

# Difference Decision Diagrams\*

Jesper Møller   Jakob Lichtenberg   Henrik Reif Andersen   Henrik Hulgaard

Department of Information Technology, Building 344

Technical University of Denmark

DK-2800 Lyngby, Denmark

E-mail: {jmr, jali, hra, henrik}@it.dtu.dk

Tel./Fax: +45 4525 3721/+45 4593 0074

## Abstract

*This paper describes a new data structure, difference decision diagrams (DDD), for representing a Boolean logic over inequalities of the form  $x - y < c$  and  $x - y \leq c$  where the variables are integer or real-valued. We give algorithms for manipulating DDDs and for determining functional properties (tautology, satisfiability, and equivalence). DDDs enable an efficient verification of timed systems modeled as, for example, timed automata or timed Petri nets, since both the states and their associated timing information can be represented symbolically, similar to how ROBDDs represent Boolean predicates. We demonstrate the efficiency of DDDs by computing the set of reachable states for a non-trivial timed system and compare the results with the state-of-the-art tools KRONOS and UPPAAL.*

**Keywords:** verification, real-time systems, symbolic model checking, decision diagrams, difference constraints.

## 1. Introduction

Model checking [14] is used extensively today for formal verification of finite state systems such as digital circuits and embedded software. The success of the technique is primarily due to the use of ROBDDs [10] for representing sets of and relations over Boolean variables *symbolically*, making it possible to verify systems with a very large number of states. However, if the model contains non-Boolean (e.g., real-valued) variables, ROBDDs and other symbolic representations of Boolean predicates are inefficient. As a consequence, state-of-the-art techniques for analyzing systems with time, modeled for example as timed automata, are only capable of analyzing systems with a handful of timers and a few thousand states.

In this paper we consider a Boolean logic extended with difference constraints, i.e., inequalities of the form  $x - y < c$  and  $x - y \leq c$ , where  $x$  and  $y$  are integer or real-valued variables and  $c$  is a constant. Difference constraints arise naturally when analyzing systems with time, expressing relations between the timers in the model, e.g., that the difference between two timers is within some bound. We call the Boolean logic over difference constraints for *difference constraint expressions* given by the following grammar:

$$\phi ::= x - y \leq c \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi, \quad (1)$$

where  $x, y \in \mathbf{Var}$  denote variables and  $c \in \mathbb{D}$  denotes a constant. We will allow the usual derived operators such as  $x - y > c$ ,  $\phi_1 \vee \phi_2$ , and  $\forall x.\phi$ . In this paper, the domain  $\mathbb{D}$  of the logic is either the real numbers  $\mathbb{R}$  or the integers  $\mathbb{Z}$ .

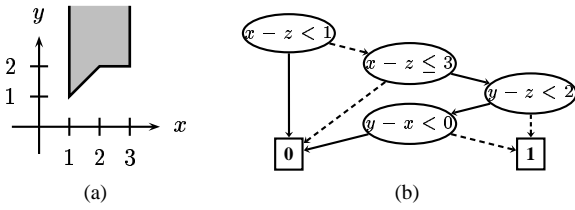
The main contribution of this paper is a data structure, called *difference decision diagrams* (DDD), for representing difference constraint expressions *symbolically*, making it possible to represent the state space of timed systems (and other systems with non-Boolean variables) efficiently. DDDs represent difference constraint expressions using a decision tree in a manner similar to the ROBDD representation of a Boolean expression. Consider the following expression  $\phi$  over  $x, y, z \in \mathbb{R}$ :

$$\phi = 1 \leq x - z \leq 3 \wedge (y - z \geq 2 \vee y - x \geq 0). \quad (2)$$

Figure 1 shows  $\phi$  as an  $(x, y)$ -plot for  $z = 0$  and the corresponding DDD. Each non-terminal vertex in a DDD contains a test expression  $\alpha$  (a difference constraint) and has two outgoing edges called the high- and low-branch which are drawn with solid and dashed lines, respectively. The high-branch is followed when  $\alpha$  evaluates to true; the low-branch when  $\alpha$  evaluates to false.

Observe that the logic in (1) subsumes Boolean logic since a Boolean variable  $b_i$  for instance can be encoded as a constraint  $x_i - x'_i \leq 0$  using two variables  $x_i, x'_i \in \mathbf{Var}$  (we use T and F to denote the two Boolean constants). This

\*Supported by a grant from the Danish Technical Research Council



**Figure 1.** The expression  $\phi$  in (2) as (a) an  $(x, y)$ -plot for  $z = 0$ , and (b) a difference decision diagram.

allows us to represent both the discrete (Boolean) states of a timed system *and* the associated timing information in the same data structure.

### 1.1. Related Work

One approach to analyze systems with time or other continuous variables is to make the dense domains discrete. For example, in a timed model it is assumed that the clocks only can take integer or rational values. Such a discretization makes it possible to use ROBDDs for representing both the state graph and the associated timing information [2, 9, 11, 12]. However, this way of representing dense domains is often inefficient; the ROBDD representation is very sensitive to the granularity of the discretization and to the size of the delay ranges. Another approach based on ROBDDs is to have a Boolean variable representing each constraint, and use an external decision procedure to determine implications among these variables [13]. These implications are used to prune the representation of the state space. The advantage is that any kind of decidable constraints can be used. Our approach can be seen as a simplified version of this where we take advantage of restricting the types of constraints to difference constraints and perform reductions on-the-fly.

Several algorithms for analyzing timed automata have been developed. The unit-cube approach [1] models time as dense but represents the timing information using a finite number of equivalence classes. Again, the number of timed states is dependent on the size of the delay ranges and easily becomes unmanageable. Several recent timing analysis methods use difference bound matrices (DBMs) [19] for representing the timing information [8, 20, 24, 29] (see [30] for an extensive description of the different approaches to model checking of timed automata.) In these approaches, a set of DBMs representing the possible timer configurations is associated with each discrete state of the system. Although DBMs provide a compact representation of a clock configuration, there are several serious problems with the approaches based on DBMs: first, the number of DBMs for representing the timing information associated with a given state can become very large, secondly, there is no sharing or reuse of DBMs among the different discrete states, and finally, each discrete state is represented explicitly, thus

these approaches are limited by the number of reachable states of the system. Several researchers have attempted to remedy these shortcomings, for example by using partial order methods [6, 25, 27] or by using approximate methods [3, 5, 28]. Although these approaches do address the first problem, they are still susceptible to the last two problems since each state is represented explicitly. Using DDDs it is possible to combat all three problems since first, unlike DBMs, DDDs are not limited to representing the timing information as a union of convex sets, secondly, DDDs represent all states and the associated timing information in a single shared data structure, and finally, states and the timing information are represented symbolically using difference constraint expressions.

Another approach [26] suggests using a partition refinement algorithm for efficient model checking. However, the reported running times are still exponential.

Based on the initial ideas of this paper, Behrmann et al. [4] have implemented a minor variation of DDDs allowing a fanout of more than two (which they call CDDs). They have shown a significant improvement in memory consumption in UPPAAL, even though the experiments in contrast to ours do not use a fully symbolic approach (the discrete states are enumerated explicitly). Thus, this approach will not be able to handle the larger instances of the timed system described in Section 6.2.

## 2. Difference Decision Diagrams

The data structure *difference decision diagrams* (DDD) is developed to efficiently represent and manipulate difference constraint expressions. Difference decision diagrams share many properties with binary decision diagrams (BDDs): they can be ordered, they can be reduced making it possible to check for tautology and satisfiability in constant time, and many of the algorithms and techniques for BDDs can be modified to apply to DDDs.

