Difference Bound Constraint Abstraction for Timed Automata Reachability Checking

Weifeng Wang[1,2] and Li Jiao[1(✉)]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{wangwf,ljiao}@ios.ac.cn

[2] University of Chinese Academy of Sciences, Beijing 100049, China

Abstract. We consider the reachability problem for timed automata. One of the most well-known solutions for this problem is the zone-based search method. Max bound abstraction and LU-bound abstraction on zones have been proposed to reduce the state space for zone based search. These abstractions use bounds collected from the timed automata structure to compute an abstract state space. In this paper we propose a difference bound constraint abstraction for zones. In this abstraction, sets of difference bound constraints collected from the symbolic run are used to compute the abstract states. Based on this new abstraction scheme, we propose an algorithm for the reachability checking of timed automata. Experiment results are reported on several timed automata benchmarks.

1 Introduction

Model checking of timed automata has been studied for a long time since it was proposed [2]. One of the most interesting properties to be verified for timed automata is the reachability property. In this paper, we will focus on the reachability problem of timed automata.

It is known that the reachability problem for timed automata is PSPACE-complete [3]. Initially region-based method [2] was used to discretize the state space, and convert the timed automata model to finite automata. However, the resulting finite automata are so large that it is not practical to perform model checking on them. BDD-based [4,8] and SAT-based [17] symbolic model checking can be used to fight the state explosion.

Zone-based method is an important approach to the reachability problem of timed automata. In zone-based method, a group of difference bound inequalities is used to symbolically represent a convex set of clock valuations (which is called a "zone"), and exhaustive search is performed on the symbolic state space [7]. Abstraction techniques for zones are used to reduce the symbolic state space, and ensure the reduced symbolic state space to be finite. In max-bound abstraction (a.k.a. k-approximation), the maximum constants appearing in the guards of the timed automata are collected, and used to compute abstractions for zones. LU-abstraction [6] improves by classifying the constants into two categories: those appearing in lower bound guards and those appearing in upper bound guards.

© IFIP International Federation for Information Processing 2015

DOI: 10.1007/978-3-319-19195-9_10

Behrmann et al. [5] used static analysis on the structure of timed automata to obtain smaller bounds, which lead to coarser abstractions. Herbreteau et al [12] proposed to calculate the bounds on-the-fly, and used a non-convex abstraction based on LU-bounds. All the above mentioned techniques are based on bounds that are collected from the timed automata, which just capture limited amount of information about the system.

In this paper, we explore the possibility to use difference bound constraints as abstractions for zones. In our abstraction scheme, a set of difference bound constraints is used as the abstraction of the zone. In fact, the conjunction of these difference bound constraints is a zone that is larger than the original zone. This abstraction is, to some extent, similar to the predicate abstraction in the program verification field.

A lazy search algorithm similar to that in [13] is used to gradually refine the abstraction. Each node is a tuple (l, Z, C), where (l, Z) is the symbolic state, and C is a set of difference bound constraints, which serves as the abstraction. Initially, the difference bound constraint set of each node is $\emptyset$, which means there is no difference bound constraint in the abstraction, i.e., the abstracted zone is the set of all clock valuations. If a transition t is disabled from a node (l, Z, C), we extract a set of difference bound constraints C_t from Z such that $Post_t(\llbracket C_t \rrbracket) = \emptyset$, C_t is sufficient to prove that the transition t from the configuration (l, Z) is disabled. The difference bound constraints in C_t are added to C, after which the change in C is propagated backward according to certain rules. The addition of difference bound constraints into the abstraction is in fact a refinement operation.

The key problem here is how to compute and propagate the set of difference bound constraints. We propose a method to propagate difference bound constraints, which makes use of structural information of the timed automata to identify "important" difference bound constraints in the zones from those that are "irrelevant".

Unfortunately, the lazy search algorithm using only difference bound constraint abstraction does not necessarily terminate. The LU-abstraction $Extra^+_{LU}$ [6] is used in our algorithm to ensure termination. The resulting algorithm can be seen as a state space reduction using difference bound constraint abstraction on top of $Extra^+_{LU}$-based symbolic search.

We performed experiments to compare our method with zone-based search and the lazy abstraction method proposed in [13]. Results show that in general our method behaves similarly to that in [13], while in some cases our method can achieve better reduction of the state spaces.

Related Work. Abstraction refinement techniques [9] have attracted much attention in recent years. This kind of techniques check the property of the system by iteratively checking and refining abstract models which tend to be smaller than the original model. Efforts have been devoted on adapting abstraction refinement techniques for the verification of timed automata [16,10].

Lazy abstraction [11] is an important abstraction refinement technique. In the lazy abstraction procedure, an abstract reachability tree is built on-the-fly, along with the refinement procedure, and predicate formulas are used to represent the abstract symbolic states. Difference bound constraint abstraction is similar to predicate abstraction, and the constraint propagation resembles interpolation [15]. In our method, abstractions only take the form of conjunctions of difference bound constraints, which is more efficient than general-purpose first order formulas. Herbreteau et al. [13] proposed a lazy search scheme to dynamically compute the LU-bounds during state space exploration, which results in smaller LU-bounds and coarser abstractions. We use a similar lazy search scheme.

Organization of the Paper. In Section 2 we have a simple review of basic concepts related to timed automata. We present the difference bound constraint abstraction, and the model checking algorithm based on this abstraction in Section 3. An example is given in Section 4 to illustrate how our method achieves state space reduction. Experiment results are reported in Section 5, and conclusions are given in Section 6.

