Partial Order Reduction for Timed Systems

Master of Science in Applied Computer Science

Serdar Oguz ATA

Examiners: Prof. Dr. Bernhard Nebel

Supervisors: Yusra Alkhazraji
Dr. Robert Mattmüller

ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Albert-Ludwigs Universität Freiburg
Faculty of Engineering
Georges-Köhler-Allee 101,
79110 Freiburg, Germany
07.08.2014

Declaration

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg,
07.08.2014

Serdar Oguz ATA

Summary

In this work, an application of a partial order reduction method for the reachability analysis of timed systems using local-time semantics is presented. Reachability analysis of timed systems requires exploring an excessive size of the state-space. The size of the state-space can be reduced by using partial order reduction methods. Partial order reduction methods make use of the independence relation between transitions. Two transitions are independent if they lead to the same state with any execution order. Exploring one of the two possible execution order is sufficient to reach the same states. Partial order reduction methods choose one of the two paths and prune the other one to decrease the size of the state space to be explored.

Application of partial order reduction methods to timed systems is not straightforward, since all of the transitions which include a clock variable are implicitly synchronized. Thus local-time semantics is proposed to enable the application of partial order reduction methods to timed automata. The idea of local-time semantics is to remove the implicit clock synchronization between the local clocks of different processes to allow local clocks to advance with different time scales. Using local-time semantics allows usage of independence of transitions in different processes thus allowing application of partial order reduction techniques to timed automata.

In this paper, stubborn sets are used as the partial order reduction method. Local-time semantics and stubborn sets are implemented in MCTA, a model checking tool for timed automata. The implementation is tested with the benchmarks available for MCTA and test results showing the size of explored state-space with global time semantics, with local time semantics and with local time semantics and stubborn sets are added.

Zusammenfassung

In dieser Arbeit wird eine Anwendung der sogenannten Partial-Order-Reduction Methode für die Erreichbarkeitsanalyse von Realzeitsystemen mit "Local-time Semantics" vorgestellt. Die Erreichbarkeitsanalyse von Realzeitsystemen erfordert die Erkundung eines exponentiell Größen Zustandsraumes. Die Größe des Zustandsraumes kann durch Partial-Order-Reduction reduziert werden. Partial-Order-Reduction Methoden nutzen die Unabhängigkeitsrelation zwischen den Übergängen. Zwei Übergänge sind unabhängig, wenn sie in den gleichen Zustand mit jeder Ausführungsreihenfolge führen. Die Erforschung einer der möglichen Reihenfolgen der Ausführung ist ausreichend, um die gleichen Zustände zu erreichen. Die Partial-Order-Reduction Methode wählt einen der beiden Pfade und beschneidet den anderen, um die Größe des zu erkundenden Zustandsraumes zu verringern.

Die Anwendung der Partial-Order-Reduction auf Realzeitsysteme ist nicht einfach, weil alle Übergänge, die eine Uhren-Variable enthalten, implizit synchronisiert sind. Daher wird die sogenannte "Local-time Semantics" vorgeschlagen, die die Anwendung der Partial-Order-Reduction auf Realzeitsysteme ermöglichen. Die Idee von Local-time Semantics ist, die implizite Zeitsynchronisierung zwischen den lokale Uhren von verschiedenen Prozessen zu entfernen und lokalen Uhren mit unterschiedlichen Zeitskalen fortschreiten zu lassen. So kann die Partial-Order-Reduction mit Realzeitautomaten verwendet werden.

In dieser Papier wird die Stubborn-Sets als Partial-Order-Reduction Methode verwendet. Local-time Semantics und Stubborn-Sets wurden in MCTA, einer Model-Checking-Anwendung für Realzeitautomaten implementiert. Die Implementierung wird mit den für MCTA verfügbaren Benchmarks getestet und die Testergebnisse präsentiert.

Table of Contents

1 Introduction 6
2 Timed Automata 7
2.1 Operational Semantics 8
2.2 Regions and Zones 10
2.3 Symbolic Semantics 13
2.4 Zone-Normalization 14
2.5 Symbolic Reachability 16
2.6 Difference Bound Matrices (DBMs) 18
3 Local-Time Semantics 19
3.1 Symbolic Local-Time Semantics 23
4 Partial Order Reduction 26
4.1 Stubborn Sets 26
4.2 Algorithm 28
5 Experimental Results and Conclusion 33
5.1 MCTA 33
5.2 Experimental Results 34
5.3 Conclusion 39
Bibliography 41

1 Introduction

For timed systems, model checking is an important verification technique. For this purpose, several tools, such as UPPAAL [1], [2], KRONOS [3] and MCTA [4], were developed. However, performing model checking for large networks of timed automata requires exploring a large search space which requires a huge memory usage [5]. The search space to be explored increases exponentially in the number of components and clocks, since the clock values, which may differ for the same untimed state when reached by different paths, are also kept as the part of timed states.

For finite-state systems, several *partial order reduction* methods [6], [7], [8], [9] were introduced in order to decrease the size of the explored state-space, thus reducing the memory and time required for model checking problems. Partial order reduction methods avoid exploring redundant interleavings of independent transitions. Intuitively, executing a sequence of independent transitions in different orders leads to the same final state, and partial order reduction methods ensure that only a reduced state-space is explored by preventing the expansion of unnecessary interleavings instead of exploring the whole state-space by executing all of the transitions.

Partial order reduction methods decreases the size of the explored state-space when applied to finite-state systems. However, for timed systems, due to the implicit synchronization of transitions caused by the clocks which increase at the same rate, the application of partial order reduction methods is not straightforward. Since the clocks advance with the same speed, the processes which include at least one clock variable becomes dependent on all other processes that include a clock variable.

In [10], independence of transitions is defined for timed automata. Basically, if executing two transitions in any order results in the same state, control vector and clock assignment, they are independent. In this approach two transitions can only be independent if they are executed in the same global time interval.

Partial order reduction is applied to Petri nets, which is less expressive than timed automata since it only has earliest and latest firing times of transitions [11]. In this work, relative firing order of enabled transitions is stored so a state implicitly keeps the history of the system. Also synchronization is not allowed as the states only has the relative differences of firing times of the transitions.

In [12], *local-time semantics* is introduced to make partial order reduction methods applicable to timed automata. The idea of local-time semantics is removing the implicit synchronization of the clocks in different processes. The clocks in different processes are allowed to advance independently and the clocks of two different processes are resynchronized only if those two processes need to communicate. This *desynchronization* allows different interleavings of independent transitions to forget their order of execution. Thus the par-

tial order reduction methods designed for finite-state systems can be applied to timed automata using local-time semantics. In this paper, the local-time semantics explained in [12] is implemented. In [12], *ample sets* is used as a partial order reduction method for timed systems; however, in this paper *stubborn sets*([6]) is implemented instead.

The local-time semantics presented in [12] is redefined with different notation and proven for model checking in [13], [14]. Ample sets are defined and used as a partial order reduction method. The local-time semantics and ample sets are implemented for UPPAAL [15]. The results show that the performance is improved when local-time semantics and ample sets are used.

This paper is organized as follows: General information about timed automata, symbolic semantics, symbolic reachability and the necessary tools for implementation of timed automata is given in Section 2. In Section 3, local-time semantics for networks of timed automata and symbolic local-time semantics are explained. In Section 4, stubborn sets which is the partial order reduction method used in the implementation is explained. Finally in Section 5, experimental results are given with the conclusion of the paper.

2 Timed Automata

The theory of timed automata is used to model the behaviour of real-time systems. A timed automaton is a finite automaton, consisting of finite number of locations and edges, extended with a finite set of real valued clocks initialized with 0 when the system starts and advance with the same speed. Guards of the edges may include clock constraints, which restrict the behaviour of the system and the edge can only be executed if the clock constraint is satisfied. The locations may include location invariants, which are clock constraints that the automaton can remain in a location as long as the location invariant is satisfied. The clocks can also be reset to zero as an effect of an edge.

To define a timed automaton formally, first some notations are required.

B is a finite set of labels. Labels can be *local* or *synchronizing*. Synchronizing labels have complements, for a synchronizing label called a, its complement is denoted by $\bar{a}$.

For a finite set of real valued variables $\mathcal{C}$, called clocks, $\mathcal{B}(\mathcal{C})$ denotes the set of clock constraints which are conjunction of atomic constraints of the form $x \sim n$ or $x - y \sim n$ where $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, >, \geq\}$ and n is a natural number.

Definition 1 *[16] A timed automaton $\mathcal{A}$ is a tuple $\langle L, B, \mathcal{C}, I, E, l_{ini} \rangle$ where*

- *L is a finite set of locations,*
- *B is a finite set of labels,*
- *$\mathcal{C}$ is a finite set of clocks,*
- *$I : L \rightarrow \mathcal{B}(\mathcal{C})$ assigns a clock constraint to each location, called invariant,*
- *$E \subseteq L \times \mathcal{B}(\mathcal{C}) \times B \times 2^{\mathcal{C}} \times L$ a set of directed edges, edges (l, g, a, r, l') from location l to l' are labelled with a, with guard g, and a set of clocks r which are to be reset,*
- *l_{ini} is the initial location.*

A *network of timed automata* is a parallel composition of $A_1 \mid ... \mid A_n$ of a collection A_1, ..., A_n of timed automata. Each automaton is defined according to Definition 1. An edge $(l_i, g_i, a, r_i, l_i') \in E$ can also be written as $l_i \xrightarrow{g_i, a, r_i} l_i'$. The edge can be executed and the automaton A_i moves from l_i to l_i' performing label a and resetting the clocks in r_i if the guard g_i which is a clock constraint is satisfied. A *local action* is an edge $l_i \xrightarrow{g_i, a, r_i} l_i'$ of some automaton A_i with the local label a. A *synchronizing action* is a pair of matching edges, $l_i \xrightarrow{g_i, a, r_i} l_i' | l_j \xrightarrow{g_j, \bar{a}, r_j} l_j'$, where a is a *synchronizing label*, and for $i \neq j$, $l_i \xrightarrow{g_i, a, r_i} l_i'$ is an edge of A_i and $l_j \xrightarrow{g_j, \bar{a}, r_j} l_j'$ is an edge of A_j.

An example network of timed automata is shown in Fig. 1. In this example there are two separate automata (for simplicity no guards or reset operations are included). There are four edges $e1, e2, e3$ and $e4$ in the system. Two edges $e1$ and $e2$ form two local transitions, $s1 \xrightarrow{g_1, a, r_1} m1$ and $s2 \xrightarrow{g_2, b, r_2} m2$ respectively. The edges $e3$ and $e4$ have the same synchronization label and these two edges form a synchronization transition $m1 \xrightarrow{g_3, c, r_3} f1 | m2 \xrightarrow{g_4, \bar{c}, r_4} f2$.

For simplicity of the implementation of local-time semantics which will be explained later, it is required that the invariants should be conjunctions of constraints in the form $x \leq n$ where x is a clock and n is a natural number. It is also required that the set of clocks $\mathcal{C}_i$ for each automata A_i is disjoint, so each automaton only has local clocks. Also, the set of nodes N_i are pairwise disjoint.

2.1 Operational Semantics

The semantics of a network of timed automata is defined as a transition system where a state consists of the current location of each automaton and the current values of each clock. For a network of timed automaton $A = A_1 \mid ... \mid A_n$, a state can be shown as a pair (l, u) where l is a *control vector*, a vector consisting of the location of each automaton and u is a *clock assignment*, containing the current values for each clock. The states can be changed by performing one of three types of transitions ([16], [12]).

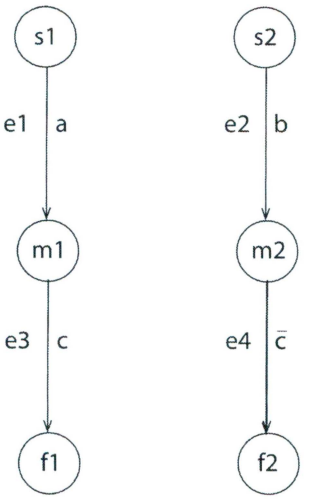

Fig. 1. A network of timed automata consisting of two different automata.

