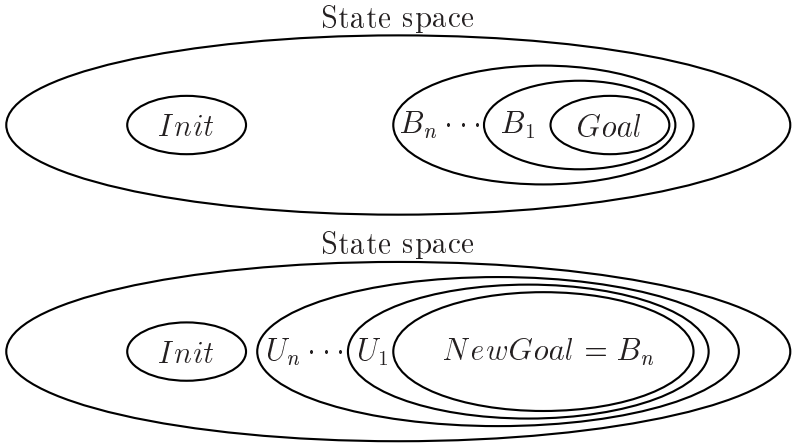

Figure 2.3: Calculation of two under-approximations.

This approach of stepwise under-approximation, was developed to cope with very large systems, consisting of many components in parallel. In the original domain of State/Event systems, each of the components, is in itself a transition system. The idea was, to only look at a subset of the components, and see if this subset could reach *Init* backwards, without involving the other components. If these components could not reach *Init* extra components were taken into consideration, until all components, or rather a dependency

closed set of components, were considered. The original CBR method used a concept of having an index set of the components. This index set was increased to give larger and larger under-approximations. This concept will in this report be replaced by a more general concept of partitioning the state-space into finer and finer partitions. The concept of partitioning will be the topic of the next section. The calculation of a series of under-approximations can theoretically lead to slower negative termination, but more likely also to a much faster positive termination. The efficiency of the method also depends on the fact that, these under-approximations can be represented and handled easily.

2.3 Partitioning

In this section we will develop the formal foundation for the CBR method. The idea is to use a succession of finer and finer partitionings of the state-space ST, to under-approximate the set of states that can reach $Goal$. By refinement of the partitioning, hence enlarging the under-approximation, we will get closer and closer to the full set of states that can reach $Goal$.
Intuitively partitioning can be described as splitting the state-space into a number of disjoint parts. For some domain we will have an infinite number of partitions in each partitioning. Formally we define a partitioning in the following way.

Definition 2.3 : PARTITIONING

$\mathcal{P} = \{St_i \mid i \in I\}$ is a partitioning of the state-space ST if the following three conditions hold:

1. $\bigcup\{St_i \mid i \in I\} = ST$
2. $\forall i \; St_i \neq \emptyset$
3. $St_i \cap St_j = \emptyset$ when $i \neq j$

One can talk of one partitioning being finer than another. We define a partial order on the set of all partitionings.

Definition 2.4 : Ordering of Partitionings

We say that $\mathcal{P}$ is finer than $\mathcal{Q}$ that is $\mathcal{P} \sqsubseteq \mathcal{Q}$ if

$$\forall i \in I \; \exists j \in J \; . \; St_i \subseteq St'_j$$

where $\mathcal{P} = \{St_i \mid i \in I\}$ and $\mathcal{Q} = \{St'_j \mid j \in J\}$

Figure 2.4 illustrates two partitionings $\mathcal{P}$ and $\mathcal{Q}$ where $\mathcal{P}$ is finer than $\mathcal{Q}$.

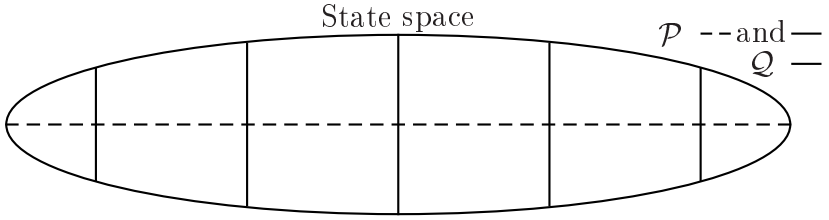

Figure 2.4: Two partitionings $\mathcal{P} \sqsubseteq \mathcal{Q}$

It is worth noticing that, because $\sqsubseteq$ is a partial order, not all pairs of partitionings can be ordered. When using the CBR method, we will start with an initial partitioning $\mathcal{P}_0$ and from that create finer and finer partitionings, until we have a partitioning, in which all states in each partition are bisimilar. This final partitioning $\mathcal{P}_{stable}$ will be defined later. The only requirement for the initial partitioning is that *Goal* is $\mathcal{P}_0$ sorted. The succession of partitionings could be written as follows: $\mathcal{P}_0 \sqsupseteq \mathcal{P}_1 \sqsupseteq \ldots \sqsupseteq \mathcal{P}_n \sqsupseteq \mathcal{P}_{stable}$
Now we define the notion of $\mathcal{P}$-equivalence.

Definition 2.5 : $\mathcal{P}$-equivalence

$s \sim_{\mathcal{P}} t \iff \forall i.(s \in St_i \iff t \in St_i)$ where $\mathcal{P} = \{St_i \mid i \in I\}$

The equivalence classes generated by a specific $\sim_{\mathcal{P}}$ equivalence, exactly follows the partitions of the corresponding partitioning $\mathcal{P}$.
We say that a subset H of the state-space is $\mathcal{P}$-sorted if, for the given partitioning $\mathcal{P}$, no partition intersects both with H and the complement of H. This is formally defined in the following way:

Definition 2.6 : $\mathcal{P}$-SORTEDNESS

Let $\mathcal{P} = \{St_i \mid i \in I\}$ and let $H \subseteq ST$. We say that H is $\mathcal{P}$-sorted if

$$\forall s, s' \in ST. s \in H \wedge s \sim_{\mathcal{P}} s' \Rightarrow s' \in H.$$

In correspondence with $\text{PRE}(S)$ defined in section 2.1.2 we define the $\mathcal{P}$-sorted predecessors.

Definition 2.7 : $\text{PRE}_{\mathcal{P}}(H)$

$$\text{PRE}_{\mathcal{P}}(H) = \{s \in ST \mid \forall t \sim_{\mathcal{P}} s. \exists t' \in H. t \longrightarrow t'\}$$

This means that if one state t can take a transition into H, then every other state that is $\sim_{\mathcal{P}}$ equivalent with t must also be able to take a transition into H, before this partition, $\sim_{\mathcal{P}}$ equivalence class, is included in $\text{PRE}_{\mathcal{P}}(H)$. We illustrate this in figure 2.5. As this definition imposes an extra condition, in comparison with the original PRE, it can only return a set of states that is equal to, or smaller than, the set of states returned by the original predecessor function. This means that we obtain an under-approximation.

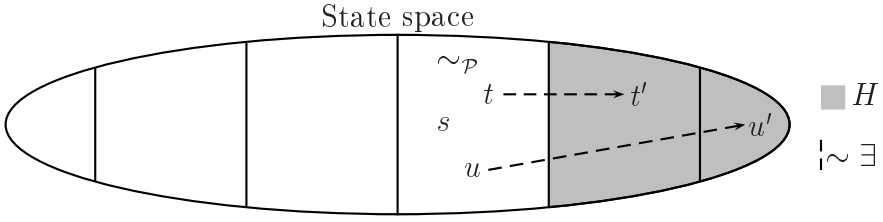

Figure 2.5: Illustration of $\text{PRE}_{\mathcal{P}}(H)$.

Here it is not necessary to require that H is $\mathcal{P}$-sorted. This requirement will be added when extending it to the $\text{PRE}^*_{\mathcal{P}}(H)$ function, where it is needed because H itself is included in the result.
This definition actually gives us one predecessor function for each partitioning $\mathcal{P}$. We will in turn use each of these functions in our CBR algorithm as we refine the partitionings.

Lemma 2.8

$\text{PRE}_{\mathcal{P}}(H)$ is $\mathcal{P}$-sorted

The proof of lemma 2.8 is very simple.

Proof of of lemma 2.8
We show that whenever $s \in \text{PRE}_{\mathcal{P}}(H)$ and $t \sim_{\mathcal{P}} s$ then $t \in \text{PRE}_{\mathcal{P}}(H)$. Thus we must show that $u \longrightarrow u'$ for some $u' \in H$ whenever $u \sim_{\mathcal{P}} t$. Because $\sim_{\mathcal{P}}$ is an equivalence relation, whenever we have that $u \sim_{\mathcal{P}} t$ (and $t \sim_{\mathcal{P}} s$) we can also conclude that $u \sim_{\mathcal{P}} s$, and as $s \in \text{PRE}_{\mathcal{P}}(H)$ it follows that indeed $u \longrightarrow u'$ for some $u' \in H$.

The following lemma states that the $\text{PRE}_{\mathcal{P}}$ operation is monotonic with regard to the input set.

Lemma 2.9

$\text{PRE}_{\mathcal{P}}(H_1) \subseteq \text{PRE}_{\mathcal{P}}(H_2)$ *if* $H_1 \subseteq H_2$

The proof of lemma 2.9 only uses the definition of $\text{PRE}_{\mathcal{P}}$ and the fact that $H_1 \subseteq H_2$.

Proof of lemma 2.9
Let $s \in \text{PRE}_{\mathcal{P}}(H_1)$. Then $t \longrightarrow t'$ for some $t' \in H_1$, for all $t \sim_{\mathcal{P}} s$. But as $H_2 \supseteq H_1$ we also have that $t \longrightarrow t'$ for some $t' \in H_2$ for all $t \sim_{\mathcal{P}} s$, and hence $s \in \text{PRE}_{\mathcal{P}}(H_2)$.

2.3.1 Central Theorems

In this section we will present the three central theorem: 2.15, 2.16, and 2.18. These theorems are needed in the construction of the CBR algorithm. First we present lemmas 2.10 and 2.13, which are simpler versions of respectively theorem 2.15 and theorem 2.16. After this the $\text{PRE}^*_{\mathcal{P}}$ operation will be introduced and the three final lemmas will be proved.
In all of the following we will use two partitionings $\mathcal{P}$ and $\mathcal{Q}$. The partitioning $\mathcal{P}$ is finer than $\mathcal{Q}$ which can be expressed in the following way; $\mathcal{P} \sqsubseteq \mathcal{Q}$.

Lemma 2.10

$\text{PRE}_{\mathcal{Q}}(H) \subseteq \text{PRE}_{\mathcal{P}}(H)$ *where* $\mathcal{P} \sqsubseteq \mathcal{Q}$

Lemma 2.10 states that $\text{PRE}_{\mathcal{P}}(H)$ is monotonic with regard to how fine the partitioning $\mathcal{P}$ is. If the partitioning becomes finer the resulting set can only become larger. This follows the intuition that if we refine the partitioning the under-approximation becomes better.

Proof of lemma 2.10

Let $s \in \text{PRE}_{\mathcal{Q}}(H)$. Then $t \longrightarrow t'$ for some $t' \in H$, for all $t \sim_{\mathcal{Q}} s$. However as $\mathcal{P}$ is a finer partitioning than $\mathcal{Q}$, giving us $\sim_{\mathcal{P}} \subseteq \sim_{\mathcal{Q}}$, we also have that $t \longrightarrow t'$ for some $t' \in H$, for all $t \sim_{\mathcal{P}} s$. Thus $s \in \text{PRE}_{\mathcal{P}}(H)$.

For the next central lemma we need the following lemma.

Lemma 2.11

$\text{PRE}_{\mathcal{P}} \subseteq \text{PRE}(H)$

Lemma 2.11 states that any under-approximation of the set of predecessors will be included in the full set of predecessors.

Proof of lemma 2.11

Assume $s \in \text{PRE}_{\mathcal{P}}(H)$. Then $t \longrightarrow t'$ with $t' \in H$ for all $t \sim_{\mathcal{P}} s$. In particular, $s \longrightarrow s'$ for some $s' \in H$ (because $s \sim_{\mathcal{P}} s$). Then $s \in \text{PRE}(H)$.

For the next lemma we need a partitioning $\mathcal{P}_{stable}$, with a certain property. For this we first need to define the bisimulation $\mathcal{B}_{Goal}$, which depends on the transition system and the set of goal states *Goal*.

Definition 2.12 : BISIMULATION $\mathcal{B}_{Goal}$

$\mathcal{B}_{Goal}$ is a bisimulation if whenever $(s,t) \in \mathcal{B}_{Goal}$ then the following three conditions hold:

1. $s \in Goal \iff t \in Goal$
2. $s \longrightarrow s' \Rightarrow \exists t'.t \longrightarrow t' \wedge (s',t') \in \mathcal{B}_{Goal}$
3. $t \longrightarrow t' \Rightarrow \exists s'.s \longrightarrow s' \wedge (s',t') \in \mathcal{B}_{Goal}$

This defines all pairs of states that are bisimilar. We call a partitioning stable if all states in each partitioning are bisimilar. This does not describe exactly

one partitioning. For a given domain, the requirement is that we find one partitioning $\mathcal{P}_{stable}$ with the give property. An example of a relation that gives us a stable partitioning is the identity relation $Id = \{(s,s) \mid s \in ST\}$.

Lemma 2.13

$\textsc{Pre}_{\mathcal{P}_{stable}}(H) = \textsc{Pre}(H)$ *where* H *is* $\mathcal{P}_{stable}$ *sorted*

Lemma 2.13 is needed in order to guarantee that, if we partition the state-space down to bisimulation, we will get the same result as with traditional backwards reachability analysis.

Proof of lemma 2.13

We split the proof into two parts:

$\subseteq$: follows from lemma 2.11.

$\supseteq$: Let $s \in \textsc{Pre}(H)$. Then $s \longrightarrow s'$ with $s' \in H$. Now let $t \approx s$. Then $t \longrightarrow t'$ for some t' with $t' \approx s'$ (by definition of $\approx$). But H was assumed to be $\mathcal{P}_{stable}$ sorted and hence $t' \in H$. It follows that $s \in \textsc{Pre}_{\mathcal{P}_{stable}}(H)$.

We now define the $\textsc{Pre}^*_{\mathcal{P}}$ operation and prove lemmas 2.10 and 2.13 in their new form. Intuitively $\textsc{Pre}^n_{\mathcal{P}}(H)$ is all the states that can reach H in exactly n transitions. From this we define $\textsc{Pre}^*_{\mathcal{P}}(H)$ to be all the states, that can reach H in any number of transitions.

Definition 2.14 : $\textsc{Pre}^*_{\mathcal{P}}(H)$

For any given partitioning $\mathcal{P}$ we define $\textsc{Pre}^*_{\mathcal{P}}(H)$ as the union:

$$\textsc{Pre}^*_{\mathcal{P}}(H) = \bigcup_{n=0}^{\infty} \textsc{Pre}^n_{\mathcal{P}}(H)$$

where $\textsc{Pre}^n_{\mathcal{P}}$ is defined recursively by the following formulas:

$$\begin{aligned} \textsc{Pre}^0_{\mathcal{P}}(H) &= H \\ \textsc{Pre}^{n+1}_{\mathcal{P}}(H) &= \textsc{Pre}_{\mathcal{P}}(\textsc{Pre}^n_{\mathcal{P}}(H)) \end{aligned}$$

and H is required to be $\mathcal{P}$-sorted.

The above definition can also be applied to the original predecessor function $\textsc{Pre}(H)$ to obtain a definition of $\textsc{Pre}^n(H)$ and $\textsc{Pre}^*(H)$. The notation $\textsc{Pre}^*(H)$ will be used in theorem 2.18 and $\textsc{Pre}^n(H)$ will be used in the proof of theorem 2.18. For $\textsc{Pre}^*_{\mathcal{P}}(H)$ to be $\mathcal{P}$-sorted we have to require that H is $\mathcal{P}$-sorted because H itself is included in $\textsc{Pre}^*_{\mathcal{P}}(H)$.
The following theorem states that whenever we refine the partitioning we will obtain an under-approximation that is a superset of the previous one. The left side uses the coarser partitioning $\mathcal{Q}$ and this set is a subset of the right side that uses the finer partitioning $\mathcal{P}$.

