NOTICE: This is the author's version of a work that was accepted for publication in Science of Computer Programming. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. A definitive version was subsequently published in Science of Computer Programming, 64(2007):115-139.

DOI: http://dx.doi.org/10.1016/j.scico.2006.03.009

The Octahedron Abstract Domain

Robert Clarisó [1,2]

Estudis d'Informàtica i Multimèdia
Universitat Oberta de Catalunya (UOC)
Barcelona, Spain

Jordi Cortadella [2,3]

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain

Abstract

An interesting area in static analysis is the study of numerical properties. Complex properties can be analyzed using *abstract interpretation*, provided that an adequate abstract domain is defined. Each domain can represent and manipulate a family of properties, providing a different trade-off between the precision and complexity of the analysis. The contribution of this paper is a new numerical abstract domain called *octahedron* that represents constraints of the form $(\sum x_i - \sum x_j \geq k)$. The implementation of octahedra is based on a new kind of decision diagrams called *Octahedron Decision Diagrams* (OhDD).

Key words: Abstract Interpretation; Numerical Abstract Domains; Relational Abstract Domains; Convex Polyhedra

[1] Part of this research was performed while this author was at the Departament de Llenguatges i Sistemes Informàtics of UPC.

[2] Supported by CICYT TIN 2004-07925 and the FPU grant AP2002-3862 from the Spanish Ministry of Education, Culture and Sports.

[3] Supported by a Distinction for the Promotion of Research by the Generalitat de Catalunya.

1 Introduction

Abstract interpretation [9] defines a generic framework for the static analysis of dynamic properties of a system. This framework can be used, for instance, to analyze termination or to discover invariants in programs automatically. However, each analysis requires the framework to be parametrized for the relevant domain of properties being studied, *e.g.* numerical properties.

There is a wide spectrum of numerical abstract domains that can be used to represent and manipulate properties. Some examples are intervals, octagons and convex polyhedra. Each domain provides a different trade-off between the precision of the properties that can be represented and the efficiency of the manipulation. An interesting problem in abstract interpretation is the study of new abstract domains that are sufficiently expressive to analyze relevant problems and allow an efficient implementation.

In this paper, a new numerical abstract domain called the *octahedron abstract domain* is described. It can represent conjunctions of restricted linear inequalities called *unit* inequalities, of the form $(\sum x_i - \sum x_j \geq k)$. A new kind of decision diagram called *Octahedron Decision Diagram* (OhDD) has been specifically designed to represent and manipulate this family of constraints efficiently. The implementation of the octahedron abstract domain allows some degree of optimization when the variables under study are non-negative. Several classes of interesting problems such as the study of the values of clocks or counters, or the analysis of the length or size of an element (string, list, ...) contain non-negative variables, so this optimization can be applied. Some examples of problems that can be solved using unit inequalities over non-negative variables are the analysis of timed systems [1, 18], the analysis of string length in C programs [12] and the discovery of bounds on the size of asynchronous communication channels. In other problems where variables are unconstrained, OhDD can still be used albeit with lower efficiency.

This paper is an extended version of the conference paper with the same title published in the *Static Analysis Symposium (SAS) 2004*. In this extended version, further details are provided about the following areas: the canonicity of the representation, the implementation of the operations, the use of OhDD with unrestricted variables and the potential benefits of dynamic reordering.

The remaining sections of the paper are organized as follows. Section 2 presents related work in the definition of numerical domains for abstract interpretation, and previous decision diagram techniques used to represent numerical constraints. Section 3 defines the numerical domain of octahedra, and Section 4 describes the data structure and its operations. This Section will also discuss different implementations depending on the class of variables being studied:

Table 1
A comparison of numerical abstract domains based on inequality properties.

Abstraction	**Reference**	**Constraints**	**Example**
Intervals	[9]	$(k_1 \leq x_i \leq k_2) \quad k_1, k_2 \in \mathbb{Q}$	$2 \leq x \leq 5$
Difference Bound Matrices (DBMs)	[11, 3]	$(k_1 \leq x_i \leq k_2) \quad k_1, k_2 \in \mathbb{Q}$ $(x_i - x_j \leq k) \quad k \in \mathbb{Q}$	$1 \leq x \leq 3$ $x - y \leq 5$
Octagons	[22]	$(\pm\, x_i \pm x_j \leq k) \quad k \in \mathbb{Q}$	$2 \leq x + y \leq 6$
Two-variables per-inequality	[30]	$(c_1 \cdot x_1 + c_2 \cdot x_2 \geq k)$ $c_1, c_2, k \in \mathbb{Q}$	$3x - 2y \geq 5$
Octahedra	This paper	$(\sum x_i - \sum x_j \geq k)$ $k \in \mathbb{Q}$	$x + z - y \geq 5$
Convex polyhedra	[10, 17]	$(\sum c_i \cdot x_i \geq k)$ $c_i, k \in \mathbb{Q}$	$x + 3y - z \geq 1$
Templates	[29]	Like convex polyhedra, but the set of possible coefficients c_i is established a priori.	

unconstrained variables or non-negative variables. In Section 5, some possible applications of the octahedron abstract domain are discussed, and some experimental results are presented. Finally, Section 6 draws some conclusions and suggests some future work.

2 Related Work

2.1 Numerical Abstract Domains

Abstract domain is a concept used to denote a computer representation for a family of constraints, together with the algorithms to perform the abstract operators such as union, intersection, widening or transfer functions. Several abstract domains have been defined for interesting families of numerical properties, such as *inequality*, *equality* or *modulo* properties. The octahedron abstract domain belongs to the first category, inequalities. Other abstract domains based on inequalities are *intervals*, *difference bound matrices*, *octagons*, *two-variables-per-inequality*, and *convex polyhedra*. An example of these abstract domains and their relation to octahedra can be seen in Table 1.

Intervals are a representation for constraints on the upper or lower bound for each variable, *e.g.* $(k_1 \leq x_i \leq k_2)$. Interval analysis is very popular due to its

simplicity and efficiency: an interval abstraction for n variables requires $O(n)$ space, and all operations require $O(n)$ time in the worst case. *Octagons* are an efficient representation for a system of inequalities on the sum or difference of variable pairs, *e.g.* $(\pm x_i \pm x_j \leq k)$ and $(x_i \leq k)$. The implementation of octagons is based on *difference bound matrices* (DBM), a data structure used to represent constraints on differences of pairs of variables, as in $(x_i - x_j \leq k)$ and $(x_i \leq k)$. Efficiency is an advantage of this representation: the spatial cost for representing constraints on n variables is $O(n^2)$, while the temporal cost is between $O(n^2)$ and $O(n^3)$, depending on the operation. *Convex polyhedra* are an efficient representation for conjunctions of linear inequality constraints. This abstraction is very popular due to its ability to express precise constraints. However, this precision comes with a very high complexity overhead, which is unbounded in theory but exponential in practice [10]. This complexity has motivated the definition of abstract domains such as *two-variables-per-inequality*, which try to retain the expressiveness of linear inequalities with a lower complexity. However, currently there is no experimental data comparing the performance or the precision of this abstract domain to that of convex polyhedra, so it is unclear whether the lower theoretical complexity is noticeable from a practical point of view.

The abstract domain presented in this paper, *octahedra*, also attempts to keep some of the flexibility of convex polyhedra with a lower complexity. Instead of limiting the number of variables per inequality, the coefficients of the variables are restricted to $\{-1, 0, +1\}$. From this point of view, octahedra provide a precision that is between octagons and convex polyhedra. The precision of octahedra and the two-variables-per-inequality domain are not comparable.

The abstract domains presented so far are restricted to a specific subclass of linear inequalities that is established in the definition of the domain. Instead, the domain of *template constraints* [29] allows the user to define the structure of the properties to be analyzed. This domain has a parameter: a "template" matrix T describing the potential invariants. This matrix has one column per variable, and one row for each potential invariant. The value of a position T_{ij} in this matrix describes the coefficient of the variable x_j in the i-th invariant. Then, all the operations of the abstract domain are posed as linear programming (LP) problems, using the template matrix to define the constraints of the problems. The number of LP problems to be solved is polynomial in terms of the size of the template, *i.e.* the number of potential invariants.

In some sense, template constraints subsume all the other abstract domains described in this section. However, the more specialized domains may be more efficient because they can exploit the structure of the constraints in the implementation. Furthermore, the template matrix may be very large, *e.g.* a $(3^n - 1) \times n$ matrix for an octahedron with n variables, or even infinite, *e.g.* in the two-variables-per-inequality domain.

2.2 Decision diagrams

One possible implementation of octahedra is based on decision diagrams. Decision diagram techniques have been applied successfully to several problems in different application domains. Binary Decision Diagrams (BDD) [5] provide an efficient mechanism to represent boolean functions. Zero Suppressed BDDs (ZDD) [21] are specially tuned to represent sparse functions more efficiently. Multi-Terminal Decision Diagrams (MTBDD) [15] represent functions from boolean variables to reals, $f : \mathbb{B}^n \to \mathbb{R}$.