- Delay Transition: $\langle l, u\rangle \rightarrow \langle l, u+d\rangle$ if $u \models I(l)$ and $(u+d) \models I(l)$ where d is a non-negative real number.

- Local Transition: $\langle l, u\rangle \rightarrow \langle l[l'_i/l_i], u'\rangle$ where $l[l'_i/l_i]$ shows that the i^{th} element of the location vector l_i is replaced by l'_i if there is a local action $l_i \xrightarrow{g,a,r} l'_i$ such that $u \models g$ and $u' = [r \mapsto 0]u$ where $[r \mapsto 0$ indicate that the clocks in r are reset to zero.

- Synchronizing Transition: $\langle l, u\rangle \rightarrow \langle l[l'_i/l_i][l'_j/l_j], u'\rangle$ if there is a synchronizing action $l_i \xrightarrow{g_i,a,r_i} l'_i | l_j \xrightarrow{g_j,\bar{a},r_j} l'_j$ such that $u \models g_i$, $u \models g_j$ and $u' = [r_i \mapsto 0][r_j \mapsto 0]u$.

When a delay transition is executed, the locations of all of the automata remain the same, only the clocks advance as long as all of the invariants are satisfied by the new clock values.

A local transition $l_i \xrightarrow{g,a,r} l'_i$ can be executed if the guard g (enabling condition of the edge) is satisfied. The automaton moves from location l_i to l'_i and the clocks in r are reset.

A synchronizing transition $l_i \xrightarrow{g_i,a,r_i} l'_i | l_j \xrightarrow{g_j,\bar{a},r_j} l'_j$ is a pair of matching edges which are on automata A_i and A_j respectively ($i \neq j$). a is called the *synchronizing label*. The synchronizing transition can be executed if both of the guards g_i and g_j are satisfied. The automaton A_i moves from l_i to l'_i and A_j moves from l_j to l'_j and the clocks in r_i and r_j are reset.

Definition 2 *The state $\langle l, u\rangle$ is reachable from the initial state $\langle l_0, u_0\rangle$ iff there exists a transition sequence of the form $\langle l_0, u_0\rangle \xrightarrow{\lambda_1} \langle l_1, u_1\rangle \xrightarrow{\lambda_2} .... \xrightarrow{\lambda_n} \langle l, u\rangle$.*

2.2 Regions and Zones

Since the clocks are real-valued variables, the semantics of timed automata is an infinite transition system so it is not an appropriate model for automated verification. This problem can be overcome with the introduction of symbolic semantics. To explain the symbolic semantics, introduction of region and zone concepts are necessary.

Definition 3 *[17] Let $\mathcal{C}$ be set of clocks, $k(x) \in \mathbb{N}_0$ be the clock ceiling, which is the maximal clock constant in the automaton, for each clock $x \in \mathcal{C}$, and u, v are clock assignments of $\mathcal{C}$. For a real number d, $\lfloor d \rfloor$ denotes the integer part and $frac(d)$ denotes the fractional part. The clock assignments u, v are region equivalent and denoted by $u \simeq v$ if and only if the following conditions are satisfied*

- *For all $x \in \mathcal{C}$,*
  $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or both $\lfloor u(x) \rfloor > k(x)$ and $\lfloor v(x) \rfloor > k(y)$,

- *For all $x \in \mathcal{C}$ with $u(x) < k(x)$,*
  $frac(u(x)) = 0$ if and only if $frac(v(x)) = 0$,

- *For all $x, y \in \mathcal{C}$,*
  $\lfloor u(x) - u(y) \rfloor = \lfloor v(x) - v(y) \rfloor$
  or both $|u(x) - u(y)| > k$ and $|v(x) - v(y)| > k$,

- *For all $x, y \in \mathcal{C}$ with $-k < u(x) - u(y) < k$,*
  $frac(u(x) - u(u)) = 0$ if and only if $frac(v(x) - v(y)) = 0$

  where $k = max\{k(x), k(y)\}$.

Definition 4 *[17] For a valuation u, the equivalence class of u induced by $\simeq$ is denoted by $[u]$ and it is called a region.*

There are fixed number of clocks in a network of timed automaton. Also each clock has a maximal value (a clock ceiling), so the number of regions is finite. The region graph of a timed automaton with two clocks x, y is shown in Fig. 2. In this example, the clock ceilings of both clocks are 1 (i.e. $k(x) = 1$ and $k(y) = 1$). In this figure each region corresponds to a clock constraint. For example, the clock constraint $(x = 0) \wedge (y > 1)$ corresponds to the region marked with (a). The constraint $(x = 0) \wedge (y = 1)$ is represented with the region shown as (b). The regions (c) and (d) are equivalent to the constraints $(x = y) \wedge (0 < x) \wedge (x < 1) \wedge (0 < y) \wedge (y < 1)$ and $(0 < x) \wedge (x < 1) \wedge (0 < y) \wedge (y < 1) \wedge (y < x)$ respectively.

The equivalence classes can be used to construct a finite-state model called *region automaton*. The number of symbolic states (consists of location and a partition of the infinite

state-space of timed automata) in region automaton is finite but the number of regions increases exponentially with the number of clocks and the clock ceiling.

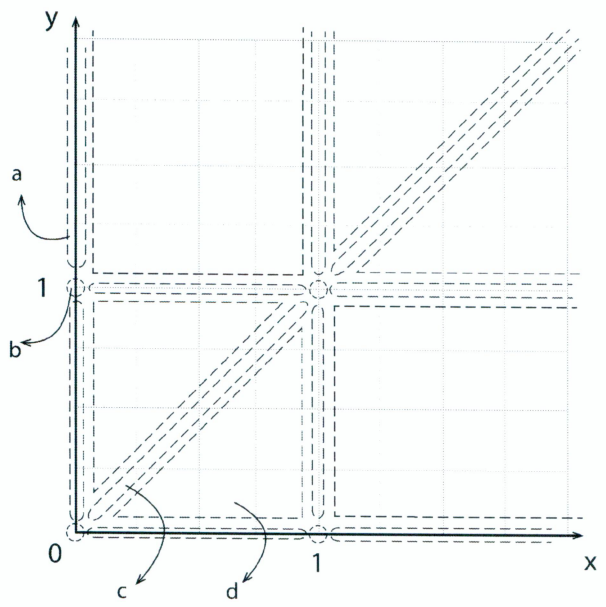

Fig. 2. The regions of a timed automaton with two clocks x and y where $k(x) = 1$, $k(y) = 1$.

Theorem 1. *[17] For a set of clocks $\mathcal{C}$, $k(x) \in \mathbb{N}_0$ the maximal constant for each $x \in \mathcal{C}$ and $k = max\{k(x)|x \in \mathcal{C}\}$, the upper bound for the number of regions is* $(2c+2)^{|\mathcal{C}|} * (4c+3)^{\frac{1}{2}|\mathcal{C}|*(|\mathcal{C}|-1)}$

Note that the result of the formula above is the upper bound not the exact number of regions. The exponential increase in the number of regions can also be observed in Fig. 3. In Fig. 3, again the region graph of a timed automaton with two clocks is shown, but for this case the clock ceilings are 3 (i.e. $k(x) = 3$ and $k(y) = 3$) and the number of regions is much higher.

The state-space for timed automata can be represented more efficiently with *zones* and *zone-graphs* ([18], [19], [20], [21], [22]). In zone graphs, the symbolic states are represented with zones instead of regions.

Definition 5 *[17] A zone is a set z of valuations of clocks $\mathcal{C}$ such that there exists $\varphi \in \mathcal{B}(\mathcal{C})$ with $v \in z$ if and only if $v \models \varphi$*

In other words, a zone is the maximal set of clock assignments satisfying a clock constraint.

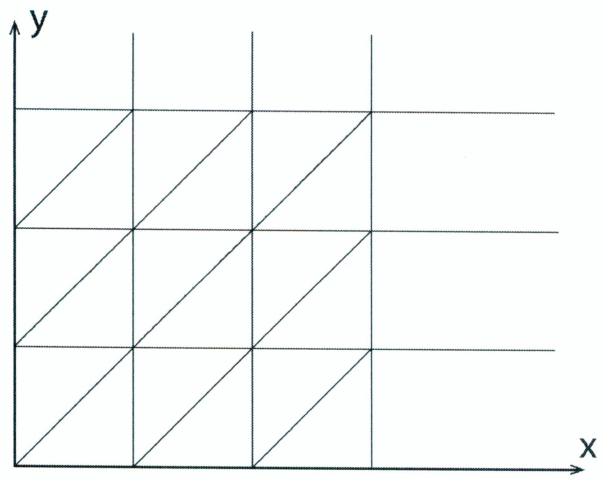

Fig. 3. A regions of a timed automaton with two clocks x and y where $k(x) = 3$ and $k(y) = 3$.

In Fig. 4, a zone example can be seen for a timed automaton with two clocks x and y. This zone represents the clock constraint $(x \geq 1) \wedge (y \geq 1) \wedge (x - y \geq 0) \wedge (x - y \leq 2)$.

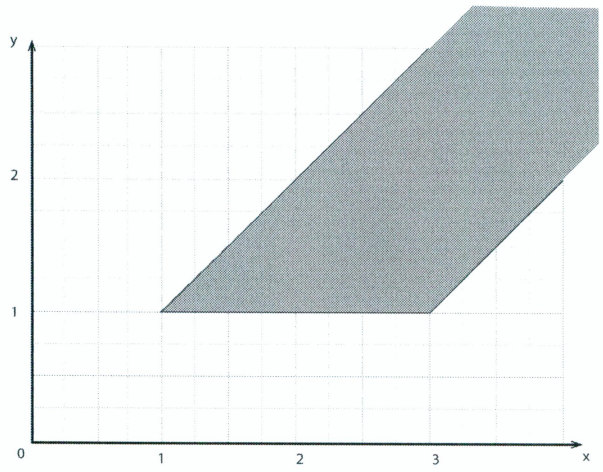

Fig. 4. An example zone for a timed automaton with two clocks x and y.

An efficient way to represent and store the set of clock assignments which represents a zone is using Difference Bound Matrices (DBMs) [23]. Intuitively, a zone can be represented as a conjunction of atomic clock constraints which can also be converted to difference constraints. These constraints can efficiently be represented with matrices which are called Difference Bound Matrices (DBMs). DBMs are explained in more detail in Section 2.6

2.3 Symbolic Semantics

For a clock constraint $D \in \mathcal{B}(\mathcal{C})$, the maximal set of clock assignments which satisfy D can be denoted by $[D]$. To simplify the notation, D will be used instead of $[D]$ for the rest of this paper.

A symbolic state of a network of timed automata is denoted by $\langle l, D\rangle$ where l is the location vector storing location of each automaton and $D \in \mathcal{B}(\mathcal{C})$ is a zone. $\langle l, D\rangle$ represents the set of all states $\langle l, u\rangle$ such that $u \models D$.