Theorem 2.15

$\textsc{Pre}^*_{\mathcal{Q}}(H) \subseteq \textsc{Pre}^*_{\mathcal{P}}(H)$ *where* $\mathcal{P} \sqsubseteq \mathcal{Q}$

The proof of theorem 2.15, which is an induction proof, uses lemma 2.10 and lemma 2.9. Lemma 2.10 is similar to theorem 2.15 except for the stars, while lemma 2.9 states that the $\textsc{Pre}_{\mathcal{P}}(H)$ operation is monotonic with regard to the input set H.

Proof of theorem 2.15 ■

The theorem obviously follows from the from the fact that $\textsc{Pre}^n_{\mathcal{Q}}(H) \subseteq \textsc{Pre}^n_{\mathcal{P}}(H)$ for all n. We prove this by induction in n.

Basis $n = 0$**:** By the definition of $\textsc{Pre}^n_{\mathcal{P}}$ we can rewrite $\textsc{Pre}^0_{\mathcal{Q}}(H) \subseteq \textsc{Pre}^0_{\mathcal{P}}(H)$ to $H \subseteq H$ which is obviously true.

Step: Assume $\textsc{Pre}^n_{\mathcal{Q}}(H) \subseteq \textsc{Pre}^n_{\mathcal{P}}(H)$ (IH) then

$$\begin{aligned} \textsc{Pre}^{n+1}_{\mathcal{Q}}(H) = \textsc{Pre}_{\mathcal{Q}}(\textsc{Pre}^n_{\mathcal{Q}}(H)) &\subseteq \text{ by lemma 2.9 and (IH)} \\ \textsc{Pre}_{\mathcal{Q}}(\textsc{Pre}^n_{\mathcal{P}}(H)) &\subseteq \text{ by lemma 2.10} \\ \textsc{Pre}_{\mathcal{P}}(\textsc{Pre}^n_{\mathcal{P}}(H)) &= \textsc{Pre}^{n+1}_{\mathcal{P}}(H). \end{aligned}$$

Now we have proven that $\forall n.\textsc{Pre}^n_{\mathcal{Q}}(H) \subseteq \textsc{Pre}^n_{\mathcal{P}}(H)$.

■ ■

We now extend lemma 2.13 by replacing $\textsc{Pre}_{\mathcal{P}_{stable}}$ with $\textsc{Pre}^*_{\mathcal{P}_{stable}}$ and $\textsc{Pre}$ with $\textsc{Pre}^*$.

Theorem 2.16

$\textsc{Pre}^*_{\mathcal{P}_{stable}}(H) = \textsc{Pre}^*(H)$ *where* H *is* $\mathcal{P}$*-sorted*

Theorem 2.16 now states that no matter how many backwards steps we take, with the final partitioning $\mathcal{P}_{stable}$ of the state-space, the result will never

differ from that obtained using the original predecessor function. The proof for theorem 2.16 follows the structure of proof for theorem 2.15.

Proof of theorem 2.16

The theorem clearly follows from the fact that $\textsc{Pre}^n_{\mathcal{P}_{stable}}(H) = \textsc{Pre}^n(H)$ for all n. We prove this by induction in n.

Basis $n = 0$**:** By the definition of $\textsc{Pre}^n_{\mathcal{P}}$ we can rewrite $\textsc{Pre}^0_{\mathcal{P}_{stable}}(H) = \textsc{Pre}^0(H)$ to $H = H$ which is obviously true.

Step: Assume $\textsc{Pre}^n_{\mathcal{P}_{stable}}(H) = \textsc{Pre}^n(H)$ (IH) then

$$\begin{aligned} \textsc{Pre}^{n+1}_{\mathcal{P}_{stable}}(H) &= \\ \textsc{Pre}_{\mathcal{P}_{stable}}(\textsc{Pre}^n_{\mathcal{P}_{stable}}(H)) &= \text{by lemma 2.13 and (IH)} \\ \textsc{Pre}_{\mathcal{P}}(\textsc{Pre}^n_{\mathcal{P}}(H)) &= \textsc{Pre}^{n+1}_{\mathcal{P}}(H). \end{aligned}$$

Hence we have proved that $\textsc{Pre}^n_{\mathcal{P}_{stable}}(H) = \textsc{Pre}^n(H)$ holds for all n.

For the final central theorem we need an extended version of the monotonicity lemma 2.9, which uses $\textsc{Pre}^*_{\mathcal{P}}$ instead of $\textsc{Pre}_{\mathcal{P}}$.

Lemma 2.17

$\textsc{Pre}^*_{\mathcal{P}}(H_1) \subseteq \textsc{Pre}^*_{\mathcal{P}}(H_2)$ *when* $H_1 \subseteq H_2$

As for the previous proof, the proof of lemma 2.17 follows the structure of the proof for theorem 2.15.

Proof of lemma 2.17

The lemma follows clearly from the fact that $\textsc{Pre}^n_{\mathcal{P}}(H_1) \subseteq \textsc{Pre}^n_{\mathcal{P}}(H_2)$ for all n whenever $H_1 \subseteq H_2$. We prove this by induction in n.

Basis $n = 0$**:** By the definition of $\textsc{Pre}^n_{\mathcal{P}}$ we can rewrite $\textsc{Pre}^0_{\mathcal{P}}(H_1) = \textsc{Pre}^0_{\mathcal{P}}(H_2)$ to $H_1 \subseteq H_2$ which is obviously true.

Step: Assume $\textsc{Pre}^n_{\mathcal{P}}(H_1) \subseteq \textsc{Pre}^n_{\mathcal{P}}(H_2)$ (IH) then

$$\begin{aligned} \textsc{Pre}^{n+1}_{\mathcal{P}}(H_1) &= \\ \textsc{Pre}_{\mathcal{P}}(\textsc{Pre}^n_{\mathcal{P}}(H_1)) &\subseteq \text{by monotonicity (lemma 2.9) and (IH)} \\ \textsc{Pre}_{\mathcal{P}}(\textsc{Pre}^n_{\mathcal{P}}(H_2)) &= \textsc{Pre}^{n+1}_{\mathcal{P}}(H_2). \end{aligned}$$

This is exactly lemma 2.9 and hence we have proved that $\textsc{Pre}^n_{\mathcal{P}}(H_1) \subseteq \textsc{Pre}^n_{\mathcal{P}}(H_2)$ holds for all n when $H_1 \subseteq H_2$.

The following and final of the three central lemmas lets us reuse states that has already been calculated. If one set of predecessors has been calculated using partitioning $\mathcal{Q}$, it can be used as the starting point of the next calculation using a finer partitioning $\mathcal{P}$.

Theorem 2.18

$\textsc{Pre}^*_{\mathcal{P}}(H) = \textsc{Pre}^*_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))$ *where* $\mathcal{P} \sqsubseteq \mathcal{Q}$

The following proof consists of two parts, where the second of them once again follows the structure of the proof for theorem 2.15.

Proof of theorem 2.18 ■

We want to prove that $\textsc{Pre}^*_{\mathcal{P}}(H) = \textsc{Pre}^*_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))$ for two partitionings $\mathcal{P}$ and $\mathcal{Q}$ where $\mathcal{P} \sqsubseteq \mathcal{Q}$. We consider the two inclusions.

$\subseteq$: Obviously $H \subseteq \textsc{Pre}^*_{\mathcal{Q}}(H)$ and $\textsc{Pre}^*_{\mathcal{P}}$ is monotonic with respect to the input set according to lemma 2.17. Hence we can conclude that $\textsc{Pre}^*_{\mathcal{P}}(H) \subseteq \textsc{Pre}^*_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))$.

$\supseteq$: The inclusion follows from the fact that $\textsc{Pre}^*_{\mathcal{P}}(H) \supseteq \textsc{Pre}^n_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))$ for all n. We prove this by induction in n.

Basis $n = 0$**:** In this case $\textsc{Pre}^*_{\mathcal{P}}(H) \supseteq \textsc{Pre}^0_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))$ reduces to $\textsc{Pre}^*_{\mathcal{P}}(H) \supseteq \textsc{Pre}^*_{\mathcal{Q}}(H)$ which is exactly theorem 2.15.

Step: Assume $\textsc{Pre}^*_{\mathcal{P}}(H) \supseteq \textsc{Pre}^n_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))$ (IH) then

$$\begin{aligned} \textsc{Pre}^{n+1}_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H)) &= \\ \textsc{Pre}_{\mathcal{P}}(\textsc{Pre}^n_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))) &\subseteq \text{ by monotonicity (lemma 2.9) and (IH)} \\ \textsc{Pre}_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{P}}(H)) &\subseteq \textsc{Pre}^*_{\mathcal{P}}(H) \end{aligned}$$

The last inclusion follows from the fact that we have a finite partitioning of the state-space. So $\textsc{Pre}^*_{\mathcal{P}}(H)$ will converge within finitely many steps.

Hence we have proved that $\forall n.\textsc{Pre}^*_{\mathcal{P}}(H) \supseteq \textsc{Pre}^n_{\mathcal{P}}(\textsc{Pre}^*_{\mathcal{Q}}(H))$.

Having proved the two inclusions we can conclude the correctness of theorem 2.18.

■ ■

Having proved the three central theorems 2.15, 2.16, and 2.18 we can now give the CBR algorithm and prove the correctness of it.

2.4 CBR Algorithms

In this section we will present two versions of the CBR algorithm. The first algorithm, which is also the simplest, resembles the original algorithm from the paper [LNAB+98]. The second one is needed when only certain subsets of the state-space can be represented efficiently. Such subsets are called representable symbolic states and are explained in section 2.4.2. The revised, second algorithm is described in the last subsection.

2.4.1 Simple Algorithm

In this section we present and prove the correctness of the simple CBR algorithm. The algorithm is shown in figure 2.6. The input for the algorithm is a transition system $(ST, \longrightarrow)$ and two sets of states *Goal* and $Init$, such that *Goal* and $Init$ both are subsets of the state-space ST.

```
REACHABLE((ST, ⟶), Goal, Init)
Select 𝒫 such that Goal is 𝒫-sorted
R ← Goal
repeat
  R_new ← PRE*_𝒫(R)

  /* Check for early positive termination. Theorem 2.15 */
  if Init ∩ R_new ≠ ∅ then return TRUE

  /* Check for negative termination. Theorem 2.16 */
  if 𝒫 = 𝒫_stable then return FALSE

  𝒫 ← 𝒫' such that 𝒫_stable ⊑ 𝒫' ⊑ 𝒫.

  /* Reuse of previously computed states. Theorem 2.18 */
  R ← R_new
forever
```

Figure 2.6: Original CBR algorithm

The algorithm gives a formal definition of the procedure that was described in section 2.2. First the initial partitioning is selected such that *Goal* is $\mathcal{P}$-sorted. After this *Goal* is assigned to R. The two variables R and R_{new}

contain unions of partitions from the currently used partitioning $\mathcal{P}$ and any of the previously used, coarser partitionings. In the top of the loop the new contents of R_{new} is calculated from R using the current partitioning. The result is then tested for intersection with $Init$, for positive termination. If we have reached the final partitioning $\mathcal{P}_{stable}$ we terminate with a negative answer, else we select a finer partitioning, that is still no finer than $\mathcal{P}_{stable}$. R_{new} is then assigned to R and we start from the top again. The loop is repeated until one of the two termination conditions is fulfilled.

Correctness

We want to conclude that the algorithm is correct. This consist of two parts; concluding that it always terminates and that it terminates with the correct answer.
To conclude that the algorithm always terminates we first need to conclude, that we can only run through the loop finitely many times. This is guaranteed by having a finite number of partitionings that are used. This is a requirement that must be taken care of for each domain to which the method is applied. The finite number of partitions guarantee that we, at some point, will end up with $\mathcal{P} = \mathcal{P}_{stable}$ and terminate with a negative answer. Secondly we require that $\textsc{Pre}^*_{\mathcal{P}}(R)$ can be computed effectively and hence always terminates. If these two requirements are fulfilled, for the domain to which the method is applied, we can conclude that the algorithm always terminates.

Now we will prove that the algorithm will terminate with the correct answer. Throughout computation $\mathcal{P}$ assumes a sequence of values $\mathcal{P}_0 \sqsubseteq \mathcal{P}_1 \sqsubseteq \ldots \sqsubseteq \mathcal{P}_n = \mathcal{P}_{stable}$. Similarly R_{new} assumes a sequence of values $R^0_{new}, R^1_{new}, \ldots, R^n_{new}$. We claim that $\forall i. R^i_{new} = \textsc{Pre}^*_{\mathcal{P}_i}(Goal)$.
We prove this by induction in i.

Basis $i = 0$**:** The first time we enter the loop we have that $R = Goal$ and $\mathcal{P} = \mathcal{P}_0$. R_{new} is given to be exactly $\textsc{Pre}^*_{\mathcal{P}}(R)$ so $R^0_{new} = \textsc{Pre}^*_{\mathcal{P}_0}(Goal)$.

Step: Assume $R^n_{new} = \textsc{Pre}^*_{\mathcal{P}_n}(Goal)$ (IH) then

$$\begin{aligned} R^{n+1}_{new} &= \textsc{Pre}^*_{\mathcal{P}_{n+1}}(R^n_{new}) \\ &= \textsc{Pre}^*_{\mathcal{P}_{n+1}}(\textsc{Pre}^*_{\mathcal{P}_n}(Goal)) \text{ by IH} \\ &= \textsc{Pre}^*_{\mathcal{P}_{n+1}}(Goal) \text{ by theorem 2.18 and } \mathcal{P}_{n+1} \sqsubseteq \mathcal{P}_n \end{aligned}$$

By having shown this we can conclude by theorem 2.16 that for the final partitioning $\mathcal{P}_n = \mathcal{P}_{stable}$ we have $R^n_{new} = \textsc{Pre}^*_{\mathcal{P}_{stable}}(Goal) = \textsc{Pre}^*(Goal)$. So if there is a path leading from $Init$ to $Goal$ the check for intersection

$Init \cap R_{new}$ will guarantee that the algorithm terminates with a positive answer. Now we need to argue that the algorithm cannot terminate with a positive answer if there is no path. The only way the algorithm can terminate with a positive answer is if R_{new} intersects with $Init$ so by proving that $\forall i.R^i_{new} \subseteq \text{PRE}^*(Goal)$. This can be concluded from the previous proof and theorem 2.15.

Requirements

The requirements that this algorithm enforces on a domain, to which it can be applied, are the following; a transition system, with a stable partitioning $\mathcal{P}_{stable}$, a finite sequence of partitionings of the state-space, and an efficiently calculable predecessor function, for each partitioning.
For the algorithm to work efficiently there are some extra requirements. Firstly the possibility of representing arbitrary unions of partitions efficiently, such that the predecessor function can be computed directly on the representation yielding a new union of partitions. Secondly an efficient way of checking for intersection between such a representation and $Init$. Finally the partitionings should be made in a sensible way, such that there is a chance, that intersection can be obtained without always reaching the finest possible partitioning $\mathcal{P}_{stable}$.

2.4.2 Symbolic States

The purpose of this section is to motivate the need for the revised algorithm presented in the next section, and define the concept of representable symbolic state, used in the revised algorithm.

In some domains, in particular the domain of Timed Automata, to which we will apply the CBR method, it is only possible to efficiently represent certain subsets of the state-space. We will call these subsets of the state-space representable symbolic states RSS. Furthermore the predecessor function operates on one such representable symbolic state at a time, and gives as result a list of representable symbolic states. This does not directly fit the framework of the simple algorithm, because the assumption here is that all predecessors can be calculated in one step.
The idea is that the symbolic state represents a set of concrete states. There is no restriction on how many concrete states a symbolic state can represent. This depends entirely on the domain. In fact, in the domain of Timed Automata, each symbolic state represents an infinite set of symbolic states. A set of representable symbolic states must satisfy the following properties:

Assumption 2.19 : Representable Symbolic States RSS

A set of representable states $RSS \subseteq \mathcal{P}(ST)$ must have the following characteristics:

- RSS must be finite.
- $Goal$ can be represented as a union of representable symbolic states.

$$Goal = \bigcup_{i \in I} J_i$$

such that $J_i \in RSS$ and I is finite.

When using representable symbolic states we will need a predecessor function that from one symbolic state $\mathcal{J}$ delivers as output a finite set of representable symbolic states. In the following we formally state the obvious extra requirement that the output set of symbolic states must cover the set of states that the original predecessor function would have given.