2 Preliminaries

2.1 Timed Automata and the Reachability Property

A set of *clock variables* X is a set of non-negative real-valued variables. A *clock constraint* is a conjunction of constraints of the form $x \sim c$, where $x, y \in X$, $c \in \mathbb{N}$, and $\sim \in \{<, \leq, >, \geq\}$. A *difference bound constraint* on X is a constraint of the form $x - y \prec c$, where $x, y \in X \cup \{0\}$, $c \in \mathbb{N}$, and $\prec \in \{<, \leq\}$. Obviously a clock constraint can be re-written as conjunctions of difference bound constraints. A *clock valuation* is a function $\nu : X \mapsto \mathbb{R}_{\geq 0}$, which assigns to each clock variable a nonnegative real value. We denote $\mathbf{0}$ the special clock valuation that assigns 0 to every clock variable. For a formula φ on X, we write $\nu \models \varphi$, if φ is satisfied by the valuation ν. Furthermore, we denote by $[\![\varphi]\!]$ the set of all clock valuations satisfying φ, i.e., $[\![\varphi]\!] = \{\nu | \nu \models \varphi\}$.

Definition 1 (Timed Automata). *A* timed automaton *is a tuple $\langle L, l_{init}, X, T\rangle$, where L is a finite set of locations, l_{init} is the initial location, X is a finite set of clocks, and T is a finite set of transitions of the form $l \xrightarrow{a,g,r} l'$, where a is an action label, g is a clock constraint, which we call* guard*, and $r \subseteq X$ is the set of clocks to be reset.*

For a transition $t = l \xrightarrow{a,g,r} l' \in T$, we use $t.a$, $t.g$, $t.r$ to denote the corresponding action, guard, and set of clocks to be reset.

Definition 2 (Semantics of Timed Automata). *A* configuration *of a timed automaton $\mathcal{A} = \langle L, l_{init}, X, T\rangle$ is a pair (l, ν) where $l \in L$ is a location, and ν is a clock valuation. The initial configuration is $(l_{init}, \mathbf{0})$. There are two kinds of transitions*

- ***Action**. For each pair of states (l, ν) and (l', ν'), $(l, \nu) \rightarrow_t (l', \nu')$ iff there is a transition $t = l \xrightarrow{a,g,r} l' \in T$, and*

    - $\nu \models g$, *and*
    - $\nu'(x) = \nu(x)$ *for each* $x \notin r$, *and*
    - $\nu'(x) = 0$ *for each* $x \in r$.
- ***Delay***. *For each pair of configurations* (l, ν) *and* (l', ν'), *and an arbitrary* $\delta \in \mathbb{R}_{\geq 0}$, $(l, \nu) \rightarrow_\delta (l', \nu')$, *iff* $l = l'$ *and* $\nu'(x) = \nu(x) + \delta$ *for each clock* $x \in X$.

A run *of a timed automaton is a (possibly infinite) sequence of configurations* $\rho = (l_0, \nu_0)(l_1, \nu_1)\cdots$, *where* $(l_0, \nu_0) = (l_{init}, \boldsymbol{0})$, *and for each* $i \geq 0$, *either* $(l_i, \nu_i) \rightarrow_t (l_{i+1}, \nu_{i+1})$ *for some* $t \in T$, *or* $(l_i, \nu_i) \rightarrow_\delta (l_{i+1}, \nu_{i+1})$ *for some* $\delta \in \mathbb{R}_{\geq 0}$.

The definition of timed automata is always extended to networks of timed automata, which is a parallel composition of timed automata. The parallel composition can be obtained by the product of these components. Usually this product is not computed directly, but on-the fly during the verification. In this paper we will describe our method based on timed automata, while it could be naturally extended on networks of timed automata.

In this paper we will consider the *reachability* problem. Basically, a location of a timed automaton is reachable iff there is a run of the timed automaton that reaches the location.

Definition 3 (Reachability). *A location* l_{acc} *of a timed automaton* $\mathcal{A}$ *is* reachable *iff there is a finite run* $\rho = (l_0, \nu_0)(l_1, \nu_1)\cdots(l_k, \nu_k)$, *where* $l_k = l_{acc}$.

2.2 Zone Based Symbolic Semantics

The symbolic semantics of timed automata has been proposed to fight state explosion. Basically, the idea is to represent a set of clock valuations using clock constraints. Zones are used in timed automata model checking to symbolically represent the sets of clock valuations. A *zone* is a convex set of clock valuations that can be represented by a set of difference bound constraints.

For a zone Z and a clock constraint g, we define $Z \wedge g$ as $\{\nu | \nu \in Z \wedge \nu \models g\}$, $Z[r := 0]$ as $\{\nu | \exists \nu' \in Z \cdot \forall x \in r(\nu(x) = 0) \wedge \forall x \notin r(\nu(x) = \nu'(x))\}$, and $Z \uparrow$ as $\{\nu | \exists \nu' \in Z, \delta \in \mathbb{R}_{\geq 0} \cdot \nu = \nu' + \delta\}$. Zones are closed under these operations [7].

Definition 4 (Symbolic Semantics of Timed Automata). *The symbolic semantics of a timed automaton* $\mathcal{A} = \langle L, l_{init}, X, T\rangle$ *is a labeled transition system* $(S, \Rightarrow, s_0)$. *Each state* $s \in S$ *is a symbolic configuration* (l, Z), *where* l *is a location, and* Z *is a zone. The initial state is* $s_0 = (l_{init}, [\![0 \leq x_1 = x_2 = \cdots = x_n]\!])$. *For each pair of states* $s = (l, Z)$ *and* $s' = (l', Z')$, $s \Rightarrow_t s'$ *iff there exists a transition* $t = l \xrightarrow{a,g,r} l' \in T$ *such that* $Z' = (Z \wedge g)[r := 0] \uparrow$. *A* symbolic run *is a sequence* $(l_0, Z_0)(l_1, Z_1)\cdots$, *where* $(l_0, Z_0) = (l_{init}, [\![0 \leq x_1 = x_2 = \cdots = x_n]\!])$, *and for each* $i \geq 0$, $(l_i, Z_i) \Rightarrow_t (l_{i+1}, Z_{i+1})$ *for some* $t \in T$.

In addition, we define the *Post* operator $Post_t(Z) \overset{\text{def}}{=} (Z \wedge t.g)[t.r := 0] \uparrow$. A transition $t = l \xrightarrow{a,g,r} l' \in T$ is *disabled* at (l, Z), if $Post_t(Z) = \emptyset$.