**Definition 1 (Difference Decision Diagram)** A *difference decision diagram* (DDD) is a directed acyclic graph  $(V, E)$ . The vertex set  $V$  contains two terminals  $\mathbf{0}$  and  $\mathbf{1}$  with out-degree zero, and a set of non-terminal vertices with out-degree two and the following attributes:

Attribute	Type	Description
$pos(v)$	<b>Var</b>	Positive variable $x_i$ .
$neg(v)$	<b>Var</b>	Negative variable $x_j$ .
$op(v)$	{LE, LEQ}	Operator $<$ or $\leq$ .
$const(v)$	$\mathbb{D}$	Constant $c$ .
$high(v)$	$V$	High-branch $h$ .
$low(v)$	$V$	Low-branch $l$ .

The set  $E$  contains the edges  $(v, low(v))$  and  $(v, high(v))$ , where  $v \in V$  is a non-terminal vertex.  $\square$

Similar to ROBDDs, the non-terminal vertices of a DDD corresponds to the if-then-else operator  $\alpha \rightarrow \phi_1, \phi_0$  defined by

$$\alpha \rightarrow \phi_1, \phi_0 = (\alpha \wedge \phi_1) \vee (\neg \alpha \wedge \phi_0), \quad (3)$$

where  $\alpha$  is a test expression and  $\phi_0, \phi_1$  are difference constraint expressions. However, unlike ROBDDs, the test expression  $\alpha$  is not a Boolean variable, but a difference constraint of the form  $x - y \lesssim c$ , where the symbol  $\lesssim$  represents either  $<$  or  $\leq$ .

Each vertex  $v$  in a DDD denotes a difference constraint expression  $\phi^v$ . If  $v$  is a terminal vertex, i.e., either  $\mathbf{0}$  or  $\mathbf{1}$ ,  $\phi^v$  is  $\mathbf{F}$  or  $\mathbf{T}$ , respectively. Otherwise,  $v$  represents the expression  $\phi^v$  given by:

$$\phi^v = x_i - x_j \lesssim c \rightarrow \phi^{high(v)}, \phi^{low(v)}, \quad (4)$$

where  $x_i = pos(v)$ ,  $x_j = neg(v)$ ,  $\lesssim = op(v)$ , and  $c = const(v)$ . We use the following notational shorthands:

$$\begin{aligned} var(v) &\stackrel{\text{def}}{=} (pos(v), neg(v)) \\ bound(v) &\stackrel{\text{def}}{=} (const(v), op(v)) \\ cstr(v) &\stackrel{\text{def}}{=} (var(v), bound(v)) \\ attr(v) &\stackrel{\text{def}}{=} (cstr(v), high(v), low(v)). \end{aligned}$$

Adding two bounds  $(c_1, o_1)$  and  $(c_2, o_2)$  gives  $(c_1 + c_2, o_1 + o_2)$ , where  $o_1 + o_2$  is LEQ if both  $o_1$  and  $o_2$  are LEQ and LE otherwise. Negating a bound  $(c, o)$  gives  $(-c, \neg o)$ , where  $\neg$ LE is LEQ and  $\neg$ LEQ is LE.

We use  $v \rightsquigarrow u$  to denote that the vertex  $u$  is reachable from  $v$  (i.e., there is a path from  $v$  to  $u$ ). The size of a DDD  $v$ , denoted  $|v|$ , is the number of vertices reachable from  $v$ ; that is,  $|v| = |\{u \in V : v \rightsquigarrow u\}|$ .

## 2.1. Ordering

To define ordered DDDs, we assume given a total ordering  $<$  of the variables  $x_1, \dots, x_n$ , which furthermore must totally order pairs of variables  $(x_i, x_j)$ .<sup>1</sup> We extend this ordering to attributes  $cstr(v)$  of vertices  $v$  in a DDD. Constants,  $const(v)$ , are ordered as usually in  $\mathbb{D}$ , and the two operators,  $op(v)$ , are ordered as  $\text{LE} < \text{LEQ}$ . Bounds,  $(const(v), op(v))$  and constraints,  $(var(v), bound(v))$ , are ordered lexicographically. For example,

$$\begin{aligned} ((x_2, x_1), (0, \text{LE})) &< ((x_2, x_1), (0, \text{LEQ})) < \\ &((x_2, x_1), (1, \text{LE})) < ((x_3, x_1), (0, \text{LE})). \end{aligned}$$

We assume that the two terminal vertices have attributes that are greater than all non-terminals.

<sup>1</sup>Pairs of variables can for example be ordered reversed lexicographically, that is  $(x_i, x_j) < (x'_i, x'_j)$  iff  $x_j < x'_j$  or  $(x_j = x'_j \wedge x_i < x'_i)$ .

**Definition 2 (Ordered DDD)** An *ordered DDD* (ODDD) is a DDD where each non-terminal vertex  $v$  satisfies:

1.  $neg(v) < pos(v)$ ,
2.  $var(v) < var(high(v))$ ,
3.  $var(v) < var(low(v))$  or  $(var(v) = var(low(v))$  and  $bound(v) < bound(low(v))$ ). □

Requirement 1 expresses that the pair of variables  $var(v) = (pos(v), neg(v)) = (x_i, x_j)$  of a vertex  $v$  is *normalized*; that is,  $x_j < x_i$ . This does not restrict what we can represent with DDDs, because the two variables in a vertex can always be swapped by negating the bound and swapping the low- and high-branches. We further require in an ordered DDD, that either the children of a vertex have variables later in the ordering (requirement 2 and first part of 3) or the variables along the low-branch are identical (second part of 3). The second part of requirement 3 makes it possible to have multiple tests on the same pair of variables, which is needed because of the disjunctive abilities of DDDs. The last two requirements imply  $cstr(v) < cstr(high(v))$  and  $cstr(v) < cstr(low(v))$ . The DDD in Figure 1 is an example of an ordered DDD with the ordering  $z < x < y$  extended reversed lexicographically to pairs of variables.

## 2.2. Locally Reduced DDDs

Similar to ROBDDs, we define a set of local reduction rules that reduce the size of the DDD representation.

**Definition 3 (Locally Reduced DDD)** A *locally reduced DDD* ( $R_L$ DDDD) is an ODDD satisfying, for all non-terminals  $u$  and  $v$ :

1.  $\mathbb{D} = \mathbb{Z}$  implies  $op(v) = \text{LEQ}$ ,
2.  $attr(u) = attr(v)$  implies  $u = v$ ,
3.  $low(v) \neq high(v)$ ,
4.  $var(v) = var(low(v))$  implies  $high(v) \neq high(low(v))$ . □

Requirement 2 and 3 are identical to the reduction requirements for ROBDDs. Thus, if we encode a Boolean variable  $b_i$  as  $x_i - x'_i \leq 0$ , any Boolean expression over  $b_1, b_2, \dots, b_n$  is represented in a canonical form using locally reduced DDDs. Requirement 4 ensures that any two consecutive vertices with the same pair of variables have different high-branches. This requirement is fulfilled using the following equivalence for ordered DDDs:

$$\begin{aligned} x - y \lesssim_1 c_1 \rightarrow h, (x - y \lesssim_2 c_2 \rightarrow h, l) &\Leftrightarrow \\ x - y \lesssim_2 c_2 \rightarrow h, l. & \end{aligned}$$

### 3. Construction of DDDs

In this section we present efficient algorithms for manipulating locally reduced DDDs. Orderedness ensures that the basic algorithm for computing the Boolean connectives is polynomial. However, for existential quantification the situation is different. Although the algorithm in polynomial time computes the modified and additional constraints, its worst-case running time is exponential since it needs to regain orderedness.

