Decision Diagrams for Linear Arithmetic

Sagar Chaki
SEI/CMU
chaki@sei.cmu.edu

Arie Gurfinkel
SEI/CMU
arie@sei.cmu.edu

Ofer Strichman
Technion
ofers@ie.technion.ac.il

***Abstract*—Boolean manipulation and existential quantification of numeric variables from linear arithmetic (LA) formulas is at the core of many program analysis and software model checking techniques (e.g., predicate abstraction). We present a new data structure, Linear Decision Diagrams (LDDs), to represent formulas in LA and its fragments, which has certain properties that make it efficient for such tasks. LDDs can be seen as an extension of Difference Decision Diagrams (DDDs) to full LA. Beyond this extension, we make three key contributions. First, we extend sifting-based dynamic variable ordering (DVO) from BDDs to LDDs. Second, we develop, implement, and evaluate several algorithms for existential quantification. Third, we implement LDDs inside CUDD, a state-of-the-art BDD package, and evaluate them on a large benchmark consisting of 850 functions derived from the source code of 25 open source programs. Overall, our experiments indicate that LDDs are an effective data structure for program analysis tasks.**

I. INTRODUCTION

Many program analysis problems – e.g., computation and application of predicate abstraction, image computation, function summarization – are ultimately reduced to manipulating propositional formulas over some theory. Typically, two types of operations are required: (i) Boolean (conjunction, negation, etc.) and (ii) existential quantification (henceforth QELIM). For example, the predicate abstraction of a transition relation R is computed as

$$\exists \vec{x}, \vec{x'} \centerdot R(\vec{x}, \vec{x'}) \wedge \left(\bigwedge_{i=1}^{n} v_i \Leftrightarrow p_i(\vec{x}) \right) \wedge \left(\bigwedge_{i=1}^{n} v'_i \Leftrightarrow p_i(\vec{x'}) \right) , \quad (1)$$

where $\vec{x}, \vec{x'}$ are current- and next-sate program variables, $\vec{v}, \vec{v'}$ are current- and next-state propositional abstract variables, and p_i is the definition of the i-th abstract variable in terms of $\vec{x}$. Thus, effective predicate abstraction – and program analysis tasks in general – requires a mechanism that combines space-efficient representation of formulas and fast QELIM.

This is challenging because QELIM algorithms require a formula in Disjunctive Normal Form (DNF), but DNF is a space-inefficient representation, e.g., relative to Conjunctive Normal Form (CNF). Common solutions to this issue follow one of three approaches: DNF, Abstract Syntax Tree (AST), and Decision Diagram (DD). In the DNF approach, a formula is represented by the set of terms of a DNF, e.g., a disjunctive invariant is represented by a set of octagons [15]. This approach is easy to implement, but does not scale to formulas with large DNFs. In the AST approach, a formula is represented as an AST (or a DAG) whose nodes correspond to variables, constants, and operators. This is the most space-efficient representation. However, converting an AST into DNF during QELIM – e.g., using a SMT-solver [12] – is expensive, in many cases exponential in the AST size. In the DD approach, a formula is represented by a DAG whose nodes are labeled by atomic terms. The DAG enables sharing of sub-expressions, and, at the same time, is easy to convert to DNF. This is the approach we explore in this paper by introducing and evaluating a new data structure, Linear Decision Diagram (LDD), for quantifier-free first-order linear arithmetic (LA) formulas.

LDDs extend DDDs – decision diagrams for difference logic – proposed by Møller et al. [16]. Predicates in difference logic are of the form $x - y \leq c$ or $x - y < c$ for variables x, y and a constant c, ranging over $\mathbb{Q}$ or $\mathbb{Z}$. The key idea of DDDs is to represent first-order quantifier-free difference logic formulas as BDDs with nodes labeled by atomic predicates, and to reduce redundancy by leveraging implications between those predicates. For example, in a DDD, a node labeled with $x - y \leq 10$ never appears as a high child of a node labeled with $x - y \leq 5$. For a fixed variable order, a DDD of a formula f is no larger than a BDD representing a propositional abstraction of f (i.e., f with all atomic terms replaced by propositional variables). An important feature of DDDs is a QELIM algorithm based on Fourier-Motzkin elimination [3]. The algorithm lends itself well to dynamic programming, thus, leveraging the DAG-structure of DDs.

DDDs have three main limitations as an instrument to aid program analysis: (a) difference logic is too restrictive for many program analysis tasks, (b) DDDs do not support dynamic variable ordering (DVO), and (c) there are no publicly available implementations of DDDs and no reports of their effectiveness in solving practical program analysis problems. Our solution via LDDs address all three limitations. Specifically, LDDs extend to full linear arithmetic, and support efficient algorithms for DVO and QELIM. Moreover, we implemented LDD within CUDD, a state-of-the-art BDD package. Finally, we have evaluated LDDs using a benchmark derived from open-source programs.

Extending the Boolean operations from DDDs to LDDs is straightforward. The key challenges are in supporting DVO and QELIM. The importance of DVO for DDs is well-known. This is especially true for LDDs since they add restrictions on the variable order. We show that the standard BDD DVO *cannot* be used "as is" for LDDs. Instead, we adapt Rudell's sifting algorithm [18], as implemented in CUDD. Our adaptation does not add any overhead over the BDD version of DVO. Our experiments confirm that DVO reduces the size of LDDs significantly.

We develop two types of QELIM for LDDs. The *black-box* QELIM applies an external QELIM-solver to each path in the LDD. It is linear in the number of paths (bad), but is compatible with any off-the-shelf solver (good). In contrast, the *white-box* QELIM is a generalized and optimized variant of the DDD version. It recursively applies pairwise resolution to DD nodes, in the style of Fourier-Motzkin. In the worst case, this algorithm is exponential in the size of the diagram. However, our experiments indicate that it performs well in practice. Black-box and white-box QELIM have success rates of 4.47% and 98.94%, respectively, when solving our benchmark problems.

The basic white-box QELIM algorithm only eliminates one variable at a time – a fundamental limitation of Fourier-Motzkin. However, many applications of QELIM in program analysis require eliminating multiple variables. To address this, we present an elimination strategy that iterates, while there are variables to be eliminated, between (a) dropping constraints with variables that are not resolved on during Fourier-Motzkin, and (b) choosing and eliminating an existentially quantified variable which results in a minimum number of resolutions. This heuristic yields a speedup of over 5 times for QELIM in our benchmark.