Symbolic semantics is sound and complete with respect to the reachability property [7]. A given location l_{acc} is reachable iff there is a symbolic run ending with a configuration (l_{acc}, Z) where $Z \neq \emptyset$.

Zones can be represented as Difference Bound Matrices (DBMs), and efficient algorithms for manipulating DBMs have already been proposed [7]. The DBM representation of a zone on the clock set X is a $(|X| + 1) \times (|X| + 1)$ matrix, each element of which is a tuple $(\prec, c)$, where $\prec \in \{<, \leq\}$ and $c \in \mathbb{N}$. In a DBM D, $D_{0x} = (\prec, c)$ means $0 - x \prec c$, i.e. $x \succ -c$, $D_{x0} = (\prec, c)$ means $x - 0 \prec c$, i.e. $x \prec c$, for $x, y \neq 0$, $D_{ij} = (\prec, c)$ represents the constraint $x_i - x_j \prec c$. Two different DBMs might correspond to the same zone. In order to tackle this problem, the *canonical forms* of DBMs can be computed by the Floyd-Warshall algorithm [7].

The zone-based semantics described in the above is not necessarily finite. Max-bound abstraction and LU-bound abstraction are proposed to reduce the state space to finite, and the former can be seen as a special case of the latter. Basically, these abstraction techniques remove from the zone those constraints that exceed certain bounds, resulting in an abstracted zone that is larger than the original one. Coarser abstractions lead to smaller symbolic state space. As far as we know, $Extra^+_{LU}$ [6] is the coarsest convex-preserving abstraction based on LU-bounds.

An LU-bound is a pair of functions LU, where $L : X \to \mathbb{N} \cup \{-\infty\}$ is called a lower bound function and $U : X \to \mathbb{N} \cup \{-\infty\}$ an upper bound function.

Definition 5 (LU-extrapolation [6]). *Let Z be a zone whose canonical DBM is $\langle c_{i,j}, \prec_{i,j} \rangle_{i,j=0,1,\ldots,|X|}$. Given an LU-bound LU , the LU-extrapolation $Extra^+_{LU}(Z)$ of Z is a zone Z' which can be represented by a DBM $\langle c'_{i,j}, \prec'_{i,j} \rangle_{i,j=0,1,\ldots,|X|}$, where*

$$\langle c'_{i,j}, \prec'_{i,j} \rangle = \begin{cases} \infty & \text{if } c_{i,j} > L(x_i) \\ \infty & \text{if } -c_{0,i} > L(x_i) \\ \infty & \text{if } -c_{0,j} > U(x_j), i \neq 0 \\ (-U(x_j), <) & \text{if } -c_{0,j} > U(x_j), i = 0 \\ (c_{i,j}, \prec_{i,j}) & \text{otherwise} \end{cases}$$

We denote by $\Rightarrow^E$ the symbolic semantics of timed automata augmented with $Extra^+_{LU}$: for two symbolic configurations (l, Z) and (l', Z'), $(l, Z) \Rightarrow^E (l', Z')$ iff there is a zone Z'' such that $(l, Z) \Rightarrow (l', Z'')$ and $Z' = Extra^+_{LU}(Z'')$. We can choose the LU-bound $L(x)$ and $U(x)$ as follows: for each clock x, $L(x)(U(x))$ is the largest constant c such that $x > c(x < c)$ or $x \geq c(x \leq c)$ appears in the guard of some transition. Intuitively, $L(x)$ $(U(x))$ collects the maximum constant appearing in the lower-bound (upper-bound) guard of x. It can be proved that $\Rightarrow^E$ preserves reachability [6].

3 Difference Bound Constraint Abstraction Based Reachability Checking

3.1 Difference Bound Constraint Abstraction and Adaptive Simulation Graph

For a zone Z and a difference bound constraint $\mathfrak{c}$ we denote $Z \models \mathfrak{c}$ as $\forall \nu \in Z \cdot \nu \models \mathfrak{c}$. For a set C of difference bound constraints we denote $[\![C]\!]$ to be the set of valuations that satisfy all the constraints in C, i.e., $[\![C]\!] = \{\nu | \forall \mathfrak{c} \in C \cdot \nu \models \mathfrak{c}\}$. Intuitively, a set C of difference bound constraints is interpreted as a conjunction of the constraints from C. In fact, $[\![C]\!]$ is also a zone, although it will not be stored as a canonical DBM in our algorithm. In the sequel, we might mix the use of set-theoretic notations (e.g. intersection) and logic notations (e.g. conjunction) on zones and constraints.

Definition 6 (Difference Bound Constraint Abstraction). *A* difference bound constraint abstraction *C for a zone Z is a set of difference bound constraints such that $Z \subseteq [\![C]\!]$.*

This definition resembles the concept of predicate abstraction in program verification. Each difference bound constraint in C can be seen as a predicate. In our algorithm, only those difference bound constraints that are useful for the reachability problem are kept, while the irrelevant constraints are ignored. The difference bound constraint abstraction of a zone is still a zone, but we store it as a set of constraints rather than a canonical DBM. Based on this abstraction, we define an Adaptive Simulation Graph (ASG) similar to that in [13].

Definition 7 (Adaptive Simulation Graph). *Given a timed automaton $\mathcal{A}$, the adaptive simulation graph $ASG_{\mathcal{A}}$ of $\mathcal{A}$ is a graph with nodes of the form (l, Z, C), where l is a location, Z is a zone, and C is a set of difference bound constraints such that $Z \subseteq [\![C]\!]$. A node could be marked* tentative *(which roughly means it is covered by another node). Three constraints should be satisfied:*
G1 *For the initial state l_0 and initial zone Z_0, there is a node (l_0, Z_0, C_0) in the graph for some C_0.*
G2 *If a node (l, Z, C) is not tentative, then for every transition $(l, Z) \Rightarrow_t (l', Z')$ s.t. $Z' \neq \emptyset$, there is a successor (l', Z', C') for this node.*
G3 *If a node (l, Z, C) is tentative, there is a non-tentative node (l, Z', C') covering it, i.e., $Z \subseteq [\![C]\!] \subseteq [\![C']\!]$.*
In addition, two invariants are required.
I1 *If a transition t is disabled from (l, Z), and (l, Z, C) is a non-tentative node, then t should also be disabled from $(l, [\![C]\!])$.*
I2 *For every edge $(l, Z, C) \Rightarrow_t (l', Z', C')$ in the ASG: $Post_t([\![C]\!]) \subseteq [\![C']\!]$.*

For a node $v = (l, Z, C)$, we use $v.l$, $v.Z$, and $v.C$ to denote the three components. We say that a node (l, Z, C) is *covered* by a node (l, Z', C'), if $Z \subseteq [\![C']\!]$.