Definition 6 *[16] For a zone D, and a set of clocks z, the constraints $D^{\uparrow}$ and $r(D)$ are defined as follows:*

- $D^{\uparrow} = \{u + d | u \in D, d \in \mathbb{R}_{+}\}$
- $r(D) = \{[r \rightarrow 0]u | u \in D\}$

The symbolic transition relations over symbolic states can be denoted by $\rightsquigarrow$ and defined as follows ([16], [12]):

- Global Delay Transition: $\langle l, D\rangle \rightsquigarrow \langle l, D^{\uparrow} \wedge I(l)\rangle$
- Local Transition: $\langle l, D\rangle \rightsquigarrow \langle l[l'_i/l_i], r(D \wedge g) \wedge I(l[l'_i/l_i])\rangle$ if there exists a local action $l_i \xrightarrow{g,a,r} l'_i$
- Synchronization Transition: $\langle l, D\rangle \rightsquigarrow \langle l[l'_i/l_i][l'_j/l_j], (r_i \cup r_j)(D \wedge g_i \wedge g_j) \wedge I(l[l'_i/l_i][l'_j/l_j])\rangle$ if there exists a synchronization action $l_i \xrightarrow{g_i,a,r_i} l'_i | l_j \xrightarrow{g_j,\bar{a},r_j} l'_j$

It can be shown that the set of zones $\mathcal{B}(\mathcal{C})$ is closed under these two operations and the result of these operations is also a zone. A zone D is said to be *closed under entailment* or *closed*, if constraints of D cannot be strengthened without decreasing the solution set. Also a zone has a *canonical form*, meaning that for each zone $D \in \mathcal{B}(\mathcal{C})$, there is a unique zone $D' \in \mathcal{B}(\mathcal{C})$ such that D and D' have exactly the same solution set and D' is closed. The DBMs make use of these properties for the efficient implementation of state-space exploration using symbolic semantics. The constraints $D^{\uparrow}$ is handled by $up(D)$ and $r(D)$ is handled by $reset(D, r := 0)$ in DBM implementation and clearly the resulting constrains are also zones.

Theorem 2. *[16]For a timed automaton with initial state $\langle l_0, D_0\rangle$,*

- *(soundness) $\langle l_0, [u_0]\rangle \rightsquigarrow^* \langle l_f, D_f\rangle$ implies $\langle l_0, u_0\rangle \rightarrow^* \langle l_f, u_f\rangle$ for all $u_f \in D_f$*

– *(completeness)* $\langle l_0, u_0 \rangle \rightarrow^* \langle l_f, u_f \rangle$ *implies* $\langle l_0, [u_0] \rangle \rightsquigarrow^* \langle l_f, D_f \rangle$ *for some* D_f *such that* $u_f \in D_f$

The soundness implies that if a set of final states $\langle l_f, D_f \rangle$ can be reached from the initial symbolic state $\langle l_0, [u_0] \rangle$ according to $\rightsquigarrow$, all the final states should also be reachable according to the operational semantics.

The completeness implies that if a state is reachable according to the operational semantics, this can also be reached using symbolic transitions.

The number of zones in a timed automaton may be infinite, implying that the relation $\rightsquigarrow$ is infinite. This problem may be overcome by *normalizing* the zones, i.e. abstracting zones according to the maximal clock constant seen in the automata [24]. The idea behind this abstraction is that if the clock value exceeds the maximal constant for that clock in the automata, only the information that the maximal value is exceeded is important, the exact value is not important.

2.4 Zone-Normalization

The methods in this paper which are used to decrease the explored part of the search space of reachability problems are implemented on MCTA. In MCTA, no difference constraints are allowed in the guards, i.e. only atomic constraints of the form $x \sim n$ are allowed. Such automata are called *diagonal-free automata* [25]. An efficient zone normalization method for the diagonal-free automata is *k-normalization* ([26], [27]).

Definition 7 *(k-normalization) For a zone D and a clock ceiling k, the semantics of the k-normalization operation is defined as:*

$$norm_k(D) = \{u | u \simeq v, v \in D\}$$

As explained before D has a canonical form, and from this form, $norm_k(D)$ can be calculated as follows ([26], [27]):

– remove all constraints of the form $x < m$, $x \leq m$, $x - y < m$ and $x - y \leq m$ where $m > k(x)$

– replace all constraints of the form $x > m$, $x \geq m$, $x - y > m$ and $x - y \geq m$ where $m > k(x)$ with $x > k(x)$ and $x - y > k(x)$ respectively.

Since there are finite number of clocks with a clock ceiling, there are only finite number of normalized zones.

Theorem 3. *[16] For a timed automaton with initial state $\langle l_0, u_0 \rangle$, with maximal clock constants bound by clock ceiling k and not containing any guard with difference constraints in the form $x - y \sim n$, the transition relation $\rightsquigarrow_k$ which is defined as follows is sound and complete:*

$\langle l, D \rangle \rightsquigarrow_k \langle l', norm_k(D') \rangle$ *if* $\langle l, D \rangle \rightsquigarrow \langle l', D' \rangle$

- *(soundness)* $\langle l_0, [u_0] \rangle \rightsquigarrow_k^* \langle l_f, D_f \rangle$ *implies* $\langle l_0, u_0 \rangle \rightarrow^* \langle l_f, u_f \rangle$ *for all* $u_f \in D_f$ *such that* $u_f(x) \leq k(x)$ *for all* x
- *(completeness)* $\langle l_0, u_0 \rangle \rightarrow^* \langle l_f, u_f \rangle$ *with* $u_f(x) \leq k(x)$ *for all* x *implies* $\langle l_0, [u_0] \rangle \rightsquigarrow_k^* \langle l_f, D_f \rangle$ *for some* D_f *such that* $u_f \in D_f$
- *(finiteness) The transition relation* $\rightsquigarrow_k$ *is finite.*

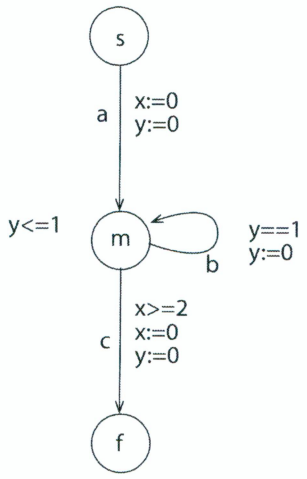

Fig. 5. An example automaton with two clocks x and y.

The effect of normalization can be demonstrated with the example in Fig. 5. The automaton in the figure has two clocks x and y, three locations s, m and f and three transitions $s \xrightarrow{a,(x:=0,y:=0)} m$, $m \xrightarrow{(y==1),b,(y:=0)} m$ and $m \xrightarrow{(x\geq 2),c,(x:=0,y:=0)} f$. Also the location m has the invariant $y \leq 1$. Without the *k-normalization*, there are infinitely many reachable zones for this automaton $((y \leq 1) \wedge (x = y))$, $((y \leq 1) \wedge (x \leq 2) \wedge (x - y = 1))$, $((y \leq 1) \wedge (x \leq 3) \wedge (x - y = 2))$ ...(Fig. 6). However, when the *k-normalization* is applied,

the number of reachable zones is finite, $((y \leq 1) \wedge (x = y))$, $((y \leq 1) \wedge (x \leq 2) \wedge (x - y = 1))$, $((y \leq 1) \wedge (x - y = 2))$ and $((y \leq 1) \wedge (x > 2) \wedge (x - y > 2))$ (Fig. 7).

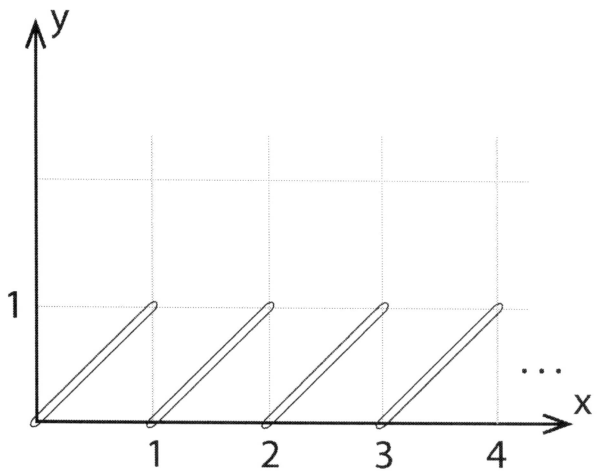

Fig. 6. The reachable zones for the automaton in Fig. 5 without *k-normalization.*

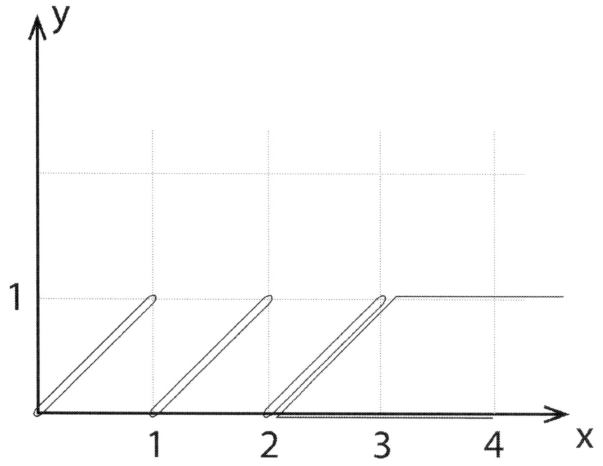

Fig. 7. The reachable zones for the automaton in Fig. 5 with *k-normalization.*

2.5 Symbolic Reachability

The methods explained in this paper are based on the reachability problem of timed automata, particularly satisfaction of location constraints. For this purpose, the state-space

of the given automata should be explored and the explored states are checked if they satisfy the location requirements. For a network of timed automata A, a distance heuristic h (a function that estimates distance to the goal), and an error property φ (a property that is used to check if the state is an error state. i.e., a negated invariant property), MCTA executes a reachability algorithm on the zone graph of A. The basic model checking algorithm of MCTA is shown in Algorithm 1 ([4]).

Algorithm 1 Basic Directed Model Checking Algorithm of MCTA

```
1:  procedure DMC(A, h, φ)
2:      open=empty priority queue
3:      closed = ∅
4:      open.insert(s_0, priority(h, s_0))
5:      while open ≠ ∅ do
6:          s=open.getMinimum()
7:          if s ⊨ φ then
8:              generateErrorTrace(s)
9:          closed = closed ∪{s}
10:         for each transition t of A that is applicable in s do
11:             if t[s] ∉ closed then
12:                 open.insert(t[s], priority(h, t[s]))
13:         return True
```

In the algorithm, the priority queue *open* contains the states whose successors have not been computed yet and the *closed* list contains the states which are already explored and whose successors are already inserted into the *open* queue. The algorithm starts with the initial state s_0 and calculates a *priority value* with the *priority function* for this state and later for each successor state according to the heuristic value h and the search algorithm chosen. Then the best state s according to the priority value is chosen from the open list and checked if it is an error state. If so, an error trace is generated for this state. If s is not an error state, it is inserted to the *closed* list, the successors and their corresponding priority values are calculated and stored into the *open* queue.

2.6 Difference Bound Matrices (DBMs)

Difference Bound Matrices (DBMs)([23], [18]) is an efficient way to represent zones. To have a standard form for clock constraints, a reference clock 0 is also added to the set of clocks which can be shown as $\mathcal{C}_0 = \mathcal{C} \cup \{0\}$. The clock constraints have the form $x - y \preceq n$ where $x, y \in \mathcal{C}_0$, $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z}$. A zone can be represented with at most $|\mathcal{C}_0|$ atomic constraints, so it can be stored with a $|\mathcal{C}_0| \times |\mathcal{C}_0|$ matrix.

Each element D_{ij} (i^{th} row and j^{th} column) of this matrix represents a bound for the difference $x_i - x_j$, thus called *Difference Bound Matrices*(DBMs). Each element stores the difference value and relation, i.e. for two clocks x_i, x_j, the constraint $x_i - x_j \preceq n$ is stored

as $D_{ij} = (n, \preceq)$. If there is no bound for a clock difference, i.e. $x_i - x_j < \infty$, then it is represented as $D_{ij} = \infty$. Since all clocks are non-negative (i.e. $\forall x_i \in \mathcal{C}_0, x_i \geq 0$), for all clocks $x_i \in \mathcal{C}_0$, $0 - x_i \leq 0$ must be added as constraints to ensure that negative clock values are not included in the zone.