The paradigm of decision diagrams has also been applied to the analysis of numerical constraints. Most of these approaches compare the value of numerical variables with constants or intervals, or compare the value of pairs of variables. Some examples of these representations are *Difference Decision Diagrams* (DDD) [23], *Region Encoding Diagrams* (RED) [35], *Numerical Decision Diagrams* (NDD) [13], and *Clock Difference Diagrams* (CDD) [4]. Although the individual constraints involve *a maximum of two* variables, these diagrams can encode conjunctions and disjunctions of these constraints. In other representations, each node encodes one complex constraint like a linear inequality. Some examples of these representations are *Decision Diagrams with Constraints* (DDC) [19] and *Hybrid-Restriction Diagrams* (HRD) [36]. Again, these diagrams can encode conjunctions and disjunctions of linear inequalities. Nevertheless, there is no sistematic procedure to combine the different constraints in the diagram, detect redundancies and simplify the representation. The Octahedron Decision Diagrams described in this paper use an innovative approach to encode linear inequalities. This approach, presented in Section 4, includes a procedure called *saturation* that combines and simplifies the inequalities of the diagram. However, contrary to other decision diagrams, there is a loss of precision when encoding the disjunction of constraints.

3 Octahedra

3.1 Definitions

The octahedron abstract domain is now introduced. In the same way as convex polyhedra, an octahedron abstracts a set of vectors in $\mathbb{Q}^n$ as a system of linear inequalities satisfied by all these vectors. The difference between convex polyhedra and octahedra is the family of constraints that are supported.

Definition 1 (Unit linear inequality) *A linear inequality is a constraint of the form* $(c_1 \cdot x_1 + \ldots + c_n \cdot x_n \geq k)$ *where the coefficients* c_i *are in* $\mathbb{Q}$ *and the*

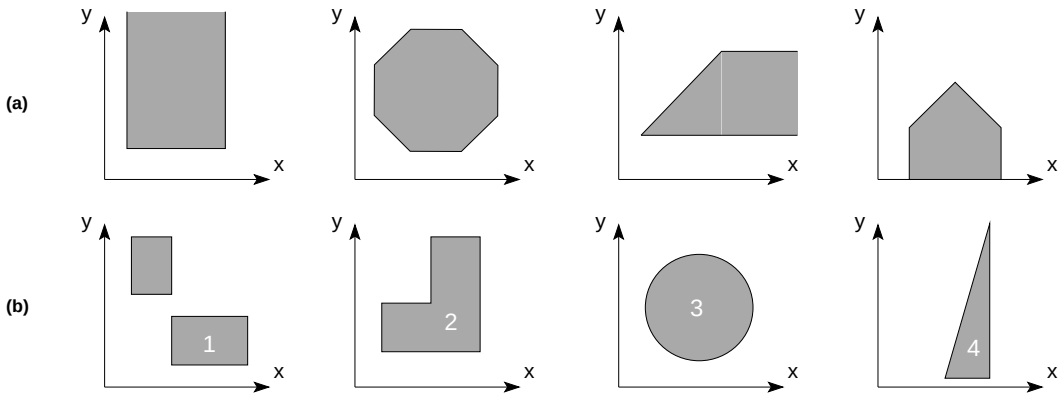


Fig. 1. Examples of (a) octahedra and (b) non-octahedra over two variables.

constant term k is in $\mathbb{Q} \cup \{-\infty\}$, e.g. $(3x+2y-z \geq -7)$. A linear inequality is called unit *if all coefficients are in $\{-1, 0, +1\}$, such as $(x + y - z \geq -7)$.*

Definition 2 (Octahedron) *An octahedron O over $\mathbb{Q}^n$ is the set of solutions to the system of m unit inequalities $O = \{X \mid AX \geq B\}$, with the vector $B \in (\mathbb{Q} \cup \{-\infty\})^m$ and the matrix $A \in \{-1, 0, +1\}^{m \times n}$. Octahedra satisfy the following properties:*

(1) Convexity: *An octahedron is a convex set,* i.e. *any segment between two points of the octahedron is fully within the octahedron.*
(2) Closed for intersection: *The intersection of two octahedra is also an octahedron.*
(3) Non-closed for union: *In general, the union of two octahedra might not be an octahedron.*

Figure 1(a) shows some examples of octahedra in a two-dimensional space. In Fig. 1(b) there are several regions of space which are not octahedra, either because they are not connected (1), they are not convex (2), they cannot be represented by a finite system of linear inequalities (3), or because they can be represented as a system of linear inequalities, but not unit linear inequalities (4). Notice that in two-dimensional space all octahedra are octagons; octahedra can only show a better precision than octagons in higher-dimensional spaces.

During the remaining of this paper, we will use C to denote a vector in $\{-1, 0, +1\}^n$ where n is the number of variables. Therefore, given a set of variables X, the expression $(C^T X \geq k)$ denotes the unit linear inequality $(c_1 \cdot x_1 + \ldots + c_n \cdot x_n \geq k)$.

Lemma 3 *An octahedron over n variables can be represented by at most $3^n - 1$ non-redundant inequalities.*

PROOF. Each variable can have at most three different coefficients in a unit linear inequality. For n variables, there are at most 3^n possible combinations of unit coefficients, where one of them is an irrelevant unit inequality with 0 in all coefficients. This means that if an octahedron has more than $3^n - 1$ unit inequalities, some of them will only differ in the constant term, *e.g.* $(C^T X \geq k_1)$ and $(C^T X \geq k_2)$. One of these inequalities is definitely redundant, namely, the one with the smaller constant. □

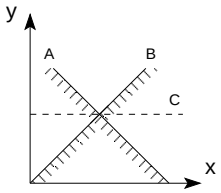
A problem when dealing with convex polyhedra and octahedra is the lack of canonicity of the systems of linear inequalities: the same polyhedron/octahedron can be represented with different systems of inequalities. For example, both systems of inequalities $(x = 3) \wedge (y \geq 5)$ and $(x = 3) \wedge (x + y \geq 8)$ define the same octahedron with different inequalities. Given a convex polyhedron, there are algorithms to minimize the number of constraints in a system of inequalities, *i.e.* removing all constraints that can be derived as linear combinations of others. However, in the previous example both representations are minimal and even then, they are different. Although it is possible to define a canonical form for convex polyhedra [2], its complexity makes it impractical. Regarding octahedra, a canonical form for octahedra can be defined using the result of Lemma 3. Even though the number of inequalities of this canonical form makes an explicit representation impractical, symbolic representations based on decision diagrams can manipulate sets of unit inequalities efficiently.

Definition 4 (Canonical form of octahedra) *The canonical form of an octahedron $O \subseteq \mathbb{Q}^n$ is either (i) the empty octahedron or (ii) a system of $3^n - 1$ unit linear inequalities, where in each inequality $(C^T X \geq k)$, k is the tightest bound satisfied by O. This bound k may be $-\infty$,* i.e. *the inequality is unbounded in the octahedron.*

Theorem 5 *Two octahedra O_1 and O_2 represent the same subset of $\mathbb{Q}^n$ if and only if they both have the same canonical form.*

Proof We need to prove both directions of the if and only if. **(→)** Given a constraint $(C^T X \geq k)$, there is a single tightest bound to that constraint. Therefore, if two octahedra are equal, they will have the same bound for each possible linear constraint, and therefore, the same canonical form. □

(←) From its definition, an octahedron is completely characterized by its system of inequalities. If two octahedra O_1 and O_2 have the same canonical form, then they satisfy exactly the same system of inequalities and therefore are equal. □



$$
\begin{aligned}
A = \{\ & (-\infty \le x \le +\infty)\ \wedge\ (-\infty \le y \le +\infty) \\
& \wedge\ (-\infty \le x - y \le +\infty)\ \wedge\ (-\infty \le x + y \le 6)\ \} \\
B = \{\ & (-\infty \le x \le +\infty)\ \wedge\ (-\infty \le y \le +\infty) \\
& \wedge\ (0 \le x - y \le +\infty)\ \wedge\ (-\infty \le x + y \le +\infty)\ \} \\
A \cap B = \{\ & (-\infty \le x \le +\infty)\ \wedge\ (-\infty \le y \le +\infty) \\
& \wedge\ (0 \le x - y \le +\infty)\ \wedge\ (-\infty \le x + y \le 6)\ \} \\
\text{canonical}(A \cap B) = \{\ & (-\infty \le x \le +\infty)\ \wedge\ (-\infty \le y \le \mathbf{3}) \\
& \wedge\ (0 \le x - y \le +\infty)\ \wedge\ (-\infty \le x + y \le 6)\ \}
\end{aligned}
$$

Fig. 2. An example where $A \cap B$ is not in canonical form.

Theorem 6 *Let A and B be two non-empty octahedra represented by systems of inequalities of the form $(C^T X \geq k_a)$ and $(C^T X \geq k_b)$ for all $C \in \{-1, 0, +1\}^n$, which may be non-canonical. The intersection $A \cap B$ is defined by the system of inequalities $(C^T X \geq \max(k_a, k_b))$, which might be in non-canonical form even if the input systems were canonical.*

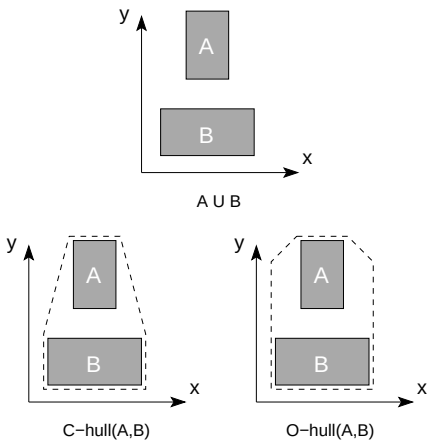
PROOF. Any point $P \in \mathbb{Q}^n$ that satisfies $(C^T P \geq \max(k_a, k_b))$ will also satisfy $(C^T P \geq k_a)$ and $(C^T P \geq k_b)$. Therefore, any point P satisfying the new system of inequalities will also appear in both A and B. □

Figure 2 shows an example where the intersection of two octahedra is not in canonical form, even though the original octahedra were in canonical form. The intersection of $A = \{\ (x + y \leq 6)\ \}$ and $B = \{\ (x \geq y)\ \}$ satisfies the inequality $(y \leq 3)$. However, this constraint does not appear simply by taking the maximum constant for all the constraints of A and B. Instead, this implicit constraint is implied by the other constraints of the intersection, *i.e.* it is computed as a linear combination of other constraints of $A \cap B$.