In order to test the suitability of LDDs for solving practical program analysis programs, we have evaluated our implementation on a benchmark derived from open source programs. The benchmark consists of transition relations (in Static Single Assignment form) of 850 functions from 25 C programs. In each case, we measured the space required (in DD nodes) to represent the transition relation, and time to compute a forward image (i.e., strongest post-condition). The experimental results lead us to believe that LDDs are an effective representation for fragments of LA for program analysis tasks.

Related Work. DDs for theories other than propositional logic have been studied extensively since the early 90's, capitalizing on the great success of BDDs and numerous BDD optimizations. Among these, we do not cover DDs – such as Binary Moment Diagrams (BMDs) [5], Algebraic Decision Diagrams (ADDs) [2], and Boolean Expression Diagrams (BEDs) [1] – which are restricted to variables with finite domains. Thus, the most relevant structures to LDDs are those that label the nodes with linear predicates over the reals.

Groote and Tveretina [9] proposed decision diagrams for full first-order logic. Assuming an input formula is in Prenex normal form, they represent the negation of its quantification suffix with a DD, and prove a contradiction using Skolemization and standard strategies, e.g., applying unifiers.

Equational (EQ) BDDs [10] are aimed at deciding equalities with uninterpreted functions (EUF). In EQ-BDDs, nodes are labeled with predicates (equalities), and reduction rules enforce substitution according to a predefined order $\preceq$ between variables. For example, if $x \preceq y$, then y is substituted by x in high sub-DD of a node labeled $x = y$. Equational BDDs are semi-canonical – they reduce to **0** (or **1**) if they represent a contradiction (or tautology), but are non-canonical otherwise.

Cavada et al. [6] propose a QELIM technique for LA that combines BDDs and SMT-solvers. They focus on predicate abstraction, and consider the problem of quantifying all numeric variables from LA formulas (see (1) for a template) over LA predicates and propositional variables. The Boolean structure of the formulas are encoded via BDDs. QELIM is done by recursively traversing the BDD, carrying, along each path, the set of linear predicates (i.e., the context) seen on it. At each recursive step, an SMT-solver is used to check whether the context is consistent. Paths with inconsistent context are removed. The use of the context precludes the use of dynamic programming. Thus, the algorithm is linear in the number of paths of the BDD. We call such an approach "black box" because it uses an external decision procedure as is. In Sec. IV, we present a similar "black box" algorithm for quantifying *some* or all numeric variables.

The most relevant prior work is on DDDs [16]. QELIM of a numeric variable from a DDD is based on Fourier-Motzkin. Although in the worst case this procedure is exponential in the size of the DD, in the best case it is the same as QELIM for BDDs. We describe it in more detail in Sec. IV, as it is the base for our improved method. *Clock Difference Diagrams* [13], is an alternative to DDDs that was developed independently around the same time. CDDs are based on DDs with arbitrary branching degree. QELIM is done in the black-box fashion by traversing all **1**-paths. We are not aware of any work that has adapted DVO to either DDDs, CDDs, or EQ-BDDs. DDDs and CDDs are extensions of Interval Decision Diagrams (IDD) [19].

We define LDDs in the next section. In Sec. III and IV we discuss the problems of dynamic variable ordering and QELIM with such diagrams, respectively. Sec. V is dedicated to experimental results, and we conclude in Sec. VI.

II. LINEAR DECISION DIAGRAMS

We assume the reader is familiar with the basics of decision diagrams. A Linear Decision Diagram (LDD) is a data structure to represent and manipulate propositional formula over (a fragment of) linear arithmetic. Formally, they are BDDs with (a) nodes labeled by linear atomic predicates, and (b) satisfying ordering and local reduction constraints. In the rest of this paper, we use T to denote a fragment of linear arithmetic, unless mentioned otherwise. For a formula p, we write $\textsc{Vars}(p)$ to mean the set of variables in p.

A. Definitions

An LDD over a theory T is a directed acyclic graph with

- Two terminal nodes labeled with **0** and **1**, respectively;
- Nonterminal nodes. Each nonterminal node u has two children, denoted by $H(u)$ and $L(u)$, and is labelled with a T-atom (i.e., an atomic predicate), denoted by $C(u)$.
- Edges $(u, H(u))$ and $(u, L(u))$ for every non-terminal node u.

We use $attr(u)$ to denote the triple $(C(u), H(u), L(u))$. An LDD with a root node u represents the formula $exp(u)$ over

T defined as follows: $exp(\mathbf{0})$ is FALSE, $exp(\mathbf{1})$ is TRUE, otherwise, $exp(u)$ is defined recursively as:

$$exp(u) = \textsc{ite}(C(u), exp(H(u)), exp(L(u)))\ ,$$

where

$$\textsc{ite}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c)\ .$$

For simplicity, we don't distinguish between a node u and $\exp(u)$.

Example 1 *An example of an LDD for the formula*

$$(z - y \leq 0 \wedge x - y \leq 10) \vee (z - y > 0 \wedge x - y \leq 5)$$

is shown in Fig. 3(a). A different LDD for the same formula, owing to a different order, is shown in Fig. 3(b).

B. Requirements from the theory

A theory T over variables Var is *LDD-adequate*, or simply *adequate*, if for any given set of consistent T-atoms A_T over Var (e.g., $x < x$ is an inconsistent T-atom for T being linear arithmetic), the following functions can be provided:

- (*Negation*) $NEG : A_T \mapsto A_T$ such that $NEG(c) \Leftrightarrow \neg c$. In the following, we write $\neg c$ to mean $NEG(c)$.
- (*Normalization*) $N : A_T \mapsto A_T$ such that if $c \Leftrightarrow c'$ or $c \Leftrightarrow \neg c'$ then $N(c) = N(c')$. Note that $\forall c \in A_T \centerdot N(c) = N(\neg c)$. If $c = N(c)$, then c and $\neg c$ are represented by $\textsc{ite}(c, \mathbf{1}, \mathbf{0})$ and $\textsc{ite}(c, \mathbf{0}, \mathbf{1})$, respectively.
- (*Negation Check*) $isNEG : A_T \mapsto Bool$ such that $isNEG(c) \Leftrightarrow (c = \neg N(c))$.
- (*Implication*) $IMP : A_T \times A_T \mapsto Bool$ such that $IMP(c_1, c_2)$ iff $(c_1 \Rightarrow c_2)$.
- (*Resolution*) $\textsc{Rslv} : A_T \times Var \times A_T \mapsto A_T \cup \{\textsc{true}\}$ such that $\textsc{Rslv}(c_1, x, c_2) = c_3$ iff $x \in \textsc{Vars}(c_1) \cap \textsc{Vars}(c_2)$, and $c_3 \Leftrightarrow \exists x \centerdot\ c_1 \wedge c_2$. We syntactically extend $\textsc{Rslv}$ so its first argument is a set of T-atoms as follows: $\textsc{Rslv}(S, x, c) = \bigwedge_{s \in S} \textsc{Rslv}(s, x, c)$.