Assumption 2.20 : Requirements for $\textsc{SymPre}_{\mathcal{P}}(\mathcal{J})$

Given a symbolic state $\mathcal{J} \in SST$ and a partitioning $\mathcal{P}$ the following must hold:

$$\textsc{SymPre}_{\mathcal{P}}(\mathcal{J}) = \{\mathcal{J}_1, \ldots, \mathcal{J}_m\}$$

$$\Downarrow$$

$$\bigcup_{i=1}^{m} \mathcal{J}_i \subseteq \textsc{Pre}_{\mathcal{P}}(\mathcal{J})$$

For the final partitioning $\mathcal{P}_{stable}$ the following must also hold:

$$\textsc{SymPre}_{\mathcal{P}_{stable}}(\mathcal{J}) = \{\mathcal{J}_1, \ldots, \mathcal{J}_m\}$$

$$\Downarrow$$

$$\bigcup_{i=1}^{m} \mathcal{J}_i = \textsc{Pre}_{\mathcal{P}_{stable}}(\mathcal{J})$$

```
REACHABLE((ST, ⟶), Init, Goal)
Select 𝒫 such that Goal is 𝒫-sorted
WAIT ← Goal
repeat
    PASSED := {}
    repeat
      begin
        get symbolic state 𝒥 from WAIT
        if 𝒥 ∩ Init ≠ ∅ then return TRUE
        else if 𝒥 ⊈ 𝒥' for all 𝒥' ∈ PASSED then
          begin
            add 𝒥 to PASSED
            NEXT := SYMPRE_𝒫(𝒥)
            for all 𝒥 in NEXT do
                  put 𝒥 to WAIT
          end
      end
    until WAIT = {}
    if 𝒫 = 𝒫_stable then return FALSE
    𝒫 ← 𝒫' such that 𝒫_stable ⊑ 𝒫' ⊑ 𝒫.
    WAIT := PASSED
forever
```

Figure 2.7: Symbolic State CBR algorithm

2.4.3 Symbolic State Algorithm

In this section we present a different version of the CBR algorithm using the concept of representable symbolic states described in the previous section. The input for the revised algorithm is much like the input of the original one. Now the two arguments *Goal* and *Init* have to be subsets of the set of representable symbolic states *RSS*.

Correctness

We want to conclude that the symbolic algorithm is correct. This, again consists of two parts; concluding that it always terminates and that it terminates with the correct answer.

This algorithm has two loops, with the one inside the other. We will call these the inner and the outer loop respectively. The outer loop is a repeat-forever loop, so the only way this loop can terminate is by the algorithm finishing, by returning either true or false. We use the same argument as for the simpler algorithm, that we have chosen a finite sequence of partitionings. This guarantees that after a finite number of runs through the outer loop we will eventually reach $\mathcal{P} = \mathcal{P}_{stable}$, and the algorithm will terminate. We also require that the inner loop terminates, in each iteration of the outer loop. This can be guaranteed due to the fact that we have a finite number of representable symbolic states. In the inner loop representable symbolic states are removed from the waiting list and added to the passed list while every representable symbolic state (RSS) that could reach this RSS are added to the waiting list for later exploration. Once a representable symbolic state has been added to the passed list, it will not be explored again. This process continues until the waiting list is empty. If not earlier, this is at least guaranteed to happen when all representable symbolic states have been added to the passed list. So we are guaranteed that the inner loop will always terminate.

Now we turn to proving that the algorithm will terminate with the correct answer when it terminates. Again we want to do this by induction. Just after exiting the inner loop, where WAIT will always be empty, we will prove that $Goal \subseteq \textsc{Passed} \subseteq \textsc{Pre}^*_{\mathcal{P}_n}(Goal)$ for the current partitioning $\mathcal{P}_n$ being used. We do this by induction in n.

Basis $n = 0$**:** Just after exiting the inner loop for the first time the following will hold:

$$\textsc{Passed} = \{\ \mathcal{J} \mid \mathcal{J} \longrightarrow^*_{\mathcal{P}_0} \mathcal{J}' \wedge \mathcal{J}' \in Goal\}$$

where $\mathcal{J} \longrightarrow_{\mathcal{P}} \mathcal{J}'$ means that $\mathcal{J} \in \textsc{SymPre}_{\mathcal{P}}(\mathcal{J}')$ ($\longrightarrow^*_{\mathcal{P}}$ denotes as usual the transitive and reflexive closure of $\longrightarrow_{\mathcal{P}}$). The passed list here contains all the symbolic states needed to represent *Goal* and all the symbolic states that can reach *Goal* using the partitioning $\mathcal{P}_0$. By iteratively applying the requirement for the symbolic predecessor function stated in assumption 2.20, we can conclude that the following holds:

$$Goal \subseteq \textsc{Passed} \subseteq \textsc{Pre}^*_{\mathcal{P}_0}(Goal)$$

Step: On entering the inner loop the waiting list will contain representable symbolic states such that $Goal \subseteq \textsc{Wait} \subseteq \textsc{Pre}^*_{\mathcal{P}_n}(Goal)$, where $\mathcal{P}_n$

is the previous partitioning. All of these states will, after some iterations in the inner loop, be added to the passed list, such that $Goal \subseteq \textsc{Passed} \subseteq \textsc{Pre}^*_{\mathcal{P}_n}(Goal)$, this is the induction hypothesis (IH). We now aim to prove that after exiting the inner loop the following will hold:

$$Goal \subseteq \textsc{Passed} \subseteq \textsc{Pre}^*_{\mathcal{P}_{n+1}}(Goal)$$

After exiting the inner loop we can clearly see that the following will hold.

$$\textsc{Passed} = \{\ \mathcal{J} \mid \mathcal{J} \longrightarrow^*_{\mathcal{P}_{n+1}} \mathcal{J}' \wedge \mathcal{J}' \in \textsc{Pre}^*_{\mathcal{P}_n}(Goal)\}$$

Again iteratively applying assumption 2.20 we can conclude the following:

$$Goal \subseteq \textsc{Passed} \subseteq \textsc{Pre}^*_{\mathcal{P}_{n+1}}(Goal)$$

By having proved that $Goal \subseteq \textsc{Passed} \subseteq \textsc{Pre}^*_{\mathcal{P}_n}(Goal)$ after each iteration we can conclude that the passed list will always contain at least $Goal$ and it will never contain any states that cannot reach $Goal$. If any of the representable symbolic states used to represent Passed intersect with $Init$, the algorithm would have terminated with a positive answer, when this state was being explored. We will eventually reach the final partitioning $\mathcal{P}_{stable}$. Because of the special assumption made for $\mathcal{P}_{stable}$, in assumption 2.20, we can conclude that after the final iteration $\textsc{Passed} = \textsc{Pre}^*_{\mathcal{P}_{stable}}(Goal)$. $\textsc{Pre}^*_{\mathcal{P}_stable}(Goal)$ is by theorem 2.16 equal to $\textsc{Pre}^*(Goal)$. Hence after the last run of the inner loop Passed will contain representable symbolic states that covers exactly all states that can reach $Goal$. This results in the fact that if any of these states intersects with $Init$, the algorithm would have terminated with a positive answer. Similarly the passed list does not contain more than what can actually reach $Goal$, and hence the algorithm will never terminate with a positive answer, when there is no path from $Init$ to $Goal$.

Requirements

The requirements that this algorithm enforces on a domain, to which it can be applied, are the following: A transition systems $(ST, \longrightarrow)$, a finite set of representable symbolic states RSS, and a symbolic predecessor function SymPre that fulfills the requirement of assumption 2.20. Again we also need a way in which to check for intersection between any representable symbolic state $\mathcal{J}$ and $Init$ and inclusion between two representable symbolic states. We also need sensible partitionings of the representable symbolic state-space.

2.5 Differences from the Original CBR

This section describes the differences between the original CBR method, [LNAB+98], and the CBR method presented in this report. It also describes differences from the CBR method that was developed in the previous report [Lar02].
The main difference is the concept on which the formal foundation is build. The original paper uses an index set of machines that is gradually increased, while we here use a partitioning of the state-space where the partitionings are gradually refined. Despite this difference, the simple algorithm presented in figure 2.6 closely resembles the original algorithm. The second algorithm adds more generality to the method by allowing the use of symbolic states. This makes the CBR method applicable to other types of domains. In the previous report the CBR method was generalized by having more the one index set, each representing one type of components. The back draw of this method was that the CBR method could not be presented once and for all and then applied to different domains. It had to be adjusted depending on the types of the components being used in each domain.
An aspect of the original method that has been lost is dependency analysis. In the original domain, a dependency analysis was performed on the State/Event machines in order to determine if all of the machines where needed in the analysis. If it could be concluded, that some of the machines could in no way, effect the reachability of $Goal$, these machines could be excluded from the analysis. Thereby leading to a faster negative termination. The concept of dependency analysis is not incorporated into the general CBR method because it depends very much on the specific domain. The dependency analysis works on components, and by analyzing what components can influence the set of components that we start with, we can stop before including all components. This would, in the new formalism, correspond to stopping at a earlier partitioning than $\mathcal{P}_{stable}$. Maybe this kind of feature could be added if extra information were added to the framework.

3 Timed Automata (TA)

This chapter contains the definition of networks of simple timed automata. By simple timed automata we mean timed automata without invariants, committed locations, urgency, and integers as are allowed in UPPAAL. First we present an informal description of timed automata. After this we formally describe the syntax and semantics of a single timed automaton. In the end we describe the syntax and semantics of the parallel composition of several time automata into a Timed Automata Network (TAN).

3.1 Informal Description

Timed automata are finite state automata extended with a number of real valued clocks. Graphically a timed automaton can be depicted as nodes with arrows going from one node to another when there is a transition. We write constraints (also known as guards) at the origin of a transition and reset sets at the destination of the transition. At the center of the arrow we write the label.

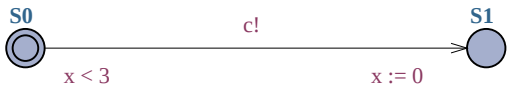

Figure 3.1: A simple automaton.

In figure 3.1 we have a very simple automaton with only two states and one transition. The transition goes from the initial state S0 to the state S1. The initial state is marked with double circles. The guard consists of only one atomic formula saying that the value of clock x should be less than 3. Similarly only one clock is reset (x := 0). The label on the transition is 'c!' this is the complement action of 'c?', which means that this transition must synchronize with an 'c?' transition in another timed automaton. As in CCS

[Mil89] we can also have transitions with no label, these transitions are in fact τ transitions that does not need to synchronize. Figure 3.1 illustration was made using the graphical interface for UPPAAL.

3.2 Preliminaries

First we need some auxiliary definitions.

Definition 3.1 : ACTIONS

Let $Chan$ be a finite set of channels, ranged over by c. We define Act to be a finite set of actions ranged over by a. For each channel in $Chan$ we define two actions such that $Act = \{c! \mid c \in Chan\} \cup \{c? \mid c \in Chan\}$. We define a complement operator $\bar{} : Act \to Act$ as $\bar{c!} = c?$ and $\bar{c?} = c!$. We define Δ to represent an infinite set of delay actions, $\Delta = \{\epsilon(d) | d \in \mathbb{R}\}$, where we use $\mathbb{R}$ to stand for the non-negative reals. The special internal action is represented by τ. We define the two sets $Act_\tau = Act \cup \tau$ and $\Delta_\tau = \Delta \cup \tau$.

Definition 3.2 : CLOCKS AND CONSTRAINTS

C is a finite set of real valued clocks ranged over by x, y, z. A clock valuation $u : C \to \mathbb{R}$ is a function that assigns to each clock a real non-negative value. We also define $\mathbb{R}^C$ to be the set of all clock valuations. We write $u(x)$ to mean the value of the clock x in the clock valuation u. We define two operations on clock valuations: Reset and Delay. Reset where a set of clocks are set to zero: $u' = u[r \mapsto 0], r \subseteq C$ defined by $\forall x \in r.u'(x) = 0, \forall x \in C \setminus r.u(x) = u'(x)$. Delay where all clocks are increased with the same value: $u + d : C \to \mathbb{R}$ where $d \in \mathbb{R}$, defined by $\forall x \in C.(u + d)(x) = u(x) + d$. We define $\mathcal{B}(C)$ to be the set of all clock constraints (also known as guards) $g ::= A \mid g \wedge g$ where A is an atomic formula of the form: $x \prec n$ or $x - y \prec n$ for $\prec \in \{\leq, \geq, <, >\}$ and n being a natural number. We write $g(u)$ to mean that the clock constraint g is true under the clock valuation u.

We extend the notion of transition system to a labelled transition system, where each transition has a label.

Definition 3.3 : Labelled Transition System

A labelled transition system relates the triple $(S, \mathcal{L}, \longrightarrow)$ in the following way. S is a set of states, $\mathcal{L}$ is a set of labels, and $\longrightarrow$ is a set of transitions $\longrightarrow \subseteq S \times \mathcal{L} \times S$. If $(S_1, \alpha, S_2) \in \longrightarrow$ we write $S_1 \xrightarrow{\alpha} S_2$

We describe the semantics of timed automata in terms of a labelled transition system.

3.3 Timed Automaton

In this section we define the syntax and semantics of a timed automaton.

Definition 3.4 : Syntax of Timed Automaton

A simple timed automaton A over actions Act and clocks C is defined by the triple (L_A, l_A^0, E_A) where L_A is a set of locations, $l_A^0 \in L_A$ is the initial location, and $E_A \subseteq L_A \times \mathcal{B}(C) \times Act_\tau \times 2^C \times L_A$.

Definition 3.5 : Semantics of Timed Automaton

The semantics of a timed automaton A is a labelled transition system defined by the triple $(S_A, \mathcal{L}_A, \longrightarrow_A)$ where the states are made up of a node and a clock valuation: $S_A = L_A \times \mathbb{R}^C$, the labels are the union $\mathcal{L}_A = Act_\tau \cup \Delta$, and the transition relation is defined as:

- $(l, u) \xrightarrow{a}_A (l', u')$ if $\exists g, r.(l, g, a, r, l') \in E_A, u' = [r \mapsto 0]u,$ and $g(u)$
- $(l, u) \xrightarrow{\epsilon(d)}_A (l, u + d)$

As an example to illustrate the semantics, we can look at the simple timed automaton depicted in figure 3.1. The start state of this automaton is $(s_0, x = 0)$ from here it can, among many other delay transitions, take the following delay transition $(s_0, x = 0) \xrightarrow{\epsilon(2,5)} (s_0, x = 2, 5)$. From here it can take the discrete transition $(s_0, x = 2, 5) \xrightarrow{a!} (s_1, x = 0)$ because $x < 3$, so the guard is true.

3.4 Timed Automata Network

We want to define how to make a parallel composition of several timed automata into a Timed Automata Network (TAN).

Definition 3.6 : Syntax of Timed Automata Network

A TAN N over actions A and clocks C has the form:

$$N = A_1 | \ldots | A_n$$

where each A_i is a timed automaton over actions Act and clocks C.

The clocks are all potentially global, but may in reality be local by being used in only one automaton. In the definition of the semantics we need some notation. We write $\vec{l}$ to mean a vector $l_1, l_2, \ldots, l_n$ of locations in each automaton.

Definition 3.7 : Semantics of Timed Automata Network

The semantics of a TAN $N = (A_1 | \ldots | A_n)$ over actions Act and clocks C is a labelled transition system $(S_N, \mathcal{L}_N, \longrightarrow_N)$ where the states is a node in each timed automaton and a clock valuation $S_N = L_1 \times \ldots \times L_n \times \mathbb{R}^C$, the labels are $\mathcal{L} = \Delta_\tau$, and the transition relation $\longrightarrow_N$ is defined by:

- $(\vec{l}, u) \xrightarrow{\tau}_N (\vec{l'}, u')$ if
  $\exists g_i, r_i.(l_i, g_i, a, r_i, l'_i) \in E_i$
  $\exists g_j, r_j.(l_j, g_j, \bar{a}, r_j, l'_j) \in E_j$
  $g_i(u), g_j(u), u' = [r_i \cup r_j \mapsto 0]u$
  $\forall k \notin \{i, j\}.l'_k = l_k$
  for some $i, j \in \{1, \ldots, n\}$ where $i \neq j$ and $a \in Act$.
- $(\vec{l}, u) \xrightarrow{\tau}_N (\vec{l'}, u')$ if
  $\exists g_i, r_i.(l_i, g_i, \tau, r_i, l'_i) \in E_i$
  $g_i(u), u' = [r_i \mapsto 0]u$
  $\forall k \notin \{i\}.l'_k = l_k.$
  for some $i \in \{1, \ldots, n\}$
- $(\vec{l}, u) \xrightarrow{\epsilon(d)}_N (\vec{l}, u + d)$

The three types of transitions presented above can be described respectively as synchronizing, private, and delay transitions. The first is synchronizing because two timed automata synchronize by taking transitions labelled with each others complement. The second is private because it involves only one timed automaton. The third is a delay transition where all clocks are increased by the same value.