The following theorem states that, the adaptive simulation graph preserves reachability of the corresponding timed automaton.

Theorem 1. *A location l_{acc} in the timed automaton $\mathcal{A}$ is reachable, iff there is a node (l_{acc}, Z, C) such that $Z \neq \emptyset$ in $ASG_{\mathcal{A}}$.*

The right-to-left direction of this theorem is obviously true: by the definition of ASG, each path in $ASG_{\mathcal{A}}$ corresponds to a symbolic run in $\mathcal{A}$. In order to prove the other direction of Theorem 1, we prove a slightly stronger lemma.

Lemma 1. *If there is a symbolic run $(l_0, Z_0) \cdots (l, Z)$ in $\mathcal{A}$ with $Z \neq \emptyset$, then there must be a non-tentative node (l, Z_1, C_1) in $ASG_{\mathcal{A}}$ such that $Z \subseteq [\![C_1]\!]$.*

Proof. We prove this lemma by induction on the length of the run. For the 0-length run and (l_0, Z_0), the lemma is trivially true. Assume the lemma is true for a run $(l_0, Z_0) \cdots (l, Z)$, now we prove that the lemma holds for every successor (l', Z') of (l, Z) with $(l, Z) \Rightarrow_t (l', Z')$ and $Z' \neq \emptyset$.

We need to prove that there is a node in $ASG_{\mathcal{A}}$ corresponding to (l', Z') as described in the lemma. By induction hypothesis, there is a non-tentative node v such that $Z \subseteq [\![v.C]\!]$. We now assert that $Post_t(v.Z) \neq \emptyset$. Because, otherwise, by **I1** of Definition 7 we will have $Post_t([\![v.C]\!]) = \emptyset$, and consequently $Post_t(Z) \subseteq Post_t([\![v.C]\!]) = \emptyset$, which contradicts the assumption $Z' \neq \emptyset$. From **G2** we know that there is a successor v' of v such that $(v.l, v.Z) \Rightarrow_t (v'.l, v'.Z)$. By **I2** we have $Post_t([\![v.C]\!]) \subseteq [\![v'.C']\!]$, so $Z' = Post_t(Z) \subseteq Post_t([\![v.C]\!]) \subseteq [\![v'.C]\!]$, If v' is non-tentative, then it is a node that we want to find, and the lemma is proved. Otherwise, there is a non-tentative node v'' covering v'. From **G3** we know that $[\![v'.C]\!] \subseteq [\![v''.C]\!]$, so $Z' \subseteq [\![v''.C]\!]$, and v'' is the qualifying node.

According to Theorem 1, the reachability problem could be solved by exploring the ASG. The algorithm for constructing the ASG will be described in the next subsection.

3.2 The ASG-Constructing Algorithm

The algorithm for constructing the ASG is shown in Algorithm 1. The main procedure repeatedly calls EXPLORE to explore the nodes until an accepting node is found (lines 10-11), or the worklist is empty (line 8).

The function EXPLORE proceeds as follows. For a node v to be explored, first it checks whether v is an accepting node. If so, the algorithm exits with the result "reachable", otherwise it checks whether there is a non-tentative node v'' covering v. If so, v is marked tentative with respect to v'' (line 13), and the set of difference bound constraints of v'' is copied to v to maintain **G3** of Definition 7 (line 14). There is no need to generate successor nodes for a tentative node. Otherwise, the difference bound constraint set of v is computed using the transitions disabled at v to maintain **I1** (line 17), after which successor nodes of v are generated (maintaining **G2**) and put into the worklist for exploration (lines 19-21).

Whenever the difference bound constraint set of a node v' is changed, PROPAGATE will be called (lines 15, 18, 38) to propagate the newly-added constraints backward to its parent v (to maintain **I2**) (lines 29-31), and further to the nodes

Algorithm 1. The ASG-constructing algorithm

```
 1: function MAIN
 2:     let $v_{root} = (l_0, Z_0, \emptyset)$
 3:     add $v_{root}$ to the worklist
 4:     while worklist not empty do
 5:         remove $v$ from the worklist
 6:         EXPLORE($v$)
 7:         RESOLVE
 8:     return "not reachable"
 9: function EXPLORE($v$)
10:     if $v.l$ is accepting then
11:         exit "reachable"
12:     else if $\exists v''$ non-tentative s.t. $v.l = v''.q \wedge v.Z \subseteq [\![v''.C]\!]$ then
13:         mark $v$ tentative w.r.t. $v''$
14:         $v.C \leftarrow v''.C$
15:         PROPAGATE($v$, $v.C$)
16:     else
17:         $v.C \leftarrow$DISABLED($v.l$, $v.Z$)
18:         PROPAGATE($v$, $v.C$)
19:         for all $(l', Z')$ s.t. $(v.l, v.Z) \Rightarrow (l', Z')$ and $Z' \neq \emptyset$ do
20:             create the successor $v' = (l', Z', \emptyset)$ of $v$
21:             add $v'$ to worklist
22: function RESOLVE
23:     for all $v$ tentative w.r.t. $v'$ do
24:         if $v.Z \nsubseteq [\![v'.C]\!]$ then
25:             mark $v$ non-tentative
26:             set $v.C \leftarrow \emptyset$
27:             add $v$ to worklist
28: function PROPAGATE($v'$, $C'$)
29:     let $v$ = parent($v'$)
30:     $C \leftarrow$ BACKPROP($v$, $v'$, $C'$)
31:     $C_1 \leftarrow$UPDATE($v$, $C$)
32:     if $C_1 \neq \emptyset$ then
33:         for all $v_t$ tentative w.r.t. $v$ do
34:             if $v_t.Z \subseteq [\![v.C]\!]$ then
35:                 $C_t \leftarrow$UPDATE($v_t$, $C_1$)
36:                 PROPAGATE($v_t$, $C_t$)
37:         if $v \neq v_{root}$ then
38:             PROPAGATE($v$, $C_1$)
```

tentative with respect to v (to maintain **G3**) (line 36). If a tentative node v_t is no longer covered by v, the function RESOLVE will eventually be called (line 7) to mark it non-tentative, remove all constraints from its difference bound constraint set, and put it into the worklist for exploration (lines 25-27).