Example 2 *Let UTVPI be the quantifier-free first order theory of Unit Two Variables Per Inequality over the integers. The set of UTVPI-atoms is*

$$\{ax + by \leq k \mid x, y \in Var, a, b \in \{-1, 1\}, k \in \mathbb{Z}\}\ .$$

This theory is also known as Octagons *[15]. $UTVPI$ is adequate, as shown below:*

- *$NEG(ax + by \leq k) = -ax - by \leq -k - 1$.*
- *Let $<_V$ be a total order on Var and $c = ax + by \leq k$. Then, $N(c)$ is (i) $N(by + ax \leq k)$ if $y <_V x$, (ii) c if $x <_V y \wedge a > 0$, and (iii) $NEG(c)$ otherwise.*
- *$IMP(c_1, c_2) =$ TRUE iff $c_1 = ax + by \leq k$ and $c_2 = ax + by \leq k'$ and $k \leq k'$.*
- *$\textsc{Rslv}(c_1, x, c_2)$ is* TRUE *if x does not appear in opposite phases in c_1 and c_2. Otherwise, it is the result of resolution on x between c_1 and c_2. For example,* $\textsc{Rslv}(x - y \leq 5, x, z - x \leq 2) = z - y \leq 7$.

Other adequate theories include: intervals, whose atoms are $\{x \leq k \mid x \in Var, k \in \mathbb{Z}\}$; difference logic, whose atoms are $\{x - y \leq k \mid x, y \in Var, k \in \mathbb{Z}\}$; and linear arithmetic over the reals. We use UTVPI for illustrations in this paper and in all of our experiments.

```
1: function MK ($A_T$ $c$, $LDD$ $f$, $LDD$ $g$)
2:     if ($IMP(c, C(f))$) then $f \leftarrow H(f)$
3:     if ($f = g$) then return $g$
4:     if ($IMP(c, C(g)) \wedge f = H(g)$) then return $g$
5:     return BDDNODE($c, f, g$)
```

Fig. 1. MK: Building an LDD.

C. Variable ordering

For LDDs, "T-atoms ordering" replaces the traditional BDD "variable ordering". Ordering between the T-atoms facilitates the construction of reduced diagrams. Let $\leq_{IMP}$ be the partial order on A_T induced by IMP: $c_1 \leq_{IMP} c_2 \Leftrightarrow IMP(c_1, c_2)$. A *$T$-atoms ordering* is any total order $\leq_T$ that extends $\leq_{IMP}$ to a total order on A_T. An LDD u is *ordered* w.r.t. $\leq_T$ iff for every node v reachable from u, $C(v) \leq_T C(H(v))$ and $C(v) \leq_T C(L(v))$. An LDD u is *well-ordered* if it is ordered with respect to some T-atoms ordering $\leq_T$. Both of the LDDs in Fig. 3 are well-ordered.

D. Local Reductions

An LDD is locally reduced iff the following five conditions hold on every internal node u and v.

1) *No duplicate nodes.* $attr(u) = attr(v) \Rightarrow u = v$.
2) *No redundant nodes.* $L(v) \neq H(v)$.
3) *Normalized labels.* $C(v) = N(C(v))$.
4) *Imply high.* $\neg IMP(C(v), C(H(v)))$.
5) *Imply low.* $IMP(C(v), C(L(v))) \Rightarrow H(v) \neq H(L(v))$.

For example, the LDD in Fig. 3(a) is reduced, but the LDD in Fig. 3(b) is not. From here on, we write LDD to mean Reduced Ordered LDD (ROLDD). The function $\textsc{mk}(c, f, g)$, shown in Fig. 1, constructs an ROLDD for $\textsc{ite}(c, f, g)$, where (i) c is a normalized constraint, (ii) f and g are ROLDDs s.t. $f \neq g$, and (iii) $c \leq_T C(f)$ and $c \leq_T C(g)$. Lines 2–4 ensure that the result is reduced, and line 5 returns a unique diagram node representing the ITE. The proof of correctness of MK is based on the following two reduction rules to enforce *Imply high* and *Imply low*, respectively:

$$\frac{\textsc{ite}(x, \textsc{ite}(y, h, l), z) \quad IMP(x, y)}{\textsc{ite}(x, h, z)}\ ,$$

$$\frac{\textsc{ite}(x, y, \textsc{ite}(z, h, l)) \quad IMP(x, z) \quad y \Leftrightarrow h}{\textsc{ite}(z, h, l)}\ .$$

E. Basic LDD Operations

For any symmetric operator *op* that distributes over ITE, the function APPLY (Op *op*, LDD f, LDD g) constructs the LDD for f *op* g. It is similar to the equivalent BDD operation. It is based on the following transformations (if multiple rules apply, the earliest is selected) and their symmetric versions obtained by swapping the arguments of *op*:

$$\frac{\textsc{ite}(x, y, z)\ op\ \textsc{ite}(u, v, w) \quad x \Leftrightarrow u}{\textsc{ite}(x, y\ op\ v, z\ op\ w)}\ ,$$

```
function SwapInPlace $(i, j)$
    replace every BDD node $F : (i, H, L)$ with $(j, G_1, G_0)$ where
    $F_{10}, F_{11}$ are the $\neg j$ and $j$ cofactors of $H$
    $F_{00}, F_{01}$ are the $\neg j$ and $j$ cofactors of $L$
    $G_1 \leftarrow (F_{11} = F_{01})\ ?\ F_{11} : (i, F_{11}, F_{01})$
    $G_0 \leftarrow (F_{00} = F_{10})\ ?\ F_{00} : (i, F_{10}, F_{00})$
```

Fig. 2. Swapping adjacent labels in a BDD.

$$\frac{\text{ITE}(x,y,z)\ op\ \text{ITE}(u,v,w) \quad IMP(x,u)}{\text{ITE}(x, y\ op\ v, z\ op\ \text{ITE}(u,v,w))}\ \text{, and}$$

$$\frac{\text{ITE}(x,y,z)\ op\ \text{ITE}(u,v,w) \quad x \leq_T u}{\text{ITE}(x, y\ op\ \text{ITE}(u,v,w), z\ op\ \text{ITE}(u,v,w))}\ .$$

LDD conjunction (And) and LDD disjunction (Or) are implemented via Apply. LDD negation (Not) is implemented as in BDDs. The implementation of ternary LDD if-then-else (ITE) is similar to Apply.