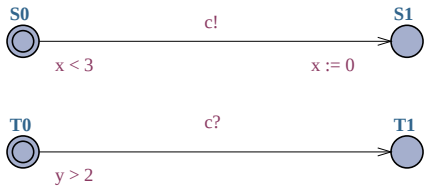

Figure 3.2: Two simple timed automata S and T.

Again to illustrate the semantics we give an example. We have, in figure 3.2, two simple timed automata that we combine into the system $N = S \mid T$. The start state of the system is $((s_0, t_0), x = 0, y = 0)$. From this state we could choose to delay for one time unit.

$$((s_0, t_0), x = 0, y = 0) \xrightarrow{\epsilon(1)} ((s_0, t_0), x = 1, y = 1)$$

From the new state we cannot take any discrete transitions because of the guard $y > 2$. So we choose to delay again, this time with 1.5 time units.

$$((s_0, t_0), x = 1, y = 1) \xrightarrow{\epsilon(1.5)} ((s_0, t_0), x = 2.5, y = 2.5)$$

Now we can take the discrete transition because the guards on both synchronizing transitions are true.

$$((s_0, t_0), x = 2.5, y = 2.5) \xrightarrow{\tau} ((s_1, t_1), x = 0, y = 2.5)$$

We notice that in the resulting state the clock x is set to zero.

4 Symbolic Analysis of TA

The semantics given in chapter 3 yields an infinite state-space and the CBR algorithm presented in chapter 2 needs a finite state-space in order to be guaranteed to terminate. To reduce the infinite state-space to a finite state-space we will represent groups of clock valuations as zones. This is done in the same manner as for the verification tool UPPAAL. We first define the concept of zones and operations on zones that we need during symbolic analysis. We then describe the data structure Difference Bounded Matrix (DBM) used to represent zones and how the needed operations are realized efficiently on DBMs. Finally we show how to perform both forward and backwards symbolic analysis using the operations described.

4.1 Zones

We introduce zones in order to be able to handle a set of states simultaneously, in one symbolic state. A zone represents an infinite set of clock valuations, it gives bounds on, both the difference between individual clocks, and on the absolute value of clocks. Figure 4.1 illustrates the difference between a single clock valuation and a zone. In general symbolic states are subsets of $L_1 \times \ldots \times L_n \times \mathbb{R}^C$. The symbolic states we use in this section has the form $(\vec{l}, Z)$. A symbolic state $(\vec{l}, Z)$ represents all states of the form $(\vec{l}, u)$ where $u \in Z$. A zone is a set of clock valuations defined by a simple constraint system which is defined in the same way as clock constraints in section 3.2.

Definition 4.1 : SIMPLE CONSTRAINT SYSTEM

$$g ::= \; x \prec n \mid x - y \prec n \mid g \wedge g$$

where $\prec \in \{\leq, \geq, <, >\}$ and $n \in \mathbb{N}$. We use $\mathcal{B}(C)$ to represent the set of all simple constraint systems over clocks C.

Figure 4.1 illustrates the difference between a single clock valuation and a zone.

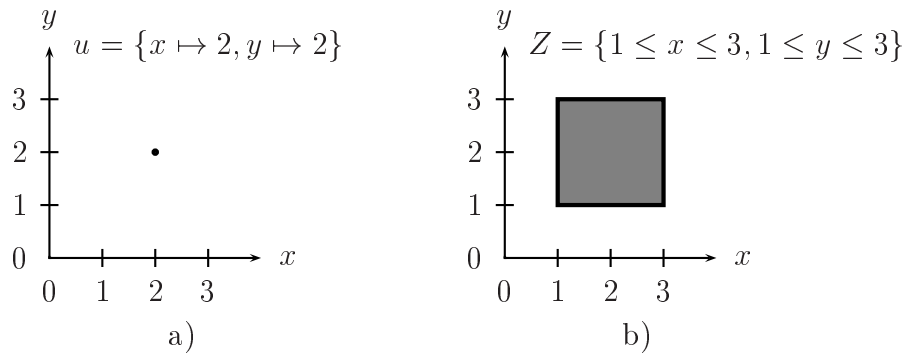

Figure 4.1: a) Clock valuation. b) Zone

Now we have a way of representing a group of states as one symbolic state $(\vec{l}, Z)$ and move on to defining useful operations on zones in order to be able to define a symbolic transition relation.

4.1.1 Operations on Zones

We define five operations on zones that we need for the symbolic reachability analysis. The *Future* and *Reset* operations are only needed for forward analysis, and *Past* and *Free* are only needed for backward analysis while we need *Conjunction* for both. We remind that $\mathbb{R}$ is defined as the non-negative reals. The five operations are defined as follows.

$$
\begin{array}{lrcl}
Future: & Z^{\nearrow} & = & \{u + d \mid u \in Z \text{ and } d \in \mathbb{R}\} \\
Past: & Z^{\swarrow} & = & \{u \mid \exists d \in \mathbb{R}. u + d \in Z\} \\
Reset: & {}^{reset}\{r\}Z & = & \{u[r \mapsto 0] \mid u \in Z\} \\
Free: & {}^{free}\{r\}Z & = & \{u \mid u[r \mapsto 0] \in Z\} \\
Conjunction: & Z \wedge Z' & = & \{u \mid u \in Z \text{ and } u \in Z'\}
\end{array}
$$

The operations are illustrated in figure 4.2. The first four operations are illustrated by the effect they have on the example zone Z that is shown in the upper left corner. The conjunction operator is illustrated with two other zones Z_1 and Z_2.

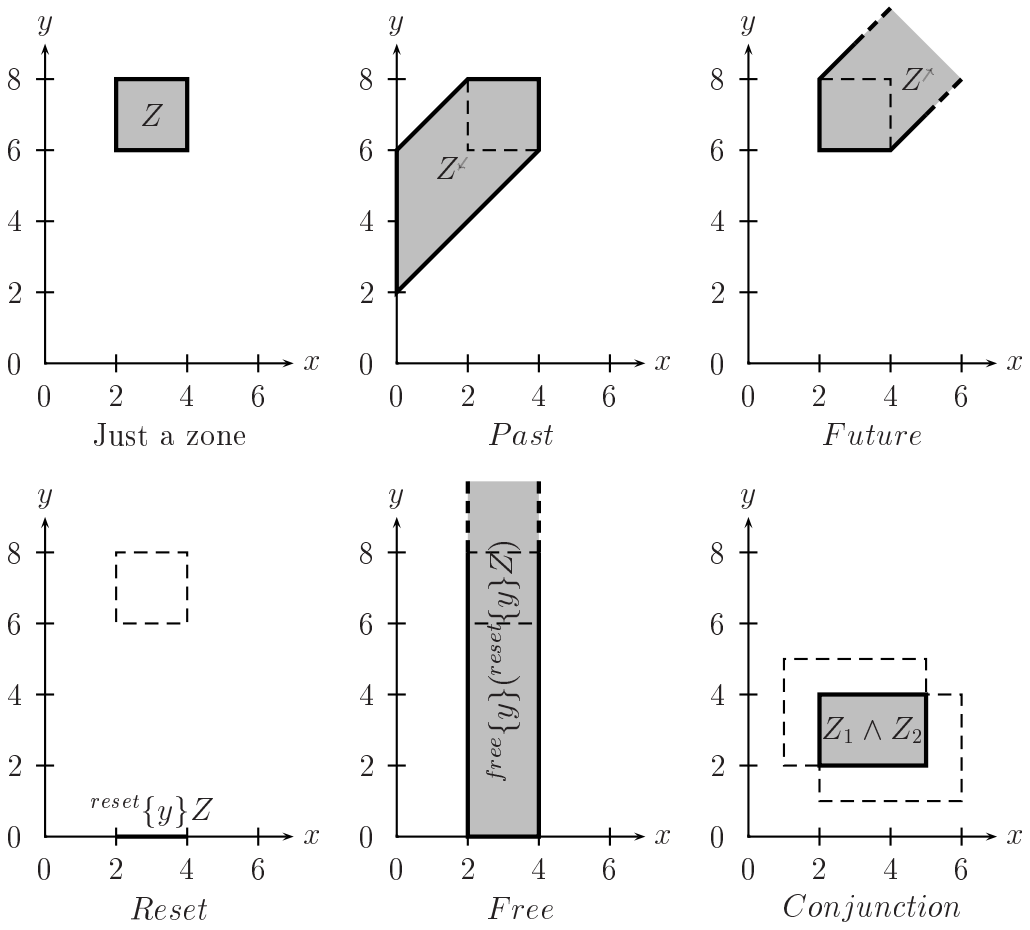

Figure 4.2: Operations on Zones

4.2 Difference Bounded Matrices

We need a data representation of zones and a definition of the five operations on this representation. A DBM is a matrix representation of a simple constraint system.

Definition 4.2 : Difference Bounded Matrix

$$M : \{x_0, x_1, \ldots, x_n\}^2 \rightarrow (\mathbb{Z} \times \{<, \leq\}) \cup \{+\infty\}$$

where x_0 is a special zero valued clock.

For every pair of clocks it gives a comparison operator and a real value or ∞. For each pair of clocks $M(x_i, x_j) = (n_{ij}, \prec_{ij})$ represents that $x_i - x_j \prec_{ij} n_{ij}$, where $\prec_{ij}$ is $<$ or $\leq$. Figure 4.2 illustrates how a number in the matrix represents a bound on the difference between two clocks. The operator is represented by an extra bit stored along with each number. In figure 4.5 we will as an example illustrate how the different zones shown in figure 4.2 can be represented as DBMs.

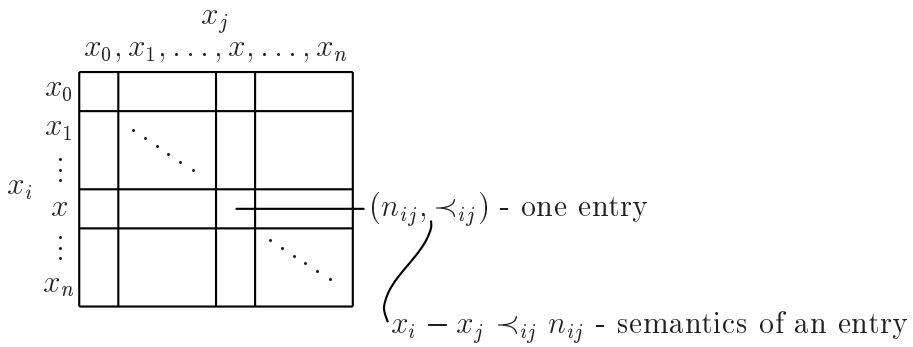

Figure 4.3: Illustration of a Difference Bounded Matrix.

In order to be able to define the operations we need to have the DBMs in a canonical form. For the definition of the canonical form we need a definition of the two operators $+$ and $\leq$ for pairs of the type $(n, \prec)$, where $n \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$. To do this we also define $+_b$ and $\leq_b$, which operate on $`<'$ and $`\leq'$.

Definition 4.3 : $+_b$ OPERATOR FOR $`<'$ AND $`\leq'$

$$\begin{aligned} `<' +_b `<' &= `<' \\ `\leq' +_b `<' &= `<' \\ `<' +_b `\leq' &= `<' \\ `\leq' +_b `\leq' &= `\leq' \end{aligned}$$

Here we observe that anything but two $\leq$'s adds up to $<$. We can now define $+$ on pairs of the type $(n, \prec)$. This is simply done by adding the integers and adding the $\prec$ operators with the newly defined $+_b$ operator.

$$(n_1, \prec_1) + (n_2, \prec_2) = (n_1 + n_2, \prec_1 +_b \prec_2)$$

Definition 4.4 : $\leq_b$ OPERATOR FOR $`<'$ AND $`\leq'$

$$\begin{array}{lll} `<' & \leq_b & `\leq' \\ `<' & \leq_b & `<' \\ `\leq' & \leq_b & `\leq' \end{array}$$

Both the operators are equal with themselves and $`<'$ is smaller than $`\leq'$. Now we are ready to define $\leq$ on pairs of the type $(n, \prec)$. This is done as a sort of lexicographic ordering. First the integers are considered, if these are equal then the $\prec$ operators are compared.

$$(n_1, \prec_1) \leq (n_2, \prec_2) = n_1 < n_2 \vee (n_1 = n_2 \wedge \prec_1 `\leq' \prec_2)$$

With the $+$ and $\leq$ operators defined we are ready to define the canonical form.

Definition 4.5 : CANONICAL FORM

A DBM M is on canonical form if and only if $\forall x_i, x_j, x_k \in C$ it is such that

$$M(x_i, x_j) + M(x_j, x_k) \geq M(x_i, x_k)$$

We define the operations on DBMs in canonical form. The following functions define the value of each entry in the resulting matrix, based on the input matrix. The first four operations are illustrated in figure 4.4, these are the operation that operate on a single matrix. The final operation, conjunction, describes the resulting matrix in terms of two input matrices.

Past

$$M^{\nwarrow}(x_i, x_j) = \begin{cases} M(x_i, x_j) & x_i \neq x_0 \\ (0, \leq) & x_i = x_0 \end{cases}$$

Future

$$M^{\nearrow}(x_i, x_j) = \begin{cases} M(x_i, x_j) & x_j \neq x_0 \\ +\infty & x_j = x_0 \end{cases}$$

Reset

$$^{reset}\{x\}M(x_i, x_j) = \begin{cases} M(x_i, x_j) & x_i, x_j \neq x \\ (0, \leq) & x_i = x \\ (0, \leq) & x_j = x \wedge x_i = x_0 \\ +\infty & x_j = x \end{cases}$$

Free

$$^{free}\{x\}M(x_i,x_j) = \begin{cases} M(x_i,x_j) & x_i, x_j \neq x \\ (0,\leq) & x_i = x_0 \wedge x_j = x \\ +\infty & x_i = x \vee (x_j = x \wedge x_i \neq x_0) \end{cases}$$

Conjunction

$$(M_1 \wedge M_2)(x_i,x_j) = \begin{cases} M_1(x_i,x_j) & M_1(x_i,x_j) \leq M_2(x_i,x_j) \\ M_2(x_i,x_j) & otherwise \end{cases}$$

The implementation of four of the operators is illustrated in figure 4.4. The gray areas represent that the values in this part of the matrix are left unchanged. The values that are assigned to the changed areas can be read from the definition of the operations.

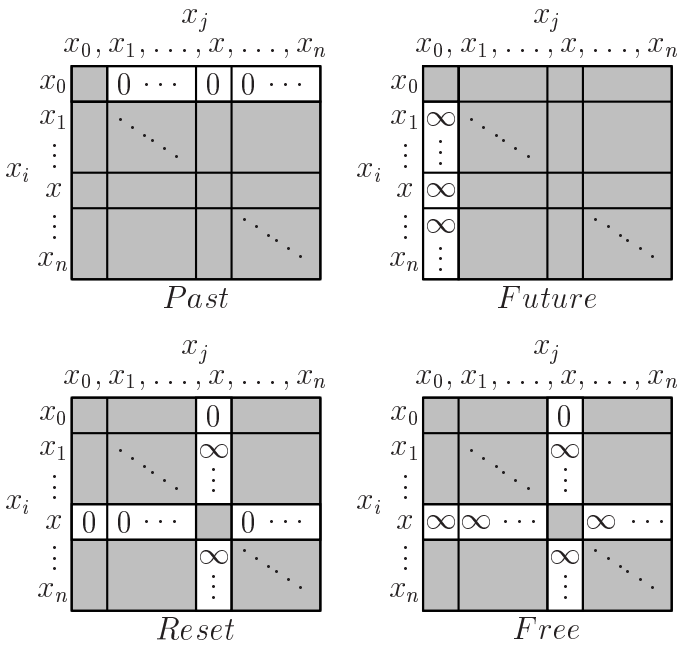

Figure 4.4: Illustration of the Operations on DBMs.