For each node v, the difference bound constraint abstraction $v.C$ is stored as a set of difference bound constraints, rather than a canonical DBM. Checking whether $Z \subseteq [\![C]\!]$ for a zone Z and a difference bound constraint set C can be accomplished by checking whether $Z \models \mathfrak{c}$ for all $\mathfrak{c} \in C$. When adding constraints to a difference bound constraint abstraction, only the strongest constraints are kept, which is handled by the function UPDATE, whose code is omitted here.

3.3 Computing the Difference Bound Constraint Sets

The algorithm for building the ASG relies on two functions DISABLED, and BACKPROP to extract the "important" difference bound constraints from zones. In this subsection we describe an implementation of the two functions. Before doing that we introduce the arithmetic on $\{<, \leq\} \times \mathbb{N}$. For two arbitrary pairs $(\prec_1, c_1), (\prec_2, c_2) \in \{<, \leq\} \times (\mathbb{N} \cup \{+\infty\})$, $(\prec_1, c_1) + (\prec_2, c_2) = (\prec_3, c_3)$, where $c_3 = c_1 + c_2$, and $\prec_3 = <$ iff $\prec_1 = <$ or $\prec_2 = <$. The order "$<$" is defined as: $(\prec_1, c_1) < (\prec_2, c_2)$ iff $c_1 < c_2$ or $c_1 = c_2 \wedge \prec_1 = < \wedge \prec_2 = \leq$.

DISABLED. In order to maintain the invariant **I1**, the result C computed by DISABLED(l, Z) should satisfy: i) $Z \subseteq [\![C]\!]$, and ii) $Post_t([\![C]\!]) = \emptyset$ for each t disabled at (l, Z).

By Definition 4 we know that $Post_t(Z) = \emptyset$ iff $Z \wedge t.g = \emptyset$. Thus the problem is reduced to: given a zone Z, and a formula φ which is conjunctions of difference bound constraints, such that $Z \wedge \varphi = \emptyset$, find a set C of difference bound constraints such that $Z \subseteq [\![C]\!]$ and $[\![C]\!] \wedge \varphi = \emptyset$. Since $Z \wedge \varphi = \emptyset$, there must be a sequence of difference bound constraints $x_0 - x_1 \prec_0 c_0, x_1 - x_2 \prec_1 c_1, \ldots x_{m-1} - x_m \prec_{m-1} c_{m-1}, x_m - x_0 \prec_m c_m$ such that

- **C1** Each of them appears either in φ, or in the canonical DBM of Z.
- **C2** $x_i \neq x_j$ for $i, j \in \{0, 1, \ldots, m\}$ and $i \neq j$
- **C3** $(\prec_0, c_0) + \cdots + (\prec_m, c_m) < (<, 0)$, i.e., this sequence of difference bound constraints forms a contradiction.

We take C to be $\{\mathfrak{c} = x_i - x_{(i+1) \bmod (m+1)} \prec_i c_i | \mathfrak{c} \text{ is from } Z\}$. Obviously, the C obtained above satisfies i) and ii).

As shown in [7], a conjunction of difference bound constraints can be seen as a directed weighted graph, where each clock variable corresponds to a node, and each constraint corresponds to a weighted edge, whose weight is a pair in $\{<, \leq\} \times (\mathbb{N} \cup \{+\infty\})$. Figure 1 illustrates the directed weighted graph for the zone $x \geq 0 \wedge y - x = 10$. There is a contradiction in the conjunction iff there is a negative cycle in the graph, i.e., the sum of weights in the cycle is less than $(\leq, 0)$.

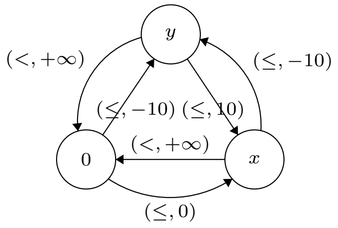


Fig. 1. Directed weighted graph of zone $x \geq 0 \wedge y - x = 10$

The difference bound constraints in C correspond to the edges in the graph of Z that form a negative cycle with the edges from the graph of φ. So the problem is reduced to finding a negative cycle in the merged graph of Z and φ, and picking the edges in the cycle that belong to Z. This task is accomplished by the function FINDCONTRA in Algorithm 2, where Floyd-Warshall algorithm is used to find a negative cycle. Function DISABLED in Algorithm 2 calls FINDCONTRA for every disabled transition, and collects all the constraints obtained.

BACKPROP. In order to maintain the invariant **I2**, the result C computed by BACKPROP(v, v', C') (where $(v.l, v.Z) \Rightarrow_t (v'.l, v'.Z)$) should satisfy: i) $Post_t([\![C]\!]) \subseteq [\![C']\!]$ and ii) $v.Z \subseteq [\![C]\!]$.

If there is a set $C_{\mathfrak{c}'}$ of difference bound constraints for each $\mathfrak{c}' \in C'$, such that $v.Z \subseteq [\![C_{\mathfrak{c}'}]\!]$ and $Post_t([\![C_{c'}]\!]) \models \mathfrak{c}'$, then we can choose C as $\bigcup\limits_{\mathfrak{c}' \in C'} C_{\mathfrak{c}'}$. One can easily check that such a C satisfies the above two conditions. We only need to consider the propagation of each difference bound constraint in C' one by one.