Lemma 7 *An octahedron B is an upper approximation of an octahedron A, noted $A \subseteq B$, iff either (i) A is empty or (ii) for any constraint $(C^T X \geq k_a)$ in the canonical form of A, the equivalent constraint $(C^T X \geq k_b)$ in B (which may be in canonical form or not) has a constant term k_b such that $(k_a \geq k_b)$.*

PROOF. By definition, $A \subseteq B$ iff $A = A \cap B$. This lemma is a direct consequence of this property and Theorem 6. □

Definition 8 (Convex and octahedral hull) *The* convex hull (C-hull) *of two convex sets A and B is the intersection of all convex sets that include both A and B. The* octahedral hull (O-hull) *of two octahedra A and B is the intersection of all octahedra that include both A and B.*



$$
\begin{aligned}
A &= \{(2 \le x \le 4) \wedge (4 \le y \le 7)\} \\
B &= \{(1 \le x \le 5) \wedge (1 \le y \le 3)\} \\
\mathsf{C\text{-}hull} &= \{(1 \le x \le 5) \wedge (1 \le y \le 7) \wedge \\
&\qquad (4x - y \ge 1) \wedge (-4x - y \ge -23)\} \\
\mathsf{O\text{-}hull} &= \{(1 \le x \le 5) \wedge (1 \le y \le 7) \wedge \\
&\qquad (x - y \ge 5) \wedge (-x - y \ge -11)\}
\end{aligned}
$$

Fig. 3. Overapproximations of the union: convex (C) and octahedral (O) hull.

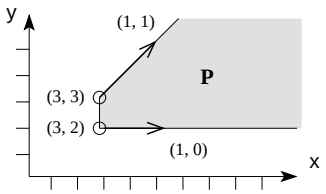
Figure 3 shows an example of the convex and octahedral hulls of two octahedra A and B. Notice that the convex hull is always an upper approximation of the union, and the octahedral hull is always an upper approximation of the convex hull, i.e. $A \cup B \subseteq \mathsf{C\text{-}hull}(A, B) \subseteq \mathsf{O\text{-}hull}(A, B)$.

Theorem 9 *Let A and B be two non-empty octahedra whose canonical form are respectively $(C^T X \ge k_a)$ and $(C^T X \ge k_b)$ for all $C \in \{-1, 0, +1\}^n$. Then, the octahedral hull $\mathsf{O\text{-}hull}(A, B)$ is defined by the system of inequalities $(C^T X \ge \min(k_a, k_b))$.*

PROOF. Given a bound k for one inequality $(C^T X \ge k)$ of $\mathsf{O\text{-}hull}(A, B)$, the proof can be split into two parts: proving that $k \le \min(k_a, k_b)$ and proving that $k \ge \min(k_a, k_b)$.

As the octahedral hull includes A and B, all points $P \in A$ and $P \in B$ should also be in $\mathsf{O\text{-}hull}(A, B)$. Therefore, any point in A or B should satisfy the constraints of $\mathsf{O\text{-}hull}(A, B)$. Given a constraint $(C^T X \ge k)$, it is known that points in A satisfy $(C^T X \ge k_a)$ and points in B satisfy $(C^T X \ge k_b)$. If both sets of points must satisfy the constraint in $\mathsf{O\text{-}hull}(A, B)$, then k must satisfy $k \le \min(k_a, k_b)$.

On the other side, the octahedral hull is the least octahedron that includes A and B. Therefore, the bounds of each constraint should be as tight as possible, i.e. as large as possible. If we know that $k \le \min(k_a, k_b)$ should hold for a given unit inequality, the tightest bound for that inequality is precisely $k = \min(k_a, k_b)$. As a *corollary*, the octahedral hull computed in this way is in canonical form. □



System of generators

$$P = \{\lambda_1 \cdot (3,3) + \lambda_2 \cdot (3,2) + \mu_1 \cdot (1,1) + \mu_2 \cdot (1,0) \mid \lambda_1 \geq 0, \lambda_2 \geq 0, \mu_1 \geq 0, \mu_2 \geq 0, \lambda_1 + \lambda_2 = 1\}$$

System of constraints

$$P = \{(x,y) \mid (y \geq 2) \wedge (x \geq 3) \wedge (x - y \geq 0)\}$$

Fig. 4. An example of a convex polyhedron (shaded area) and its double description.

3.2 Computing the Canonical Form

3.2.1 Preliminaries: Dual Representations of Polyhedra

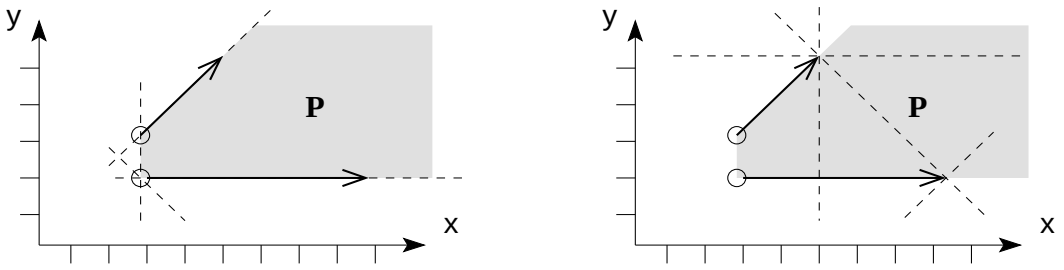
Any convex polyhedron has two dual representations: the system of *constraints* and the system of *generators* [6]. The system of constraints defines the polyhedron as a conjunction of linear inequalities. Meanwhile, the system of generators defines the polyhedron as the convex combination of a set of points (vertices) and the positive linear combination of a set of rays (vectors). Both representations are useful in the implementation of the abstract domain of convex polyhedra, as operations like intersection are trivial in the constraint representation, while the convex hull is naturally expressed in the generator representation. Also, both representations can be minimized when the dual is available. Figure 4 shows an example of a convex polyhedron and its double description.

There is a procedure that translates from one representation into the other. This procedure was described in [6], and further improved in [14, 34]. Given a system of c constraints over $\mathbb{Q}^d$, the computation of the dual representation requires $O(c^{\lfloor \frac{d}{2} \rfloor})$ time [20]. Furthermore, even if we consider only minimized representations, the size of a representation can grow exponentially with this translation. For example, an hypercube in d-dimensions is defined by $2d$ constraints but it has 2^d vertices. The worst case is achieved by *cyclic polytopes* [20], which have up to $O(c^{\lfloor \frac{d}{2} \rfloor})$ vertices with a system of inequalities with c constraints.

3.2.2 The algorithm

The computation of the canonical form of octahedra will be based on the generator representation of the octahedron. The pseudocode for a possible algorithm is presented in Figure 6. The output of the algorithm should be either the empty octahedron or the bounds for each of the $3^d - 1$ unit inequalities of the canonical form.

The algorithm is based on the two following observations. First, a ray is a



$$x \geq 3$$
$$y \geq 2$$
$$x + y \geq 5$$
$$x - y \geq 0$$

(a)

$$-x \geq -\infty$$
$$-y \geq -\infty$$
$$-x - y \geq -\infty$$
$$-x + y \geq -\infty$$

(b)

Fig. 5. (a) Bounded unit inequalities, (b) unbounded unit inequalities.

vector that represents a direction of unbounded growth in the octahedron, *i.e.* no constraint can impose a bound that "crosses" the ray. Therefore, a unit inequality will be unbounded in an octahedron O if it crosses one of the rays of O. And second, if a unit inequality is bounded, its tightest bound will occur in one of the vertices of the octahedron. Figure 5 shows an example of these observations in the system of generators from Figure 4. All the unit inequalities that cross the rays are unbounded as shown in Fig. 5(b). On the other hand, all the bounded inequalities achieve the tightest bound in one of the vertices of the octahedron, as shown in Fig. 5(a).

If a unit inequality is defined as $(C^T X \geq k)$, it is possible to determine whether it crosses a ray r using the scalar product: if $(C^T \cdot r < 0)$, then the unit inequality crosses the ray. For example, the ray $(1, 0)$ is crossed by the inequality $(-x + y \geq k)$ because $(-1, 1)^T \cdot (1, 0) = (-1) \cdot 1 + 1 \cdot 0 = -1$. Also, given a vertex v, the tightest bound k achieved by a unit inequality $(C^T X \geq k)$ can be computed as $k = C^T \cdot v$. For instance, the bound of the inequality $(x - y \geq k)$ in a vertex $(3, 2)$ is $(1, -1)^T \cdot (3, 2) = 1 \cdot 3 + (-1) \cdot 2 = 1$.