The conjunction operator, which is not illustrated, combines two matrices. For each entry in the matrix the values are compared and the smallest is selected as the entry in the resulting matrix. Figure 4.5 illustrates all the operations by use of the zones from figure 4.2. We only use the $\leq$ operator in these examples to keep it simpler.

	x_0	x	y
x_0		-2	-6
x	4		-2
y	8	6	

Just a zone

	x_0	x	y
x_0		$\boxed{0}$	$\boxed{0}$
x	4		-2
y	8	6	

Past

	x_0	x	y
x_0		-2	-6
x	$\boxed{0}$		-2
y	$\boxed{0}$	6	

Future

	x_0	x	y
x_0		-2	$\boxed{0}$
x	4		$\boxed{\infty}$
y	$\boxed{0}$	$\boxed{0}$	

Reset

	x_0	x	y
x_0		-2	$\boxed{0}$
x	4		-2
y	$\boxed{\infty}$	$\boxed{\infty}$	

Free

	x_0	x	y
x_0		-1	-2
x	5		3
y	5	4	

Z_1

	x_0	x	y
x_0		-2	-1
x	6		5
y	4	2	

Z_2

	x_0	x	y
x_0		-2	-2
x	5		3
y	4	2	

Conjunction

Figure 4.5: Illustration of the Operations on DBMs.

The matrices are no longer in the canonical form after the operations have been performed. They are restored to canonical form by calculating the shortest path closure. This can best be illustrated by viewing the matrix as a graph. In figure 4.6 we calculate the canonical form for the matrix after the *Past* operation. The values on the edges are given by taking the from node as the row and the to node as the column. The shortest path closure is calculated by checking for shorter paths between two nodes via other nodes. In figure 4.6 only the edge from x_0 to y is changed. The new value is -2 because the path via the x node is $-2 + 0 = -2$.

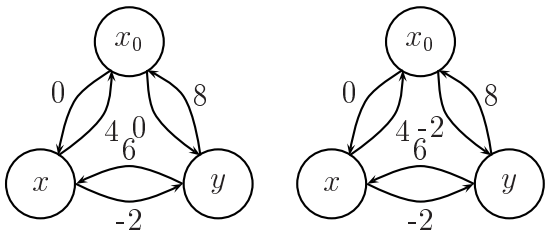

Figure 4.6: The shortest path closure of the *Past* matrix.

In the algorithms presented we also need to check for inclusion. This is done by comparing each pair of entries in the matrices. If for every pair of entries the entry of matrix A is smaller then that of matrix B, then A is included in B.
To guarantee termination we introduce a normalization operation. Initially we have infinitely many Zones and cannot guarantee that the algorithm terminates. Two clock valuations that cannot be distinguished in the model are time-abstracted bisimilar, illustrated in figure 4.7. This means that when the one can take a delay transition the other can also take a delay transition, not necessarily with the same amount of delay and end up in a state that is time-abstracted bisimilar with the end state of the first. The same is true for discrete transitions. Knowing this we only need to represent one of such time-abstracted bisimilar states. This is done by applying the normalization operation to all Zones after each operation.

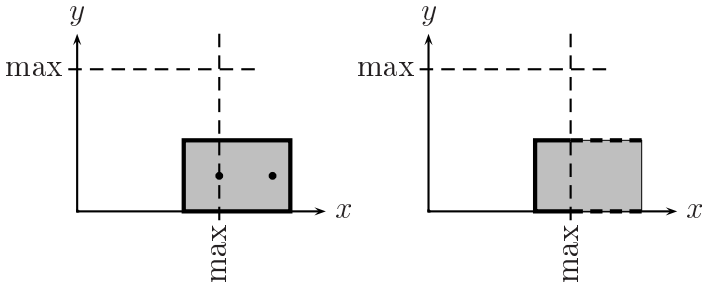

Figure 4.7: Illustration of normalization. All clock valuations to the right of the max line will have a point exactly on the max line that it is time-abstracted bisimilar to. Such two points are illustrated on the left figure. Therefor the Zone on the right figure can reach the same states as the figure on the left.

First the maximum constant N, used in the model or in the properties to be checked, is found. Any integer lager than N can be replaced with ∞ and all integers smaller than $-N$ can be replaced with $-N$. This can be done because it never will be compared to anything larger than itself. The normalization is performed after each operation. When N is known we can also calculate how many bits we need to represent each entry in the DBM. The operation is described below.

Normalization

$$norm_N M(x_i, x_j) = \begin{cases} (-N, <) & M(x_i, x_j) \leq (-N, <) \\ M(x_i, x_j) & (-N, \leq) \leq M(x_i, x_j) \leq (N, \leq) \\ +\infty & (-N, <) \leq M(x_i, x_j) \end{cases}$$

4.3 Symbolic Reachability

In this section we use the operations defined in section 4.1 to do symbolic reachability analysis on networks of Timed Automata.

Definition 4.6 : Forward Symbolic Transition

We define two types of transition, in contrast to the three types defined in the normal semantics. The first represents a delay action followed by a single transition while the other is a delay followed by a synchronization. We calculate the new zone by use of the operations that we have defined. In both cases we first take the future operation on the original zone, after this we conjunct it with the guard(s), and last reset the clocks defined by the reset set(s).

- $(\vec{l}, Z) \Longrightarrow_F (\vec{l'}, Z')$ if
  $\exists g_i, r_i.(l_i, g_i, \tau, r_i, l'_i) \in E_i$
  $Z' = {}^{reset}\{r_i\}(Z^{\nearrow} \wedge g_i)$
  $\forall k \notin \{i\}.l'_k = l_k$
  for some $i \in \{1, \ldots, n\}$

- $(\vec{l}, Z) \Longrightarrow_F (\vec{l'}, Z')$ if
  $\exists g_i, r_i.(l_i, g_i, a, r_i, l'_i) \in E_i$
  $\exists g_j, r_j.(l_j, g_j, \bar{a}, r_j, l'_j) \in E_j$
  $Z' = {}^{reset}\{r_i \cup r_j\}(Z^{\nearrow} \wedge g_i \wedge g_j)$
  $\forall k \notin \{i, j\}.l'_k = l_k$
  for some $i, j \in \{1, \ldots, n\}$ where $i \neq j$ and $a \in Act$.

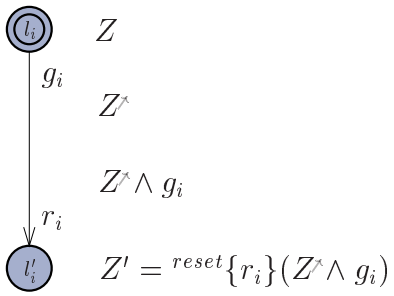

Figure 4.8: Illustrates how Z' is calculated when taking one forward symbolic transition.

Definition 4.7 : Backward Symbolic Transition

As with the forward symbolic transitions we define two transition rules. The first represents a single transition followed by a delay action while the other is a synchronization followed by a delay. It is worth noticing that the order is not the same as for the forward symbolic transitions. This does not have any impact on the reachable state-space. The essential thing is that we alternate between discrete and delay actions. This results in that we have to take the future operation on the initial location before checking for intersection in the backwards algorithm. Here we start by using the past operation and conjunct the result with the reset set(s). After this the clocks in the reset set(s) are freed, this is then conjuncted with the guards.

- $(\vec{l'}, Z') \Longleftarrow_B (\vec{l}, Z)$ if
  $\exists g_i, r_i.(l_i, g_i, \tau, r_i, l'_i) \in E_i$
  $Z = ({}^{free}\{r_i\}(r_i \wedge Z'^{\not\downarrow})) \wedge g_i$
  $\forall k \notin \{i\}.l'_k = l_k$
  for some $i \in \{1, \ldots, n\}$

- $(\vec{l'}, Z') \Longleftarrow_B (\vec{l}, Z)$ if
  $\exists g_i, r_i.(l_i, g_i, a, r_i, l'_i) \in E_i$
  $\exists g_j, r_j.(l_j, g_j, \bar{a}, r_j, l'_j) \in E_j$
  $Z = ({}^{free}\{r_i \cup r_j\}(r_i \wedge r_j \wedge Z'^{\not\downarrow})) \wedge g_i \wedge g_j$
  $\forall k \notin \{i, j\}.l'_k = l_k$
  for some $i, j \in \{1, \ldots, n\}$ where $i \neq j$ and $a \in Act$.

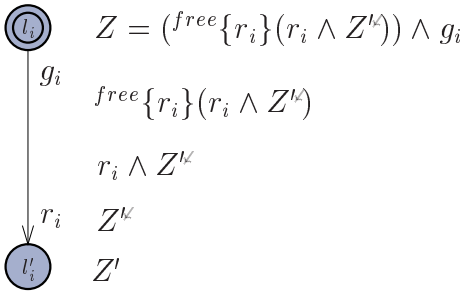

Figure 4.9: Illustrates how Z is calculated when taking one backwards symbolic transition.

4.3.1 Algorithms

With the symbolic transition rules we define two similarly looking algorithms presented in figures 4.10 and 4.11. First we describe the forward symbolic

reachability algorithm. The algorithm has a passed-list (*Passed*) and a waiting-list (*Wait*). Initially the passed list is empty and the waiting list contains the initial state. For each cycle in the **repeat-until** loop one symbolic state is removed from the waiting list. After having added all states, that can be reached from it, to the waiting list, the state is itself added to the passed list. This is continued until either; the waiting list is empty, or a state is found, that intersects with *Goal*. The target that we want to check if we can reach, *Goal*, is a set of symbolic states.

PASSED := {}
WAIT := $\{(\vec{l}_0, Z_0)\}$
repeat
  begin
    get $(\vec{l}, Z)$ from WAIT
    if $(\vec{l}, Z) \cap Goal \neq \emptyset$ **then return TRUE**
    else if $Z \not\subseteq Z'$ **for all** $(\vec{l}, Z') \in$ PASSED **then**
      begin
        add $(\vec{l}, Z)$ to PASSED
        NEXT := $\{(\vec{l_s}, Z_s) \mid (\vec{l}, Z) \Longrightarrow_F (\vec{l_s}, Z_s) \wedge Z_s \neq \emptyset\}$
        for all $(\vec{l_{s'}}, Z_{s'})$ in NEXT **do**
            put $(\vec{l_{s'}}, Z_{s'})$ to WAIT
      end
  end
until WAIT = {}
return FALSE

Figure 4.10: Algorithm for forward symbolic reachability analysis

The backwards symbolic reachability algorithm differs in three ways. Firstly the waiting list is initialized to contain *Goal* in stead of the initial state. Secondly there is tested for intersection with the initial state instead of *Goal*. Thirdly the transition relation that is used to find new symbolic states, to put in the waiting list, is $\Longleftarrow_B$.

4.3.2 Theorems

In this section we present some theorems, and a single sample proof, needed to argument for the correctness of the algorithms presented in the previous

```
PASSED := {}
WAIT := Goal
repeat
  begin
    get (l⃗, Z) from WAIT
    if (l⃗, Z) ∩ (l⃗_0, Z_0)^↑ ≠ ∅ then return TRUE
    else if Z ⊈ Z' for all (l⃗, Z') ∈ PASSED then
      begin
        add (l⃗, Z) to PASSED
        NEXT := {(l⃗_s, Z_s) | (l⃗, Z) ⟸_B (l⃗_s, Z_s) ∧ Z_s ≠ ∅}
        for all (l⃗_s', Z_s') in NEXT do
              put (l⃗_s', Z_s') to WAIT
      end
  end
until WAIT = {}
return FALSE
```

Figure 4.11: Algorithm for backwards symbolic reachability analysis

section. Correctness in the sense that the symbolic algorithm gives the correct result in terms of concrete reachability. Theorem 4.8 implies, by a simple inductive argument, that if we have a sequence of symbolic transitions leading from the initial state to *Bad* we also have a sequence of concrete states leading from the initial state to *Bad*.

Theorem 4.8

Forward: symbolic to concrete

$$(\vec{l}, Z) \Longrightarrow_F (\vec{l'}, Z')$$
$$\Downarrow$$
$$\forall u' \in Z'.\exists d, \exists u \in Z.(\vec{l}, u) \xrightarrow{\epsilon(d)} \xrightarrow{\tau} (\vec{l'}, u')$$

Proof of theorem 4.8 ■

The proof can be split into two cases depending on how $(\vec{l}, Z) \Longrightarrow_F (\vec{l'}, Z')$ is

obtained. We only treat the first simple case. Thus we have from the right side of the definition of the $\Longrightarrow_F$ transition relation that:

$$
\begin{array}{l}
(l_i, g_i, \tau, r_i, l_i') \in E_i \\
Z' = {}^{reset}\{r_i\}(Z^{\nearrow} \wedge g_i) \\
\forall k \notin \{i\}.l_k' = l_k \\
\text{for some } i \in \{1, \ldots, n\}
\end{array}
$$

We assume that $u' \in Z'$. Thus for some $u'' \in Z^{\nearrow} \wedge g_i$ we have that $u' = [r_i \mapsto 0]u''$. Then we also have for some $u \in Z$ and some $d \in \mathbb{R}$ that $u'' = u + d$. It follows that $(\vec{l}, u) \xrightarrow{\epsilon(d)} (\vec{l}, u + d) \xrightarrow{\tau} (\vec{l'}, u')$.

■ ■

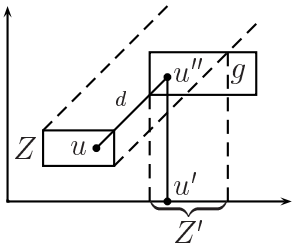

Figure 4.12: Illustration of proof of theorem 4.8

The relation between u, u', and u'' in the proof of theorem 4.8 is illustrated in figure 4.12. The theorem itself is illustrated in figure 4.13. Here we use dashed arrows to illustrate $\exists$ and full arrows to illustrate $\forall$. The same is true for the illustration of the following theorems.

$$
\begin{array}{ccc}
(\vec{l}, Z) & \Longrightarrow_F & (\vec{l'}, Z') \\
\cup & & \cup \\
(\vec{l}, u) & \overset{\epsilon(d)}{\dashrightarrow} \overset{\tau}{\dashrightarrow} & (\vec{l'}, u')
\end{array}
\qquad
\begin{array}{l}
\| \sim \forall \\
\| \sim \exists
\end{array}
$$

Figure 4.13: Illustration of theorem 4.8

Theorem 4.9 states that if we have a sequence of concrete states leading from the initial state to *Bad* we can match this with a sequence of symbolic states.

Theorem 4.9

Forward: concrete to symbolic

$$\begin{array}{c} (\vec{l}, u) \xrightarrow{\epsilon(d)} \xrightarrow{\tau} (\vec{l'}, u') \\ \Downarrow \\ \forall Z.u \in Z.\exists Z'.(\vec{l}, Z) \Longrightarrow_F (\vec{l'}, Z') \wedge u' \in Z' \end{array}$$

Theorem 4.9 is illustrated in figure 4.14. We omit the proof which is similar to that of theorem 4.8.

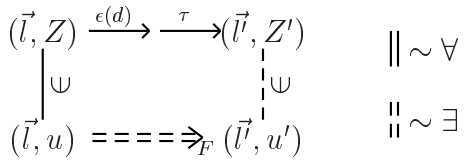

Figure 4.14: Illustration of theorem 4.9

In order to conclude that the forward symbolic reachability algorithm presented in figure 4.10 always terminates we need to ensure that there are only finitely many reachable symbolic states. In fact exactly one is generated for each transition that can be taken. We also want to conclude that when the algorithm terminates it terminates with the correct answer. This we can conclude from the two theorems 4.9 and 4.8. If we find a sequence of symbolic states that leads from the initial state to Bad there also exists a concrete sequence of states. On the other hand if we do not find a sequence of symbolic states we can conclude that there is now sequence of concrete states leading from the initial state to Bad.

In the following we present two theorems that state the exact same things as theorems 4.8 and 4.9 only for the backward transition relation $\Longleftarrow_B$. Since the arrows goes in the other direction the states are not in the same order in the top and bottom of the theorem. This also has the effect that the concrete arrows in figure 4.15 and 4.16 goes from right to left.