The algorithms are all based on a function MK for creating DDD vertices. The function MK normalizes the two variables and ensures that the created vertex is locally reduced:  $\text{MK}((x, y), (c, o), h, l)$  returns the identity of a vertex, equivalent to a vertex  $v$  with  $\text{var}(v) = (x, y)$ ,  $\text{bound}(v) = (c, o)$ ,  $\text{high}(v) = h$ , and  $\text{low}(v) = l$ . Using MK as the only function for constructing a DDD ensures that it is locally reduced. As for BDDs, MK can be implemented with an expected running time of  $O(1)$ . We use  $\text{MK}'(v, b)$  as a shorthand for  $\text{MK}(v, b, \mathbf{1}, \mathbf{0})$ , which returns a vertex corresponding to a single normalized difference constraint.

#### 3.1. Boolean Combination of DDDs

The function  $\text{APPLY}(op, u, v)$  is used to combine two DDDs rooted at  $u$  and  $v$  with a Boolean operator  $op$ . APPLY is a generalization of the version used for ROBDDs, which is based on the fact that any binary Boolean operator  $op$  distributes over the if-then-else operator:

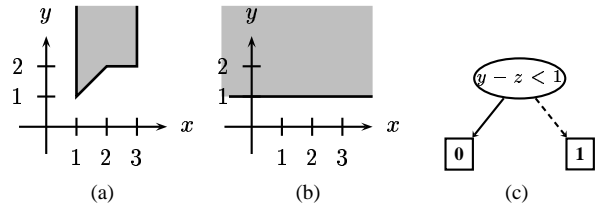
$$\begin{aligned} (\alpha \rightarrow h, l) \text{ op } (\alpha' \rightarrow h', l') &\Leftrightarrow \\ \alpha \rightarrow (h \text{ op } (\alpha' \rightarrow h', l')), (l \text{ op } (\alpha' \rightarrow h', l')) &. \end{aligned} \quad (5)$$

This equivalence provides a method to combine two DDDs with a Boolean operator. Reading the equivalence from left to right, we see that we can move the Boolean operator down one level in the DDD. If we continuously do so until both arguments of  $op$  are  $\mathbf{0}$  or  $\mathbf{1}$ , we can evaluate the expression and return the appropriate result.

If the two pairs of variables are equal, we can simplify (5):

$$\begin{aligned} (\alpha \rightarrow h, l) \text{ op } (\alpha' \rightarrow h', l') &\Leftrightarrow \\ \begin{cases} \alpha \rightarrow (h \text{ op } h'), (l \text{ op } (\alpha' \rightarrow h', l')) & \text{if } \alpha \prec \alpha', \\ \alpha \rightarrow (h \text{ op } h'), (l \text{ op } l') & \text{if } \alpha = \alpha', \\ \alpha' \rightarrow (h \text{ op } h'), ((\alpha \rightarrow h, l) \text{ op } l') & \text{if } \alpha \succ \alpha'. \end{cases} & \quad (6) \end{aligned}$$

Together, (5) and (6) can be used to give the algorithm APPLY: We use (5) when  $(x, y) \prec (x', y')$  or  $(x, y) \succ (x', y')$  and (6) when  $(x, y) = (x', y')$ . Using MK to construct new vertices and applying dynamic programming, the runtime of APPLY is the same as the ROBDD version, that



**Figure 2.** Existential quantification of  $x$  in (2). (a) An  $(x, y)$ -plot of  $\phi$  for  $z = 0$ . (b) An  $(x, y)$ -plot of  $\exists x.\phi$  for  $z = 0$ . (c) The DDD for  $\exists x.\phi$ .

is,  $O(|u||v|)$ . Negation of a DDD  $u$  can therefore be done in time  $O(|u|)$ . In the following we use  $u \text{ op } v$  as a shorthand for  $\text{APPLY}(op, u, v)$  when  $u$  and  $v$  are DDDs.

#### 3.2. Quantifications

Since the domain of the variables is infinite, quantification is more complicated than the binary Boolean connectives. Existential quantification of a variable  $x$  in a DDD rooted at  $u$  is performed by removing all vertices reachable from  $u$  containing  $x$ , but keeping all implicit constraints induced by  $x$  among the other variables. For example, quantifying out  $x$  in the expression  $\phi$  given in (2) yields  $\exists x.\phi = y - z \geq 1$ , see Figure 2. Here, the constraint  $y - z \geq 1$  does not occur explicitly in  $\phi$ , but implicitly because of  $y - x \geq 0$  and  $x - z \geq 1$ .

To compute  $\exists x.(x_i - x_j \lesssim c \rightarrow h, l)$ , we consider two cases: If  $x$  is different from both  $x_i$  and  $x_j$ , we can push down the quantifier one level in the DDD:

$$\begin{aligned} \exists x.(x_i - x_j \lesssim c \rightarrow h, l) &= \\ x_i - x_j \lesssim c \rightarrow \exists x.h, \exists x.l & \quad \text{if } x \notin \{x_i, x_j\}. \end{aligned} \quad (7)$$

If  $x$  is equal to  $x_i$  or  $x_j$ , we *relax* all paths in  $h$  and  $l$  with  $x_i - x_j \leq c$  and  $x_i - x_j > c$ , respectively, and combine the results with disjunction:

$$\begin{aligned} \exists x.(x_i - x_j \leq c \rightarrow h, l) &= \\ \exists x.\text{RELAX}(h, x, x_i - x_j \leq c) & \\ \vee \exists x.\text{RELAX}(l, x, x_j - x_i < -c) & \quad \text{if } x \in \{x_i, x_j\}. \end{aligned} \quad (8)$$

The case for  $\exists x.(x_i - x_j < c \rightarrow h, l)$  is analogous.

If  $x$  is equal to  $x_i$ , relaxation of a path  $p$  with a constraint  $x_i - x_j \lesssim c$  consists of adding a new constraint  $x'_i - x_j \lesssim c + c'$  to  $p$  for each constraint  $x'_i - x_i \lesssim c'$  in  $p^2$ . The case where  $x$  is equal to  $x_j$  is symmetric. These observations lead to the following algorithm:

$$\mathbf{1} \text{ RELAX}(u, x, x_i, x_j, b) =$$

<sup>2</sup>In terms of the constraint graph [15, p. 541] defined by  $p$ , relaxation with  $x_i - x_j \lesssim c$  corresponding to an edge from  $x_j$  to  $x_i$  creates a new edge from  $x_j$  to  $x'_i$  with weight  $c + c'$  for each edge from  $x_i$  to  $x'_i$  with weight  $c'$  (i.e., the edge from  $x_j$  to  $x'_i$  is now explicit, not implicit via  $x_i$ ).

```

2  if  $u \in \{0, 1\}$  then return  $u$ 
3  else
4     $h \leftarrow \text{RELAX}(\text{high}(u), x, x_i, x_j, b)$ 
5     $l \leftarrow \text{RELAX}(\text{low}(u), x, x_i, x_j, b)$ 
6     $\alpha \leftarrow \text{MK}'(\text{var}(u), \text{bound}(u))$ 
7    if  $\text{neg}(u) = x_i = x$  then
8       $h \leftarrow h \wedge \text{MK}'((\text{pos}(u), x_j), b + \text{bound}(u))$ 
9    elseif  $\text{pos}(u) = x_i = x$  then
10      $l \leftarrow l \wedge \text{MK}'((\text{neg}(u), x_j), b - \text{bound}(u))$ 
11    elseif  $\text{neg}(u) = x_j = x$  then
12      $l \leftarrow l \wedge \text{MK}'((x_i, \text{pos}(u)), b - \text{bound}(u))$ 
13    elseif  $\text{pos}(u) = x_j = x$  then
14      $h \leftarrow h \wedge \text{MK}'((x_i, \text{neg}(u)), b + \text{bound}(u))$ 
15    return  $(\alpha \wedge h) \vee (\neg\alpha \wedge l)$ 

```

We can use dynamic programming in both EXISTS and RELAX. In the worst case RELAX generates  $|u|^2$  new constraints in lines 8–14. A (pessimistic) bound on the number of added constraints is therefore  $O(|u|^3)$ . However, in order to maintain orderedness these new constraints cannot be added where they are discovered through calls to MK, but need to be added through calls to APPLY (lines 8–15). The repeated calls to APPLY imply that the running time of EXISTS is worst-case exponential.