In summary, extending Boolean operations from BDDs to LDDs is straightforward. In the next two sections, we show how to extend dynamic variable ordering and Qelim, which are more challenging.

III. Dynamic Variable Ordering

It is well known that the variable order of a decision diagram has a crucial effect on its size. Both finding the best order and deciding whether an order is optimal are NP-hard [4]. A lot of BDD research (e.g., [18], [8], [17]) has been dedicated to heuristics for finding a good variable order. In this section, we show how to adapt the sifting heuristic of Rudell [18], as implemented in CUDD, to LDDs. The exact details of the heuristic are beyond the scope of this paper. We only focus on the parts that we changed.

Sifting heuristic is based on trial-and-error. Each node label (i.e., a BDD variable) is moved up and down in the order by swapping the order of two adjacent labels. The best position is recorded and restored at the end. There are several factors that make sifting very efficient: a set of Boolean functions is represented as one multi-rooted DAG; a *unique table* is used to locate nodes in the DAG in constant time; the table is partitioned into subtables, one per label, to locate all nodes with a given label in O(1); and, finally, swapping adjacent labels i and j in all diagrams in the unique table is done in time linear in the number of nodes labeled with i or j.

Only the swapping algorithm needs to be adapted for LDDs. Swapping adjacent labels amounts to reordering diagrams in the unique table. In CUDD, this operation is called SwapInPlace. We first show how SwapInPlace works for BDDs, and then, that it does not work for LDDs.

Pseudocode for SwapInPlace is shown in Fig. 2. Note that the swapping is done *in place* – any edge that was pointing to a node labeled with i before the swap points to the same node, but now labeled with j after the swap. Furthermore, SwapInPlace maintains an invariant that every diagram in the unique table is well-ordered and reduced.

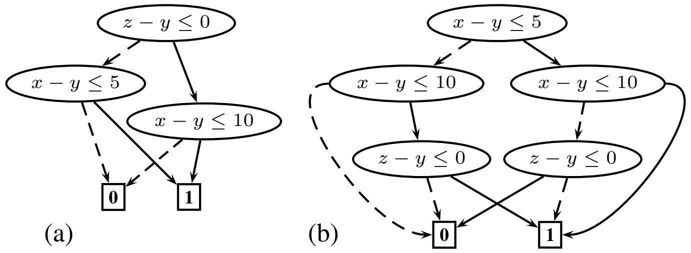

Fig. 3. (a) An ROLDD ordered by $z-y \leq 0$,$x-y \leq 5$, $x-y \leq 10$; (b) An OLDD ordered by $x-y \leq 5$, $x-y \leq$, $z-y \leq 0$.

```
function GroupMove $(X \subseteq A_T\ ,\ Y \subseteq A_T)$
    for $(i = |X|; i \geq 1; i \leftarrow i-1)$ do
        for $(j = 1; j \leq |Y|; j \leftarrow j+1)$ do
            LDDSwapInPlace$(x_i, y_j)$
```

Fig. 4. Swapping two adjacent sets of labels.

SwapInPlace does not work for LDDs. Consider an LDD shown in Fig. 3(a). Completely unrestricted swapping of adjacent labels conflicts with LDD well-orderedness constraints. Say, we swap $z-y \leq 0$ and $x-y \leq 5$. Then, the result is not well-ordered since a node labeled $z-y \leq 0$ appears between the nodes labeled with $x-y \leq 5$ and $x-y \leq 10$. Thus, we must swap groups of variables at a time. Say, we further swap $z-y \leq 0$ with $x-y \leq 10$. The result is shown in Fig. 3(b). This LDD is well-ordered, but it is not reduced: the *Imply high* rule (see Sec. II-D) is violated. It is easy to construct an example where the *Imply low* rule is violated as well.

To overcome the problems shown above, we propose a new algorithm, called GroupMove. GroupMove takes two ordered sets of T-atoms $X = \{x_1, \ldots, x_{|X|}\}$ and $Y = \{y_1, \ldots, y_{|Y|}\}$, and swaps them in the variable order. The order before GroupMove is $x_1, \ldots, x_{|X|}, y_1, \ldots, y_{|Y|}$, and after GroupMove is $y_1, \ldots, y_{|Y|}, x_1, \ldots, x_{|X|}$. This is done with $|X| \times |Y|$ calls to the helper function LDDSwapInPlace. Intuitively, GroupMove sifts each atom in the top class, one at a time, through the bottom class.

Function LDDSwapInPlace (shown in Fig. 5) is a variant of the original SwapInPlace that also maintains LDD reductions. Lines 7 and 10 ensure that any newly constructed LDD has no redundant nodes and satisfies *Imply low* rule. Lines 5 and 6 establish a stronger variant of *Imply high*: if a node v is reachable from a node u through $H(u)$, then $C(u)$ does not imply $C(v)$. This is crucial for ensuring that *Imply high* rule is established at the end of GroupMove.

```
1:  function LDDSwapInPlace $(A_T\ x\ ,\ A_T\ y)$
2:      replace every LDD $F{:}(x, H, L)$ with $(y, G_1, G_0)$, where
3:      $F_{00}, F_{01}$ are the $\neg y$ and $y$ cofactors of $L$
4:      $F_{10}, F_{11}$ are the $\neg y$ and $y$ cofactors of $H$
5:      $F'_{11} \leftarrow IMP(y, C(F_{11}))\ ?\ H(F_{11}) : F_{11}$
6:      $F'_{01} \leftarrow IMP(y, C(F_{01}))\ ?\ H(F_{01}) : F_{01}$
7:      if $(F'_{11} = F'_{01} \vee (IMP(x, C(F'_{01})) \wedge F_{11} = H(F'_{01})))$ then
8:          $G_1 = F'_{01}$
9:      else $G_1 \leftarrow (x, F'_{11}, F'_{01})$
10:     if $(F_{00} = F_{10} \vee (IMP(x, C(F_{00})) \wedge F_{10} = H(F_{00})))$ then
11:         $G_0 \leftarrow F_{00}$
12:     else $G_0 \leftarrow (x, F_{10}, F_{00})$
```