Theorem 4.10

Backward: symbolic to concrete

$$(\vec{l'}, Z') \Longleftarrow_B (\vec{l}, Z)$$
$$\Downarrow$$
$$\forall u \in Z. \exists d\ \exists u' \in Z'. (\vec{l}, u) \xrightarrow{\tau} \xrightarrow{\epsilon(d)} (\vec{l'}, u')$$

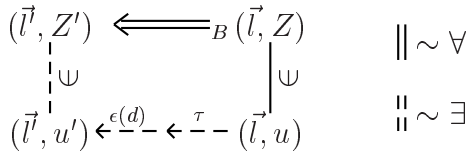

Figure 4.15: Illustration of theorem 4.10

Theorem 4.11

Backward: concrete to symbolic

$$(\vec{l}, u) \xrightarrow{\tau} \xrightarrow{\epsilon(d)} (\vec{l'}, u')$$
$$\Downarrow$$
$$\forall Z'. u' \in Z'. \exists Z. (\vec{l'}, Z') \Longleftarrow_B (\vec{l}, Z) \wedge u \in Z$$

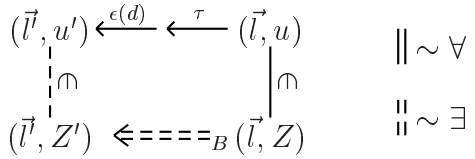

Figure 4.16: Illustration of theorem 4.11

4.3.3 Correctness of Backwards Algorithm

We will use theorems 4.10 and 4.11 to prove the correctness of the algorithm for symbolic backwards reachability with regard to reachability. This could

be done in a similar fashion for the forward algorithm using theorem 4.8 and 4.9.

Theorem 4.12

The symbolic backwards reachability is partially correct with regard to reachability. That is whenever the algorithm terminates it gives the correct answer.

We need some definitions for the proof of theorem 4.12.

Definition 4.13 : LEADS-TO OPERATOR

We write $(\vec{l}, u) \rightsquigarrow (\vec{l'}, u')$ if $(\vec{l}, u) \xrightarrow{\tau} \xrightarrow{\epsilon(d_1)} \ldots \xrightarrow{\tau} \xrightarrow{\epsilon(d_n)} (\vec{l'}, u')$.
We also write $(\vec{l}, u) \rightsquigarrow g$ if $(\vec{l}, u) \rightsquigarrow (\vec{l'}, u')$ for some $(\vec{l'}, u') \in g$.

Proof of theorem 4.12 ■

We prove the partial correctness by splitting into two cases.

Case 1: Assume that the algorithm terminates with the answer **TRUE**. We claim that whenever $(\vec{l}, Z) \in$ PASSED and $u \in Z$ then $(\vec{l}, u) \rightsquigarrow Bad$. To see this note that if $(\vec{l}, Z) \in$ PASSED then we have a symbolic sequence of the form:

$$(\vec{l}_{Bad}, Z_{Bad}) \Longleftarrow_B (\vec{l}_1, Z_1) \Longleftarrow_B (\vec{l}_2, Z_2) \Longleftarrow_B \ldots \Longleftarrow_B (\vec{l}, Z)$$

Now applying theorem 4.10 repeatedly proves the claim.

It follows immediately from the proven claim that if ever an encountered symbolic state intersects with INIT, $(\vec{l_0}, Z_0)$, then indeed there is a path from INIT to some state in Bad.

Case 2: Assume that the algorithm terminates with the answer **FALSE**. We claim that in this case $(\vec{l}, u) \rightsquigarrow Bad$ cannot hold for any state $(\vec{l}, u) \in$ INIT. We prove this by contradiction. Assume on the contrary that $(\vec{l}, u) \rightsquigarrow Bad$ for some state $(\vec{l}, u) \in$ INIT. This is:

$$(\vec{l}, u) \xrightarrow{\tau} \xrightarrow{\epsilon(d_1)} \ldots \xrightarrow{\tau} \xrightarrow{\epsilon(d_n)} Bad$$

Then because of theorem 4.11 we have a symbolic sequence of the form:

$$(\vec{l}_{Bad}, Z_{Bad}) \Longleftarrow_B (\vec{l}_1, Z_1) \Longleftarrow_B (\vec{l}_2, Z_2) \Longleftarrow_B \ldots \Longleftarrow_B (\vec{l}, Z)$$

with $(\vec{l}_{Bad}, Z_{Bad}) \subseteq Bad$ and $u \in Z$.

But then $(\vec{l}, Z)$ must be present in the WAIT-list at some point during computation and hence we would have obtained termination with the answer **TRUE** in contradiction with the assumption.

■ ■

For total correctness of the algorithm we observe that the algorithm terminates due to the finiteness of the symbolic state-space, induced by normalization.
A similar correctness proof could be given for symbolic forward reachability algorithm.

5 Application of CBR on TA

In this section we apply the CBR method to the domain of Timed Automata Network (TAN). We choose the symbolic CBR algorithm, from section 2.4.3, because it exactly fits the domain.

5.1 Fulfilling the Requirements

We will in the following sections describe how the domain fits the requirements stated in section 2.4.3. Here we will introduce the CBR method for timed automata, not unlike how it was introduced in the previous report [Lar02]. At the same time we will show that this fits exactly within the framework of the symbolic algorithm presented in section 2.4.3. To be able to use only some of the components, automata and clocks, we define two subsets: $M \subseteq \{1, \ldots, n\}$ an index subset of the timed automata and $K \subseteq C$ a subset of clocks. We will base the partitioning of the state-space on an equivalence derived from these two subsets.

Representable Symbolic States

First we define the representable symbolic states (RSS), which we are going to use in the analysis. Firstly these states are symbolic in the representation of the clock values, in the use of Zones, as described in chapter 4. Secondly they are symbolic in the representation of the location vector. We introduce a partial location vector, in which we only need to specify the location for some components. The location of the rest of the components are represented by a $*$ (star) meaning that this automata can be in any of its locations. We will refer to these states as double symbolic states since they can be symbolic both in the use of zones and the representation of the discrete location. Again such symbolic states will be subsets of $L_1 \times \ldots \times L_n \times \mathbb{R}^C$ as with the symbolic states defined in section 4.1.

An M-sorted partial location vector only contains information about the automata in M, and semanticly it represents the set of all location vectors that agree with it with regard to the locations of all automata in M. For a

zone to be K-sorted it cannot include any constraints on clocks not included in K.

Definition 5.1 : DOUBLE SYMBOLIC STATE

A double symbolic state $(\vec{p}, Z)$ consists of an M-sorted location vector $\vec{p}$ and a K-sorted zone Z. For a given $M \subseteq \{1, \ldots, n\}$ an M-sorted location vector is defined as follows:

$$\vec{p} = (p_1, \ldots, p_n) \text{ where } \begin{cases} i \in M & p_i \in L_i \cup \{*\} \\ i \notin M & p_i = * \end{cases}$$

A K-sorted zone only contains constraints on clocks in K: $Z \in \mathcal{B}(K)$.

By an M, K-sorted symbolic state we mean a double symbolic state where the location vector is M-sorted and the zone is K-sorted. We notice that a double symbolic state that is M, K-sorted for a given M and K also is M, K-sorted for any larger M or K. We have that there are only finitely many zones, given normalization. Given that we have also finitely many automata and finitely many locations in each automata, we can only create a finite number of different representable symbolic states. This was one of the requirements of the symbolic CBR framework.

Partitioning of the State Space

We define the partitioning of the state-space on the basis of the M, K-equivalence. First we define M-equivalence for the discrete part of the state and K equivalence for the continuous part of the state.

Definition 5.2 : M-EQUIVALENCE

$$\vec{l} =_M \vec{l'} \iff \forall i \in M . l_i = l'_i$$

Definition 5.3 : K-EQUIVALENCE

$$u =_K u' \iff \forall x \in K . u(x) = u'(x)$$

We define the M, K-equivalence in terms of the two other equivalences.

Definition 5.4 : M, K-EQUIVALENCE

We define M, K-equivalence in the following way:

$$(\vec{l}, u) =_{M,K} (\vec{l'}, u') \iff \vec{l} =_M \vec{l'} \text{ and } u =_K u'$$

We partition the state-space based on the number of automata and clocks included in the analysis. We start with a the subset of automata and clocks needed to represent *Goal*. After this we gradually extend with more clocks and automata. Since we have a finite amount of clocks and automata, we will in a finite number of steps reach a point where all clocks and automata are included. For each M, Kcombination we define a partitioning where all states that are M, K-equivalent are in the same partition. When we have included all clocks and automata the M, K-equivalence will correspond to the identity relation $Id = \{(s, s) \mid s \in ST\}$. This will result in the fact that the partitioning defined by this equivalence satisfies the property of being a stable partitioning. The actual order in which to include the components, does not affect the method in general. Different heuristics will be considered in section 7.3. The partitioning induced by a given equivalence $=_{M,K}$, is called $\mathcal{P}_{M,K}$, instead of writing $\mathcal{P}_{=_{M,K}}$. If we have that $M \subseteq M'$ and $K \subseteq K'$ then the equivalence induced by $=_{M',K'}$ is finer than or equal to the one induced by $=_{M,K}$, because M' and K' have more elements. In general the following holds:

$$M \subseteq M' \wedge K \subseteq K' \iff \mathcal{P}_{M',K'} \sqsubseteq \mathcal{P}_{M,K}$$

Sorted Symbolic Predecessor

In this section we describe how to interpret a timed automata network (TAN) as a global transition system. The concrete states where in section 3.4 interpreted as a transition system. In this section we define how to interpret a TAN as a transition system, where the states are double symbolic states. We do this by defining a new transition relation the combines the calculation of the new zones from the $\Longleftarrow_B$ transition relation and the concept of M, K-sortedness. The idea is to relate M, K-sorted symbolic states with other M, K-sorted symbolic states. This means that we will only consider taking transitions in automata specified by M and where the constraints on the guards only range over clocks in K.

For the definition of the new transition relation we need some notation for what it means for a concrete location vector l_i to be included in a partial location vector p_i.

$$l_i \in p_i \iff \begin{cases} l_i = p_i & p_i \in L_i \\ true & p_i = * \end{cases}$$

Definition 5.5 : Backward M,K-sorted Transition Relation

The definition of the $\Longleftarrow_{M,K}$ transition relation is based on the $\Longleftarrow_B$ transition relation and adds the concept of M-sorted location vectors and K-sorted zones.
We know that $\vec{p'}$ is M-sorted and that Z' is K-sorted.

- $(\vec{p'}, Z') \Longleftarrow_{M,K} (\vec{p}, Z)$ if
  $\exists g_i \in \mathcal{B}(K), \exists r_i.(l_i, g_i, \tau, r_i, l'_i) \in E_i$
  $l_i = p_i,\ l'_i \in p'_i$
  $Z = ({}^{free}\{r_i\}(r_i \wedge Z'^{\nwarrow})) \wedge g_i$
  $\forall k \notin \{i\}.p'_k = p_k$
  for some $i \in M$

- $(\vec{p'}, Z') \Longleftarrow_{M,K} (\vec{p}, Z)$ if
  $\exists g_i \in \mathcal{B}(K), \exists r_i.(l_i, g_i, a, r_i, l'_i) \in E_i$
  $\exists g_j \in \mathcal{B}(K), \exists r_j.(l_j, g_j, \bar{a}, r_j, l'_j) \in E_j$
  $l_i = p_i,\ l'_i \in p'_i,\ l_j = p_j,\ l'_j \in p'_j$
  $Z = ({}^{free}\{r_i \cup r_j\}(r_i \wedge r_j \wedge Z'^{\nwarrow})) \wedge g_i \wedge g_j$
  $\forall k \notin \{i,j\}.p'_k = p_k$
  for some $i, j \in M$ where $i \neq j$ and $a \in Act$.

We need to prove that the new $Z \in \mathcal{B}(K)$ and that $\vec{p}$ is M-sorted. We can conclude that $Z \in \mathcal{B}(K)$ because the guards that are conjuncted are from $\mathcal{B}(K)$. The clocks that are reset are also freed again, this means that they will not bring Z out of $\mathcal{B}(K)$. We can also conclude that $\vec{p}$ is M-sorted because the index set remains the same.
We intend to prove the two assumptions made in assumption 2.20. In the following we write $\text{Pre}_{M,K}$ as a shorthand for $\text{Pre}_{\mathcal{P}_{M,K}}$. $\text{Pre}_{M,K}(H)$ is defined as

$$\{s \mid \forall t =_{M,K} s.\exists t' \in H.t \xrightarrow{\tau} \xrightarrow{\epsilon(d)} t'\}$$

Here we stretch the original definition 2.7 of Pre, by taking both a discrete and a delay step. The first line of each of the two assumptions can be rewritten as follows:

$$\text{SymPre}_{M,K}(\vec{p'}, Z') = \{(\vec{p_1}, Z_1), \ldots, (\vec{p_n}, Z_n)\}$$

First assumption In order to prove that $\bigcup_{i=1}^{n}(\vec{p_i}, Z_i) \subseteq \textsc{Pre}_{M,K}(\vec{p'}, Z')$, we must show that whenever $\vec{l} \in \vec{p_i}$ and $u \in Z_i$ then it follows that $(\vec{l_i}, u) \in \textsc{Pre}_{M,K}(\vec{p'}, Z')$.
Whenever $\vec{l} \in \vec{p_i}$ and $u \in Z_i$ then

$$(\vec{l_i}, u) \xrightarrow{\tau} \xrightarrow{\epsilon(d)} (\vec{l'}, v')$$

for some $\vec{l'} \in \vec{p'}$ and $v' \in Z'$.
Hence as $(\vec{p_i}, Z_i)$ is M, K-sorted it follows that $(\vec{l_i}, u) \in \textsc{Pre}_{M,K}(\vec{p'}, Z')$.

Second assumption We already have the $\bigcup_{i=1}^{n}(\vec{p_i}, Z_i) \subseteq \textsc{Pre}_{\mathcal{P}_{stable}}(\vec{p'}, Z')$ and only need to show that $\bigcup_{i=1}^{n}(\vec{p_i}, Z_i) \supseteq \textsc{Pre}_{\mathcal{P}_{stable}}(\vec{p'}, Z')$ in order to prove the equality. We have that $\mathcal{P}_{stable}$ is equal to the identity, so we must show that every element in $\textsc{Pre}_{Id}(\vec{p'}, Z')$ is in the set of symbolic states returned by SymPre. We have that $\textsc{Pre}_{Id}(\vec{p'}, Z') = \{s \mid \forall t =_{Id} s.\exists t' \in H.t \xrightarrow{\tau} \xrightarrow{\epsilon(d)} t'\} = \{t \mid \exists t' \in H.t \xrightarrow{\tau} \xrightarrow{\epsilon(d)} t'\}$. Since t has two transitions, which can bring it into H, we can see from the definition of $\Longleftarrow_{M,K}$ that there will exist a $(\vec{p_i}, Z_i) = t$.

Check for Inclusion and Intersection

The symbolic CBR framework also requires that, we can check for inclusion between two symbolic states, and check for intersection between a symbolic state and $Init$. When performing inclusion checks between two double symbolic state, we will first compare the partial location vectors. If the two states does not agree in one of the automata where they both specify a specific location, then they neither intersect nor does the one include the other. After this the zones are checked for intersection by the method described in chapter 4.

Definition 5.6 : Inclusion

One double symbolic state $(\vec{p'}, Z')$ covers another $(\vec{p}, Z)$ if:

$$(\vec{p}, Z) \subseteq (\vec{p'}, Z') \iff \vec{p} \sqsubseteq \vec{p'} \wedge Z \subseteq Z'$$

$$\vec{p} \sqsubseteq \vec{p'} \iff \forall i \in M.p_i = p'_i \vee p'_i = *$$

The only type of intersection check performed is intersection with $Init$. This is done first by checking if the partial location vector contains the location

vector of $Init$. After this we can perform an intersection check between the two zones. Because of the fact that we take both a discrete and a delay step, in each exploration step, we actually perform this intersection check with a zone $Z_{init}{}^{\nearrow}$, that is the future operation performed on the initial zone Z_{init}, where all clocks are zero.

In the previous section we have fulfilled the following requirements. A finite number of representable symbolic states, and a way to check for inclusion and intersection for such states. A finite sequence of partitionings, with a final partitioning with the desired property. And a symbolic predecessor function, in compliance with assumption 2.20. Having fulfilled all of the requirements for the symbolic CBR framework, we can conclude the correctness of the algorithm, when applied to the domain of TAN.

6 Extensions

This chapter describes certain extensions, which can be added to the model of timed automata, and the effect that these have on the compositional backwards reachability analysis. The extensions are; integers, invariants, urgent locations, urgent channels, and committed locations. We deal with exactly these extension because they are the ones implemented in UPPAAL. Each of the extension will be described in the following sections.