### 3.3. Assignment

The operations of *assignment* and *replacement* are often used in verification. Although these can be implemented using existential quantification, we give more efficient algorithms for general replacement and a special frequent case of assignment.

After performing an *assignment*  $\phi[x \leftarrow y + c]$ , when  $x \neq y$ , the variable  $x$  is given the value of another variable  $y$  plus a constant  $c$  in the expression  $\phi$ . Performing an assignment corresponds to removing all explicit bounds on  $x$ , and then updating  $x$  with a new value. The assignment operation  $\text{ASSIGN}(u, x, y, c)$  can therefore be performed as:

$$\phi[x \leftarrow y + c] = (\exists x. \phi) \wedge (x - y = c). \quad (9)$$

If  $x$  is equal to  $y$ , an assignment corresponds to incrementing  $x$  by the value  $c$ . In these cases, the assignment can be performed without the existential quantification. This is done by the algorithm  $\text{INCREMENT}(u, x, c)$  for computing  $\phi[x \leftarrow x + c]$  with running time  $O(|u|)$ :

```

1   $\text{INCREMENT}(u, x, c) =$ 
2  if  $u \in \{0, 1\}$  then return  $u$ 
3  else
4     $h \leftarrow \text{INCREMENT}(\text{high}(u), x, c)$ 
5     $l \leftarrow \text{INCREMENT}(\text{low}(u), x, c)$ 
6    if  $\text{pos}(u) = x$  then
7       $\alpha \leftarrow \text{MK}(\text{var}(u), \text{bound}(u) + (c, \text{LEQ}), h, l)$ 
8    elseif  $\text{neg}(u) = x$  then

```

```

9       $\alpha \leftarrow \text{MK}(\text{var}(u), \text{bound}(u) - (c, \text{LE}), h, l)$ 
10   else
11      $\alpha \leftarrow \text{MK}(\text{var}(u), \text{bound}(u), h, l)$ 
12   return  $\alpha$ 

```

### 3.4. Replacement

The *replacement* operator  $\phi[y + c/x]$  syntactically substitutes all occurrences of  $x$  in  $\phi$  with another variable  $y$  plus a constant  $c$ . The replacement operator can be implemented using conjunction and existential quantification:

$$\phi[y + c/x] = \exists x. (\phi \wedge (x - y = c)). \quad (10)$$

However, we can avoid the existential quantification by performing the replacement directly:

```

1   $\text{REPLACE}(u, x, y, c) =$ 
2  if  $u \in \{0, 1\}$  then return  $u$ 
3  else
4     $h \leftarrow \text{REPLACE}(\text{high}(u), x, y, c)$ 
5     $l \leftarrow \text{REPLACE}(\text{low}(u), x, y, c)$ 
6    if  $\text{pos}(u) = x$  then
7       $\alpha \leftarrow \text{MK}'((y, \text{neg}(u)), \text{bound}(u) - (c, \text{LE}))$ 
8    elseif  $\text{neg}(u) = x$  then
9       $\alpha \leftarrow \text{MK}'((\text{pos}(u), y), \text{bound}(u) + (c, \text{LEQ}))$ 
10   else
11      $\alpha \leftarrow \text{MK}'(\text{var}(u), \text{bound}(u))$ 
12   return  $(\alpha \wedge h) \vee (\neg\alpha \wedge l)$ 

```

Again we can use dynamic programming and only a linear number of new constraints are constructed. However, REPLACE suffers from the same problem as EXISTS, namely that in order to maintain orderedness the worst-case running time is exponential. In practice, one often replaces a variable with another variable which is close in the ordering.

## 4. Path Reduced DDDs

The previous section gave algorithms for constructing locally reduced DDDs. However, locally reduced DDDs are not a canonical representation of difference constraint expressions. In this section we show how to remove some of the redundant constraints in a path, making the representation semi-canonical. In a semi-canonical representation, there is exactly one DDD for a tautology (the terminal **1**) and exactly one DDD for an unsatisfiable expression (the terminal **0**). Thus, with semi-canonical DDDs it is straightforward to test for tautology, satisfiability, and equivalence (after using APPLY with a biimplication).

### 4.1. Paths and Semi-canonical DDDs

A *path* in a DDD corresponds to a conjunction of difference constraints or negated difference constraints (whenever

er the path follows a low-branch). Since the negations always can be removed by swapping the variables, changing the comparison operator, and negating the constant, a path corresponds to a conjunction of difference constraints, also called a *system of difference constraints* [15, Sect. 25.5]. We denote the system of difference constraints induced by a path  $p$  by  $[p]$ . A path  $p$  is defined to be *feasible* if the corresponding system of difference constraints has a feasible solution. If the constraint system has no solution, the path is *infeasible*.

**Definition 4 (Path-reduced DDD)** A *path-reduced DDD* ( $R_p$ DDD) is a locally reduced DDD where all paths are feasible.  $\square$

Paths ending at the terminals  $\mathbf{0}$  and  $\mathbf{1}$  are called  $\mathbf{0}$ -paths and  $\mathbf{1}$ -paths, respectively. If a DDD has no infeasible  $\mathbf{0}$ -paths and  $\mathbf{1}$ -paths, then it has no infeasible paths because a feasible constraint system will still be feasible if we remove some of the difference constraints from it. So if all  $\mathbf{0}$ -paths and  $\mathbf{1}$ -paths in a DDD  $u$  are feasible, then  $u$  is path reduced.

For  $R_p$ DDDs it is straightforward to decide satisfiability and tautologyness:

**Theorem 1 ( $R_p$ DDDs are semi-canonical)** *In an  $R_p$ DDD, the terminal vertex  $\mathbf{1}$  is the only representation of a tautology and the terminal vertex  $\mathbf{0}$  is the only representation of an unsatisfiable expression.*

**Proof:** We show that if  $v$  is a non-terminal in a path reduced DDD, then  $v$  represents neither a tautology nor an unsatisfiable expression.

Because  $v$  is path reduced, it is also locally reduced, so all vertices  $u$  reachable from  $v$  satisfy  $low(u) \neq high(u)$ . Furthermore, because  $v$  is a non-terminal vertex in an (acyclic) ordered DDD, there exists some vertex  $u'$  reachable from  $v$  that has  $low(u') = \mathbf{0}$  and  $high(u') = \mathbf{1}$  or  $low(u') = \mathbf{1}$  and  $high(u') = \mathbf{0}$ . Consequently, both  $\mathbf{0}$  and  $\mathbf{1}$  are reachable from  $v$ . Let  $p$  be some  $\mathbf{0}$ -path from  $v$ . Per definition of path reducedness, we know that  $p$  is feasible. This implies that there exists a variable assignment satisfying  $[p]$ , meaning that there exists a falsifying variable assignment for  $v$ . Thus,  $v$  cannot represent a tautology. Similarly, because there is a feasible  $\mathbf{1}$ -path from  $v$ ,  $v$  is satisfiable.  $\blacksquare$

Theorem 1 also shows how to check equivalence between two  $R_p$ DDDs: Use APPLY with the biimplication operator and observe whether the result is the tautology  $\mathbf{1}$ .