Fig. 5. Swapping adjacent labels in an LDD.

```
function BBQE (LDD $f$, $P \subseteq T$-atoms, $V \subseteq Var$)
    if ($f = \mathbf{1}$) then return TOTDD (THQELIM ($V, P$))
    if ($f = \mathbf{0} \vee$ THUNSAT($P$)) then return $\mathbf{0}$
    $c \leftarrow C(f)$
    if ($V \cap \textsc{Vars}(c) \neq \emptyset$) then
        $t \leftarrow \textsc{bbQE}(H(f), P \cup \{c\}, V)$
        $e \leftarrow \textsc{bbQE}(L(f), P \cup \{\neg c\}, V)$
        return $\textsc{Or}(t, e)$
    else
        $t \leftarrow \textsc{bbQE}(H(f), P, V)$
        $e \leftarrow \textsc{bbQE}(L(f), P, V)$
        return ITE $(c, t, e)$
```

Fig. 6. Black-box QELIM.

Correctness of GROUPMOVE . Let $G = (V, E)$ be a multi-rooted DAG of LDDs. A set of T-atoms X is *closed for G* iff whenever $x \in X$ and there is a $v \in V$ s.t. $IMP(C(v), x)$ or $IMP(x, C(v))$, then $C(v) \in X$. The correctness follows from Theorem 1.

Theorem 1 *Let $G = (V, E)$ be a multi-rooted DAG of ROLDDs, and X, Y be two non-empty sets of T-atoms that are closed and adjacent in G. Let $G' = (V', E')$ be the DAG after* GROUPMOVE(X, Y). *Then, (i) G' is a DAG of ROLDDs, and (ii) for any node v in G and the same node v' in G',* $\exp(v) \Leftrightarrow \exp(v')$.

IV. EXISTENTIAL QUANTIFICATION FOR LDDS

In this section, we describe three techniques for QELIM over LDDs. In the algorithms, TOTDD is used to convert a set P of T-atoms into an LDD for $\bigwedge P$; ITE and OR mean the corresponding LDD operations. For simplicity, we do not distinguish between a single T-atom and the corresponding LDD.

A. Black-box QELIM

Our black-box QELIM (see Fig.6) is called BBQE. It applies an external theory QELIM to each path of an LDD. BBQE requires the following helper functions from the theory:

- THUNSAT(P) decides whether $\bigwedge P$ is unsatisfiable;
- THQELIM(V, P) given a set of variables V and a set of T-atoms P, computes a set of T-atoms P' s.t. $\bigwedge P'$ is equivalent to $\exists V \centerdot \bigwedge P$.

Running time of BBQE(f, P, V) is linear in the number of paths of f. BBQE is not compatible with dynamic programming since it propagates the context P along every branch of an LDD.

B. White-box QELIM

Our white-box QELIM applies Fourier-Motzkin (FM) elimination directly to LDDs. It extends the QELIM algorithm of DDDs [16] to any adequate fragment T of LA.

First, we briefly recall FM elimination [3]. Let φ be a conjunction of T-atoms, and x be a numeric variable. FM elimination of x from φ proceeds as follows: Initially S is the set of all atoms of φ, and $S' = \emptyset$. For each $p \in S$, remove p from S. If $x \notin \textsc{Vars}(p)$ then add p to S'; otherwise, for each $t \in S$ s.t. $x \in \textsc{Vars}(t)$ add $\textsc{Rslv}(p, x, t)$ to S'. Note that $\textsc{Rslv}(p, x, t)$ is defined to be TRUE when x occurs in the same phase in both p and t. Upon termination, $\bigwedge S'$ is equivalent to $\exists x \centerdot \varphi$.

```
1: function WBQE1 ($x \in Var$, LDD $f$)
2:     if ($f = \mathbf{1} \vee f = \mathbf{0}$) then return $f$
3:     if ($x \notin \textsc{Vars}(C(f))$) then
4:         return $\text{ITE}(f, \textsc{wbQE1}(x, H(f)), \textsc{wbQE1}(x, L(f)))$
5:     $t \leftarrow \textsc{wbQE1}(x, \text{DR}(\{C(f)\}, x, H(f)))$
6:     $e \leftarrow \textsc{wbQE1}(x, \text{DR}(\{\neg C(f)\}, x, L(f)))$
7:     return OR (t,e)
```

Fig. 8. Basic white-box QELIM.

The key insight of white-box QELIM is to apply FM *simultaneously* to every $\mathbf{1}$-path of an LDD. The main step is simultaneous resolution. It is done by a function DR(S, x, f) (short for DAGRESOLVE) that takes a set S of T-atoms, a variable x, and an LDD f, and returns an LDD obtained by adding to each $\mathbf{1}$-path π of f the resolvents of S and π on x.

DR is implemented as follows: if f is a constant, it returns f; otherwise, it applies one of the recurrences shown in Fig. 7.

When implemented with dynamic programming, the number of recursive calls to DR is linear in the number of nodes in f. However, at the end it needs to restore orderedness, which, like in DDDs, is exponential in the size of f in the worst case.

The basic white-box algorithm WBQE1(x, f) is shown in Fig. 8. At each iteration, the algorithm either descends to branches of f (line 4), or removes a top-node of f whose label contains x (lines 5–7). The algorithm terminates since at each iteration the number of nodes labeled with an atom containing x decreases.

Recall that variable ordering in LDDs always respects chains of implications among T-atoms. Consider a "low implication chain" of LDD nodes $u_1, \ldots, u_n$. That is,

$$\forall 1 \leq i < n \centerdot L(u_i) = u_{i+1} \wedge C(u_i) \Rightarrow C(u_{i+1}) \,.$$

Then, $\forall 1 \leq i \leq n \centerdot \neg C(u_n) \Rightarrow \neg C(u_i)$. Let c be a T-atom and $x \in Var$. Then,

$$\forall 1 \leq i \leq n \centerdot \ \textsc{Rslv}(\neg C(u_n), x, c) \Rightarrow \textsc{Rslv}(\neg C(u_i), x, c) \,.$$

Let S be a set of T-atoms. Then,

$$\textsc{Rslv}(\{\neg C(u_1), \ldots, \neg C(u_n)\} \cup S, x, c) \Leftrightarrow \textsc{Rslv}(\{\neg C(u_n)\} \cup S, x, c) \,.$$

We use this observation in the algorithm WBQE2 (see Fig. 9) to reduce the number of calls to DR.