For simplicity, we break up each transition into three steps: guard, reset, and time delay, and explain the propagation for each step. The overall description of BACKPROP is shown in Algorithm 2.

We assume two zones Z_1, Z_2 and a difference bound constraint $\mathfrak{c}_2$ which corresponds to Z_2 (i.e. $Z_2 \models \mathfrak{c}_2$).

Delay. Now we have $Z_1 \uparrow = Z_2$, we want to find a $C_{\mathfrak{c}_2}$ such that $Z_1 \subseteq [\![C_{\mathfrak{c}_2}]\!]$ and $[\![C_{\mathfrak{c}_2}]\!] \uparrow \models \mathfrak{c}_2$. Observe that $\mathfrak{c}_2$ must be in one of the three cases: $x - 0 \prec c$, $0 - x \prec c$, and $x - y \prec c$, where $x, y \in X$ and $c \in \mathbb{N}$. Since $Z_1 \uparrow = Z_2$, $\mathfrak{c}_2$ can not be of the form $x - 0 \prec c$, and for the other two cases we have $Z_1 \models \mathfrak{c}_2$ and $[\![\{\mathfrak{c}_2\}]\!] \uparrow \models \mathfrak{c}_2$. So we can just choose $C_{\mathfrak{c}_2}$ to be $\{\mathfrak{c}_2\}$. Intuitively, for time delay, we just copy $\mathfrak{c}_2$ from the successor to the predecessor.

Reset. For a set r of clocks such that $Z_1[r] = Z_2$, we want to find a $C_{\mathfrak{c}_2}$ such that $Z_1 \subseteq [\![C_{\mathfrak{c}_2}]\!]$ and $[\![C_{\mathfrak{c}_2}]\!][r] \models \mathfrak{c}_2$. Let $\mathfrak{c}_2$ be $x - y \prec c$, where $x \in X \cup \{0\}$, $y \in X$ and $c \in \mathbb{N}$, we choose $C_{\mathfrak{c}_2}$ as $\{\mathfrak{c}_1\}$, where:

$$\mathfrak{c}_1 = \begin{cases} x - y \prec c, & \text{if } x, y \notin r \\ 0 - y \prec c, & \text{if } x \in r, y \notin r \\ x - 0 \prec c, & \text{if } x \notin r, y \in r \\ 0 - 0 \prec c, & \text{if } x, y \in r \end{cases} \tag{1}$$

The first and the last cases are trivial. Let's look at the second case. Since $x \in r$, according to the definition of reset operation, it must be the case that $Z_2 \models x = 0$. According to the assumption, $Z_2 \models \mathfrak{c}_2$. Note that $\mathfrak{c}_2$ is $x - y \prec c$. Combining the above two results we have $Z_2 \models 0 - y \prec c$. Since $y \notin r$, the value of y has not changed during the reset operation, thus we have $Z_1 \models 0 - y \prec c$. Conversely, we can check that $[\![\{0 - y \prec c\}]\!][r] \models x - y \prec c$. Thus $\{\mathfrak{c}_1\}$ is a qualified candidate for $C_{\mathfrak{c}_2}$. It is similar for the third case.

Here we ignore the case when $\mathfrak{c}_2$ is $x - 0 \prec c$ because, according to the time delay operation (which always follows the reset operation in the timed automata run), this is impossible.

Guard. For a guard g such that $Z_1 \wedge g = Z_2$, we want to find a $C_{\mathfrak{c}_2}$ such that $Z_1 \subseteq [\![C_{\mathfrak{c}_2}]\!]$ and $[\![C_{\mathfrak{c}_2}]\!] \wedge g \models \mathfrak{c}_2$. Notice that $Z_1 \wedge g \models \mathfrak{c}_2$ iff $Z_1 \wedge (g \wedge \neg\mathfrak{c}_2) = \emptyset$, similarly, $[\![C_{\mathfrak{c}_2}]\!] \wedge g \models \mathfrak{c}_2$ iff $[\![C_{\mathfrak{c}_2}]\!] \wedge (g \wedge \neg\mathfrak{c}_2) = \emptyset$. Like $\mathfrak{c}_2$, its negation $\neg\mathfrak{c}_2$ is also a difference bound constraint, so $g \wedge \neg\mathfrak{c}_2$ is a conjunction of difference bound constraints, and we can use FINDCONTRA to compute the set $C_{\mathfrak{c}_2}$.

Algorithm 2

```
1: function FINDCONTRA($Z$, $\varphi$)
2:     Find a negative cycle using Floyd-Warshall algorithm on the merged graph of $Z$ and $\varphi$
3:     Take the sequence of constraints corresponding to the negative cycle in $Z \wedge \varphi$: $\mathfrak{c}_0, \mathfrak{c}_1, \ldots \mathfrak{c}_{m-1}$.
4:     return $\{\mathfrak{c}_i | \mathfrak{c}_i$ is from $Z\}$
5: function DISABLED($l$, $Z$)
6:     $C \leftarrow \emptyset$
7:     for all $t$ disabled at $(l, Z)$ do
8:         $C \leftarrow C \cup$ FINDCONTRA$(Z, t.g)$
9:     return C
10: function BACKPROP($v$, $v'$, $C'$)
11:     Given $v \Rightarrow_t v'$
12:     $C \leftarrow \emptyset$
13:     for all $\mathfrak{c}_2 \in C'$ do
14:         Compute $\mathfrak{c}_1$ according to (1)
15:         $C \leftarrow C \cup$ FINDCONTRA$(v.Z, g \wedge \neg \mathfrak{c}_2)$
16:     return C
```

3.4 Termination of the ASG-Constructing Algorithm

In order to ensure that the ASG-constructing algorithm terminates, we make slight modifications on Definition 7 and on Algorithm 1. The condition **G2** of Definition 7 is modified to:

G2' If a node (l, Z, C) is not tentative, then for every transition $(l, Z) \Rightarrow_t^E (l', Z')$ s.t. $Z' \neq \emptyset$, there is a successor (l', Z', C') for this node.