## 4.2. Reduce

An algorithm for making a DDD rooted at  $u$  path reduced is:

```
1 PATHREDUCE( $u$ ) = REDUCE( $u$ ,  $\langle \rangle$ )
```

```
2 where REDUCE( $v$ ,  $p$ ) =
3   if  $\neg$ FEASIBLE( $p$ ) then return  $\perp$ 
4   elseif  $v \in \{\mathbf{0}, \mathbf{1}\}$  then return  $v$ 
5   else
6      $h \leftarrow$  REDUCE( $high(v)$ ,  $p \hat{\ } (v, high(v))$ )
7      $l \leftarrow$  REDUCE( $low(v)$ ,  $p \hat{\ } (v, low(v))$ )
8     if  $l \neq \perp \wedge h \neq \perp$  then
9       return MK( $var(v)$ ,  $bound(v)$ ,  $h$ ,  $l$ )
10    elseif  $h \neq \perp$  then return  $h$ 
11    else return  $l$ 
```

The operator  $\hat{\ }$  denotes path concatenation and  $\langle \rangle$  denotes the empty path. The function FEASIBLE( $p$ ) determines whether a system of difference constraints corresponding to the path  $p$  is feasible.

The function REDUCE( $v$ ,  $p$ ) returns  $\perp$  if and only if the path  $p$  is infeasible. Clearly, if  $p$  is infeasible, REDUCE( $v$ ,  $p$ ) returns  $\perp$  in line 3. On the other hand, if  $p$  is feasible, it is simple to see that either  $p \hat{\ } (v, high(v))$  or  $p \hat{\ } (v, low(v))$  is feasible, and thus REDUCE( $v$ ,  $p$ ) cannot return  $\perp$  in line 11. Hence, REDUCE( $v$ ,  $p$ ) =  $\perp$  if and only if  $p$  is infeasible.

The correctness of PATHREDUCE then follows from the following observation: if either  $h = \perp$  or  $l = \perp$  in lines 6 and 7, the vertex  $v$  can be removed. To see this, let  $[p]$  denote the system of difference constraints corresponding to the path  $p$  and let  $\alpha = cstr(v)$  denote the difference constraint of vertex  $v$ . Assume  $l = \perp$ , i.e., the path  $p \hat{\ } (v, low(v))$  is infeasible, and thus  $[p] \wedge \neg \alpha = \mathbf{false}$ . Then,

$$[p] \wedge \alpha = ([p] \wedge \alpha) \vee ([p] \wedge \neg \alpha) = [p] \wedge (\alpha \vee \neg \alpha) = [p].$$

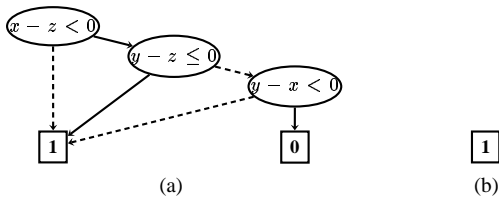
It follows from a symmetric argument that the vertex  $v$  can be removed if  $h = \perp$ .

Let us consider a small example. Figure 3 shows a locally reduced DDD for the expression

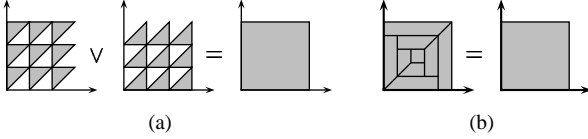
$$\phi = x - z \geq 0 \vee y - z \leq 0 \vee y - x \geq 0. \quad (11)$$

The DDD has three  $\mathbf{1}$ -paths and one  $\mathbf{0}$ -path. The  $\mathbf{0}$ -path corresponds to the system of difference constraints  $x - z < 0$ ,  $z - y < 0$ , and  $y - x < 0$ , which has no feasible solution (i.e., the  $\mathbf{0}$ -path is infeasible). Thus, if we call PATHREDUCE on the root vertex, the REDUCE-call on the vertex containing  $y - x < 0$  returns  $\mathbf{1}$ , and because of the third local reduction requirement, the result is the terminal  $\mathbf{1}$ .

There are several algorithms for determining whether a system of difference constraints is feasible. Two well-known algorithms are Floyd-Warshall's algorithm and Bellman-Ford's algorithm [15], which both have worst-case running times  $O(n^3)$ , where  $n$  is the number of variables. PATHREDUCE( $u$ ) enumerates all paths in  $u$ , and because the number of paths can be exponential in the size of  $u$ , the complexity of PATHREDUCE( $u$ ) is  $O(2^{|u|} n^3)$ .



**Figure 3.** The expression  $\phi$  from (11) as (a) a locally reduced DDD, and (b) a path reduced DDD.



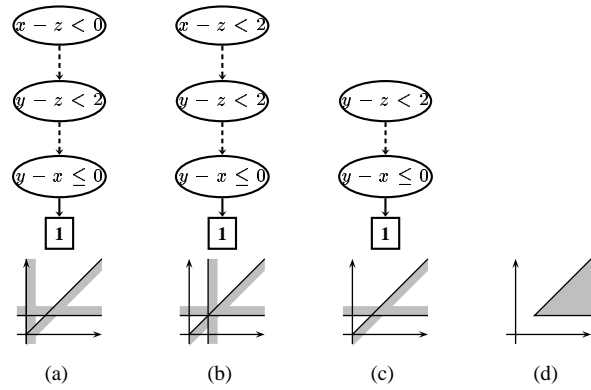
**Figure 4.** Disjunctions of complex sets can reduce to simple  $R_p$ DDD.

PATHREDUCE can be improved by using a faster algorithm to determine feasibility of a path, and by reusing the result of the feasibility check in the two recursive calls. These optimizations can be realized by an *incremental* version of the Bellman-Ford algorithm, but although these optimizations in practice improve the performance of PATHREDUCE, they do not improve on the worst-case runtime.

As shown in Theorem 1, it is straightforward to determine whether a path reduced DDD represents a tautology and whether it is satisfiable. However, in practice it is often more efficient to search for a counterexample when checking for tautology or satisfiability. For instance, when checking for tautology, the PATHREDUCE algorithm can be modified to stop (and report **false**) if a feasible **0**-path is found. Similarly, when checking for satisfiability, the algorithm can stop (and report **true**) if a feasible **1**-path is found. This approach also leads to a practical algorithm for finding a satisfying variable assignment, called ANYSAT. The algorithm searches for a feasible **1**-path and if one is found, the corresponding system of difference constraints is solved, yielding a satisfying assignment.

### 5. Fully Reduced DDDs

The reductions ensuring local and path reducedness are quite powerful. As an example consider the two sets built from nine triangles as shown in Figure 4(a). They each contain nine convex regions representable by 15 non-terminal DDD vertices using the ordering  $(x, z) \prec (y, z) \prec (y, x)$ . Computing the disjunction of the two sets using APPLY results in the  $3 \times 3$ -square represented with only four non-terminal vertices in an  $R_p$ DDD. As another example consider the nine sets shown in Figure 4(b). Combined they yield a simple convex square although no two sets together form a convex region. Using difference bound matrices similar powerful reductions are very expensive to obtain.