Since WBQE2(x, f) uses DR, in the worst case it is exponential in the number of nodes in f. However, in the best case it has the same complexity as BDD QELIM.

C. Eliminating Multiple Variables

Unlike BBQE, white-box QELIM only eliminates one variable at a time – this is a fundamental limitation of Fourier-Motzkin. When eliminating multiple variables, the order in which they are eliminated is crucial. For example, consider a formula

$$x - y \leq 1 \wedge z - x \leq 2 \wedge w - z \leq 3 \,.$$

$$\frac{\mathrm{DR}(S, x, \textsc{ite}(u, v, w)) \qquad x \notin \textsc{Vars}(u)}{\textsc{ite}(u, \mathrm{DR}(S, x, v), \mathrm{DR}(S, x, w))} \qquad \frac{\mathrm{DR}(S, x, \textsc{ite}(u, v, w)) \qquad x \in \textsc{Vars}(u)}{(\textsc{Rslv}(S, x, u) \wedge u \wedge \mathrm{DR}(S, x, v)) \vee (\textsc{Rslv}(S, x, \neg u) \wedge \neg u \wedge \mathrm{DR}(S, x, w))}$$

Fig. 7. Two recurrences used by DR.

```
function wbQE2 ($x \in Var$, LDD $f$)
    if ($f = \mathbf{1} \vee f = \mathbf{0}$) then return $f$
    if ($x \notin \textsc{Vars}(C(f))$) then
        return $\mathrm{ITE}(f, \textsc{wb}\mathrm{QE2}(x, H(f)), \textsc{wb}\mathrm{QE2}(x, L(f)))$
    $S \leftarrow \emptyset; K \leftarrow \emptyset$
    repeat
        $c \leftarrow C(f)$; $S \leftarrow S \cup \{c\}$
        $d \leftarrow \mathrm{DR}(S, x, H(f))$
        $K \leftarrow K \cup \{\textsc{wb}\mathrm{QE2}(x, d)\}$
        $S \leftarrow \{\neg c\}$ ; $c' \leftarrow C(L(f))$
        if ($IMP(c, c')$) then $f \leftarrow L(f)$;
    until ($\neg IMP(c, c')$)
    $d \leftarrow \mathrm{DR}(S, x, L(f))$
    $K \leftarrow K \cup \{\textsc{wb}\mathrm{QE2}(x, d)\}$
    return Or ($K$)
```

Fig. 9. Improved white-box Qelim.

```
1:  function wbMVQE ($V \subseteq Var$, LDD $f$)
2:      $res \leftarrow f$
3:      while ($V \neq \emptyset$) do
4:          $V' \leftarrow \textsc{findDropVars}(V, res)$
5:          if ($V' \neq \emptyset$) then
6:              $res \leftarrow \textsc{drVar}(V', res)$
7:              $V \leftarrow V \setminus V'$
8:              continue
9:          $x \leftarrow \textsc{chooseVar}(V, res)$
10:         $res \leftarrow \textsc{wb}\mathrm{QE2}(x, res)$
11:         $V \leftarrow V \setminus \{x\}$
12:     return $res$
```

Fig. 10. White-box Qelim for multiple variables.

Assume we want to eliminate x and y. There are two elimination orders: (a) x, y, and (b) y, x. In case (a), first $x - y \leq 1$ is removed and resolved with $z - x \leq 2$ to get: $z - y \leq 3 \wedge z - x \leq 2 \wedge w - z \leq 3$. Then, $z - x \leq 2$ is dropped, and finally $z - y \leq 3$ is dropped to get: $w - z \leq 3$. In case (b), first $x - y \leq 1$, and then $z - x \leq 2$ are dropped. No resolution is needed.

This example highlights two things. First, eliminating variables that occur in fewer atoms (like y above) leads to fewer resolutions and potentially smaller intermediate results. Second, variables that occur in a single atom (like y above), or, more generally, variables that occur in pure polarity in a disjunct of a DNF, are eliminated by dropping their atoms without any resolution steps. Since there is no resolution, multiple such variables can be eliminated at once.

We use these observations in a multi-variable elimination strategy called wbMVQE shown in Fig. 10. wbMVQE is parametrized by two functions:

- findDropVars(V, f) returns the subset of variables in V that do not need to be resolved on when they are eliminated from an LDD f.
- chooseVar(V, f) chooses a variable in V to be eliminated from an LDD f.

```
function drVar ($V \subseteq Var$, LDD $f$)
    if ($f = \mathbf{1} \vee f = \mathbf{0}$) then return $f$
    if ($\textsc{Vars}(C(f)) \cap V = \emptyset$) then
        return $\mathrm{ITE}(C(f), \textsc{drVar}(V, H(f)), \textsc{drVar}(V, L(f)))$
    else
        return $\textsc{Or}(\textsc{drVar}(V, H(f)), \textsc{drVar}(V, L(f)))$
```

Fig. 11. Dropping variables from an LDD without resolution.

It also uses a helper function drVar(V, f) shown in Fig. 11 that disjunctively drops all nodes from an LDD f that are labeled with an atom containing a variable in V.

wbMVQE iterates through two phases. First, it eliminates all variables that do not need resolution (lines 4–8). This is repeated until no more variables can be dropped. Second, it eliminates a variable using wbQE2 (lines 9–10). This process repeats until all variables are eliminated.

Note that wbMVQE is only a strategy – it must be instantiated with an implementation of findDropVars and chooseVar. For our experiments, we have instantiated wbMVQE by counting the number of occurrences of variables in V in the atoms labeling the nodes of f (i.e., the support of f). In our case, findDropVars returns all variables that occur in only a single atom; chooseVar picks a variable randomly from all variables that occur in the least number of atoms.