The symbolic transition relation $\Rightarrow$ is replaced with $\Rightarrow^E$, which means that the operator $Extra_{LU}^+$ is used when computing the successor nodes. Accordingly, in Line 19 of Algorithm 1, $(v.l, v.Z) \Rightarrow (v'.q, v'.Z)$ should be changed to $(v.l, v.Z) \Rightarrow^E (v'.q, v'.Z)$.

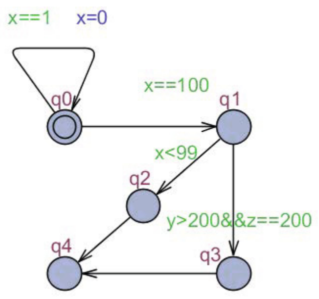
Now our algorithm can be seen as a further reduction made on top of $Extra_{LU}^+$-based search.

Theorem 2. *For an arbitrary timed automaton $\mathcal{A}$, the modified version of Algorithm 1 as described above will terminate.*

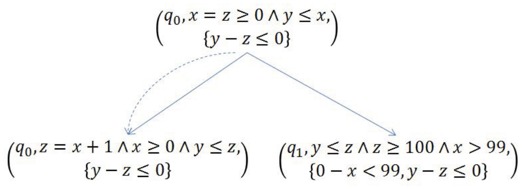
Proof. Assume, to the contrary, that the algorithm does not terminate. There must be an infinite sequence of explored nodes $v_1, v_2, \ldots$ (listed in the order of exploration) such that $v_1.l = v_2.l = \cdots$. Since the symbolic state space with $Extra_{LU}^+$ abstraction is finite, there must be two nodes v_i, v_j (with $i < j$) in the sequence such that $v_i.Z = v_j.Z$. From Definition 6 we know that $v_j.Z = v_i.Z \subseteq [\![v_i.C]\!]$, so v_j will never be explored, which contradicts the assumption.

4 An Example

Here we illustrate how our method works on the example timed automaton $\mathcal{A}_1$ shown in Figure 2a, where q_4 is the accepting location. Following [5], instead of considering one global LU-bound, we associate a local LU-bound to each location using static guard analysis. The LU-bound at each location is given in Figure 2c.



(a)


$(q_0, x = z \geq 0 \land y \leq x, \{y - z \leq 0\})$
$(q_0, z = x + 1 \land x \geq 0 \land y \leq z, \{y - z \leq 0\})$
$(q_1, y \leq z \land z \geq 100 \land x > 99, \{0 - x < 99, y - z \leq 0\})$

(b)

Location	L_x	U_x	L_y	U_y	L_z	U_z
q_0	100	100	200	$-\infty$	200	200
q_1	$-\infty$	99	200	$-\infty$	200	200
q_2	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
q_3	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
q_4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

(c)

Fig. 2. (2a): a timed automaton $\mathcal{A}_1$, (2b): the ASG of $\mathcal{A}_1$. Solid arrows represent the transition relation, while doted arrows represent the cover relation. (2c): the LU-bounds of $\mathcal{A}_1$

Using zone-based search, the following symbolic configurations will be generated: $(q_0, x = z \geq 0 \land y \leq z), (q_0, z = x + 1 \land y \leq z \land x \geq 0), (q_0, z = x + 2 \land y \leq z \land x \geq 0), \ldots, (q_0, z = x + 100 \land y \leq z \land x \geq 0), \ldots$, which is more than 100 nodes. When using our method, the resulting ASG has only 3 nodes, as shown in Figure 2b. In this example our method successfully ignores many of the constraints that are irrelevant to the reachability problem, achieving a huge reduction.

The algorithm in [13] can not avoid generating too many nodes either. The reason is that, LU-bound based abstractions can not find that the difference bound constraints on $z - x$ and $y - x$ are irrelevant. In our abstraction scheme, we consider more information than just LU-bounds, so our method can identify these constraints to be irrelevant.

5 Experiments

We have implemented UPPAAL's search algorithm, the algorithm in [13], and our algorithm. Similar to [14], an improvement is made on Algorithm 1 in the implementation: for each node v, if there is a node v' such that $v.l = v'.l$ and $v.Z \subseteq v'.Z$, then v will be deleted, parents of v will be inherited by v', and the difference bound constraints in $v'.C$ will be propagated to the parents of v, and nodes that are covered by v (if there is any) will be marked non-tentative.

We performed experiments on several benchmarks. The results are shown in Table 1 and Table 2, which are the results for breadth-first search (bfs) and

Table 1. Experiment results of the three methods using bfs search. "**Search-Extra**$^+_{LU}$" stands for $Extra^+_{LU}$-based search similar to UPPAAL. "$\mathfrak{a}_{\preceq LU}$, **disabled**" is the algorithm in [13]. "**DBCA-Extra**$^+_{LU}$" is difference bound constraint abstraction combined with $Extra^+_{LU}$. "tnds" is the total number of nodes generated, "fnds" is the number of nodes finally left, and "tm(s)" is the running time (in seconds). "to" stands for time-out(200s).

Model	**Search-Extra**$^+_{LU}$			$\mathfrak{a}_{\preceq LU}$, **disabled**			**DBCA-Extra**$^+_{LU}$		
	tnds	fnds	tm(s)	tnds	fnds	tm(s)	tnds	fnds	tm(s)
$\mathcal{A}_1$	405	204	0.03	405	203	0.14	3	3	0.00
D''_7	12869	12869	6.77	113	113	0.00	113	113	0.01
D''_8	48619	48619	100.08	145	145	0.01	145	145	0.01
D''_{70}			to	9941	9941	4.35	9941	9941	90.19
CSMA/CD 9	99288	45836	3.13	78552	35084	7.33	78552	35084	4.38
CSMA/CD 10	258249	120845	8.44	200649	90125	12.15	200649	90125	14.91
CSMA/CD 11	656312	311310	25.30	501432	226830	65.16	501432	226830	38.27
CSMA/CD 12	1636261	786447	68.81	1230757	561167	118.12	1230757	561167	98.58
FDDI 12	52555	727	13.72	422	341	0.56	176	154	0.13
FDDI 30			to	2923	2227	26.84	464	406	2.29
Fischer 8	132593	25080	2.61	132593	25080	8.44	132593	25080	7.55
Fischer 9	487459	81035	11.24	487459	81035	30.82	487459	81035	22.35
Critical 4	434421	53937	6.83	499441	53697	31.58	548781	54180	23.59
Lynch 4	46432	12700	1.19	46432	12700	1.26	46432	12700	1.05