**Figure 5.** Three  $R_p$ DDD representing the same set.

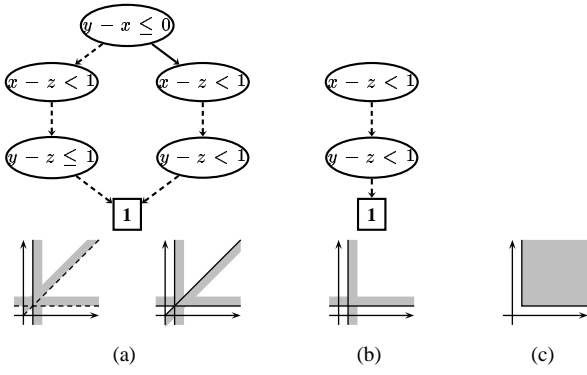
However, path-reducedness is not enough to ensure a canonical representation. As an example, consider the three path-reduced DDDs of Figure 5 which all represent the same triangular area (shown in Figure 5(d)). Local and path reductions are too weak to identify them. One problem (shown in Figure 5(a)) is that the constraints may contain a certain amount of slack. For instance, the constraint  $x - z \geq 0$  could be tightened to  $x - z \geq 2$  without changing the semantics. To avoid this kind of slack we introduce a notion of a path being *tight*, which strengthens the notion of path reducedness.

To introduce tightness we need to distinguish the dominating constraints in a path. Formally, a constraint  $x_i - x_j \lesssim c$  is *dominating* in a path  $p$  if all other constraints  $x_i - x_j \lesssim c'$  on the same pair of variables in  $p$ , are less restrictive, i.e.,  $(c, \lesssim) < (c', \lesssim')$ . Non-dominating constraints occur only in paths that through low-edges pass through several vertices with constraints on the same pair of variables.

**Definition 5 (Tightness)** A dominating constraint  $\alpha = x_i - x_j \lesssim c$  is *tight* in a feasible path  $[p] = [p_1] \wedge \alpha \wedge [p_2]$  if for all tighter constraints  $(c', \lesssim') < (c, \lesssim)$ , the systems  $[p_1] \wedge (x_i - x_j \lesssim' c') \wedge [p_2]$  and  $[p]$  have different solutions. A path  $p$  is *tight* if it is feasible and all dominating constraints on it are tight. An  $R_L$ DDD  $u$  is *tight* if all paths from  $u$  are tight.  $\square$

From the definition it is clear that tightness generalizes path reducedness since any tight DDD is also an  $R_p$ DDD. Hence, Theorem 1 implies that it is trivial to determine satisfiability and tautologiness of tight DDDs.

Adding tightness as a condition prevents the existence of the DDD in Figure 5(a). A DDD can be made tight by enumerating all paths, for each path solve the associated all-pairs shortest path problem, replacing the bounds of the constraints by the bounds from the solution, and finally combine all the tight paths by disjunction using APPLY. Hence, the DDD (a) will get reduced to the DDD (b).



**Figure 6.** An example where the mergeability test is necessary to merge two paths into one, making the top constraint redundant.

Tight DDDs are still not canonical due to implicit constraints that arise as consequences of the constraints in the vertices. The solution set will not depend on how many of these implicit constraints are made explicit but the resulting DDDs will be different. To remove this arbitrariness, vertices will be added to the DDD:

**Definition 6 (Saturation)** A tight path  $p$  from an  $R_P$ DDD is *saturated* if for all constraints  $\alpha$  not on  $p$ , if  $\alpha$  is added to  $p$  either (1)  $\alpha$  is not dominating and tight, or (2) the constraint system  $[p_1] \wedge \neg \alpha$  is infeasible, when  $[p]$  is written  $[p] = [p_1] \wedge [p_2]$  with all constraints on  $p_1$  smaller than  $\alpha$  with respect to  $\prec$ , and all constraints on  $p_2$  larger than  $\alpha$ . An  $R_P$ DDD  $u$  is *saturated* if all paths from  $u$  are saturated.  $\square$

Saturation can be obtained by making as many implicit constraints as possible explicit without introducing any infeasible paths in the DDD. As an example, the DDD in Figure 5(c) will be saturated into the DDD in Figure 5(b). However, tight and saturated DDDs are still not canonical. Figure 6 shows an example of two tight, saturated  $R_P$ DDDs that have the same solution. Intuitively, the problem is that the vertex with constraint  $y - x \leq 0$  is redundant, since the solution set is the area  $x \geq 1, y \geq 1$  (assuming  $z = 0$ ). To detect such situations, a further check is necessary:

**Definition 7 (Disjunctive vertex)** Let  $p$  be a path leading to the vertex  $u$  in a DDD, and assume  $\alpha = cstr(u)$ ,  $h = high(u)$ , and  $l = low(u)$ . Then  $u$  is *disjunctive* in  $p$  if  $[p] \wedge (\alpha \rightarrow h, l)$  and  $[p] \wedge (h \vee l)$  have the same solution.  $\square$

This leads us to the following definition and accompanying conjecture:

**Definition 8 (Fully reduced DDD)** An  $R_P$ DDD  $u$  is a *fully-reduced DDD* ( $R_F$ DDD) if it is tight, saturated, and has no disjunctive vertices.  $\square$

**Conjecture 2 (Canonicity)** If  $u$  and  $v$  are  $R_F$ DDDs with the same set of solutions then  $u = v$ .

As it is illustrated by the above discussion, canonicity is rather difficult to obtain in DDDs. This is quite unlike the situation for BDDs, where local reductions and a total ordering of the variables is enough to obtain it. The reason is that, in DDDs there are non-local dependencies among the various constraints giving rise to not only untight constraints but also implied constraints that may or may not be explicitly present. Pragmatically, the lack of canonicity of path-reduced DDDs might not be a problem. The main benefit of the canonicity of ROBDDs is that the questions of equivalence, satisfiability, and tautologiness are trivial to answer. However, as pointed out in Theorem 1, satisfiability and tautologiness is trivial for path-reduced DDDs and even for just local-reduced DDDs the questions can be solved by a simple on-the-fly search for feasible paths. The crucial issue is whether the representation during computations stay compact which can occur with just a semi-canonical representation.

**Corollary 3** If  $u$  is an  $R_F$ DDD such that the solution to  $u$  is convex then  $u$  contains exactly one path to the terminal vertex 1.

## 6. Experimental Results

DDD can be used to analyze timed system efficiently by representing sets of discrete states and their associated timing information implicitly. The DDD algorithms implement the operations necessary for analyzing general systems with time such as timed automata [1] or timed Petri nets [8].

To demonstrate the applicability and efficacy of the DDD approach, we analyze two different versions of Milner's scheduler with time and compare the runtimes with those obtained with two state-of-the-art tools, KRONOS [16–18] and UPPAAL [7, 21].

### 6.1. Analyzing Timed Systems

We sketch how to analyze a timed system using difference constraint expressions (for a more complete description, see [23]). We describe a timed system by a set of transitions  $T$  defined as guarded assignments over Boolean variables  $B$  and real-valued clock variables  $C$ . A transition  $t \in T$  has the form

$$g \rightarrow \vec{v} := \vec{d},$$

where  $g$  is a guard expressed as a difference constraint expression over the variables in  $B$  and  $C$  and  $\vec{v} := \vec{d}$  is a multi-assignment of  $n$  constant values  $\vec{d}$  to Boolean and

clock variables  $\vec{v}$ . Absolute constraints on the clock variables (such as  $c_i \geq 0$ ) are expressed as difference constraints relative to a special variable  $z \in C$  which is interpreted as having the value 0 (for example  $c_i - z \geq 0$ ). The assignment  $[\vec{v} := \vec{d}]$  is a shorthand for  $c_i := z + d_i$  for each clock  $c_i$  in  $\vec{v}$  and  $b_i := d_i$  for each Boolean variable in  $\vec{v}$ . Clearly, guards can now be represented as DDDs, and the multi-assignment can be performed by ASSIGN.