3.2.3 Complexity

For an octahedron over $\mathbb{Q}^d$ with v vertices and r rays, the algorithm requires $O(d \cdot (v + r) \cdot 3^d)$ time. The space requirement is $O(d \cdot (v + r + 3^d))$ in the worst case. We can establish upper bounds for v and r in terms of d. For example, we know that $r \leq 2d$ as further rays would be redundant in terms of positive linear combinations. With respect to v, a possible upper bound is $v \leq (3^d - 1)^{\lfloor \frac{d}{2} \rfloor}$, as there are at most $3^d - 1$ non-redundant inequalities. It is possible that a better bound can be established by taking into account that all constraints are unit inequalities, but currently this is an open problem.

Function *Canonical_form*(O)
Input: An octahedron O defined as a system of inequalities over d variables.
Output: The canonical form of O, which is either the empty octahedron (if O is empty) or the canonical system of $3^d - 1$ unit inequalities for O.

```
Compute the system of generators of O: the set of vertices V and rays R
if V = ∅ then
  return empty
endif
{ Compute the bound of each inequality in the canonical form }
for each unit inequality C ∈ {−1, 0, +1}^d in O do
  { Evaluate the rays for this inequality }
  unbounded := false
  for each ray r ∈ R do
    unbounded := (C^T · r < 0)?
    if unbounded then
      break
    endif
  endfor
  if unbounded then
    tightest_bound := −∞
  else
    { Evaluate all the vertices for this inequality }
    { Keep the tightest bound satisfied by all vertices }
    tightest_bound := +∞
    for each vertex v ∈ V do
      new_bound := C^T · v
      { (C^T · X ≥ new_bound) holds for this vertex }
      tightest_bound := min ( new_bound, tightest_bound )
      { (C^T · X ≥ tightest_bound) holds for all vertices visited so far }
    endfor
  endif
  bounds[ C ] = tightest_bound
  { bounds[ C ] = k means (C^T · X ≥ k) }
endfor
return bounds
```

Fig. 6. Pseudocode to compute the canonical form of an octahedron.

The complexity of these algorithms makes the computation of the canonical form impractical. Even though the canonical form is useful to define the semantics of the operations, it is not practical for the implementation. Instead of the canonical form, we will define a relaxed version called the *saturated form*. Operations performed using the saturated form may lose some precision, while

always yielding an upper approximation of the exact result.

3.2.4 Canonicity in other abstract domains

The problem of computing a canonical form appears in other abstract domains as well. In domains based on inequality properties, it might be possible to compute a tighter bound to a constraint by combining several other constraints. A possible way to compute the canonical form consists in calculating all possible combinations of constraints.

In the case of DBMs and octagons, only pairwise combinations of constraints should be considered. The algorithms that compute the canonical form in these abstract domains require $O(n^3)$ time [11, 22] when variables are rational.

A canonical form for template constraints can also be computed. This computation requires solving c linear programming (LP) problems, where c is the number of linear inequalities in the template. Given a template for all possible unit constraints, it is also possible to use this method to compute the canonical form of an octahedron. However, this method requires solving $3^d - 1$ LP problems so, like the algorithm presented in this paper, it is impractical except for small values of d.

Convex polyhedra also have a canonical form [2], but the complexity of its computation is very high. For instance, it involves discovering all redundant constraints in the system of inequalities, which can be posed as several LP problems.

3.3 Abstractions of Octahedra

As it was shown in the previous section, the canonical form of an octahedron provides a useful mechanism to define operations such as the test for inclusion, the intersection or the octahedral hull. However, it is not convenient to implement the operations using the canonical form.

On the other hand, octahedra are defined in the context of abstract interpretation of numerical properties. In this context, the problem is the abstraction of a set of values in $\mathbb{Q}^n$, and the main concern is ensuring that the abstraction is an *upper approximation* of the concrete set of values. Thus, as long as an upper approximation can be guaranteed, an exact representation of octahedra is not required, as octahedra are already abstractions of more complex sets. Keeping this fact in mind, efficient algorithms that operate with upper approximations of the canonical form can be designed.

The first step is the definition of a relaxed version of the canonical form, which is called *saturated* form [4]. While the canonical form has the tightest bound in each of its inequalities, the bounds in the saturated form may be more relaxed. A system of unit inequalities is in saturated form as long as the bounds imposed by the sum of any pair of constraints in the system appear explicitly. For example, a saturated form of the octahedron $(a \geq 3) \wedge (b \geq 0) \wedge (c \geq 0) \wedge (b - c \geq 7) \wedge (a + b \geq 8) \wedge (a + c \geq 6)$ can be defined by the following system of inequalities:

$$(a \geq 3) \wedge (b \geq 7) \wedge (c \geq 0) \wedge (a + b \geq 10) \wedge (a + c \geq 6) \wedge (b + c \geq 7)$$
$$\wedge\ (b - c \geq 7) \wedge (a + b - c \geq 10) \wedge (a + b + c \geq 13)$$

where the constraints with a bound of $-\infty$ have been removed for brevity. In this example, saturation has exposed explicitly that $(a + b \geq 10)$. This inequality is the sum of $(a \geq 3)$, $(b - c \geq 7)$ and $(c \geq 0)$.

A saturated form O^* of an octahedron $O = \{\ X \mid AX \geq B\ \}$ can be computed using an iterative procedure called *saturation*. At each step of this procedure, a sum between two unit inequalities is computed. If this sum has a tighter bound than the one already known, the bound is updated, and so on until a fixpoint is reached. The formal description of saturation is the following:

(1) Initialize the system of $3^n - 1$ unit inequalities for all possible values of the coefficients $C \in \{-1, 0, +1\}^n$. The bound k of a given inequality $(C^T X \geq k)$ is chosen as:

$$k = \begin{cases} b & \text{if } C^T X \geq b \text{ appears in } AX \geq B \text{ and } C \not\geq 0^n. \\ -\infty & \text{otherwise} \end{cases}$$

(2) Select two inequalities $C_1^T X \geq k_1$ and $C_2^T X \geq k_2$ such that $k_1 > -\infty$ and $k_2 > -\infty$. Let us define $C_* = C_1 + C_2$ and $k_* = k_1 + k_2$.
(3) If $C_* \notin \{-1, 0, +1\}^n$ return to step 2.
(4) If $C_*^T X \geq k$ appears in the system of inequalities with $k \geq k_*$, return to step 2.
(5) Replace the inequality $C_*^T X \geq k$ by $C_*^T X \geq k_*$.
(6) Repeat steps 2-5 until:
- A fixpoint is reached *or*
- An inequality $C_*^T X \geq k$ with $C = 0^n$ and $k > 0$ is found. In this case, the octahedron is empty.

[4] In the context of a system of constraints, the term *saturated* is usually used with a different meaning: to denote constraints which cannot become tighter without changing the set of solutions. In this paper we will only use the term to refer to the saturated form, even though some inequalities in this form may be non-saturated.

The implementation of the saturation algorithm will differ slightly from this definition: in step 2, instead of choosing a pair of inequalities and computing the sum, the implementation will select *all possible pairs of unit inequalities* and compute the sum simultaneously. This operation can be performed efficiently in the decision diagram. In any case, it does not affect the termination of the algorithm.

Theorem 10 *Let* $O = \{X \mid AX \geq B\}$ *be a non-empty octahedron. The* saturation *algorithm applied to* O *terminates.*

PROOF. Each step of the saturation algorithm defines a tighter bound for an inequality of the octahedron. The new inequality $(C_3^T X \geq k'_3)$ is obtained from two previously known inequalities $(C_1^T X \geq k_1)$ and $(C_2^T X \geq k_2)$, so that $C_3 = C_1 + C_2$ and $k'_3 = k_1 + k_2$, and $k'_3 > k_3$, where k_3 is the previously known bound for the inequality. If inequalities 1 and 2 were computed in previous rounds of the saturation algorithm, this dependency chain can be expanded, e.g. if inequality 2 comes from inequalities 4 and 5, then $C_3 = C_1 + C_4 + C_5$ and $k'_3 = k_1 + k_4 + k_5$. Non-termination of the saturation algorithm implies that there will be infinitely many sums of pairs of inequalities. Ignoring the bound k, there are only finitely many inequalities over n variables. Therefore, it is always possible to find a step that computes a bound k'_j that depends on a previously known bound k_j, i.e. $C_j = C_j + \sum C_l$ and $k'_j = k_j + \sum k_l$. As $C_j - C_j = \sum C_l = 0^n$ and $k'_j - k_j = \sum k_l > 0$, the linear combination $((\sum C_l)^T X \geq (\sum k_l))$ is equivalent to $(0 > 0)$, which implies that O is empty. □

The fixpoint in saturation *may not be reached* if the octahedron is empty. For example, the octahedron in Fig. 7(a) is empty because the sum of the last four inequalities is $(0 \geq 4)$. The saturation algorithm applied to this octahedron does not terminate. Adding the constraints in top-down order allows the saturation algorithm to produce $(x_2 - x_4 \geq 5)$, which can again be used to produce $(x_2 - x_4 \geq 9)$ and so on. Even then, the saturation algorithm is used to perform the emptiness test because of three reasons:

- There are special kinds of octahedra for which termination is guaranteed. For instance, if all inequalities describe constraints between symbols (all constant terms are zero), saturation is guaranteed to terminate. This occurs because any sum among unit constraints will either leave a constant as 0, or replace a $-\infty$ constant by a 0.
- The conditions required to build an octahedron for which the saturation algorithm does not terminate are complex and artificial, and therefore we expect them to occur rarely in practical examples. Note that non-termination may arise only for *some* systems of unit inequalities that define an empty octahedron.