Consider an automaton with 3 clocks x, y, z as an example. To represent zones of this automaton a 4×4 matrix is necessary since $|\mathcal{C}_0| = 4$ as $\mathcal{C}_0 = \{0, x, y, z\}$. For this automaton consider the zone D which is defined as $D = (x < 3) \wedge (y \leq 2) \wedge (x - y = 1) \wedge (z > 4)$. To find the DBM, the zone should be represented with the standard form mentioned before, which results $D = (x - 0 < 3) \wedge (y - 0 \leq 2) \wedge (x - y \leq 1) \wedge (y - x \leq 1) \wedge (0 - z < -4)$. Note that the constraint $(x - y = 1)$ is represented with two inequality constraints $(x - y \leq 1) \wedge (y - z \leq 1)$. Since the clocks are always equal to themselves, i.e. $(x - x = 0)$, the diagonal entries of the matrix should be $x - x \leq 0$. Also the constraints which do not have a bound are represented with ∞ to show that they are unbounded. So for this zone the resulting DBM is as follows:

$$DBM(D) = \begin{bmatrix} (0, \leq) & (0, \leq) & (0, \leq) & (-4, <) \\ (3, <) & (0, \leq) & (1, \leq) & \infty \\ (2, \leq) & (1, \leq) & (0, \leq) & \infty \\ \infty & \infty & \infty & (0, \leq) \end{bmatrix}$$

Usually many zones with different conjunction of clock constraints may represent the same solution set, but the *canonical form* of a DBM is a unique constraint for each family of zones which cannot be strengthened without loss of solution. For this purpose the tightest constraint for each clock difference must be found. This requires a quite expensive operation and often done with Floyd-Warshall algorithm ([28]) (cubic in the number of clocks, i.e. $O(|\mathcal{C}_0|^3)$). To decrease this cost, most of the operations on DBMs are ensured to conserve the canonical form.

3 Local-Time Semantics

To prevent exploration of unnecessary interleavings caused by execution of independent transitions, partial order reduction methods are used. Independent transitions are executed by different processes so their order of execution is not important and do not affect each other. For two transitions t_i and t_j in processes A_i and A_j, if the transitions are independent, execution of one does not affect the other, i.e. execution of t_i has no effect on execution of t_j, or vice versa. Executing first t_i after t_j and first t_j after t_i must lead to the same symbolic state; so after executing one of these transitions, execution of the other one is delayed to prevent exploration of already explored state-space. The transitions t_i and t_j are independent if $(t_i, t_j)((l, D)) = (t_j, t_i)((l, D))$ for any symbolic state (l, D). Local

transitions on different processes are independent if time is not involved in the system, i.e. finite-state systems. However, for networks of timed automata, time is present in the system and the transitions t_i and t_j are not independent if the processes they belong have clock variables. Consider the example ([12]) in Fig. 8.

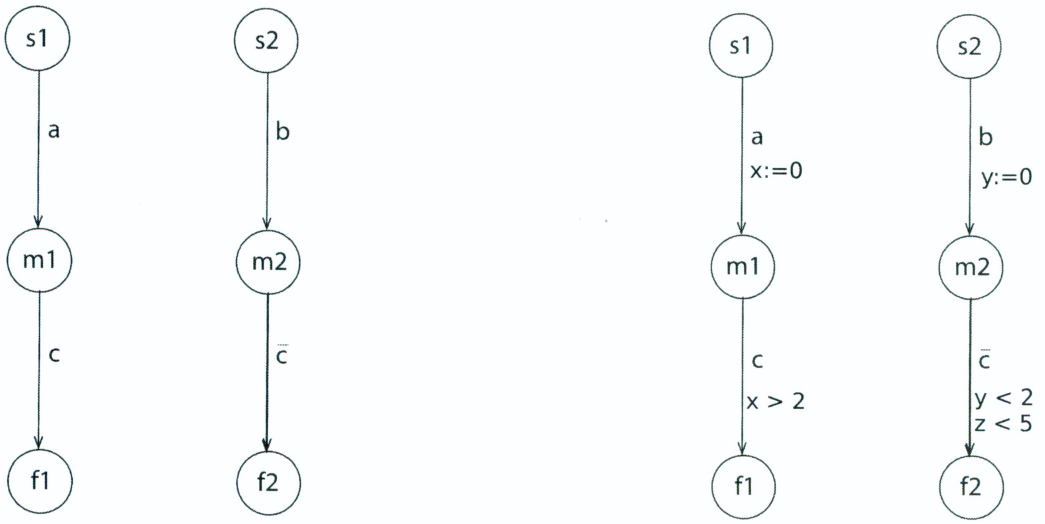

Fig. 8. Finite-state automata (left) and timed automata (right).

On the left, there is a finite-state automata with two processes. The initial state for the automata is (s_1, s_2). From the initial state, two local transitions a and b can be executed by the two processes from their local initial locations. From the state (m_1, m_2), a synchronization transition c can be executed which leads to the state (f_1, f_2). From the state (s_1, s_2) to (m_1, m_2), there are two possible transition sequences; $(s_1, s_2) \xrightarrow{a} (m_1, s_2) \xrightarrow{b} (m_1, m_2)$ and $(s_1, s_2) \xrightarrow{b} (s_1, m_2) \xrightarrow{a} (m_1, m_2)$. Since a and b are two independent transitions, one of the two interleavings above which lead to the same state (m_1, m_2) can be pruned if a partial order reduction method is used.

On the right side of the Fig. 8, there is a network of timed automata with two processes. The state-space of the timed automata is shown with an illustration on the left side of the Fig. 9. The system is initialized with the initial state (s_1, s_2) and all the clocks reset to zero. The left automaton can wait at the initial location arbitrarily and then execute the transition $(s_1, s_2) \xrightarrow{a,x:=0} (m_1, s_2)$ afterwards while resetting the clock x while y and z can take any value greater or equal to zero depending on the time the system waited at the initial state. The left automaton has to wait at location m_1 for at least 2 time units until the synchronization transition can be executed. Similarly from the initial state the right automaton can execute the transition $(s_1, s_2) \xrightarrow{b,y:=0} (s_1, m_2)$ after after waiting for some

time while resetting clock y. The right automaton can stay in location m_2 at most 2 time units after executing transition b or 5 time units after start of the system, whichever is lower. There are two possible transition sequences $(s_1, s_2) \xrightarrow{a,x:=0} (m_1, s_2) \xrightarrow{b,y:=0} (m_1, m_2)$ and $(s_1, s_2) \xrightarrow{b,y:=0} (s_1, m_2) \xrightarrow{a,x:=0} (m_1, m_2)$ leading to the state (m_1, m_2) as it was for the finite-state automata case. However, for the first interleaving resulting from executing first a then b (the left branch of the state-space shown in Fig. 8), since x is reset before y, the clocks x and y has the relation $x \geq y$ at the state (m_1, m_2); while for the second interleaving resulting from first executing b then a (the right branch of the state-space shown in Fig. 8), the clocks satisfy $x \leq y$. From the first interleaving, since $x > y$, the synchronization transition c can be executed leading to the desired goal state (f_1, f_2); however, from the second interleaving, since $x < y$, the guards $x > 2$ and $y < 2$ cannot be satisfied simultaneously and the synchronization transition c is not enabled. So, in the presence of time, executing these two transitions x and y in different orders produces different results, thus a and b are dependent because of the implicit synchronization of the clocks. So it is not sufficient to explore only one of the interleavings unlike finite-state automata.

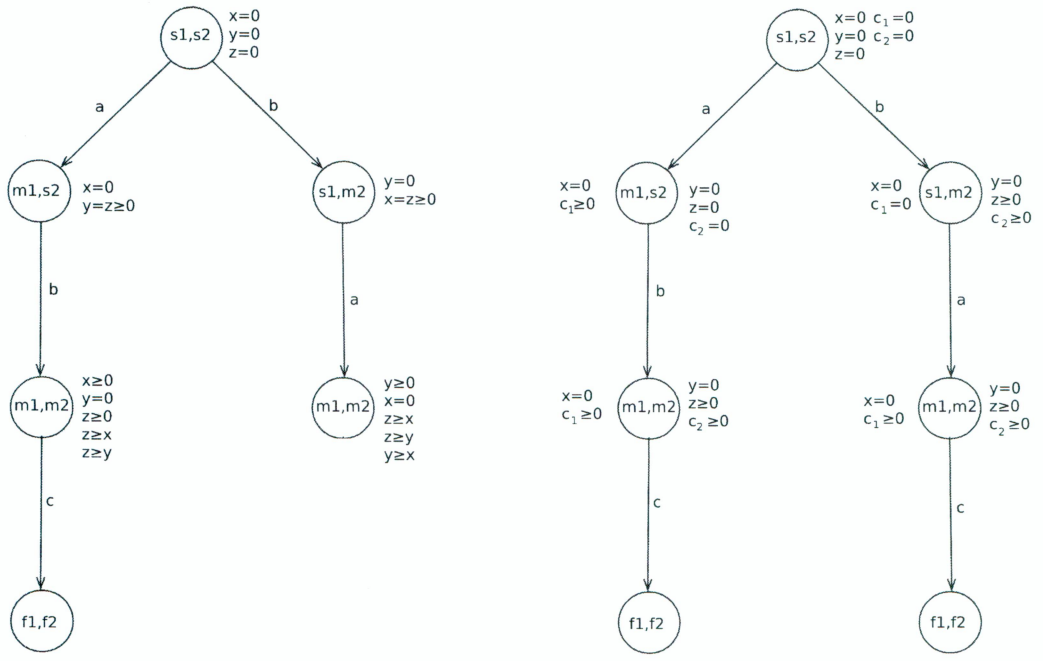

Fig. 9. State space of the timed automata in Fig. 8 with normal semantics (left) and local time semantics (right).

To solve the problem explained above for timed automata, local-time semantics is proposed([12]). The idea of local-time semantics is to allow clocks advance independently from

other clocks, thus to remove implicit synchronization of clocks. The execution order of transitions will be indefinite for the interleavings, so the transitions which are independent in the absence of clocks are also independent in the presence of clocks. After this *desynchronization*, standard partial order reduction methods can be used for timed automata.

When local-time semantics is applied to the example in Fig. 8, the resulting clock values for both of the interleavings need to satisfy only $y \leq z$. With the standard semantics, the clocks have the constraint $x = y = z$, but there is no condition between x and y or z when local-time semantics is used since they are in different automata.

Removing the clock synchronization between automata enables application of partial order reduction techniques, but the timing information required for synchronization transitions is also lost. In the previous example the edge of synchronization transition which is on the right automaton has the guard condition $z < 5$, and since z is never reset, the synchronization transition should be executed within 5 time units from the system initialization. This is also an implicit constraint on the left automaton; however, with the desynchronization of clocks, this constraint is lost. So to be able to execute the synchronization transition properly , the local time scales of the two automata should be *resynchronized.* For this purpose, a *local reference clock* is added to each automaton. The local reference clocks measure the local time advance of the automaton. So when a synchronization transition is to be executed, the local reference clocks must be equal. So the constraint $c_1 = c_2$ must also be added as a constraint to synchronization transitions.

In the example above, the local reference clocks c_1 and c_2 should be added to the left and right automata respectively. The state-space of the timed automata is shown on the right side of the Fig. 9. After executing the local transition a, the system reaches the state (m_1, s_2). Since the system can wait at the initial state arbitrarily, the local reference clock c_1 must satisfy $c_1 \geq 0$. Since the clocks on different automata advance independently, y, z and c_2 remains zero. Executing b from this state leads to the state (m_1, m_2) while clocks satisfy $x = 0, y = 0, z \geq 0, c_1 \geq 0, c_2 \geq 0$. If the transitions a and b are executed in the reverse order, the state reached and the clock values are identical as it can be seen from the right branch of the state-space in Fig. 9. Applying the required constraints $(x > 2, y < 2, z < 5, c_1 = c_2)$ for synchronizing transition to both of the reached states leads to the goal state (f_1, f_2) showing that the transitions a and b can be interpreted as independent when the local time semantics is used.