The set of states reachable by firing the transitions in  $T$  and advancing the time can be determined using a constrained image computation. Two basic operations are needed: one for *firing a transition* and one for *advancing time*. Let  $\varphi$  denote a difference constraint expression representing a set of states, i.e., a set of discrete states and their associated timing information. A transition is fired on each state in  $\varphi$  implicitly by constraining  $\varphi$  to the subset where the guard is true and then performing the assignment on this subset:

$$\text{FIRE}(\varphi, g \rightarrow \vec{v} := \vec{d}) = (\varphi \wedge g)[\vec{v} := \vec{d}] \wedge \text{INV},$$

where INV is an invariant which must hold at the destination states. Typically, the invariant is used to ensure progress similar to state invariants in timed automata.

The  $z$ -variable plays a central rôle when determining the set of states that can be reached from a state in  $\varphi$  by advancing time. The set of states that can be reached from  $\varphi$  by advancing time by some amount  $\delta \in \mathbb{R}$  is determined by<sup>3</sup>

$$\text{ADV}(\varphi, \delta) = (\exists z'. (\varphi \wedge (z - z' = \delta) \wedge (z' \leq z) \wedge \text{INV}[z'/z]))[z/z'].$$

The set of states that can be reached from  $\varphi$  by advancing time by an arbitrary amount is given by:

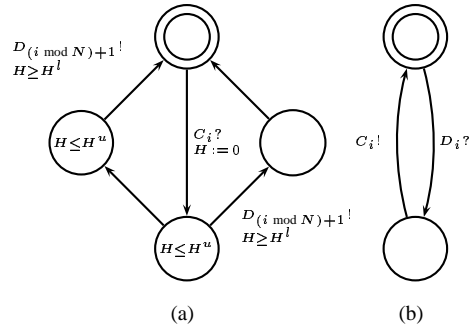
$$\begin{aligned} \text{ADV}(\varphi) &= \bigvee_{\delta \in \mathbb{R}} \text{ADV}(\varphi, \delta) \\ &= (\exists z'. (\varphi \wedge (z' \leq z) \wedge \text{INV}[z'/z]))[z/z']. \end{aligned}$$

Finally, some transitions may be *urgent* meaning that they must fire immediately when they become enabled. An urgency predicate URG is used to represent the set of states that have an urgent transition and thus time cannot progress in these states<sup>4</sup>. Using this predicate, the set of reachable states is now determined by calculating the least fixed point of the function NEXT starting from an expression  $I$  characterizing the set of initial states:

$$\text{NEXT}(\varphi) = \varphi \vee \text{ADV}(\varphi \wedge \neg \text{URG}) \vee \bigvee_{t \in T} \text{FIRE}(\varphi, t).$$

<sup>3</sup>If the bounds expressed by the state invariant are not convex, a further conjunct must be added expressing that the invariant holds at all intermediate points in time, i.e.,  $\forall z''. ((z' < z'' \leq z) \rightarrow \text{INV}[z''/z])$ .

<sup>4</sup>If the urgency predicate involves clock variables,  $\neg \text{URG}$  also needs to hold invariantly when advancing time, obtained by adding a conjunct in the formula for ADV:  $\forall z''. ((z' < z'' \leq z) \rightarrow \neg \text{URG}[z''/z])$ .



**Figure 7.** The  $i^{\text{th}}$  cyclers of Milner's scheduler with one clock. The cyclers are modeled with two timed automata synchronizing via CCS-like channels as in UPPAAL. Initial locations are drawn with double circles (for the first cycler the initial location in (b) is the opposite location). The locations of the automaton in (a) represent the four possible combinations of the variables  $h_i$  and  $t_i$ , and the automaton in (b) represents the variable  $c_i$ .

## 6.2. Milner's Scheduler with One Clock

Milner's scheduler [22] consists of  $N$  cyclers, connected in a ring, that cooperate on controlling  $N$  tasks. We associate three Boolean variables  $c_i$ ,  $h_i$ , and  $t_i$  with each cycler and use a global clock  $H$  to ensure that a cycler passes the token on to the following cycler within a bounded amount of time. The  $i^{\text{th}}$  cycler is described by the transitions:

$$\begin{aligned} c_i \wedge \neg t_i &\rightarrow H, t_i, c_i, h_i := 0, \text{T}, \text{F}, \text{T} \\ h_i \wedge (H^l \leq H \leq H^u) &\rightarrow c_{(i \bmod N)+1}, h_i := \text{T}, \text{F} \\ t_i &\rightarrow t_i := \text{F}. \end{aligned}$$

The amount of time a cycler can keep the token is determined by the interval  $[H^l, H^u]$ . The first transition is urgent, i.e., time is not allowed to advance when a cycler can take the token. Thus, the urgency predicate is

$$\text{URG} = \bigvee_{i=1}^N c_i \wedge \neg t_i.$$

For Milner's scheduler, the invariant is given by

$$\text{INV} = \bigwedge_{i=1}^N h_i \Rightarrow H \leq H^u,$$

expressing that cycler  $i$  must pass on the token no later than time  $H^u$ .

It is straightforward to model Milner's scheduler with timed automata. Figure 7 shows the  $i^{\text{th}}$  cycler in the notation used by UPPAAL.

**Table 1.** Experimental results for Milner’s scheduler with one clock using the bounds  $[H^l, H^u] = [25, 200]$ . The first column shows the number of cyclers. The following three columns show the CPU time (in seconds) to build the reachable state space using KRONOS, UPPAAL, and DDDs, respectively. The results were obtained on a Pentium II PC with 64 MB of memory running Linux. A ‘–’ denotes that the analysis did not complete within an hour.

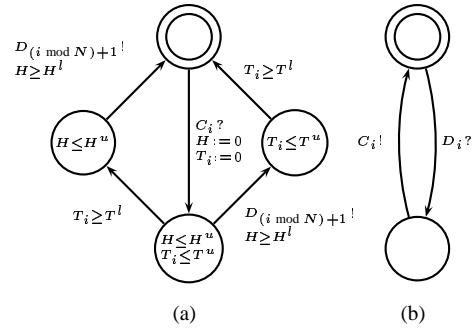
$N$	KRONOS	UPPAAL	DDD
4	0.2	0.1	0.1
5	0.7	0.2	0.1
6	22.6	0.6	0.1
7	339.2	2.3	0.1
8	–	9.0	0.2
9	–	35.0	0.2
10	–	138.4	0.2
11	–	529.8	0.2
12	–	2560.7	0.3
16	–	–	0.5
32	–	–	2.2
64	–	–	15.9
128	–	–	123.3
256	–	–	1104.8

We have computed the reachable state space for increasing  $N$  using a fixed-point iteration with front sets. The results are shown in Table 1 together with the runtimes obtained with KRONOS (version 2.2b) and UPPAAL (version 2.17).

This version of Milner’s scheduler has a number of discrete states which is exponential in  $N$  since a task can terminate independently of the other tasks. Thus, state space exploration based on enumerating all discrete states (as in UPPAAL and KRONOS) only succeeds for small systems. In the DDD-based approach, discrete states are represented implicitly (as when using ROBDDs for purely discrete systems) and choosing a good ordering of the variables gives polynomial runtimes (and state space representations).