- Another reason is the good behavior of the saturation procedure in practical examples. If the input is an octahedron obtained by performing minor changes to previously saturated octahedra, *e.g.* the intersection of two saturated octahedra, typically very few iterations are required to reach the fixpoint. This makes the prediction of non-termination possible in practice.

Even if the saturation algorithm terminates, in some cases it might fail to discover the tightest bound for an inequality. For example, in the octahedron in Fig. 7(b), saturation will fail to discover the constraint $(x_1 - x_2 + x_3 + x_4 + x_5 + x_6 \geq 6)$, as any sum of two inequalities will yield a non-unit linear inequality. Therefore, given a constraint $(C^T X \geq k_s)$ in the saturated form, the bound k_c for the same inequality in the canonical form may be different, $k_c \not\leq k_s$. But $k_c \geq k_s$ always holds, as k_c is the tightest bound for that inequality. In this sense, the result will always be an upper approximation of the exact canonical result, as $k_c \geq k_s$ is the exact definition for upper approximation of octahedra (Lemma 7)

Several operations which have been defined for the canonical form can also be used in the saturated form. For instance, the definition of the intersection in Theorem 6 did not require that the operands were in canonical form. Regarding the union, we were able to prove in Theorem 9 that the octahedral hull is in canonical form if the two initial octahedra are in canonical form. A similar result holds for octahedra in saturated form.

Lemma 11 *Given two octahedra A and B in saturated form, the union implemented as $(C^T X \geq \min(k_a, k_b))$ is also saturated.*

PROOF. The result can only be non-saturated if we can find three inequalities $(C_1^T X \geq k_1)$, $(C_2^T X \geq k_2)$ and $(C_3^T X \geq k_3)$ such that the first two can be combined to produce the third $(C_1 + C_2 = C_3)$ with a tighter bound $(k_1 + k_2 > k_3)$. Considering the relationship between these bounds and the bounds of A and B in the same inequalities C_1, C_2 and C_3, we can define $k_1 = min(k_1^a, k_1^b)$, $k_2 = min(k_2^a, k_2^b)$, $k_3 = min(k_3^a, k_3^b)$. As A is saturated, we know that $(k_3^a \geq k_1^a + k_2^a)$, and conversely for B, $(k_3^b \geq k_1^b + k_2^b)$. Therefore, we know that $k_3 = min(k_3^a, k_3^b) \geq min(k_1^a + k_2^a, k_1^b + k_2^b)$. Proving that the result is saturated is then reduced to checking that $(k_1 + k_2 \leq k_3)$, which is equivalent to: $min(k_1^a, k_1^b) + min(k_2^a, k_2^b) \leq min(k_1^a + k_2^a, k_1^b + k_2^b)$. This property can be checked using a proof by cases. $\square$

$$
\begin{array}{rcl}
+x_2 \qquad -x_4 & \geq & 1 \\
-x_1 - x_2 + x_3 + x_4 + x_5 - x_6 & \geq & 1 \\
+x_1 - x_2 - x_3 + x_4 - x_5 + x_6 & \geq & 1 \\
+x_1 + x_2 + x_3 - x_4 - x_5 - x_6 & \geq & 1 \\
-x_1 + x_2 - x_3 - x_4 + x_5 + x_6 & \geq & 1
\end{array}
$$

(a)

$$
\begin{array}{l}
+x_1 - x_2 - x_3 + x_4 \geq 1 \\
-x_1 - x_2 + x_3 + x_5 \geq 2 \\
+x_1 + x_2 + x_3 + x_6 \geq 3
\end{array}
$$

(b)

Fig. 7. (a) Empty octahedron where the saturation algorithm does not terminate and (b) Non-empty octahedron where the saturated and canonical form are different.

3.4 Abstract Semantics of the Operators

In order to characterize the octahedron abstract domain, the abstract semantics of the abstract interpretation operators must be defined. Intuitively, this abstract semantics is defined as simple manipulations of the saturated form of octahedra. All operations are guaranteed to produce upper approximations of the exact result, as it was justified in section 3.3. Some operations like the intersection can deal with non-saturated forms without any loss of precision, while others like the union can only do so at the cost of additional over-approximation.

In the definition of the semantics, A and B will denote octahedra, whose saturated forms contain inequalities of the form $(C^T X \geq k_a)$ and $(C^T X \geq k_b)$, respectively.

- **Intersection** $A \cap B$ is represented by a system of inequalities where $(C^T X \geq \max(k_a, k_b))$, which might be in non-saturated form.
- **Union** $A \cup B$ is approximated by the saturated form $(C^T X \geq \min(k_a, k_b))$. Lemma 11 proves that the result is saturated.
- **Inclusion** Let A and B be two octahedra. If $k_a \geq k_b$ for all inequalities in the saturated form of A and B (which may be in saturated form or not), then $A \subseteq B$. Notice that the implication does not work in the other direction, i.e. if $k_a \not\geq k_b$ then we don't know whether $A \subseteq B$ or $A \not\subseteq B$.
- **Widening** $A \nabla B$ is defined as the octahedron with inequalities $(C^T X \geq k)$ such that k:

$$
k = \begin{cases} -\infty & \text{if} \quad k_a > k_b \\ k_a & \text{otherwise} \end{cases}
$$

  As established in [22], the result should *not* be saturated in order to guarantee convergence in a finite number of steps.
- **Extension** An octahedron O can be extended with a new variable y by modifying the constraints of its saturated form O^*. Let $(c_1 \cdot x_1 + \ldots + c_n \cdot x_n \geq k)$ be a constraint of O^*, the inequalities that will appear in the saturated

form of the extension are:

$$\begin{array}{lll} c_1 \cdot x_1 + \ldots + c_n \cdot x_n - 1 \cdot y & \geq & -\infty \\ c_1 \cdot x_1 + \ldots + c_n \cdot x_n + 0 \cdot y & \geq & k \\ c_1 \cdot x_1 + \ldots + c_n \cdot x_n + 1 \cdot y & \geq & -\infty \end{array}$$

If the new variable is known to be non-negative, the last constraint can be changed to a more precise ($c_1 \cdot x_1 + \ldots + c_n \cdot x_n + 1 \cdot y \geq k$) as the known bound k cannot be decreased by adding a non-negative value.

- **Projection** A projection of an octahedron O removing a dimension x_i can be performed by removing from its saturated form O^* all inequalities where x_i has a coefficient that is not zero.
- **Unit linear assignment** A unit linear assignment $[x_i := \sum_{j=1}^{m} c_j \cdot x_j]$ with coefficients $c_i \in \{-1, 0, +1\}$ can be defined using the following steps:
  · Extend the octahedron with a new variable t.
  · Intersect the octahedron with the octahedron ($t = \sum_{j=1}^{m} c_j \cdot x_j$)
  · Project the variable x_i.
  · Rename t as x_i.

3.5 Impact of the conservative inclusion test

Several operations defined using the saturated form cause a loss of precision because they do not use the tightest possible bound in each inequality. This loss of precision is always an upper approximation: the most precise result is included in the octahedron produced by the operators. In the context of abstract interpretation, upper approximations are acceptable: instead of studying the exact set of states of a system, an upper approximation of state space is computed. A possible way to perform this computation is called *increasing chain*: initially, our state space S^0 only contains the initial states. Each iteration after the first attempts to discover new reachable states that can be accessed from previously reached states. The new states are added to state space, which grows after each iteration: ($S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$). The analysis terminates when a fixpoint is reached, *i.e.* when $S_{i+1} \subseteq S_i$.

However, the inclusion operator defined for saturated forms is not accurate: even if $A \subseteq B$, the answer may be inconclusive when the saturation procedure fails to discover the tightest bound for an inequality. In this section, we discuss whether this has any effect on the termination of an abstract interpretation analysis. Our claim is that even with this inclusion operator, the analysis can be guaranteed to terminate, even though there may be a loss of precision.

Let us consider the following scenario: after the iteration $i+1$ of an increasing analysis, the fixpoint $S_{i+1} \subseteq S_i$ has been reached but the test of inclusion does not give a conclusive answer. In this situation, there must be one constraint

$(C^T X \geq k)$ in the solutions such that $k_{i+1} < k_i$, where k_j is the bound of the inequality in S_j. The saturated form of S_{i+1} is unable to compute the tightest bound k_i for that inequality. However, after another iteration $(i+2)$, the state space S_{i+2} will satisfy $S_{i+1} \subseteq S_{i+2}$, because the analysis is an increasing hain. Therefore, the coefficient k_{i+2} will always be $k_{i+2} \leq k_{i+1}$. Hence, in S_{i+2} it does not matter that the saturated form is unable to compute the tightest bound for $(C^T X \geq k)$. We can conclude that the analysis will terminate, even though more iterations may be required. However, in practical examples, this theoretical scenario does not seem to arise, as constraints tend to be generated in a structured way that allows saturation to obtain good approximations of the exact canonical form.