The idea of local-time semantics explained on an example above can be summarised as follows. The partial order reduction methods cannot be applied on timed automata efficiently due to the dependence of the transitions caused by the implicit synchronization of the clocks. To overcome this problem, the implicit synchronization of the clocks in different automata is removed and the clocks in each automaton are allowed to increase independent of the clocks of other automata. Executing local transitions only changes local clocks so removing synchronization of clocks has no effect on local transitions. However, to

execute synchronizing transitions, the local-time scales of the automata participating the synchronizing transitions must be synchronized. To accomplish this resynchronization, a local reference clock which measures the time advancement since the initialization of the system is added to each automaton. A constraint indicating that the local reference clocks of the participating automata must be equal is added to the synchronizing transitions to ensure the resynchronization of the local-time scales of the automata. Thus the local clocks of different automata can advance independently and they are only synchronized for synchronizing transitions.

For a network of automata $A_1|...|A_n$, each automaton A_i has the set of clocks C_i which also include the local reference clock c_i. The clock assignment $u +_i d$ shows that the clocks $x \in C_i$ has the value $u(x)+d$ and the clocks $x \in C \setminus C_i$ has the value $u(x)$. In the definitions below for local-time semantics and the symbolic local-time semantics, the initial state is shown as (l^0, u^0), the set of clocks include reference clocks which are initialized as zero.

The network of timed automata can change its state with the following three types of transitions [12]:

- Local Delay Transition: $(l, u) \mapsto (l, u +_i d)$ if $I(l_i)(u +_i d)$
- Local Discrete Transition: $(l, u) \mapsto (l[l'_i/l_i], u')$ if there exists a local transition $l_i \xrightarrow{g,a,r} l'_i$ such that $u \models g$ and $u' = [r \to 0]u$
- Synchronizing Transition: $(l, u) \mapsto (l[l'_i/l_i][l'_j/l_j], u')$ if there exists a synchronizing transition $l_i \xrightarrow{g_i,a,r_i} l'_i | l_j \xrightarrow{g_j,\bar{a},r_j} l'_j$ such that $u \models g_i$, $u \models g_j$ and $u' = [r_i \to 0][r_j \to 0]u$ and $u(c_i) = u(c_j)$

The local delay transition indicate that the local clocks of an automaton can increase as long as the clock invariant is satisfied. Local discrete transition is the same as local transition for operational semantics without local-time semantics (i.e. global-time semantics). The synchronizing transition has an additional condition that it can be executed only if the reference clocks of the two participating automata is the same, i.e. $u(c_i) = u(c_j)$

Definition 8 *[12] A local time state (l, u) with reference clocks $c_1, ...c_n$ is synchronized if $u(c_1) = ... = u(c_n)$.*

The set of states that can be reached with global time semantics (when all clocks advance with the same rate) can also be reached with local time semantics when the reference clocks are synchronized, so the local-time semantics can be used instead of standard global time semantics.

Theorem 4. *[12] For all networks,* $(l_0, u_0)(\rightarrow)^*(l, u)$ *iff for all synchronized local time states* (l, u), $(l_0, u_0)(\mapsto)^*(l, u)$.

3.1 Symbolic Local-Time Semantics

To create a reachability algorithm working with partial order reduction methods, symbolic semantics should also be modified to develop symbolic local-time semantics. For this purpose, the local-time states should be represented by constraints. To prove that the constraints that symbolic local-time states uses are the same as the clock constraints, the constraints of symbolic local-time states are denoted with $\widehat{D}, \widehat{D}'...$ for now. For all $d \in \mathbb{R}$, the constraint $\widehat{D}^{\uparrow i}$ is the constraint which satisfy $u +_i d \models \widehat{D}^{\uparrow i}$ iff $u \models \widehat{D}$. The *local-time predicate transformer* denoted by $\widehat{sp_t}(\delta_i)$, which only allows the local clocks C_i to advance is defined as follows:

- $\widehat{sp_t}(\delta_i)(l, \widehat{D}) \stackrel{def}{=} (l, \widehat{D}^{\uparrow i} \wedge I(l))$

The local-time predicate transformer action $\widehat{sp_t}(\alpha)$ for a local or synchronizing transition α is defined as:

- For a local transition α defined as $l_i \xrightarrow{g,a,r} l_i'$;

  $\widehat{sp_t}(\alpha) \stackrel{def}{=} sp_t(\alpha); \widehat{sp_t}(\delta_i)$
- For a synchronizing transition α defined as $l_i \xrightarrow{g_i,a,r_i} l_i' | l_j \xrightarrow{g_j,\bar{a},r_j} l_j'$;

  $\widehat{sp_t}(\alpha) \stackrel{def}{=} \{c_i = c_j\}; sp_t(\alpha); \widehat{sp_t}(\delta_i); \widehat{sp_t}(\delta_j)$

The clock constraint $c_i = c_j$ necessary for synchronizing transition is added as a predicate transformer which is defined as $\{c_i = c_j\}(l, \widehat{D}) \stackrel{def}{=} (l, \widehat{D} \wedge c_i = c_j)$. The initial symbolic local-time state is denoted by $(l^0, \widehat{D}^0)$ where $\widehat{D}^0 = \widehat{sp_t}(\delta_1); ...\widehat{sp_t}(\delta_n)(\{u^0\})$. For a transition α, $(l, \widehat{D}) \mapsto (l', \widehat{D}')$ if $(l', \widehat{D}') = \widehat{sp_t}(\alpha)(l, \widehat{D})$.

Theorem 5. *[12] For all networks, a synchronized state* (l, u), $(l^0, u^0) \rightarrow^* (l, u)$ *if and only if* $(l^0, \widehat{D}^0)(\mapsto)^*(l, \widehat{D})$ *for a symbolic local time state* $(l, \widehat{D})$ *such that* $u \models \widehat{D}$.

This theorem indicate that the symbolic local-time semantics can be used instead of the global time semantics in terms of reachability analysis. But the requirement that the local time state must be synchronized is time and space consuming to compute. However, for a class of networks containing *no local time-stop*, this condition is weaker.

Definition 9 *[12] A network is local time-stop free if for all* (l, u), $(l^0, u^0)(\mapsto)^*(l, u)$ *implies* $(l, u)(\mapsto)^*(l', u')$ *for some synchronized state* (l', u').

A network of timed automata can be ensured to be local time-stop free by syntactical restrictions, like adding an edge with a guard weaker than the invariant to each control node. This ensures that the synchronization of clocks does not fail because of the invariant of a node which becomes false.

Theorem 6. *[12] For a local time-stop free network A and a local control node* l_k *in A,* $(l^0, D^0)(\Rightarrow)^*(l, D)$ *for some* (l, D) *such that* $l[k] = l_k$ *if and only if* $(l^0, \widehat{D}^0)(\mapsto)^*(l', \widehat{D}')$ *for some* $(l', \widehat{D}')$ *such that* $l'[k] = l_k$.

This theorem shows that the reachability analysis for local time-stop free network with symbolic local-time semantics can be accomplished without the synchronization of symbolic states.

Theorem 7. *[12] For a network of timed automata A and two transitions* α_1 *and* α_2, *if the sets of processes of A involved in* α_1 *and* α_2 *are disjoint,* $\widehat{sp}(\alpha_1)$ *and* $\widehat{sp}(\alpha_2)$ *are independent.*

This theorem shows that the timed predicate transformers defined for local-time semantics has the commutativity property. The global time semantics lacks this property which prevents application of partial order methods.

To ensure that the reachability analysis with symbolic local time semantics terminates, the number of symbolic states which is in general infinite must be finite. It can be shown that by using regions as in the case of standard semantics, there exists a finite partitioning of the state space. For this purpose, the standard region equivalence must be extended to synchronized states.

Definition 10 *[12] Two synchronized local time states with the same control vector l,* (l, u) *and* (l, u') *are synchronized-equivalent if* $([C_r \mapsto 0]u) \sim ([C_r \mapsto 0]u')$ *where* $\sim$ *is the standard region equivalence for timed automata and* C_r *is the set of reference clocks.*

The condition $([C_r \mapsto 0]u) \sim ([C_r \mapsto 0]u')$ implies that only the clocks which are not reference clocks in the local time states are region equivalent and such equivalence classes are called *synchronized regions.* To extend this definitions to the local time states which are not synchronized, *catch-up transition* is defined.

Definition 11 *[12] A local delay transition $(l, u) \rightarrow (l, u')$ of a network is a catch-up transition if $max(u(C_r)) \leq max(u'(C_r))$.*

A catch-up transition represents executing a delay transition in one of the automata whose reference clock is smaller, so it makes the system closer to synchronized state. So two non-synchronized states can be equivalent if the automata with lower reference clock advances in time to catch up with the one with higher clock value so that they can become equivalent synchronized states.

Definition 12 *[12] For a local time state (l, u) of a network of timed automata, $R((l, u))$ denotes the set of synchronized regions reachable from (l, u) only by discrete transitions or catch-up transitions.*

Definition 13 *[12] Two local time states (l, u) and (l', u') are catch-up equivalent denoted by $(l, u) \sim_c (l', u')$ if $R((l, u)) = R((l', u'))$. The equivalence class of local time states with respect to $\sim_c$ is denoted by $|(l, u)|_{\sim_c}$.*

This definition states that two catch-up equivalent local time states can reach the same set of states that all the clocks in the network are synchronized. The number of synchronized regions is finite as it is explained for regions for timed automata, so the number of catch-up classes is also finite. Since the processes generally never have a stopping time, i.e. they run indefinitely, the reference clocks c_i have no upper bound. Having unbounded clocks and finite number of regions imply that the region graph must be periodic after some initial steps.

Theorem 8. *[12] For any network of timed automata, the number of catch-up equivalence classes $|(l, u)|_{\sim_c}$ for any vector of control nodes l is bounded by a function of the number of regions in the standard region graph construction for timed automata.*

The number of nodes for a network of timed automata is finite, so the number of vectors of control nodes is also finite. Thus the symbolic local-time semantics presented above satisfy the finiteness property.

4 Partial Order Reduction

The main problem affecting the performance of verification techniques is the excessive size of the state-space. The growth of the state-space can be exponential and the main reason

for this increase is exploration of all possible interleavings. It can be shown that exploring all interleavings is not necessary to verify a given property. Partial order reduction methods explore only a reduced part of the state-space that is sufficient to check the given property. The partial order reduction methods prevents exploration of interleavings that can be reached by executing independent transitions. The order of execution of independent transitions is not important. An example showing that two independent transitions reaching the same state was explained in Fig. 8.

4.1 Stubborn Sets

Some partial order reduction methods computes a set of transitions called *"persistent sets"* ([29], [6]) for each state. Basically, a persistent set is a subset T of the all enabled transitions in a state s. All the transitions in T are independent with all the transitions which are not in T that are enabled in s or in a state reachable from s through transitions not in T. Briefly, taking any transition from s outside T does not affect T. It is sufficient to execute only the transitions in persistent sets to solve the model checking problem.

Stubborn set technique is one of the methods for computing persistent sets. Some definitions are required for explanation of stubborn sets.

Definition 14 *[6] Two transitions t_1 $t_2 \in T$, the set of all transitions, are independent in state $s \in S$ if:*

1. the set of active processes of t_1 and t_2 (i.e. the processes that t_1 and t_2 belong) are disjoint and

2. $\forall op_1 \in used(t_1)$ (i.e. all of the the clock and integer operations used by the transitions t_1) and $\forall op_2 \in used(t_2)$, if op_1 and op_2 are two operations on the same object, then op_1 and op_2 are independent in s.

This definition of independence simply implies that two transitions are dependent if one of them writes to and the other one reads from the same variable or both of them write to the same variable. They are independent if they don't operate on the same variable, or one transition doesn't write to a variable which is read or written by the other transition or they only read the common variable.