V. Experimental Results

We have implemented LDDs within CUDD. Except for DVO, our implementation is an external library that adds, but does not modify, CUDD. Following CUDD, we use *complemented edges* which allow for constant time negation. For DVO, we reuse all CUDD's heuristics, but modify how adjacent labels are swapped (see Sec. III). Fortunately, CUDD already can group variables and sift groups simultaneously (i.e., GroupMove), but we had to change its implementation of GroupMove to sift top-down instead of bottom-up.

To evaluate our implementation, we used a benchmark derived from 25 real C programs, including `mplayer`, `ff_mpeg`, `gzip`, `tcsh`, and `CUDD`. We created the benchmark by compiling each program using LLVM [14] with optimization, loop unrolling, and inlining enabled, and then approximating the SSA control-flow graph of each function by a UTVPI formula. In general, our approximation is unsound since we replaced LA formulas by UTVPI. However, each such formula is similar in structure to a bounded model-checking problem, e.g., as in CBMC [7]. We narrowed down the initial 10,000 formulas to our benchmark of 850 using the criteria: (a) more than 1,000 nodes in LDD or BDD representation, or (b) more than 2 seconds to build an LDD or a BDD, or (c) more than 2 seconds to solve with SVO.

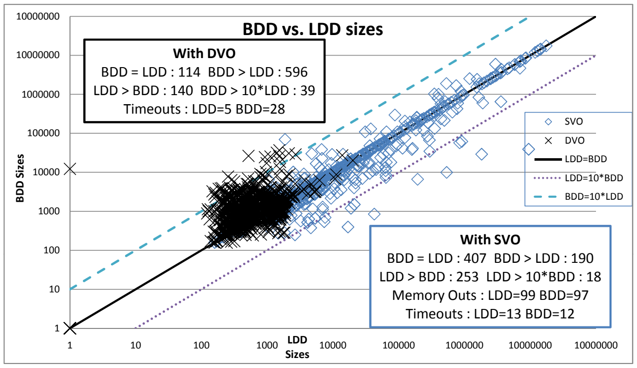

Fig. 12. Size comparison in DD nodes between LDD and BDD.

These formulas range in sizes from 4KB to 700KB, and have between 30 to 7,956 variables.

We conducted two sets of experiments to evaluate the effect of local LDD reductions and the effectiveness of QELIM, respectively. All experiments were done on an 8-core 2.00GHz Xeon with 3GB of RAM. Each test-case was limited to 300s of CPU and 512MB of RAM. The benchmark and detailed experimental results are available at `www.sei.cmu.edu/staff/chaki/FMCAD-09.html`.

Local Reductions. To measure the effect of local reductions, we compared the sizes of LDD v.s. BDD for each test-case using both SVO and DVO. By "BDD" we mean a BDD constructed by abstracting all UTVPI predicates by propositional variables. For BDDs, SVO is a syntactic variable order where each new predicate is placed at the end of the current order. It is the same for LDDs, except when it violates the LDD ordering restriction: in that case, the new predicate is inserted at an appropriate position in the order. For example, for the formula $x - y \leq 5 \wedge y - z \leq 0 \wedge x - y \leq 15$, the BDD SVO is as seen in the formula, while the LDD SVO is $x - y \leq 5, x - y \leq 15, y - z \leq 0$. DVO means that automatic DVO was enabled and DD manager reordered whenever memory usage was high.

The results are summarized in the scatter plot in Fig. 12. It shows LDD sizes (x-axis) vs. BDD sizes (y-axis). For example, the point $(325, 720)$ means that some test-case had an LDD with 325 nodes and a BDD with 720 nodes. Both axes are in log-scale. SVO and DVO experiments are marked by diamonds and crosses, respectively. As expected, DVO leads to significantly smaller diagrams for both BDDs and LDDs. Interestingly, with DVO LDDs are significantly smaller (sometimes by an order of magnitude) than BDDs, whereas with SVO the situation is reversed. This validates our intuition that DVO is even more significant for LDDs than for BDDs.

Quantification. To evaluate the effectiveness of our QELIM algorithms, we measured the time to quantify out the first $4/5^{th}$ syntactically appearing variables in each test-case. This roughly corresponds to a forward-image (a.k.a. strongest post-condition) in program analysis. We compared BBQE and 4 variants of WBMVQE (DVO is used unless stated otherwise):

TABLE I

Overall results for QELIM algorithms. All times are in seconds. *Total* = total times; *QE* = QELIM time; *TO* = Timeout; *MO* = Memoryout. No MOs for the Easy cases. No *Total*/*QE* for BB+DVO since it TO in most cases.

	Hard (154 cases)				Easy (696 cases)		
Alg.	*Total*	*QE*	*TO*	*MO*	*Total*	*QE*	*TO*
BBQE	—	—	141	0	—	—	670
WB+DVO	10,953	3,329	9	0	784	219	0
WB+SVO	38,739	36,511	21	99	395	80	0
WBWB1	11,047	3,761	11	0	829	264	0
WB-DV	17,043	13,358	34	0	5,649	5,151	8

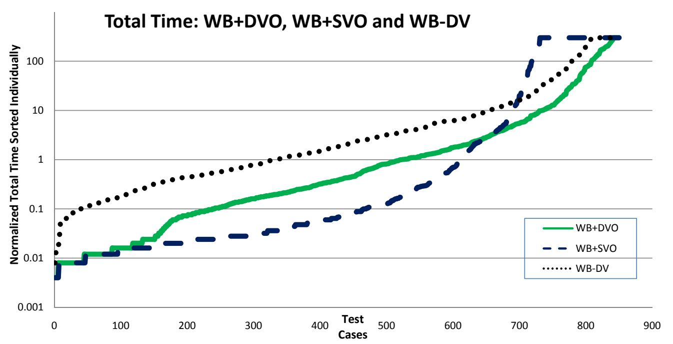

Fig. 13. Total times for white-box QELIM.

- WB+SVO: with SVO,
- WB+DVO: with DVO,
- WB-DV: with lines 4–8 removed, and
- WBWB1: with a call to WBQE2 on line 10 replaced with the call to WBQE1.

The overall results are summarized in Table I. We call the 696 test-cases that are solved by WB+SVO in under 15s *easy*, and the remaining 154 *hard*. *Total* is the time including parsing, building an LDD, and QELIM; *QE* is the time spent in BBQE or WBMVQE. All times are in seconds. Each failure is normalized to 300s. Size of the final LDDs ranged from $140,092$ to 1.