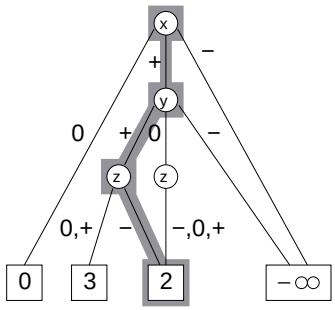
4 Octahedra Decision Diagrams

4.1 Overview

The constraints of an octahedron can be represented compactly using a specially devised decision diagram representation. This representation is called *Octahedron Decision Diagram* (OhDD). Intuitively, it can be described as a Multi-Terminal Zero-Suppressed Ternary Decision Diagram:

- *Ternary*: Each non-terminal node represents a variable x_i and has three output arcs, labelled as $\{-1, 0, +1\}$. Each arc represents a coefficient of x_i in a linear constraint.
- *Multi-Terminal* [15]: Terminal nodes can be constants in $\mathbb{R} \cup \{-\infty\}$. The semantics of a path σ from the root to a terminal node k is the linear constraint $(c_1 \cdot x_1 + c_2 \cdot x_2 + \ldots + c_n \cdot x_n \geq k)$, where c_i is the coefficient of the arc taken from the variable x_i in the path σ.
- *Zero-Suppressed* [21]: If a variable does not appear in any linear constraint, it also does not appear in the OhDD. This is achieved by using special reduction rules as it is done in Zero-Suppressed Decision Diagrams.

The reduction rules of decision diagrams have an essential role: ensuring that the representation is canonical and efficient. In the context of octahedra, canonicity means that saturated octahedra with the same system of inequalities are encoded by the same OhDD. Furthermore, a careful choice of reduction rules may decrease the size of the decision diagram, improving both memory and CPU time for all operations. In the case of OhDD, two reduction rules are defined: one for octahedra with unbounded variables and another specially tuned for octahedra with non-negative variables. In both cases, the overall manipulation of OhDD is the same, with only small changes in the implementation. These issues will be described in detail in the following section.



$$
\begin{aligned}
x &\geq 2 \\
y &\geq 0 \\
z &\geq 0 \\
x + y &\geq 3 \\
x - z &\geq 2 \\
x + y - z &\geq 2 \\
x + y + z &\geq 3
\end{aligned}
$$

Fig. 8. An example of a OhDD. On the right, the constraints of the octahedron.

Figure 8 shows an example of an OhDD and the octahedron it represents on the right, using reduction rules for non-negative variables. The shadowed path highlights one constraint of the octahedron, $(x + y - z \geq 2)$. All constraints that end in a terminal node with $-\infty$ represent constraints with an unknown bound, such as $(x - y \geq -\infty)$. As the OhDD represents the saturated form of the octahedron, some redundant constraints such as $(x + y + z \geq 3)$ appear explicitly.

This representation based on decision diagrams provides three main advantages. First, decision diagrams provide many opportunities for reuse. For example, nodes in a OhDD can be shared. Furthermore, different OhDD can share internal nodes, leading to a greater reduction in the memory usage. Second, the reduction rules avoid representing the zero coefficients of the linear inequalities. Finally, symbolic algorithms on OhDD can deal with sets of inequalities instead of one inequality at a time. All these factors combined improve the efficiency of operations with octahedra.

4.2 Definitions

Definition 12 (Octahedron Decision Diagram - OhDD) *An Octahedron Decision Diagram is a tuple (V, G) where V is a finite set of positive real-valued variables, and $G = (N \cup K, E)$ is a labeled single rooted directed acyclic graph with the following properties. Each node in K, the set of* terminal *nodes, is labeled with a constant in $\mathbb{Q} \cup \{-\infty\}$, and has no outgoing arcs. Each node $n \in N$ is labeled with a variable $v(n) \in V$, and it has three outgoing arcs, labeled $-$, 0 and $+$.*

By establishing an order among the variables of the OhDD, the notion of *ordered* OhDD can be defined. The intuitive meaning of ordered is the same as in BDDs, that is, in every path from the root to the terminal nodes, the variables of the decision diagram always appear in the same order. For example, the OhDD in Fig. 8 is an ordered OhDD.

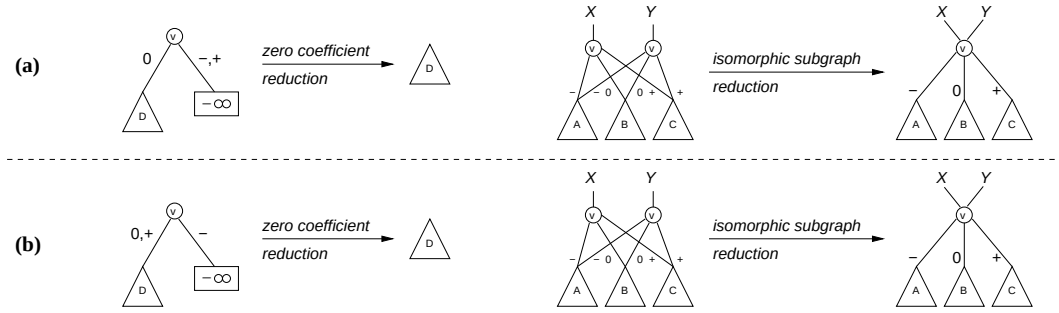


Fig. 9. Reduction rules: (a) Unconstrained variables, (b) non-negative variables.

Definition 13 (Ordered OhDD) *Let $\succ$ be a total order on the variables V of a OhDD. The OhDD is ordered if, for any node $n \in N$, all of its descendants $d \in N$ satisfy $v(d) \succ v(n)$.*

In the same way, the notion of a *reduced* OhDD can be introduced. However, the reduction rules will be different in order to take advantage of the structure of the constraints. In an octahedron, most variables will not appear in all the constraints. Avoiding the representation of these variables with a zero coefficient would improve the efficiency of OhDD. This can be achieved as in ZDDs by using a special reduction rule.

Let us consider an octahedron like $(x - y \geq 4)$. Other variables, *e.g.* z, do not affect the bound of the constraint. For example, as there is no information about z, constraints that involve x, y and z will have a bound like $-\infty$, like $(x - y + z \geq -\infty)$ or $(x - y - z \geq -\infty)$. This scenario can be described as follows: there is a node n in the OhDD with a variable z, where the outgoing arcs $-$ and $+$ point towards $-\infty$, and the 0 arc points to a node m. In this case, the node n can be replaced by node m to avoid encoding the irrelevant variable z. This reduction rule is displayed in Figure 9(a).

If the variables are known to be non-negative, the reduction rule can be refined. For example, in the case of a constraint like $(x - y \geq 4)$ and an irrelevant variable z, the constraints where the variable z appears are $(x - y + z \geq 4)$ and $(x - y - z \geq -\infty)$. Contrary to the previous case with arbitrary variables, the constraint $(x - y + z \geq 4)$ has now a known bound as $(z \geq 0)$. Therefore, the reduction rule should be rephrased to take into account this information: there should be a node n in the OhDD with a variable z, where the outgoing arc $-$ points to $-\infty$ and both the 0 and $+$ arcs point to a node m. Then, the node n can be replaced by m. Figure 9(b) depicts this alternative reduction rule. Remarkably, using this reduction rule, the set of constraints stating that "any sum of variables is greater or equal to zero" is represented as the single terminal node 0.

Figure 9 shows an example of the two alternatives, together with the other reduction rule, which merges isomorphic subgraphs of the decision diagram.

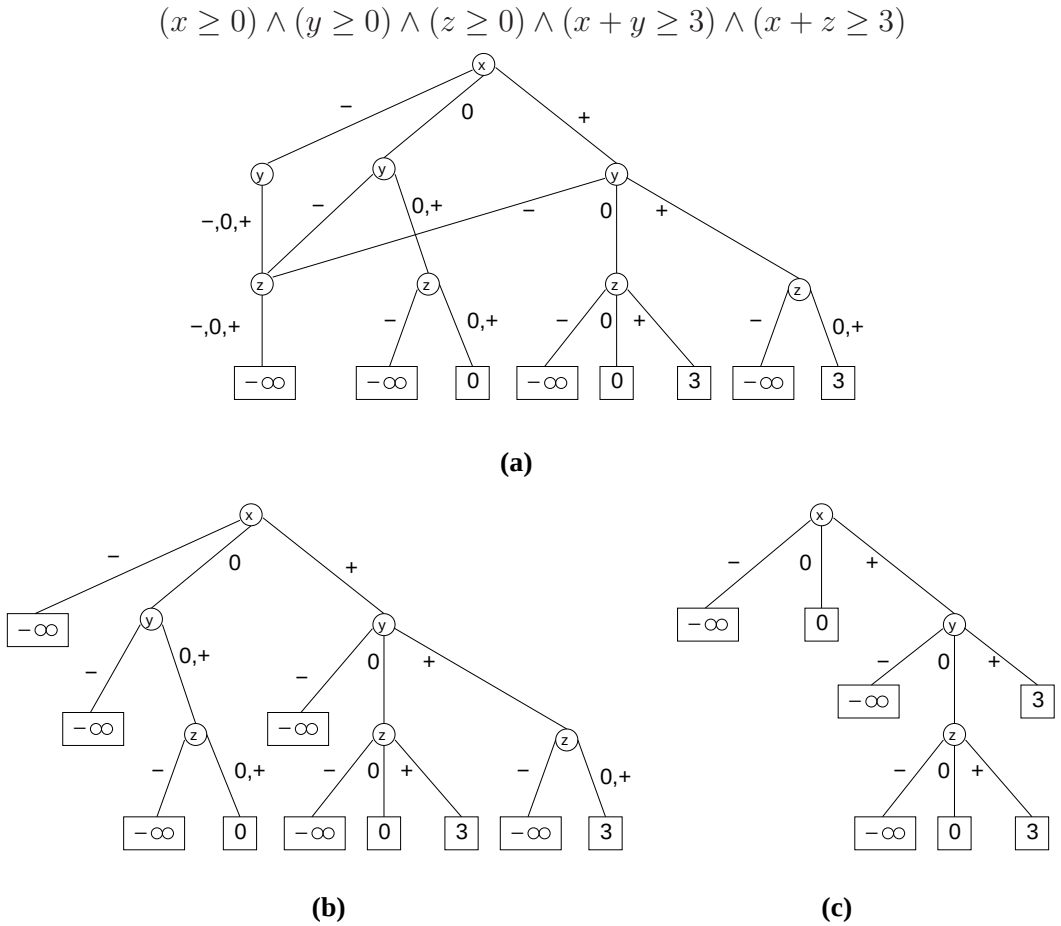


Fig. 10. Comparison of reduction rules. On the top, the octahedron being encoded. On the bottom, (a) OhDD with reduction rule 1 - isomorphic subgraphs, (b) OhDD with reduction rule 2 - unconstrained variables and (c) OhDD with reduction rule 3 - non-negative variables.