### 6.3. Milner’s Scheduler with a Clock per Task

We now restrict the time a task can be executing by introducing a clock  $T_i$  that measures the execution time of task  $t_i$ . The task  $t_i$  must terminate within a certain bound  $[T^l, T^u]$  after it is started. The resulting system potentially has  $N + 1$  concurrently running clocks (one for each task plus one for the token), but the system will have fewer discrete states than the previous version since the bounded execution time of the tasks limits the number of reachable discrete states. The  $i^{\text{th}}$  cycler is now described by the tran-



**Figure 8.** The timed automata modeling the  $i^{\text{th}}$  cycler of Milner’s scheduler with a clock for each task. Initial locations are drawn with double circles (for the first cycler the initial location in (b) is the opposite location).

sitions

$$\begin{aligned}
 c_i \wedge \neg t_i &\rightarrow H, T_i, t_i, c_i, h_i := 0, 0, \text{T}, \text{F}, \text{T} \\
 h_i \wedge (H^l \leq H \leq H^u) &\rightarrow c^{(i \bmod N)+1}, h_i := \text{T}, \text{F} \\
 t_i \wedge (T^l \leq T_i \leq T^u) &\rightarrow t_i := \text{F}.
 \end{aligned}$$

Introducing the new clocks changes the invariant slightly:

$$\text{INV} = \bigwedge_{i=1}^N (h_i \Rightarrow H \leq H^u) \wedge (t_i \Rightarrow T_i \leq T^u),$$

while the urgency predicate remains unchanged. Figure 8 shows the  $i^{\text{th}}$  cycler in the notation used by UPPAAL.

The runtimes for computing the reachable state space for increasing  $N$  are shown in Table 2. Again, the runtimes of KRONOS and UPPAAL are exponential in  $N$ , while using the DDD data structure results in polynomial runtimes.

In this version of Milner’s scheduler, the problem for KRONOS and UPPAAL is the large number of clock variables. This is handled in the DDD-based approach by eliminating the unused clocks from the representation (i.e., we quantify out  $T_i$  whenever a transition that sets  $t_i$  to false is fired in FIRE).

## 7. Conclusion

The problem addressed in this paper is how to efficiently represent and manipulate a Boolean logic over inequalities of the form  $x - y < c$  and  $x - y \leq c$ , where the variables are integer or real-valued. We have proposed a data structure inspired by ROBDDs for representing the expressions from the logic as a decision diagram in which the test conditions are difference constraints.

Introducing an ordering of the constraints makes it possible to extend the APPLY algorithm for ordered BDDs to

**Table 2.** Experimental results for Milner’s scheduler with one clock per task using the bounds  $[H^l, H^u] = [25, 200]$  and  $[T^l, T^u] = [80, 100]$ . The first column shows the number of cyclers. The following three columns show the CPU time (in seconds) to build the reachable state space using KRONOS, and UPPAAL, and DDDs, respectively. The results were obtained on a Pentium II PC with 64 MB of memory running Linux. A ‘–’ denotes that the analysis did not complete within an hour.

$N$	KRONOS	UPPAAL	DDD
4	0.4	0.2	0.2
5	2.4	1.7	0.3
6	24.2	17.6	0.5
7	346.6	201.7	0.5
8	–	2460.2	0.6
16	–	–	1.5
32	–	–	5.7
64	–	–	31.7
128	–	–	217.3

ordered DDDs without changing its runtime complexity. However, since the domain of the variables in the logic is infinitary, other operations such as existential quantification, are more difficult than for BDDs. For ordered DDDs, these algorithms are basically polynomial, but they become exponential due to the ordering requirement. Another complication is that there are implicit constraints among the variables causing the DDD data structure to be non-canonical even when local reductions are used. A first step towards canonicity is to eliminate all infeasible paths. Such a path-reduced DDD, can be tested for tautology and satisfiability in constant time. However, semantically equivalent DDDs may still have different representations. We have defined several additional restricting conditions, which we conjecture will result in canonical DDDs. It is clearly difficult to obtain an efficient canonical representation. Although canonicity would be intriguing to obtain and allow one to check for equivalence in constant time, it is not necessarily desirable in practice. A canonical representation will not be more compact than a non-canonical representation and the equivalence check can be performed as a tautology check.

Boolean variables can be modeled as difference constraints, making it possible to combine Boolean, continuous, and integer variables within a single data structure. All operations on the Boolean variables in the DDD can be performed as efficiently as with BDDs. One use of combining Boolean and real-valued variables is in constructing the set of reachable states for a concurrent timed system. The effectiveness of the data structure and associated algorithms is demonstrated by constructing the set of reachable states for two timed versions of Milner’s scheduler. For both version-

s, the set of reachable states can be computed in polynomial time using DDDs, while the tools KRONOS and UPPAAL both take exponential time.

One path that could be taken when extending the results of the paper would be to generalize the difference constraints to linear inequalities  $\sum_{i=1}^n a_i x_i \lesssim c$  ordered by a total ordering. The basic data structure and the APPLY algorithm would be unchanged. In the existential quantification the only change is in RELAX, where  $x$  is isolated and new inequalities are obtained by substituting the inequality for  $x$ . In eliminating infeasible paths, a general linear programming solver must be used, e.g, the simplex algorithm.

## References

- [1] R. Alur and D. Dill. The theory of timed automata. In *Real-Time: Theory in Practice*, number 600 in LNCS, pages 28–73. Springer, 1991.
- [2] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In *Int. Workshop on Hybrid and Real-Time System*, Mar. 1997.
- [3] F. Balarin. Approximate reachability analysis of timed automata. In *Proc. Real-Time Systems Symposium*, pages 52–61. IEEE, 1996.
- [4] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. Technical Report DoCS Technical Report Nr 99/105, Uppsala University, 1999.
- [5] W. Belluomini and C. J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 1997.
- [6] W. Belluomini and C. J. Myers. Verification of timed systems using POSETs. In *Proc. Computer Aided Verification (CAV)*, June 1998.
- [7] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *Proc. International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 1055 in LNCS, pages 431–434. Springer-Verlag, Mar. 1996.
- [8] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.
- [9] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *CAV’97*, LNCS 1254, pages 179–190, 1997.
- [10] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
- [11] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon, Aug. 1992.
- [12] S. V. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraiishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposium*, pages 266–70. IEEE, Dec. 1994.

- [13] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *Computer Aided Verification*, pages 316–27, 1997.
- [14] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
- [15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1994.
- [16] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proc. of Workshop on Verification and Control of Hybrid Systems*, number 1066 in LNCS, pages 208–219. Springer-Verlag, Oct. 1995.
- [17] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. 7th IFIP WG G.1 International Conference of Formal Description Techniques (FORTE)*, pages 227–242. Chapman & Hall, Oct. 1994.
- [18] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. 16th IEEE Real-Time Systems Symposium (RTSS)*, pages 66–75. IEEE Computer Society Press, Dec. 1995.
- [19] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, LNCS 407. Springer, 1989.
- [20] K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *Proc. of the 10th Int. Conference on Fundamentals of Computation Theory*, number 965 in LNCS, pages 62–88, Aug. 1995.
- [21] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.
- [22] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [23] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology, Technical University of Denmark, 1999. Available on the web: <http://ddd.it.dtu.dk>.
- [24] T. G. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.
- [25] T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In D. L. Dill, editor, *Computer Aided Verification (CAV)*, number 818 in LNCS, pages 468–480, 1994.
- [26] R. L. Spelberg, H. Toetenel, and M. Ammerlaan. Partition refinement in real-time model checking. In *Proceedings of FTRTFT'98*, pages 143–57, 1998.
- [27] E. Verlind, G. de Jong, and B. Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, 1996.
- [28] H. Wong-Toi and D. Dill. Approximations for verifying timing properties. In *Theories and Experiences for Real-Time Systems Development*. World Scientific Publishing, 1994.
- [29] S. Yovine. Kronos: A verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1/2), Oct. 1997.
- [30] S. Yovine. Model checking timed automata. In *Embedded Systems*, LNCS. Springer-Verlag, 1998.