BBQE performed the worst, solving only 38 cases. This is not surprising. The average path-size in the benchmark is 1.4×10^{131}, and, in 9 cases, it cannot even be represented with a C `double`. Furthermore, enumerating all paths of an LDD succeeds only in 155 cases.

WB+DVO and WB+SVO performed the best for hard and easy cases, respectively. Thus, for hard cases, the extra time spent searching for a better order pays off.

The chart in Fig. 13 shows the total time (y-axis, in log-scale) for white-box QELIM, sorted individually in increasing order. For harder cases, the time increases gradually with DVO (solid and dotted series, and WBWB1, which is not shown, but is similar to WB+DVO). However, the time jumps dramatically for SVO (dashed series). This indicates that DVO is more robust than SVO during QELIM. This is further illustrated by the chart in Fig. 14 that compares WBMVQE under SVO and DVO. In the chart, y-axis is the total time in log-scale, and the x-axis of both series is sorted by WB+SVO time. For

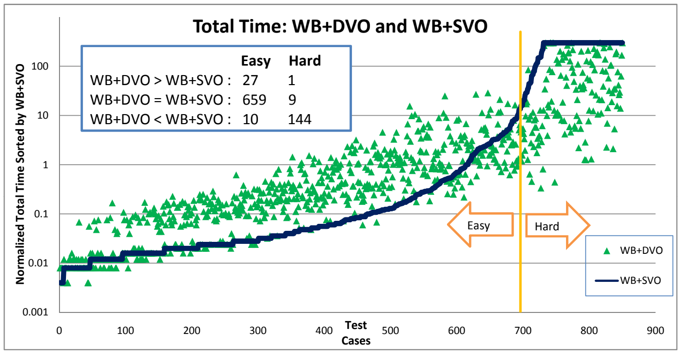

Fig. 14. Total time comparison between WB+SVO and WB+DVO.

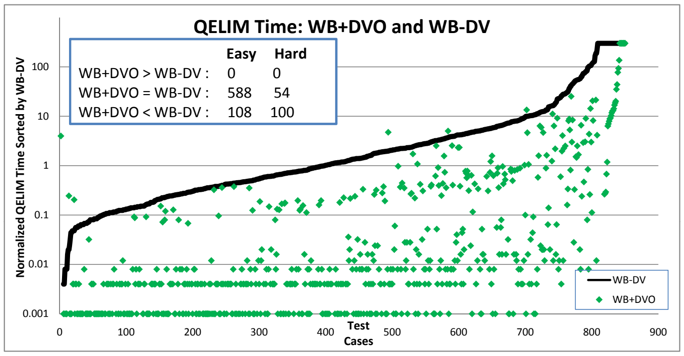

Fig. 15. QELIM time comparison between WB-DV and WB+DVO.

hard instances, in many cases WB+DVO is several orders of magnitude faster than WB+SVO. Note that due to log-scale this difference is much more significant than it appears.

The chart in Fig. 15 compares the effect of multiple-variable elimination heuristic (lines 4–8 of WBMVQE). In the chart, y-axis is the QELIM time in log-scale, and the x-axis of both series is sorted by WB-DV time. From the chart, WB+DVO almost always outperforms WB-DV, often by several orders of magnitude. Overall, WB+DVO is about 5 times faster than WB-DV for QELIM.

We also compared WBQE1 and WBQE2 (i.e., WB+DVO and WBWB1). WBWB1 timed out in 2 more cases than WB+DVO. Other than that, the two algorithms performed similarly.

We are not aware of other tools for existential quantification of arbitrary LA formulas. Thus, we have only compared between our own implementations. LDDs can be used as an SMT-solver for LA: to decide whether a formula is satisfiable first build an LDD for the formula and then quantify out all numeric variables. In our preliminary experiments this approach was not competitive with current DPLL-style SMT-solvers. We have not pursued it further.

VI. Conclusion

In this paper, we have tackled the problem of space-efficient representation of LA formulas with support for Boolean operations and QELIM. This problem is at the heart of many program analysis tasks. To this end, we have extended Difference Decision Diagrams to Linear Arithmetic. Our key contributions are: support for Dynamic Variable Ordering, QELIM algorithms, an implementation inside CUDD, and empirical evaluation on a large benchmark derived from real programs.

Overall, we found that LDDs in combination with DVO and dynamic programming-based QELIM algorithm leads to an effective data structure for program analysis tasks.

We believe that LDDs is a good basis for a combined predicate and numeric abstract domain, in the style of [11]. We plan to explore this direction in future work.

References

[1] H. R. Andersen and H. Hulgaard. Boolean Expression Diagrams, 1997.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. *FMSD*, 10(2/3):171–206, 1997.

[3] A. Bik and H. Wijshoff. Implementation of Fourier-Motzkin Elimination. Technical Report 94-42, Dept. of Computer Sci., Leiden University, 1994.

[4] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[5] R. Bryant and Y.-A. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *DAC'95*, June 1995.

[6] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In *FMCAD'07*, pages 69–76, 2007.

[7] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS'04*, volume 2988 of *LNCS*, pages 168–176, 2004.

[8] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli. Dynamic Variable Reordering for BDD Minimization. In *EURO-DAC'93/EURO-VHDL'93*, pages 130–135, 1993.

[9] J. F. Groote and O. Tveretina. Binary Decision Diagrams For First-order Predicate Logic. *J. Log. Algebr. Program.*, 57(1-2):1–22, 2003.

[10] J. F. Groote and J. van de Pol. Equational Binary Decision Diagrams. In *LPAR'00*, pages 161–178, 2000.

[11] A. Gurfinkel and S. Chaki. Combining Predicate and Numeric Abstraction for Software Model Checking. In *FMCAD'08*, 2008.

[12] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV'06*, volume 4144 of *LNCS*, pages 413–426, 2006.

[13] K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Clock Difference Diagrams. *Nord. J. Comput.*, 6(3):271–298, 1999.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, March 2004.

[15] A. Miné. The Octagon Abstract Domain. *Higher Order Symbol. Comput.*, 19(1):31–100, 2006.

[16] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proceedings of 13th International Conference on Computer Science Logic*, volume 1683 of *LNCS*, pages 111–125, 1999.

[17] S. Panda, F. Somenzi, and B. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. In *ICCAD'94*, pages 628–631, 1994.

[18] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *ICCAD'93*, pages 42–47. IEEE, 1993.

[19] K. Strehl and L. Thiele. Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In *ICCAD'98*, pages 686–692, 1998.