Notice that contrary to BDDs, nodes where all arcs have the same target will not be reduced (unless the three arcs point to $-\infty$). It is possible to have both types of variables in the same OhDD, using the suitable reduction rule for each variable.

Definition 14 (Reduced OhDD) *A reduced OhDD is an ordered OhDD where none of the following rules can be applied:*

(1) Reduction of isomorphic subgraphs*: Let D_1 and D_2 be two isomorphic subgraphs of the OhDD. Merge D_1 and D_2.*
(2) Reduction of zero coefficients (with unconstrained variables)*: Let $n \in N$ be a node with the $-$ and $+$ arcs going to the terminal $-\infty$, and with the arc 0 pointing to a node m. Replace n by m.* Or
(3) Reduction of zero coefficients (with non-negative variables)*: Let $n \in N$ be a node with the $-$ arc going to the terminal $-\infty$, and with the arcs 0 and $+$ pointing to a node m. Replace n by m.*

Figure 10 shows the effect of the different reduction rules on a OhDD. In this case, the octahedron being represented is $(x \geq 0) \wedge (y \geq 0) \wedge (z \geq 0) \wedge (x + y \geq 3) \wedge (x + z \geq 3)$. The detection of isomorphic subgraphs is illustrated in Figure 10(a). To improve the clarity of the Figure, terminal nodes with the same constant value have not been collapsed into a single node. As all the variables in this example are non-negative, the two alternative reduction rules 2 and 3 can be compared, as shown in Figures 10(b) and (c). Notice that the reduction that assumes non-negative variables produces a more compact representation. Therefore, whenever all the variables in a problem are known to be non-negative, reduction rule number 3 should be used instead of reduction rule number 2.

4.3 Implementation of the Operations

The octahedra abstract domain and its operations have been implemented as OhDD on top of the CUDD decision diagram package [32]. Each operation on octahedra performs simple manipulations such as computing the maximum or the minimum between two systems of inequalities, where each inequality is encoded as a path in a OhDD.

Two concepts from BDDs are used to present the implementation of the operations. First, the *top variable* of a OhDD is the variable that appears in the root of the OhDD. Two OhDD may have different top variables, because some variables are not encoded when the reduction rules are applied. Given several ordered OhDD, the top variable among all of them is the one which appears before in the ordering. The other concept is the term *cofactor* from Boolean algebra. The two cofactors of a Boolean formula $f(x_1, \ldots, x_n) : \mathbb{B}^n \to \mathbb{B}$ are the pair of formulas obtained by replacing variable x_i by constants 0 and 1 respectively inside f. In a BDD, the cofactors of a node f with respect to the top variable are the two children of f. In the context of OhDD, the term cofactor is also used to denote the children of a node. Each node f of the decision diagram has three cofactors f^-, f^0 and f^+ with respect to a variable x. Each cofactor denotes the set of inequalities in f where the variable x has a given coefficient, *i.e.* f^0 contains all the inequalities where x does not appear. The cofactors of a OhDD f with respect to the top variable are the targets of the three arcs $-$, 0 and $+$, while the cofactors for the other variables are defined by the chosen reduction rule.

The operations on octahedra can be implemented as recursive procedures on a OhDD. The algorithm may take as arguments one or more decision diagrams, depending on the operation. All these recursive algorithms share the same overall structure:

(1) Check if the call is a terminal case, *e.g.* all arguments are constant decision diagrams. In that case, the result can be computed directly.
(2) Look up the cache to see if the result of this call was computed previously and is available. In that case, return the precomputed result.
(3) Select the top variable t in all the arguments according to the ordering. The algorithm will only consider this variable during this call, leaving the rest of the variables to be handled by the subsequent recursive calls.
(4) Obtain the cofactors of t in each of the arguments of the call.
(5) Perform recursive calls on the cofactors of t.
(6) Combine the results of the different calls into the new top node for variable t.
(7) Store the result of this recursive call in the cache. Future calls to this method with the same arguments will use the cached result instead of repeating the computation.
(8) Return the result to the caller.

The saturation algorithm is a special case: all sums of pairs of constraints are computed by a single traversal; but if new inequalities have been discovered, the traversal must be repeated. The process continues until a fixpoint is reached. Even though this fixpoint might not be reached, as seen in Fig. 7, the number of iterations required to saturate an octahedron tends to be very low (1-4 iterations) if it is derived from saturated octahedra, *e.g.* the intersection of two saturated octahedra.

These traversals might have to visit 3^n inequalities/paths in the OhDD in the worst case. However, as OhDD are directed graphs, many paths share nodes so many recursive calls will have been computed previously, and the results will be reused without the need to recompute. The efficiency of the operations on decision diagrams depends upon two very important factors. The first one is the *order of the variables* in the decision diagram. Intuitively, each call should perform as much work as possible. Therefore, the variables that appear early in the decision diagram should discriminate the result as much as possible. A second factor in the performance of these algorithms is the *effectivity of the cache* to reuse previously computed results.

4.3.1 Reduction rules

In order to implement the reduction rules, two basic operations should be defined. First, how to obtain the OhDD for the three cofactors (coefficients) of a given variable. And second, how to build a new OhDD from the three cofactors of a given variable. Figure 11 shows the pseudocode for these two procedures, called *DD_GetCofactors* and *DD_CombineCofactors*.

The only remarkable aspect of *DD_GetCofactors* is how to compute the cofac-

Function *DD_GetCofactors*(var, f)
Input: An OhDD f over non-negative variables and a variable var. If var appears in the decision diagram f, it must appear as the top variable.
Output: The 3 cofactors of f for variable var, $< f^-, f^0, f^+ >$.

if DD_IsConstant(f) $\vee$ DD_TopVariable(f) $\neq$ var **then**
  $< f^-, f^0, f^+ > := < -\infty, f, f >$
else
  $< f^-, f^0, f^+ > :=$ < DD_NegArc(f), DD_ZeroArc(f), DD_PosArc(f) >
endif
return $< f^-, f^0, f^+ >$

Function *DD_CombineCofactors*(var, f^-, f^0, f^+)
Input: The three cofactors of a OhDD for a given variable var. Any variable in the cofactors must appear after var in the ordering.
Output: A OhDD where $< f^-, f^0, f^+ >$ are the three cofactors for variable var.

if $f^0 = f^+ \wedge f^- = -\infty$ **then**
  return f^0
else
  return DD_UniqueNode(var, f^-, f^0, f^+)
endif

Fig. 11. Implementation of the reduction rules for non-negative variables.

tors of a variable that does not appear in a OhDD. If a variable is missing, it means that it has been reduced. In a OhDD with non-negative variables, this means that its negative cofactor is $-\infty$ while its positive and zero cofactor are equal to the OhDD. A similar implementation produces the reduction rules for unconstrained variables.

Regarding *DD_CombineCofactors*, its pseudocode assumes that there is a procedure called *DD_UniqueNode* that detects isomorphic nodes in the decision diagram, so if an isomorphic node already exists it is returned, otherwise a new one is created instead. This operation is typically provided by all decision diagram packages [5].

4.3.2 Saturation

The saturation procedure can be implemented symbolically. Instead of choosing two constraints and computing its linear combination, the linear combination of a whole set of constraints is computed in one step. Figures 12 and 13 show the pseudocode that performs this saturation.

Function *Saturate*(f)
Input: A OhDD f.
Output: The saturation of the OhDD f.

```
do
  old := f
  res := SaturateRecur(f, f)
  f := MaximumRecur(f, res)
  { Emptiness test - find a constraint (0 ≥ k) with (k > 0) }
  aux := f
  while ¬DD_IsConstant(aux) do
    aux := DD_ZeroArc(aux)
  endwhile
  if aux > 0 then return +∞ endif
until f = old
return res
```

Fig. 12. Pseudocode of the saturation procedure and the emptiness test.

The recursive saturation algorithm *SaturateRecur* computes the linear combination of its two parameters. Intuitively, the computation is split according to the top variable of the decision diagram. The only cases relevant to our computation are those where the top variable will have a coefficient in $\{-1, 0, +1\}$ in the result of the linear combination. For example, the linear combination has a coefficient $+1$ if one of the arguments has coefficient 0 and the other has coefficient $+1$. The remaining variables of the linear combination are computed recursively using the same algorithm.