Table 2. Experiment results of the three methods using dfs search. All settings are the same as in Table 1, except that dfs search is performed here

Model	**Search-Extra**$^+_{LU}$			$\mathfrak{a}_{\preceq LU}$, **disabled**			**DBCA-Extra**$^+_{LU}$		
	tnds	fnds	tm(s)	tnds	fnds	tm(s)	tnds	fnds	tm(s)
$\mathcal{A}_1$	405	204	0.02	405	203	0.23	3	3	0.00
D''_7	12869	12869	6.61	113	113	0.01	113	113	0.01
D''_8	48619	48619	110.15	145	145	0.01	145	145	0.01
D''_{70}			to	9941	9941	4.68	9941	9941	55.30
CSMA/CD 9	246072	45836	9.43	136813	36901	18.63	129718	35415	11.58
CSMA/CD 10	822699	120845	37.98	452788	98731	96.94	362407	90769	35.57
CSMA/CD 11	2758945	311310	150.23			to	997243	228054	120.85
FDDI 12	1016	727	0.27	96	96	0.02	96	96	0.03
FDDI 30	6308	4507	5.73	240	240	0.26	240	240	0.41
FDDI 50	17508	12507	53.21	400	400	0.72	400	400	1.35
Fischer 8	218017	25080	4.37	196738	25080	14.00	156634	25080	7.84
Fischer 9	1058685	81035	25.80	906766	81035	86.21	642739	81035	44.13
Critical 4	1067979	54469	15.19	1025269	53731	66.45	1009995	54488	38.25
Lynch 4	84421	12700	1.21	83171	12700	3.81	83967	12700	3.25

depth-first search (dfs), respectively. D'' is from [13], Fischer comes from the demos in the UPPAAL tool. The other models are from [1]. In our experiment settings, no location is set to be accepting, thus forcing the algorithms to perform exhaustive state space exploration. The programs are run on a VMware virtual machine with Ubuntu 10.04 operating system, which is allocated 2GB of memory. The underlying PC has an Intel Core i7 CPU of 2.93GHz and 3GB of RAM.

The results on the model $\mathcal{A}_1$ show the advantage of our method over LU-bound based abstractions. This is the case when LU-bound information is not sufficient to identify irrelevant constraints. On the other models, our method behaves similarly as $\mathfrak{a}_{\preceq LU}$, **disabled**. This is quite reasonable, since we use the same lazy search framework.

In many cases **DBCA-Extra**$^+_{LU}$ generates less nodes, but costs more time than **Search-Extra**$^+_{LU}$. This is due to the overhead of the effort to maintain the invariants of ASGs The situation is similar for $\mathfrak{a}_{\preceq LU}$, **disabled**. However, for some of the models we can see that the state space reduction of our method over **Search-Extra**$^+_{LU}$ is large enough to cover the overhead.

6 Conclusion

In this paper we proposed a difference bound constraint abstraction on zones for timed automata reachability checking. Difference bound constraint sets are used as abstractions in a lazy search algorithm, and $Extra^+_{LU}$ is used to ensure termination. Experiments show that in some of the cases the new abstraction scheme reduces the state spaces. A future work would be to perform experiments on other models to further investigate the performance.

Our abstraction is not necessarily coarser than [13], because non-convex abstractions [12] are used in [13], while difference bound constraint abstraction is in fact conjunctions of constraints, which is convex. However, our abstraction scheme makes it possible to make use of more information than just LU-bounds, achieving state space reduction in some cases.

The difference bound constraint abstraction is used in a forward lazy search scheme, and $Extra^+_{LU}$ is used to ensure termination. In fact, backward zone-based search is also possible [18], and does not need Max-bound abstraction or LU-abstraction to ensure termination. A possible future work is to explore the possibility to perform lazy search using difference bound constraint abstraction in a backward manner.

Acknowledgements. We thank the anonymous reviewers for their helpful comments on the earlier version of this paper.

References

1. Benchmarks of Timed Automata, http://www.comp.nus.edu.sg/~pat/bddlib/timedexp.html
2. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)

3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
4. Asarin, E., Bozga, M., Kerbrat, A., Maler, O., Pnueli, A., Rasse, A.: Data-structures for the verification of timed automata. In: Maler, O. (ed.) HART 1997. LNCS, vol. 1201, pp. 346–360. Springer, Heidelberg (1997)
5. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)
6. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 312–326. Springer, Heidelberg (2004)
7. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
8. Beyer, D.: Improvements in BDD-based reachability analysis of timed automata. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 318–343. Springer, Heidelberg (2001)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
10. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic abstraction refinement for timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer, Heidelberg (2007)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM (2002)
12. Herbreteau, F., Kini, D., Srivathsan, B., Walukiewicz, I.: Using non-convex approximations for efficient analysis of timed automata. In: FSTTCS 2011, pp. 78–89 (2011)
13. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 990–1005. Springer, Heidelberg (2013)
14. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. CoRR abs/1301.3127 (2013)
15. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
16. Sorea, M.: Lazy approximation for dense real-time systems. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT 2004. LNCS, vol. 3253, pp. 363–378. Springer, Heidelberg (2004)
17. Wozna, B., Zbrzezny, A., Penczek, W.: Checking reachability properties for timed automata via sat. Fundam. Inf. 55(2), 223–241 (2003)
18. Yovine, S.: Model checking timed automata. In: Rozenberg, G., Vaandrager, F.W. (eds.) Lectures on Embedded Systems. LNCS, vol. 1494, pp. 114–152. Springer, Heidelberg (1998)