6.1 Integers

In this section we will first describe how integers can be used in UPPAAL. After this we will discuss the possibility of adding this to the CBR for TA method. We choose integers with some simple operations, and show how the CBR for TA method from chapter 5 can be extended.
In UPPAAL one can use both simple integers and arrays of integers. The integers can be used in guards, and in assignments. Examples of integer guards are: `L < 2`, `I == 4`, and `I <= L * 2`. Similarly we can give some examples of integer assignments: `L := 2`, `I := L / 2`, and `L := L + 1`. Both the guards and assignments can contain addition, subtraction, multiplication, and division. There is also a possibility of using a maximum and a minimum function.
It would be possible to implement all of this in the CBR for TA method. A suitable data structure could be binary decision diagrams (BDD). A BDD could represent the possible values that an integer could have in a give symbolic state. The complicated part is to calculate, which possible values, an integer could have had before it was assign the current value. In the implementation described in chapter 7 we have chosen a simpler solution. We have only two possible representations of an integer, either a concrete value or * (star) denoting any possible value. We allow only certain simple guards and assignments. The guards can only be of the form: $L == 3$. Where an integer is tested for equality with a constant. This gives us the advantage that after having taken a backwards step, with an integer guard on the transition, we know the exact value of the integer. In the assignments we only allow the use of, addition, subtraction, constants, and the integer to which the value is

being assigned. Examples of such integer assignment are: `L := 2`, `L := L + 1`, and `L := 1 - L`. This makes it easier to calculate the value of the integer prior to the assignment, because there always will be only one such value. When taking a backwards discrete transition, the calculation of integer values is carried out in two steps. First we use the assignment to calculate an intermediate value. Here we have four possible scenarios, illustrated by four examples in figure 6.1.

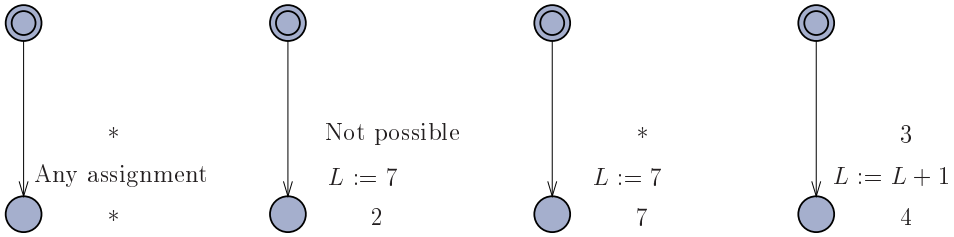

Figure 6.1: Illustrates how we calculate the intermediate value of an integer based on the value after the assignment and the assignment.

In the first case, if the integer can have any value after the assignment, it could also have had any value before the assignment, because we use unbounded integers. In the second case, if the integer is assigned a constant and it does not have this value after the assignment, then we know that this transition cannot be taken into such a symbolic state. The third case illustrates the case where the integer has exactly the value that is assigned to it. In this case we know nothing about the prior value of the integer, which is then $*$. In the final case the integer has a concrete value and it is either incremented or decremented in the assignment. In this case we can calculate what value it must have had before the assignment. There actually exists one last case. The case where we have no assignment. If this is the case, the integers intermediate value is the same as the value after the assignment.
After having calculated the intermediate value, we must check if the intermediate value agrees with the value in the guard. Here we only have three cases, illustrated in figure 6.2.
In the first case if the intermediate value is $*$, we know that the assignment could have been true. If the integer has a concrete value that is exactly the same as in the guard we know that the guard was true. In both of these cases we know that the value of the value of the integer must have been the value in the guard. The last case represents where the intermediate value and the value in the guard disagree. In this case the transition cannot lead us to the symbolic state in question.

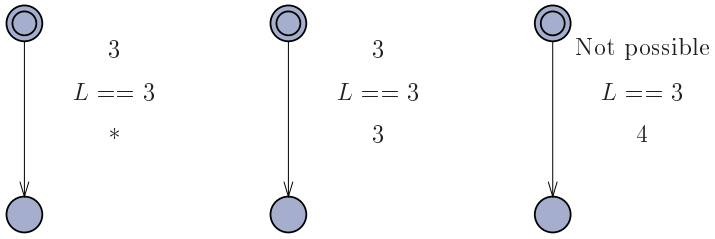

Figure 6.2: Illustrates how we calculate the value of the integer before the guard, depending on the intermediate value of an integer.

When we want to extend the CBR for TA method with integers. First we define the new representable symbolic states (RSS). The concrete states will now have the form $(\vec{l}, \vec{i}, Z)$ instead of the form $(\vec{l}, Z)$, where $\vec{i} = (i_1, \ldots, i_n)$ with $i_m \in \mathbb{Z}$. We limit guards to the form $i := c$ where c is a constant and assignments to the form $i := d$, where d is composed of the integer i itself, constants, addition and subtraction. The symbolic states will now have the form $(\vec{p}, \vec{q}, Z)$ instead of the form $(\vec{p}, Z)$, where $\vec{q}$ is defined as partial location vectors, just for integers. We also need an extra index set $I \subseteq (i_1, \ldots, i_n)$, to range over the integers. With this we can define partial integer vector as follows.

$$\vec{q_i} = (q_1, \ldots, q_n) \text{ where } \begin{cases} m \in I & q_m \in \mathbb{Z} \cup \{*\} \\ m \notin I & q_m = * \end{cases}$$

We will obtain new equivalences based on $=_{M,K}$ and the equivalence of the integer values. We will denote these new equivalences by $=_{M,K,I}$. The new symbolic predecessor function $\Longleftarrow_{M,K,I}$ will be $\Longleftarrow_{M,K}$ with the added concept of integers.

6.2 Invariants

In this section we describe what invariants are, and possible solutions on how to include them in the backwards reachability analysis. An invariant is an upper bound on clock values in a given state. For each state we can have a requirement that a set of clocks does not exceed some value. Having invariants in the system changes the computation of the zones of the states that can reach a current state. The problem lies in the fact that we can no longer guarantee that we can delay indefinitely backwards. We illustrate this fact by an example. In figure 6.3 we have two simple timed automata. The system contains two invariants one in state `A1` and one in state `B1`. The problem arises from the fact that the system can time deadlock, meaning

it can enter a state in which no further time can elapse. This happens if first time elapses such that $x == 7$ and automata `B` takes a transition into `B3`. After this time elapses such the $y == 9$. Now no more time can elapse in state `A1`, because of the invariant. At the same time the one outgoing transition is not enabled, hence we have a time deadlock.

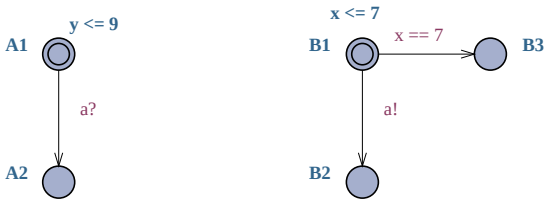

Figure 6.3: Two simple timed automata A and B.

Time deadlock poses a problem for compositional backwards reachability analysis. In order to detect if we are taking a backwards step over a time deadlock, we have to consider all components. This strongly contradict the compositionality. One possible solution to this problem is to restrict the models on which the method works. We want models that cannot time deadlock, which is described by the following property.

Definition 6.1 : No Time Deadlock

$\forall(\vec{p}, v)$ that is reachable.

$\exists(\vec{l}, v) \xrightarrow{\tau} \xrightarrow{\epsilon(d_1)} \xrightarrow{\tau} \xrightarrow{\epsilon(d_2)}$ ..... such that $\lim_{n\to\infty}(\sum_{i=1}^{n}(d_i)) \to \infty$

We believe that the following syntactical properties ensures that a system never time deadlocks:

- Each state that has an invariant must have an outgoing tau transition which is enabled when the invariant prevents any further delaying. If no such tau transition exists one must be added that leads to a special deadlock state.
- There may not be any cycles in the model in which time does not elapse. This can be ensured by checking that, in each cycle there exists a clock that is reset and later checked to be larger than a non-zero constant.

We have no proof of these assumptions and this is a very interesting area for future work.

6.3 Urgent Locations

A location in an automata can be specified as being urgent. No time can elapse in the system while an automata is in an urgent location. Urgent locations can simply be modeled by the use of invariants, so if we have a solution for invariants we have also solved the problem of urgent locations. An extra clock x_u is added which is reset on all transitions going into the urgent location. Then an invariant, which specifies that no time can elapse, $x_u \leq 0$, is added to the urgent location.

6.4 Committed Locations

A location in an automata can be specified as being committed. Similarly as for urgent locations, no time can elapse when an automata is in a committed location. There is also the extra requirement that no other automata may take any transitions before the one automata has left the committed location. Before the reachability analysis is started, the set of committed locations is examined. Any committed location that has an outgoing tau-transition is registered as well as any pair of committed locations that can synchronize. Whenever in the backwards reachability analysis that we take a backwards delay or discrete transition, we know that none of the automata not in M could have been in any of the previously mentioned states or combinations of states. If the committed location is in an automata in M we will treat them as urgent locations. When we take a backwards step into a committed location or a pair of committed locations, we cannot delay due to the invariants that was added by the treatment as urgent locations. In addition to this we must only look at transitions that can bring us out of the committed locations again, when choosing the next discrete step.

6.5 Urgent Channels

An urgent channel is a channel on which the automata must synchronize as soon as they are able to. As for the committed locations we must register all pairs of states that can synchronize over an urgent channel before the reachability analysis begins. When such a pair is not included in M, we know when we delay, that all the components outside of M cannot be in these location combinations. For pairs of locations where the one automata is included in M, and the other one is not, we must do the following. If the automata in M isn't in the location that can synchronize over an urgent channel, we must do nothing. On the other hand if it is in this exact location, we can conclude

that the automata outside M, with which it could synchronize, isn't in the corresponding location.

7 Implementation

A test implementation of the CBR method was created in order to produce some experimental results. This implementation will in the rest of the report be known as Cbr-verifyta. In the next chapter the experimental results from Cbr-verifyta will be compared with results from Uppaal. This chapter describes what the test implementation includes and how it was implemented.

7.1 Code Reuse

The CBR test implementation is implemented in the programming language `C++`. This is done in order to be able to use the Uppaal source code as a basis for the development of a test implementation of the CBR for TA method. Firstly the parsers, both for the models and the verification properties, could be reused. This results in the fact that, the same models and verification properties can be fed to both Uppaal and Cbr-verifyta. A result of using the Uppaal source code as a basis, was that there was no real design faze. The Uppaal source code was slowly replaced and changed to transform it into Cbr-verifyta. The remaining sections will describe parts of the implementation. We will only describe things that has not been covered elsewhere in the report. Although the double symbolic states and the symbolic predecessor function represents the main part of the implementation effort, they will not be described in this section. This is due to the fact that they have already received thorough treatment.

7.2 Focus of the Implementation

Due to the limited time resources, we have in this project it wasn't an option to do a full implementation of the CBR method. The priority was on being able to compare CBR and Uppaal by being able to verify relevant properties on a set of models. The main deficiency of the implementation is it's inability to handle verification properties containing negations or properties starting with `A[]`. The problem lies in the step where symbolic states are

generated from the parsed property. If first the symbolic states has been generated, there is no set of states, that we cannot check the reachability of. In the implementation we also chose to implement simple integers as described in section 6.1 because many of the models contains integers, and only use them in such a simple fashion. A point where the method differs from the algorithm, is that there is performed an extra inclusion check when inserting a state into the waiting list, to avoid duplicate states in the waiting list. This is inspired by UPPAAL, which does exactly the same, although it is not described in the forward reachability algorithm of section 4.3.1.

7.3 Dependency Analysis

In this section we will first describe the purpose of the dependency analysis. After this we consider how to perform the analysis on a timed automata network (TAN). Finally we consider some heuristics for different orders in which to include the components.
The dependency analysis is carried out in order to avoid doing unnecessary work. If we can show that a number of components are dependently closed, and that the property we intend to verify only concerns components from within the dependency closed set, then we know that we only need to include the components from the dependency closed set, in order to check the property. Having a dependency closed set means that no matter how the components outside the set behave, they cannot affect the set of states, that the dependency closed set can reach. This analysis is carried out before the actual verification.
In a TAN we have three types of components; automata, clocks, and integers. These components depend on each other in different ways. First of all automata can depend on each other by use of the same channel to communicate. All automata that write to a given channel `a!` depends on all the automata that read from the channel `a?`. Likewise each reader of a channel depend on every single writer to the same channel. Integers can only depend on automata. An integer depends on the automata that assigns it a value on one of it's transitions. Likewise clocks only depend on automata that it is resets by. Finally automata also depends on the integers and clocks used in a guard on one of it's transitions. Using these rules, we build a dependency graph. Starting with the set of components used to represent *Goal*. We simply add all the components that these components depend on. In this way, when we have no more components to include, we have reached a dependency closed system. This need not bee all components in the system. As a heuristic for which components to add we have chosen a very simple

one, of in each step adding all the components that the current components depend directly on. In the implementation we already count the number of dependent relationships between two components. So that for instance for each reset of a clock appearing in an automata, the clock depends with one point on the automata. These numbers gives some sort of representation of how closely connected two components are. One could easily imagine this being utilized in some form of heuristics where a certain number of point had to be added for each step. Coming up with some good heuristics could be a possible direction of future work.

8 Experimental Results

In this chapter we will compare the performance of the CBR method (CBR-VERIFYTA) against both full backwards reachability (FBR) and the algorithm used in the UPPAAL tool. What we mean by FBR will be explained in section 8.2.1. We choose to compare CBR and UPPAAL in terms of the number of inclusion checks and exploration steps because these are the most complex operations of the algorithms. We do not want to measure the execution time, because this will show the efficiency of the implementations instead of the relative strength of the individual methods.

8.1 Performance Parameters

We choose inclusion checks and exploration steps, as performance parameters because we believe them to be the dominant factors in the execution time. Inclusion checks are performed in two locations in the algorithms. Before a state is explored it is checked against the passed list to see if it has already been explored. Inclusion checks are also performed when inserting a state into the waiting list. Exploration steps represent the number of times a state is explored, i.e. the number of times we look at one state and determine what new states we can reach by a forward or a backwards step. The UPPAAL version used in this section is 3.2 Beta 5 (3.1.68) of September 2001.

8.2 Test Cases

As test cases we choose the two problems also described in the previous report: The soldiers problem and Fischer's mutual exclusion algorithm. Another reason for choosing these problems is that they are standard problems that are distributed as examples with UPPAAL, and the fact that they can be scaled in size.

8.2.1 Fischer's Mutual Exclusion Algorithm

The purpose of Fischer's mutual exclusion algorithm is to ensure that a number of processes all can have access to a shared resource, but never at the same time. In figure 8.1 we show a prototype for the processes in Fischer's algorithm. These prototypes are created using the graphical user interface for UPPAAL. From this prototype we can save a system with the desired number of processes. In each process the `pid` is then replaced by a unique constant, not zero.

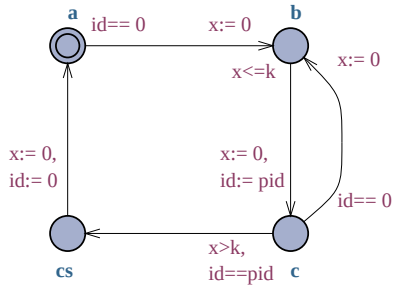

Figure 8.1: Prototype for each process in Fischer's algorithm.

We use Fischer's mutual exclusion algorithm to test CBR against both UPPAAL and FBR. We verify a property that the first two processes can both reach the critical section at the same time. This can be written as: `E<> (P1.cs and P2.cs)`. This property is never satisfied for a correct constructed Fischer's algorithm, so in all the test cases the answer is NO. We verify this property for different numbers of processes. The reason we do not change the property to include all processes, is that if we did so the compositional aspect of the CBR method would not be tested. We achieve the full backwards reachability method (FBR) method by rewriting the property to all possible combinations of states that are covered by the symbolic state generated by `E<> (P1.cs and P2.cs)`. By using CBR-VERIFYTA to verify this new property, we are guaranteed that all components are included from the start, and hence we get what corresponds to full backwards reachability analysis. The FBR property for three processes is:

```
E<>( (P1.cs and P2.cs)
     and (P3.a or P3.b or P3.c or P3.cs)
     and (id == 0 or id == 1 or id == 2 or id == 3))
```

This property is rewritten for each number of processes, by adding an extra line for each process, and an extra possible value for `id`. For four processes the property is:

```
E<>( (P1.cs and P2.cs)
     and (P3.a or P3.b or P3.c or P3.cs)
     and (P4.a or P4.b or P4.c or P4.cs)
     and (id == 0 or id == 1 or id == 2 or id == 3 or id == 4))
```

Uppaal is tested both with and without an optimization option `-a`, which tells it to detect inactive clocks. This option improves the performance of UPPAAL for the models in question. Table 8.1 contains the number of inclusion checks performed by each method in verifying the property on models of different size. Likewise table 8.2 contains the number exploration steps.