Definition 15 *[6] Let op_1 and op_2 be two operations on the same object. op_1 and op_2 can-be-dependent if there exists a state s such that op_1 and op_2 are dependent in s.*

Definition 16 *[6] Two transitions t_1 and t_2 do-not-accord with each other if there exists a state $s \in S$ such that t_1 and t_2 are enabled (i.e. can be executed at the current state) in s and are dependent in s.*

Definition 17 *[6] Two transitions t_1 and t_2 are parallel iff active processes of t_1 and active processes of t_2 are disjoint (i.e. $active(t_1) \bigcap active(t_2) = \emptyset$).*

Definition 18 *[6] Two transitions t_1 and t_2 are conflicting iff there exists a process P_i that is active for both t_1 and t_2 and from its local state, P_i can choose between t_1 and t_2 ($pre(t_1) \bigcap pre(t_2) \bigcap P_i$ where $pre(t_i)$ is the local state that t_i can be executed) (i.e. $pre(t_1) \bigcap pre(t_2) \neq \emptyset$).*

Definition 19 *[6] A necessary enabling set for a disabled transition t in a state s, denoted NES(t, s), is a set of transitions such that, for all states s' such that t is enabled in s', for all sequences w of transitions from s to s', w contains at least one transition of NES(t, s).*

Shortly, a disabled transition t in s cannot become enabled in some successor s' of s unless a transition from the necessary enabling set $NES(t, s)$ is executed.

Definition 20 *[6] A set T_s of transitions is a stubborn set in a state s if T_s contains at least one enabled transition, and if for all transitions $t \in T_s$, the two following conditions hold:*
1. if t is disabled in s, then all transitions in one necessary enabling set NES (t, s) for t in s are also in T_s;
2. if t is enabled in s, then all transitions t_0 that do-not-accord with t are also in T_s.

Since T_s contains at least one enabled transition by definition, thus set of all enabled states in T_s, call T, is non-empty. So the following theorem can be proposed.

Theorem 9. *[6] Let T be the set of all transitions in a stubborn set T_s in state s that are enabled in s. Then, T is a persistent set in s.*

The theorem is proven in [6], so stubborn sets can be used to calculate persistent sets and thus can be used to prune transitions without losing essential information.

Algorithm 2 Stubborn Sets

1: **procedure** SELECT TRANSITIONS TO INITIALIZE THE STUBBORN SET T_s
2: **for** all unreached goal locations l **do**
3: add all incoming transitions to l to T_s
4: **procedure** CALCULATE STUBBORN SET
5: **for** all transitions t in T_s **do**
6: **if** t is disabled in s **then** either
7: - choose a Process $P_j \in active(t)$ such that $s_j \neq (pre(t) \bigcap P_j)$; then add all transitions t' such that $(pre(t) \bigcap P_j) \in post(t')$ to T_s
8: - choose a condition c_j in the guard G of t that evaluates to false in s; then for all operations op used by t to evaluate c_j, add all transitions t' such that $\exists op' \in used(t')$ and op and op' can-be-dependent to T_s
9: **if** t is enabled in s **then**
10: add all transitions t' such that t and t' are in conflict
11: add all transitions t' such that t and t' are in parallel and $\exists op \in used(t)$, $\exists op' \in used(t')$: op and op' do-not-accord
12: **if** There are no more transitions that need to be added to T_s **then**
13: **return** All transitions in T_s that are enabled in s

4.2 Algorithm

From the definition of stubborn sets, an efficient algorithm which computes stubborn sets T_s is obtained in [6]. The implementation explained in this paper is added on MCTA, so some modifications on the algorithm are required for efficient and correct computation of stubborn sets.

The modified algorithm presented in Algorithm 2 can be divided into two parts. The second part (lines 4-13) is the same as the original stubborn sets algorithm from [6]. The first part (lines 1-3) chooses the transition with which the second part starts computation of stubborn sets. The Algorithm 2 is executed at every step. The input to this algorithm is the current state of the system s and set of all enabled transitions t in s. The output of the algorithm is the enabled transitions in the stubborn set T_s

To understand the reasoning behind the modifications, some background information about MCTA is required. In MCTA the states are stored according to the current location of each automata. The states are stored in two lists, one is the *open list* whose successors are to be created; the other one is the *closed list* whose successors are already created. While searching, MCTA takes a state from open list, creates the successors, stores the successors to the open list and moves the parent state to the closed list. The program searches for a solution and exits either with success when a solution is found or with failure when there is no more states in the open list to be expanded.

The first part of the algorithm is added to prevent premature termination with failure. Consider the example shown in Fig. 10. The initial state is (s_1, s_2) and it is the only state in the open list. The goal state is (f_1, f_2) If the transition tr_1 is taken as input to the stubborn sets algorithm, only tr_1 itself will be returned as the output of the second part of the algorithm. So the only successor which is put into the open list is (f_1, s_2). State (s_1, s_2)

is moved to the closed list. At the second step, if tr_2 is taken as input to the stubborn sets algorithm, again only t_2 will be the output (t_3 will be pruned again). Thus (a_1, b_0) will be the only successor and it should be removed to closed list as it is already expanded. This leaves no state in the open list and the algorithm fails to find a solution although there is obviously a solution to the goal state (applying first tr_1 then tr_3 or first tr_3 then tr_1). To prevent such problems, the first part is added to the stubborn sets algorithm.

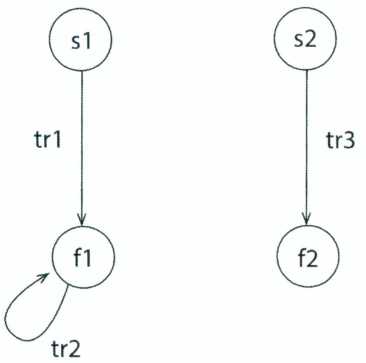

Fig. 10. An example with a self looping transition.

The first part of the Algorithm 2 (lines 1-3) initializes T_s with the *disjunctive action landmarks* [30], [31]. To reach the goal state, at least a transition from disjunctive action landmarks must be executed. So, for every unreached goal location, all of the incoming transitions to that location is added to T_s. This ensures that the transitions leading towards the goal state are not completely pruned by the stubborn sets algorithm. The stubborn sets algorithm in [6] requires an enabled transition to start computation and there is no condition on how to choose this enabled transition. The additional part of the algorithm (lines 1-3) provides transitions which prevents possible problems that might be caused by the implementation of MCTA; however, all of the transitions added by the first part might be disabled. However, every path from the initial state to goal state must include at least one transition from disjunctive action landmarks, so completing the sleep sets algorithm in the second part leads to at least one enabled transition, if the goal state is reachable. If there are no enabled states after completion of the Algorithm 2, the goal state cannot be reached from its current state with its current enabled transitions.

The second part of the algorithm (lines 4-13) computes the stubborn sets. Initially, the only transition in T_s which stores the transitions added by the algorithm is the enabled transition calculated by the first part. Each transition t in T_s is examined and some other transitions may need to be added according to the rules in the algorithm depending on whether t is disabled or enabled in the current state s.

If the transition t is disabled in a state s (line 6), there may be two reasons; either the location constraint is not satisfied (i.e. the active process P_j of t cannot execute t from its current local state $s(j)$), or there is a constraint c_j in the guard G of t that is not satisfied in s. If the transition t is disabled because of the first case (line 7), i.e. the transition cannot be executed from the current state s, the set of all transitions t' such that $(pre(t) \bigcap P_j) \in post(t')$ is a necessary enabling set $NES(t, s)$ for t in s and should be added to T_s since execution of t' is required to enable t. If the transition t is disabled because of the second case (line 8), i.e. a condition c_j in G evaluates to false in s, the set of all transitions t' that use an operation op' which can-be-dependent with an operation op that evaluates c_j is a necessary enabling set $NES(t, s)$ for t in s and should be added to T_s since only op' can change the output of op thus the truth value of c_j.

If the transition t is enabled in s (line 9), the set of all transitions t' such that t and t' do-not-accord should be added to T_s.

If t and t' are not parallel, there should be a process P_j which is active for both t and t'. If t and t' are in conflict (line 10), t' is added to T_s; if not, they cannot be enabled simultaneously as process P_j cannot be in two different local state at the same time and this contradicts to t and t' do-not-accord assumption.

If t and t' are parallel (line 11), since they do-not-accord,there exists a state s' where t and t' are enabled and dependent. Since t and t' are dependent in s', according to the definition of independence, this implies that $\exists op \in used(t), \exists op' \in used(t')$: op and op' are dependent in s'. Also t and t' are enabled implying op and op' are defined in s' so op and op' do-not-accord. As a result, t' should be added to T_s.

The second part should be looped for each transition $t \in T_s$ until no more transition t' need not to be added to T_s (line 12). Finally, picking only enabled transitions from T_s gives the persistent sets which is required to decrease the search space to find a solution to the reachability problem (line 13).

The algorithm explained above and the local time semantics can be illustrated with the example in Fig. 11 which contain three processes denoted by A_1, A_2 and A_3. Let the initial state of this automata be $(s1, s2, s3)$ and the goal state be $(f1, f3)$, i.e. the final location of A_2 is not important. Each process has one local clock each denoted by x, y and z respectively. The transitions are shown with a, b, c, d, e, f and g. The location $m1$ has the clock invariant $x > 5$, $m2$ has the invariant $y < 3$ and $m3$ has the invariant $z > 3$. Also the transition c has the guard $k == 0$ where k is an integer variable which is initialized as 0 and the transition g has the guard $z < 5$. The transition a sets k to 1 and d sets k to 0. Let the reference clocks of each automata be denoted by c_1, c_2 and c_3.

When the system is initialized, the system is at the initial state $(s1, s2, s3)$ and the enabled transitions are a, d and f. The algorithm starts with finding the unsatisfied goal

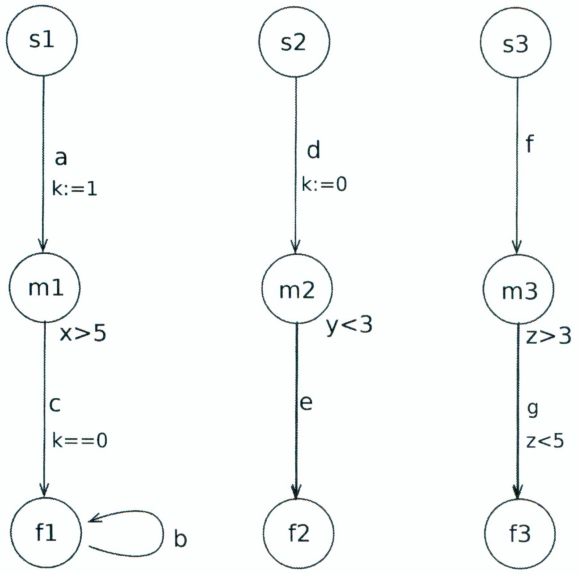

Fig. 11. A network of timed automata with three processes.

locations. Since none of the processes reached its local goal location, all of the incoming transitions to the goal locations $f1$ and $f2$ is added to T_s. So the first part of the algorithm adds the transitions b, c and g to T_s. The loop between lines 5- 13 starts with b, since it is disabled, lines 6- 8 are executed for b. Since the location constraint of b is not satisfied (line 7) c is added to T_s, and since it is already in T_s, nothing changes. Also b has no unsatisfied guard condition, so the loop continues with the next transition which is c. The location constraint of c is not satisfied (line 7) so a is added to T_s and since the guard of c only has the integer constraint $k == 0$ which evaluates to true (line 8), no other transitions are added to T_s. Next g is examined and only f is added to T_s because of the failing location constraint (line 7). Next a is checked. Since a is enabled, conditions in lines 9-11 are checked. Since no transitions satisfy any of the conditions, no new transitions are added. Lastly, f which is enabled is checked and similarly no new transitions are added. When the loop ends, T_s include the transitions a, b, c, f and g and the enabled transitions a and f are returned by the algorithm and d is pruned.