	Inclusion Checks			
# processes	UPPAAL	UPPAAL -a	FBR	CBR
2	81	66	39	27
3	967	593	344	393
4	14729	6850	2247	1197
5	275391	97077	12679	2818
6	6113281	1633538	65537	5556

Table 8.1: Inclusion checks. Fischer's algorithm.

	Exploration Steps			
# processes	UPPAAL	UPPAAL -a	FBR	CBR
2	29	23	17	12
3	301	181	108	85
4	4121	1889	563	168
5	70381	24701	2658	283
6	1441885	387925	11833	430

Table 8.2: Exploration steps. Fischer's algorithm.

From table 8.1 and 8.2 we can see that both FBR and CBR performs significantly better than UPPAAL, both with and without the `-a` option. CBR also outperforms the FBR method. In table 8.3 and 8.4 we will calculate the factor by which the number of operations grow when we increase the number of components. We calculate this growth factor by dividing the number of operations required for 3 processes with the number of operations required for 2 processes, and so on. For both versions of UPPAAL the growth factor increases as the number of processes increase, indication greater than exponential growth. For both FBR and CBR the growth factor decreases as the

number of processes grow, indicating sub exponential growth. The growth is still far from linear. We cannot conclude that this will be the case for all models, but for this particular case CBR-VERIFYTA has both the lowest number of operations and the lowest growth factor. The only exception for this is the growth from two to three processes, where FBR has a lower growth factor.

	Inclusion Checks			
# processes	UPPAAL	UPPAAL -a	FBR	CBR
3/2	11.9	9.0	8.8	14.6
4/3	15.2	11.6	6.5	3.0
5/4	18.7	14.2	5.6	2.4
6/5	22.2	16.8	5.2	2.0

Table 8.3: Growth factor. Inclusion checks. Fischer's algorithm.

	Exploration Steps			
# processes	UPPAAL	UPPAAL -a	FBR	CBR
3/2	10.4	7.9	6.4	7.1
4/3	13.7	10.4	5.2	2.0
5/4	17.1	13.1	4.7	1.7
6/5	20.5	15.7	4.5	1.5

Table 8.4: Growth factor. Exploration steps. Fischer's algorithm.

When verifying a property that cannot be satisfied UPPAAL will eventually search the entire reachable state-space. This means that no matter what unsatisfiable property we verify on the models used above, we will get the exact same number of operations. On the other hand the efficiency of the FBR and CBR methods is very dependent on the property we want to verify.

8.2.2 Soldiers Problem

We choose this problem for several reasons. It is compositional in nature, it is distributed with UPPAAL and it was also analyzed in the previous report [Lar02]. In the previous report it was remodeled compared to the version distributed together with UPPAAL. Here we choose to include test data for both models.

The problem can be described as follows. A bunch of soldiers have to cross a river over a narrow bridge, in the middle of the night. They have only one torch, which they need to cross the bridge, at the same time the bridge can only carry the weight of two of the soldiers at a time. This means that when two soldiers have crossed the bridge one of them must walk back to the original side with the torch. So a new pair of soldiers can cross the bridge. The soldiers walk at different speeds across the bridge, and if two of them cross the bridge together, they walk at the speed of the slowest of the two. The usual question to solve is; can all the soldiers cross the bridge within X time units?

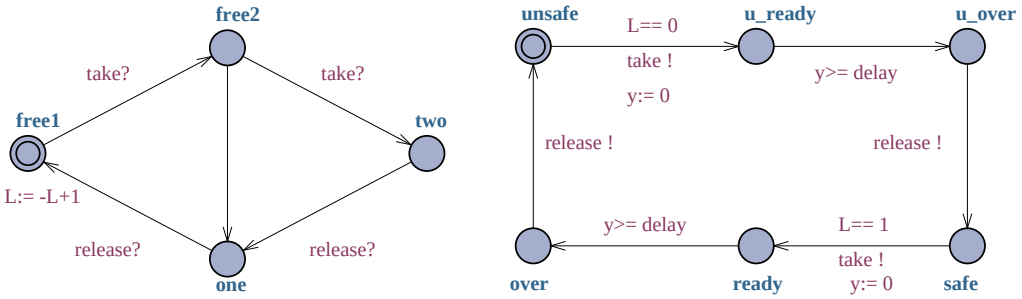

Figure 8.2: The Torch automata and the prototype for the Soldier automata.

In the version distributed with UPPAAL, shown in figure 8.2, the torch is modeled by an automaton, and so is each of the soldiers. In the version from the previous report, we modeled the location of the torch by an integer variable with three values. When one or two soldiers begin their journey across the bridge they change the value of the integer, such that no other soldiers can cross the bridge. When they get to the other side they change the value of the integer to reflect, on which side of the bridge the torch currently is located. In this model we do not use prototypes but individual automata. But the automata still resemble each other so much, that we have chosen to show only one of them in figure 8.3.

In the following the problem is modeled with four soldiers, `S1`, `S2`, `S3`, and `S4`. The four soldiers take respectively 5, 10, 20, and 25 time units to cross the bridge. In table 8.5 we show the properties that will be tested on both models. These properties will also be tested on UPPAAL, with the detect inactive clocks option.

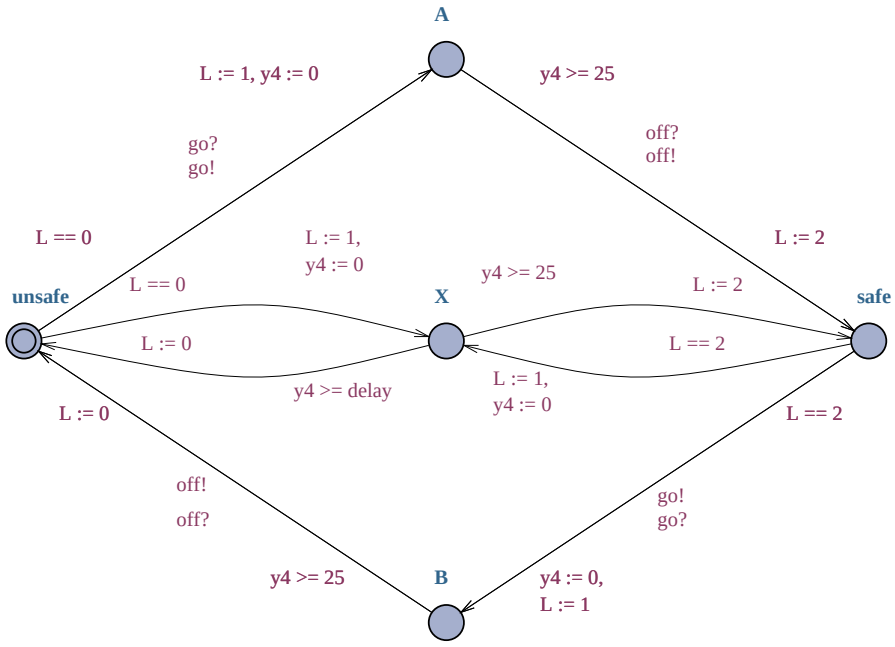

Figure 8.3: Soldier4, one of the timed automata used in this example. On this figure the transitions marked with two synchronization labels, e.g. go! and go?, are actually two transitions with the same reset sets and guards.

The percentage columns in table 8.6 and 8.7 are calculate by dividing the number of operations needed in the CBR method with the number of operations used in UPPAAL. The tables show that especially for the properties that are not satisfied, UPPAAL uses a lot fewer operations. This is also the case for the most complex property, property 8. It is only for the first two simple properties that the CBR method is consistently better. The remodeled soldiers problem generally requires both less inclusion checks and exploration steps. There is only one exception from this, which is the number of inclusion checks required by the CBR method for property number 1. With this model UPPAAL generally performs better than CBR-VERIFYTA.

8.3 Conclusion on Tests

The tests carried out in the previous section show varying results. In the most extreme case, Fischer's algorithm with six processes, where the CBR method is best, UPPAAL uses 29.401% more inclusion checks and 90.215% more exploration steps. These results are obtained by dividing the num-

1	`E<> S1.safe`	YES
2	`E<> S1.safe and S2.safe`	YES
3	`E<> S1.safe and S2.safe and S3.safe`	YES
4	`E<> S1.safe and S2.safe and S3.safe and S4.safe`	YES
5	`E<> S2.safe and S3.safe and S4.safe and time <= 39`	NO
6	`E<> S2.safe and S3.safe and S4.safe and time <= 60`	YES
7	`E<> S1.safe and S2.safe and S3.safe and S4.safe and time <= 59`	NO
8	`E<> S1.safe and S2.safe and S3.safe and S4.safe and time <= 60`	YES

Table 8.5: Properties to be verified on soldiers problem.

	Inclusion Checks			Exploration Steps		
Property	UPPAAL -a	CBR	%	UPPAAL -a	CBR	%
1	139	21	15.1%	42	9	21.4%
2	226	77	34.1%	80	27	33.8%
3	876	898	102.5%	288	271	94.1%
4	1485	3724	250.8%	468	1057	225.9%
5	1709	26714	1563.1%	534	5869	1099.1%
6	892	1527	171.2%	294	465	158.2%
7	1707	9139	535.4%	532	2701	507.7%
8	1497	10445	697.7%	474	3103	654.6%

Table 8.6: Verification of different properties on the original model of the soldiers problem. With UPPAAL -a option.

ber of operation used by UPPAAL with the number of operations used by CBR-VERIFYTA. In the case where CBR-VERIFYTA uses the most operations compared to UPPAAL, it uses 1563% more inclusion checks and 1099% more exploration steps. We can conclude that neither the one nor the other method generally is better than the other. Which method that uses the fewest operations depend both on the model and on the property to be verified. For Fischer's algorithm, where CBR always used fewer operations, it outperforms UPPAAL by more than, what UPPAAL does for the Soldiers problem. The conclusion is that CBR has considerable strengths, when applied to the domain of TAN, and it is worth doing some extra work to try and explore the full potential of the method. It might be a possibility to develop heuristics that can help choosing if forward or compositional backwards analysis is best suited for a specific verification job. Another possibility would be to combine forward analysis with CBR and check for intersection between the forward and backwards reachable state-space. This method would properly perform

	Inclusion Checks			Exploration Steps		
Property	UPPAAL -a	CBR	%	UPPAAL -a	CBR	%
1	49	30	61.2%	12	3	25.0%
2	55	33	60.0%	13	4	30.8%
3	249	646	259.4%	56	92	164.3%
4	436	3031	695.2%	96	441	459.4%
5	496	6189	1247.8%	113	1096	969.9%
6	227	810	356.8%	61	128	209.8%
7	496	6598	1330.2%	113	1036	916.8%
8	447	6630	1483.2%	99	1043	1053.5%

Table 8.7: Verification of different properties on the remodeled soldiers problem. With UPPAAL -a option

worse for properties that cannot be satisfied, but might deliver better results for properties that are true.
To perform a thorough test of different verification methods, we really need a very broad spectra of real-world verification scenarios, instead of two classical verification examples. I spite of the limited test material we can still conclude that CBR for TA is a potentially very efficient verification method.

9 Conclusion

In this chapter we will first discuss possible directions for future work. Finally we will conclude on the different parts of the report.

9.1 Future Work

In this section we will describe several directions for future work.
The most important direction of future work, for the usefulness of the CBR for TA method, is the extensions described in chapter 6. If the CBR for TA method should be a serious competitor to Uppaal, one would have to be able to handle all of these extension. The major challenge lies in handling invariants.
Another natural line of future work would also be to develop a full implementation of the CBR for TA method described in this report. A remaining field of work is also to test the efficiency of different data structures to hold the past and waiting list. This implementation should off cause include the extensions if solutions are found on ways to handle invariants and so on.
A third option is to look into the possibility of combining CBR with other verification methods. As mentioned in section 8.3 it would first of all be a possibility to combine CBR with some form of forward reachability analysis. Both by doing forward a backwards reachability analysis at the same time. But also by creating some heuristics to determine whether the CBR method or forward reachability analysis is best suited for a particular verification task. Yet another possibility is to design heuristics for the order in which to include the components in the analysis.
Finally one could apply the CBR method to some new domain. With the new generalized framework, the CBR method should be directly applicable to many useful domains, both discrete and real-time domains.

9.2 Conclusion

We have in this report succeeded in developing two versions of a more general CBR framework. We have proven the correctness of the two accompanying

CBR algorithms. The one based on the original CBR algorithm and the other based on the concept of symbolic states. We specified what is required of a domain for each of the two CBR methods to be applicable. We have as in the previous report [Lar02] introduced the domain of Timed Automata (TA), and the algorithm implemented in UPPAAL. We apply the CBR method to the domain of TA by fulfilling the requirements, and thereby without having to reprove the correctness of the algorithm. Several extensions of the domain were also considered. A test implementation has been developed and experimental results have shown the potential of the CBR for TA method. The experimental results show that the effectiveness of the method depends on both the model and the property to be verified.
The conclusion of the report is that CBR for TA is a very powerful method for some models, and the method is worth further investigation.

10 Danish Resume

Denne rapport beskæftiger sig med udvikling af en metode til Kompositionel Baglæns analyse af om tilstande kan nås. Denne metode hedder Compositional Backwards Reachability (CBR). I kapitel 2 beskrives den generelle CBR metode, der baserer sig på en finere og finere partitionering af tilstands rummet. Først introduceres en algoritme inspireret af den oprindelige CBR algoritme fra [LNAB+98]. Herefter udvikles en lignende algoritme, som baseres på brugen af symbolske tilstande. Korrektheden af begge algoritmer vises og betingelser opstilles for anvendelsen af metoden på et givet domæne. I kapitel 3 introduceres Tids Automater (Timed Automata (TA)). I kapitel 4 forklares den analyse metode for TA, som bruges i værktøjet Uppaal. I kapitel 5 anvendes CBR metoden på domænet TA ved at opfylde betingelserne for den symbolske algoritme. I kapitel 6 diskuteres muligheder for udvidelser af TA domænet. Disse udvidelser betragtes da det netop er dem, der er implementeret i værktøjet Uppaal. Herefter beskrives i kapitel 7 den test implementation, som er blevet udviklet med grundlag i kildekoden fra Uppaal. Denne implementation bliver i kapitel 8 sommenlignet med blandt andet Uppaal, for at undersøge CBR for TA metodens styrker og svagheder. Til slut beskrives mulige retninger for fremtidigt arbejde og en konklusion drages i kapitel 9.

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AL] Henrik R. Andersen and Kim G. Larsen. Kompositionel og trinvis analyse af tilstandssystemer baseret på afhængighedsanalyse. Part of patent application.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proc. 1998 Computer-Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[Bey01] Dirk Beyer. Improvements in BDD-based reachability analysis of timed automata. In *FME*, pages 318–343, 2001.

[BLL+98] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Yi Wang, and Carsten Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[BLP+99] G. Behrmann, K. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams . In *Proceedings of CAV99*, pages 22–24. Springer Verlag, 1999.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[Kat98] Joost-Pieter Katoen. Concepts, Algorithms, and Tools for Model Checking. Lecture Notes of the Course "Mechanised Validation of Parallel Systems", Friedrich-Alexander Universität Erlang-Nürnberg, 1998.

[K.L93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[KLL+97] K. J. Kristoffersen, F. Laroussinie, K. G. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *Proc. 7th Int. Joint Conf. Theory and Practice of Software Development (TAPSOFT'97), Lille, France, Apr. 1997*, volume 1214, pages 565–579. Springer, 1997.

[Lar02] Ulrik Larsen. Dat5-report: Compositional backwards reachability for simple timed automata. January 2002.

[LNAB+98] Jørn Lind-Nielsen, Henrik Reif Andersen, Gerd Behrmann, Henrik Hulgaard, Kåre Kristoffersen, and Kim G. Larsen. Verification of Large State/Event Systems using Compositionality and Dependency Analysis. In *TACAS'98 Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, 1998.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[Min99] Marius Minea. *Partial Order Reduction for Verification of Timed Systems*. PhD thesis, School of Computer Science Carnegie Mellon University, December 1999.

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, 1999.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.