Assume that the search algorithm of the system chooses to execute a, so in the next step, the system is in the state $(m1, s2, s3)$, k is set to 1 and the enabled transitions are d and f. The first part of the algorithm starts testing the unsatisfied goal locations and T_s is initialized with b, c and g again. For b, only transition c is added to the system as in the previous step. Next, c is examined. c is disabled since the integer constraint $k == 0$ is not satisfied (line 8). So the transitions which operate on k and can be dependent with c are added to T_s. The condition reads k so the transitions which write to the variable k (a and d) are added to T_s. Next, g is checked and f is added as in the previous step. Next,

a is checked and since none of the conditions in lines 7-8 satisfied, no new transitions are added. Similar to the previous step, no new transitions are added because of d and f, so the loop ends with $T_s = \{a, b, c, d, f, g\}$ and the enabled transitions d and f are returned with no pruning for this step.

Assume that d is executed at the next step and the algorithm starts in state $(m1, m2, s3)$ and $k = 0$ with the enabled transitions c, e and f. Again the first part initializes T_s with b, c and g. At this step, because of b, c is added to T_s. Since c is now enabled conditions in lines 10-11 are checked and nothing is added. Similar to the previous steps, f is added because of g and after the termination of the loop, T_s includes b, c, f and g. The enabled transitions c and f are returned and e is pruned.

Assume that the system chooses to execute c and the system reaches to the state $(f1, m2, s3)$. For the first automaton A_1 now the goal condition is satisfied so the first part only adds g to T_s. f is added to T_s because of g and at T_s includes f and g after termination of the loop and the enabled transition f is returned and e is pruned.

At the next step, the only enabled transition f is executed by the system leading to the state $(f1, m2, m3)$. In the next run of the algorithm, T_s is again initialized with g and it is enabled, requiring no more addition to T_s, so g is returned and e is pruned again.

After execution of g, the goal condition is satisfied, so the program exits with success. Note that without local time semantics and partial order reduction, because of the clock invariants at locations $m1$ and $m3$ and the clock constraint of g, g must be executed before c; however, since the implicit synchronization between the clocks is removed, c is executed before g. Also note that e must be executed before g because of the invariants if global time semantics is used. However, when local time semantics and partial order reduction is used, e is pruned by the algorithm and the system reaches the goal state without executing e since it is independent and has no effect on the found trace. Since the clock synchronization is not required for each state, the program can terminate with unsynchronized clocks and without executing e. As a result, the trace found by partial order reduction is 1 step shorter than the trace which can be found with global time semantics.

5 Experimental Results and Conclusion

5.1 MCTA

The local-time semantics and stubborn sets explained in this paper are implemented on MCTA [4]. MCTA is a tool for model checking large systems of concurrent timed automata. MCTA is optimized for falsification, i.e. for efficient detection of violation of safety properties, and creating an error trace; however, it can also be used for verification. MCTA applies directed model checking [32], which applies a distance heuristic and a special search

algorithm to guide the search towards target states. Distance heuristic computes an estimated length of the shortest path for each state, from the state to a target state. The search algorithm decides which transition should be executed and which state should be expanded at each step. If the heuristic is admissible, distance heuristic is guaranteed to never overestimate the actual distance, so the optimal path (i.e. the shortest path) is found.

The input models required for MCTA is in the form of the UPPAAL input language ([2]); however, some features like urgent channels arrays, etc. are not supported. The parser module of MCTA uses UPPAAL's parser library, after parsing, MCTA uses its own representations to store the input system. MCTA also uses UPPAAL's difference bound matrices library to represent the zones. The search module of MCTA which uses these features is explained by Algorithm 1.

In the experiments presented below, the benchmarks provided by MCTA are used. As heuristic function *hu* heuristics [33] and *pdb* heuristics [34]. *hu* heuristics is based on the monotonicity abstraction. The abstraction is based on the simplifying assumption that *every state variable, once it obtained a value, keeps that value forever.* The value of an element becomes a subset of its domain instead of a single element. When a transition is executed, if a new value is obtained for a variable, it is added to the subset, and this is done for every variable. The subset grows monotonically. *pdb* heuristics are a family of abstraction-based heuristics. A subset of the components of the system called pattern defines the *pdb* heuristics. The entire reachable state-space is created for the abstract system and every abstract state is stored to a look-up table with its abstract error distance. While solving the model checking problem, the error distances of the abstract states are used for their corresponding states. The main problem of *pdb* heuristics is pattern selection; however, MCTA can automatically detect efficient abstractions ([35]).

The search methods used in the experiments are greedy search ([36]) and A* search ([37]) algorithms. The greedy algorithm makes the locally optimum choice at each stage according to the heuristic value and generally fails to find the optimum solution. The A* search finds the path which has the lowest expected cost from the initial state to the given goal state. It uses a distance-plus-heuristic function (distance covered until the current state plus the heuristic value). At each step the algorithm chooses the lowest distance-plus-heuristic value. If the heuristic function is admissible, A* can find the optimal solution.

5.2 Experimental Results

The implementation of local time semantics and stubborn sets as a partial order reduction method is tested with several benchmarks. The tests are executed with normal semantics (called global time semantics below), with local time semantics, and with local time semantics and stubborn sets as partial order reduction method. For each of these cases, the number of explored states (denoted by exp. states), the number of generated states (gen.

states), the trace length and time elapsed are shown. Additionally, for the tests run with local time semantics and partial order reduction, the number of successor states pruned by the stubborn sets is also shown. Explored states are the states whose successors are generated and ready to be explored. Note that when partial order reduction is used, the sum of the pruned and explored states is not the same as the number of explored states of local time semantics without partial order reduction. The reason for this is the fact that when a successor state is pruned, the possible future successors of the pruned state is also implicitly pruned.

The first two set of benchmarks tested are denoted by "M" and "N" [38]. The benchmarks correspond to a mutual exclusion problem. In these problems, two automata "A_1" and "A_2" have states called "unsafe" and there is a "Controller" automaton which is designed to ensure the mutual exclusion of both unsafe states. The system is disturbed by the environment. There are not many automata in these systems but there are significant number of clocks and variables.

	global time semantics				**local time semantics**				**local time semantics and POR**				
	exp. states	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**succ. pruned**
M1	12285	37007	2787	0.07	729	2270	427	0.01	733	2285	436	0.03	28
M2	50686	167778	13709	0.25	3367	11492	1581	0.04	923	3554	751	0.02	357
M3	72554	223124	11524	0.38	2669	9440	1494	0.02	508	2161	501	0.02	164
M4	235480	782264	52634	1.27	3458	17312	3045	0.05	1046	4900	1017	0.04	668
N1	15747	48740	3576	0.17	1379	3959	567	0.02	1655	4693	611	0.03	53
N2	102351	365986	15894	1.16	4977	15633	1915	0.06	3752	11611	1543	0.07	1204
N3	107334	396592	20086	1.37	5001	15637	2055	0.05	4956	14134	1550	0.08	1451
N4	743510	3036640	86236	11.65	25110	76460	6502	0.21	10170	31085	3583	0.18	5418

Table 1. The results for *pdb* heuristics and greedy algorithm.

	global time semantics				**local time semantics**				**local time semantics and POR**				
	exp. states	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**succ. pruned**
M1	7668	19803	71	0.11	37	110	13	0.01	37	110	13	0.01	0
M2	18847	53407	119	0.29	91	258	15	0.01	19	59	18	0.01	16
M3	19597	54064	124	0.3	68	229	14	0.01	45	157	14	0.01	34
M4	46170	132609	160	0.78	202	609	16	0.01	24	92	23	0.01	25
N1	9117	22691	99	0.17	56	176	13	0.01	56	176	13	0.01	0
N2	23462	71523	154	0.49	175	502	16	0.02	19	59	18	0.01	16
N3	43767	131003	147	0.91	113	343	14	0.02	68	218	14	0.01	41
N4	152163	500656	314	3.52	331	939	17	0.02	26	90	25	0.01	37

Table 2. The results for *hu* heuristics and greedy algorithm.

The results shown in Table 1 and Table 2 are executed with greedy algorithm, and with *pdb* and *hu* heuristics respectively. It can be seen clearly that application of local time semantics significantly decreases the number explored and generated states as well as required time. Also the found paths to the goal state is much shorter when the local time semantics is used is used. It can also be observed that *hu* heuristics give better results compared to the *pdb* heuristics. When a partial order reduction method is used with the local time semantics, the results are generally better or similar to the results obtained by using local time semantics without a partial order reduction method. When *pdb* heuristics is used, using only local time semantics may lead to the solution with smaller number of generated and explored states but with a longer trace for some benchmarks.

	global time semantics				**local time semantics**				**local time semantics and POR**				
	exp. states	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**succ. pruned**
M1	54972	61783	47	0.25	2204	3473	13	0.02	2198	3449	13	0.02	24
M2	221857	278975	50	1.12	8224	13877	14	0.04	5577	8947	14	0.05	1148
M3	228131	282415	50	1.19	8165	14137	14	0.04	5732	9463	14	0.04	1158
M4	889594	1196731	53	5.71	27657	49934	15	0.1	13203	22122	15	0.11	4892
N1	110371	132495	49	1.29	3009	4842	13	0.02	2993	4810	13	0.04	32
N2	537929	651051	52	7.37	12972	21614	14	0.08	8507	14176	14	0.11	1447
N3	577424	687468	52	8.28	13946	23342	14	0.09	9838	16541	14	0.12	1635
N4	2550820	3054497	55	44.15	56201	95170	15	0.34	25959	44709	15	0.32	7551

Table 3. The results for *pdb* heuristics and A* algorithm.

	global time semantics				**local time semantics**				**local time semantics and POR**				
	exp. states	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**exp. states**	**gen. states**	**trace length**	**time elapsed**	**succ. pruned**
M1	40769	50468	47	0.51	31	108	13	0.01	31	108	13	0.01	0
M2	151963	202989	50	2.09	37	157	14	0.02	25	86	14	0.01	16
M3	190974	249250	50	2.65	34	144	14	0.02	26	102	14	0.01	13
M4	696473	962086	53	10.8	57	296	15	0.01	27	106	15	0.01	28
N1	93838	118284	49	1.86	31	108	13	0.01	31	108	13	0.01	0
N2	372534	486964	52	8.43	37	157	14	0.01	25	86	14	0.01	16
N3	510444	633414	52	11.77	39	173	14	0.01	26	102	14	0.01	13
N4	1923889	2460962	55	50.77	49	240	15	0.01	27	106	15	0.01	28

Table 4. The results for *hu* heuristics and A* algorithm.

Table 3 and Table 4, shows the tests of the same benchmarks with A* algorithm instead of greedy algorithm. Without local time semantics, A* algorithm finds shorter paths by generally exploring and generating more states and always requiring more time than the greedy algorithm. With the local time semantics and partial order reduction, A* finds shorter paths than greedy algorithm with similar timing especially with *hu* heuristics. Also

the number of explored and generated states are decreased significantly.

	global time semantics				local time semantics				local time semantics and POR				
	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	succ. pruned
C1	12780	19510	922	0.04	8938	13242	520	0.04	7907	11573	476	0.21	247
C2	38158	67246	1386	0.08	26963	45756	784	0.12	13205	19838	534	0.35	3347
C3	54871	94937	1524	0.11	38007	63387	701	0.15	25471	38697	485	0.62	4467
C4	519921	1004101	7725	1.12	356777	657330	2302	1.52	92564	153574	721	2.56	40626
C5	4763592	9326739	25647	10.95	3721565	6982160	5630	16.41	412905	732159	1897	11.52	276012
C6	49070353	98083606	286447	122.28	-	-	-	-	2444357	4697526	7221	64.69	1462933

Table 5. The results for *pdb* heuristics and greedy algorithm.

	global time semantics				local time semantics				local time semantics and POR				
	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	succ. pruned
C1	32359	36018	54	0.06	19476	23461	47	0.05	17358	20520	47	0.37	402
C2	95730	114916	54	0.17	54517	69982	47	0.15	25127	30464	47	0.57	4944
C3	123406	150361	54	0.27	69492	88876	47	0.19	45348	55913	47	1.02	6500
C4	1114824	1516550	55	2.84	602828	855978	48	1.95	150898	206660	48	3.64	59397
C5	9132765	13413292	56	23.94	4975825	7549105	49	18.75	574893	862608	49	14.27	343576
C6	-	-	-	-	-	-	-	-	3119370	5409199	49	78.19	1820078

Table 6. The results for *pdb* heuristics and A* algorithm.

The next set of benchmark denoted by "C" corresponds to the Single-tracked Line Segment study taken from [39]. The problem is the safety of tram traffic on a line segment where only a single track is available. A distributed controller is designed which never permits trams coming from both directions to enter the segment simultaneously to prevent collision. The asynchronous communication between the control automata is made faulty with the introduction of delay.

Table 5 and Table 6 shows the results achieved with *pdb* heuristics, and greedy and A* algorithms. It can be seen that greedy algorithm requires less states to be generated and explored than A* with *pdb* heuristics. Application of local time semantics always decreases the number of explored and generated states to reach the goal. However, the benchmark "C6" cannot be solved using *pdb* heuristics and local time semantics due to insufficient memory. The reason for this is that although the number of states explored and generated states decreases, it is not sufficient to compensate the increase in the size of DBMs because of the additional reference clocks. Using partial order reduction together with the local time semantics decreases the number of states generated and explored further as well as the trace length. The partial order reduction slows down smaller problems; however, it can reach the goal state faster when the problem is harder. Also it can be seen that the

benchmark C6 cannot be solved with A* and *pdb* heuristics due to insufficient memory with global time semantics or local time semantics; however, application of partial order reduction ensures the solution. The benchmarks C7-C9 cannot be solved with *pdb* heuristics with any neither search algorithm because of insufficient memory.

	global time semantics				local time semantics				local time semantics and POR				
	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	succ. pruned
C1	429	729	67	0.02	167	309	55	0.02	134	242	48	0.02	12
C2	828	1503	83	0.03	263	535	69	0.02	117	209	48	0.02	61
C3	1033	1836	79	0.04	296	606	65	0.03	166	312	50	0.02	62
C4	12938	25655	112	0.37	9220	17966	92	0.32	280	594	58	0.03	266
C5	65506	135680	176	1.99	7499	17012	151	0.27	728	1567	118	0.07	895
C6	453763	941550	432	14.01	23233	52413	263	0.82	1845	3931	222	0.14	2141
C7	4230394	8677170	924	132.85	84634	188351	803	2.99	5337	11562	558	0.31	3563
C8	3403395	7147013	2221	114.28	2133016	4350150	955	78.28	171463	349297	594	9.86	169608
C9	21656780	44060729	1952	743.2	15718025	31816351	1652	610.87	1139845	2403990	148	65.24	471349

Table 7. The results for *hu* heuristics and greedy algorithm.

	global time semantics				local time semantics				local time semantics and POR				
	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	succ. pruned
C1	7827	10618	54	0.18	4791	7174	47	0.12	4252	6302	47	0.22	96
C2	19183	27582	54	0.45	11033	17548	47	0.29	5099	7538	47	0.29	1306
C3	21464	30675	54	0.51	12117	19171	47	0.35	7986	11932	47	0.42	1540
C4	159116	251643	55	4.28	81452	140721	48	2.51	19536	30607	48	1.15	11462
C5	1081800	1842812	56	32.14	513272	946964	49	18.07	53483	89620	49	3.26	46697
C6	10043635	17948840	56	312.36	4364730	8445983	49	160.26	244296	452462	49	15.21	215907
C7	-	-	-	-	-	-	-	-	1485645	3072444	49	90.17	951621
C8	-	-	-	-	-	-	-	-	940354	2042832	49	57.04	542623
C9	-	-	-	-	-	-	-	-	4155649	8990957	50	250.72	883620

Table 8. The results for *hu* heuristics and A* algorithm.

Table 7 and Table 8 shows the results for *hu* heuristics, and greedy and A* algorithms respectively. It can be observed that the problems which are unsolvable with *pdb* heuristics are solvable with *hu* heuristics. Using local time semantics greatly improves the performance of both search algorithms in terms of both generated and explored states and elapsed time as well as the trace length when *hu* heuristics is used. Application of partial order reduction improves the performance further ensuring even faster operation with less states. Also the problems which cannot be solved with A* and *hu* heuristics can be solved with the application of partial order reduction.

	global time semantics				local time semantics				local time semantics and POR				
	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	succ. pruned
FA5	1374	2425	8	0.01	1352	2690	8	0.03	1352	2690	8	0.01	0
FA10	37154	96820	8	0.25	36512	102300	8	0.54	36512	102300	8	1.46	0
FA15	345994	1084260	8	4.78	341507	1112280	8	11.05	-	-	-	-	-
FB5	371	742	6	0.01	351	742	6	0.01	351	742	6	0.01	0
FB10	5451	14202	6	0.04	5361	14202	6	0.07	5361	14202	6	0.14	0
FB15	34356	111622	6	0.48	34146	111622	6	1.04	34146	111622	6	1.61	0

Table 9. The results for *pdb* heuristics and A* algorithm.

	global time semantics				local time semantics				local time semantics and POR				
	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	exp. states	gen. states	trace length	time elapsed	succ. pruned
FA5	9	27	8	0.01	9	27	8	0.01	9	27	8	0.01	0
FA10	9	42	8	0.01	9	42	8	0.01	9	42	8	0.01	0
FA15	9	57	8	0.01	9	57	8	0.01	9	57	8	0.01	0
FB5	7	21	6	0.01	7	21	6	0.01	7	21	6	0.01	0
FB10	7	36	6	0.01	7	36	6	0.01	7	36	6	0.01	0
FB15	7	51	6	0.01	7	51	6	0.01	7	51	6	0.01	0

Table 10. The results for *hu* heuristics and A* algorithm.

The next set of benchmarks tested are denoted by "FA" and "FB" and they correspond to two variants of Fischer protocol for mutual exclusion. FA set has an additional automata with synchronization and FB set selects two automata for the error condition. The error condition is the simultaneous presence of two of the automata in a certain location. In the automata, one of the temporal conditions is weakened to make the error possible. The numbers following the letters show the number of automata in each benchmark. Each automata in these benchmarks has a local clock.

Table 9 and Table 10 shows the results with A* algorithm and *pdb* and *hu* heuristics respectively. *hu* heuristics perform very well in these benchmarks so no more improvement can be achieved with local time semantics or partial order reduction. When *pdb* heuristics is used, the number of states generated and explored is slightly less with local time semantics. Almost all of the transitions in these benchmarks use the same variable, so there are not any independent transitions, so none of the successors can be pruned by the partial order reduction. Moreover, since local time semantics adds a reference clock to each process, the size of DBMs is increased greatly since the number of clocks is doubled (Note that each process has only one clock). Also the number of processes is relatively high for these benchmarks; so despite the slight decrease in the number of explored and generated states, the time required to find a solution is higher due to the computations with much larger DBMs and insufficient amount of decrease in the number of successor states. Also note that with partial order reduction, one of the benchmarks cannot be solved because of insufficiency of the memory.

5.3 Conclusion

In this work, application of partial order reduction method to the reachability analysis of timed systems is presented. Partial order reduction is an efficient method to decrease the size of the state-space required to be explored for model checking of finite-state systems using the independence of transitions. However, because of the implicit synchronization of transitions due to the clocks in timed systems, improvements provided by partial order reduction is limited. Introduction of local time semantics which removes the implicit synchronization between clocks and allows them to advance independent of other clocks enables efficient application of partial order reduction to timed systems.

The experimental results show that partial order reduction can improve the performance of reachability analysis of timed automata significantly by decreasing the number of explored and generated states greatly. It also decreases the required time to find a path to the goal state. Using local time semantics without partial order reduction may also improve the performance; however, it is almost always better to use with a partial order reduction method. The size of the DBMs increase because of the additional reference clocks and this causes the DBM operations to take longer time; however, the decrease in the number of generated states generally compensate for this increase in the computation time. However, if the dependency between transitions is very high in a network of timed automata, the performance improvement is limited; and even the time required to find a solution may be longer due to the increase in the size of the DBMs because of the additional reference clocks. Also, the performance improvement provided by partial order reduction generally increases with the increasing size of the problem as long as the dependency between the transitions is not very high.

In this work, the stubborn sets is implemented as the partial order reduction method. The local time semantics can be used with any partial order reduction method. So other partial order reduction methods, such as sleep sets [6], can be implemented and the performance of different partial order reduction methods can be compared as a future work.

References

1. Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal in 1995. In Tiziana Margaria and Bernhard Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 431–434. Springer, 1996.
2. Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
3. Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool kronos. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer, 1995.
4. Martin Wehrle and Sebastian Kupferschmid. Mcta: Heuristics and search for timed systems. In Marcin Jurdzinski and Dejan Nickovic, editors, *FORMATS*, volume 7595 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2012.
5. Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using uppaal. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer, 1996.
6. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
7. Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer, 1990.
8. Antti Valmari. On-the-fly verification with stubborn sets. In Courcoubetis [40], pages 397–408.
9. Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.
10. Florence Pagani. Partial orders and verification of real-time systems. In Bengt Jonsson and Joachim Parrow, editors, *FTRTFT*, volume 1135 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 1996.
11. Tomohiro Yoneda and Bernd-Holger Schlingloff. Efficient verification of parallel real-time systems. *Formal Methods in System Design*, 11(2):187–215, 1997.
12. Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 485–500. Springer, 1998.
13. Marius Minea. Partial order reduction for model checking of timed automata. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 431–446. Springer, 1999.
14. Marius Minea. *Partial order reduction for verification of timed systems.* PhD thesis, Citeseer, 1999.
15. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In Horst Reichel, editor, *FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer, 1995.
16. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.
17. Ernst-Rüdiger Olderog and Henning Dierks. *Real-time systems - formal specification and automatic verification.* Cambridge University Press, 2008.
18. David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1989.
19. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *LICS*, pages 394–406, 1992.
20. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.
21. Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. In Courcoubetis [40], pages 210–224.
22. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In Dieter Hogrefe and Stefan Leue, editors, *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 243–258. Chapman & Hall, 1994.
23. Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1957.

24. Nicolas Halbwachs. Delay analysis in synchronous programs. In Courcoubetis [40], pages 333–346.
25. Béatrice Bérard, Antoine Petit, Volker Diekert, and Paul Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundam. Inform.*, 36(2-3):145–182, 1998.
26. T. ROKICKI. Representing and modeling circuits. *Ph. D. thesis, Stanford University*, 1993.
27. Paul Pettersson. *Modelling and verification of real-time systems using timed automata: theory and practice.* Department of Computer systems, Uppsala Univ., 1999.
28. Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.
29. Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In Courcoubetis [40], pages 438–449.
30. Martin Wehrle and Malte Helmert. About partial order reduction in planning and computer aided verification. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *ICAPS*. AAAI, 2012.
31. Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*. AAAI, 2009.
32. Stefan Edelkamp, Viktor Schuppan, Dragan Bosnacki, Anton Wijs, Ansgar Fehnker, and Husain Aljazzar. Survey on directed model checking. In Doron Peled and Michael Wooldridge, editors, *MoChArt*, volume 5348 of *Lecture Notes in Computer Science*, pages 65–89. Springer, 2008.
33. Sebastian Kupferschmid, Jörg Hoffmann, Henning Dierks, and Gerd Behrmann. Adapting an ai planning heuristic for directed model checking. In Antti Valmari, editor, *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52. Springer, 2006.
34. Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
35. Sebastian Kupferschmid and Martin Wehrle. Abstractions and pattern databases: The quest for succinctness and accuracy. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2011.
36. Judea Pearl. *Heuristics - intelligent search strategies for computer problem solving.* Addison-Wesley series in artificial intelligence. Addison-Wesley, 1984.
37. Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.
38. Henning Dierks. Comparing model checking and logical reasoning for real-time systems. *Formal Asp. Comput.*, 16(2):104–120, 2004.
39. Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, and Alexander Baer. The uniform workbench, a universal development environment for formal methods. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1186–1205. Springer, 1999.
40. Costas Courcoubetis, editor. *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*. Springer, 1993.