Saturation is performed by computing these linear combinations and adding them to the system of inequalities until a fixpoint is reached. This computation is described in the procedure *Saturate* in Figure 12. Notice that after computing each set of linear combinations, they are added to the OhDD using the maximum operator, which in OhDD corresponds to the intersection. If a constraint of the form $(0 \geq k)$ with $(k > 0)$ appears during the computation of the fixpoint, then we know that the octahedron is empty.

In order to encode the empty octahedron, an additional terminal node is used in our implementation: $+\infty$. This terminal was not presented in Definition 12 to separate the concept of OhDD from implementation issues. Intuitively, any constraint with a constant term of $+\infty$, *i.e.* $(\pm x_1 \pm \ldots \pm x_n \geq +\infty)$ is false, so the semantics of the decision diagram is preserved. Moreover, choosing this terminal node to encode empty octahedra is also natural in the following sense:

Function *SaturateRecur*(f, g)
Input: Two OhDD called f and g.
Output: The OhDD describing the pairwise linear combinations of f and g, ignoring constraints with a coefficient outside $\{-1, 0, +1\}$.

```
{ Terminal cases }
if DD_IsConstant(f) ∧ DD_IsConstant(g) then
  return DD_Sum(f, g)
endif
if f = +∞ ∨ g = +∞ then return +∞ endif
if f = −∞ ∨ g = −∞ then return −∞ endif

{ Lookup the result in the cache }
res := DD_CacheLookup(SaturateRecur, f, g)
if res ≠ null then return res endif
top := DD_TopVariable(f, g)
< f⁻, f⁰, f⁺ > := DD_GetCofactors(top, f)
< g⁻, g⁰, g⁺ > := DD_GetCofactors(top, g)

{ The shorthand MAX(a,b) is used instead of MaximumRecur(a,b) }

{ Recursive calls for top coefficient = −1 }
res⁻ := MAX( SaturateRecur(f⁻, g⁰), SaturateRecur(f⁰, g⁻) )

{ Recursive calls for top coefficient = 0 }
res⁰ := MAX( MAX( SaturateRecur(f⁰, g⁰), SaturateRecur(f⁺, g⁻) ),
             SaturateRecur(f⁻, g⁺) )

{ Recursive calls for top coefficient = +1 }
res⁺ := MAX( SaturateRecur(f⁺, g⁰), SaturateRecur(f⁰, g⁺) )

{ Combine the cofactors and update the cache }
res := DD_CombineCofactors(top, res⁻, res⁰, res⁺)
DD_CacheInsert(SaturateRecur, f, g, res)
return res
```

Fig. 13. Pseudocode of one iteration of the saturation procedure.

- Neutral element of the union: The union is implemented using the minimum operator so $+\infty$ is the neutral element.
- Absorbent element of the intersection: The intersection is implemented using the maximum operator so $+\infty$ is the absorbent element.

It would also have been possible to encode empty octahedra in a different

Function *MaximumRecur*(f, g)
Input: Two OhDD called f and g.
Output: An OhDD that has, at the bottom of each path from the root to the terminal nodes, the maximum terminal found in the same path in f and g.

```
{ Terminal cases }
if f = g then return f endif
if f = +∞ ∨ g = −∞ then return f endif
if f = −∞ ∨ g = +∞ then return g endif
if DD_IsConstant(f) ∧ DD_IsConstant(g) then
  return DD_Max(f, g)
endif

{ Lookup the result in the cache }
res := DD_CacheLookup(MaximumRecur, f, g)
if (res ≠ null) then return res endif

{ Recursive calls for each cofactor }
top := DD_TopVariable(f, g)
< f⁻, f⁰, f⁺ > := DD_GetCofactors(top, f)
< g⁻, g⁰, g⁺ > := DD_GetCofactors(top, g)

{ The shorthand MAX(a,b) is used instead of MaximumRecur(a,b) }
<res⁻, res⁰, res⁺ > := < MAX(f⁻, g⁻), MAX(f⁰, g⁰), MAX(f⁺, g⁺) >

{ Combine the cofactors and update the cache }
res := DD_CombineCofactors(top, res⁻, res⁰, res⁺)
DD_CacheInsert(MaximumRecur, f, g, res)
return res
```

Function *Intersection*(f, g)
Input: Two OhDD called f and g.
Output: The intersection of f and g.

```
res := MaximumRecur(f, g)
return Saturate(res)
```

Fig. 14. Pseudocode of the intersection procedure.

way, *i.e.* adding a boolean variable to OhDD stating whether the octahedron is empty or not. Our approach was chosen for convenience in the implementation of the operations.

Function *ProjRecur*(f, v)
Input: A OhDD called f and a variable v.
Output: An OhDD that is equal to f except in all nodes labeled with variable v. If there is a node labeled with variable v inside f, it is replaced in the result by one of its children, the one targeted by the arc labeled with coefficient 0.

```
{ Terminal cases }
if DD_IsConstant(f) then return f endif

top := DD_TopVariable(f);
< f^-, f^0, f^+ > := DD_GetCofactors(top, f)
{ Test if this is the variable to be projected }
if v = top then return f^0 endif
{ Test if v precedes top in the top-down variable order. }
if v ≤ top then
  { Variable v does not appear inside f, it should have appeared before. }
  return f
endif

{ Lookup the result in the cache }
res := DD_CacheLookup(MaximumRecur, f, v)
if res ≠ null then return res endif

{ Recursive calls for each cofactor }
< res^-, res^0, res^+ > := < ProjRecur(f^-, v), ProjRecur(f^0, v), ProjRecur(f^+, v) >

{ Combine the cofactors and update the cache }
res := DD_CombineCofactors(top, res^-, res^0, res^+)
DD_CacheInsert(MaximumRecur, f, g, res)
return res
```

Function *Projection*(f, v)
Input: A OhDD called f and a variable v.
Output: The OhDD computed from f after projecting variable v. The projection of this variables "forgets" all the constraints where the variable has a coefficient other than zero.

```
{ Assumption: f is in saturated form. Otherwise we should saturate it here. }
res := ProjectRecur(f, g)
return Saturate(res)
```

Fig. 15. Pseudocode of the projection procedure.

4.3.3 Other operations

The intersection of two octahedra has been defined as the union of the sets of constraints of both octahedra, choosing the maximum constant for those constraints that appear in both octahedra. In a OhDD, constraints that do not appear in an octahedron are represented by the terminal $-\infty$. Therefore, the intersection of OhDD can be implemented by taking the *maximum* of the two arguments for each path between the root and the terminal nodes. The pseudocode that computes this maximum is shown in Fig. 14. The result of this operation is not necessarily saturated.

The same concept can be applied to the union of octahedra. The union can be computed as the *minimum* of the two arguments for each path between the root and the terminal nodes.

The projection can also be implemented symbolically, as it is shown in Figure 15. The concept behind the implementation is the following: this operation should remove all inequalities where a variable v has a non-zero coefficient. Any inequality where v has a non-zero coefficient will appear in the OhDD as a node labeled with that variable. These nodes have a child, the target of the arc labeled with 0, that represents the inequalities where v appears with a coefficient 0. Therefore, the implementation of projection can be summarized as follows: traverse the OhDD until a node n labeled with variable v is discovered; replace the reference to n with a reference to n^0.

Extension, adding a new variable to a OhDD that does not participate in any inequality, is a trivial operation due to the choice of the reduction rules. Both the reduction rules for unconstrained and non-negative variables detect variables which do not participate in an inequality and remove them from the diagram. Therefore, extending a OhDD with a new variable does not change the diagram in any way; we only need to decide the position of the new variable in the top-down variable order.

Other operations can be defined similarly or be rewritten in terms of the intersection and the union. For instance, an approximate inclusion test ($f \subseteq g$)? can be rewritten as ($f = f \cap g$)?, using an intersection plus an equality test, which compares whether the top nodes of the decision diagram are equal.

5 Applications of the Octahedron Abstract Domain

5.1 Motivating Application

Asynchronous circuits [24] are a class of circuits where there is no global clock to synchronize its different components. Asynchronous circuits replace the global clock by a local handshake between components, gaining several advantages such as lower power consumption. However, the absence of a clock makes the verification of asynchronous circuits more complex, since the correctness of the circuit is more dependent on *timing constraints*. This means that the correctness of the circuit depends on the delays of its gates and wires.

In many asynchronous circuits implementing control logic, the timing constraints that arise are unit inequalities. Intuitively, they correspond to constraints of the type

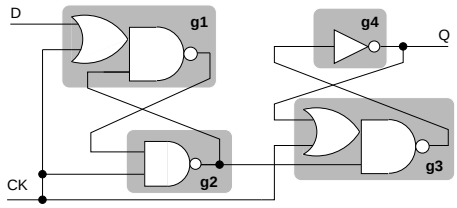
$$(\underbrace{\delta_1 + \cdots + \delta_i}_{\text{delay}(\text{path}_1)}) - (\underbrace{\delta_{i+1} + \cdots + \delta_n}_{\text{delay}(\text{path}_2)}) \geq k$$

indicating that certain paths in the circuit must be longer than other paths. In very rare occasions, coefficients different from ± 1 are necessary. A typical counterexample would be a circuit where one path must be c times longer than another one, *e.g.* a fast counter.

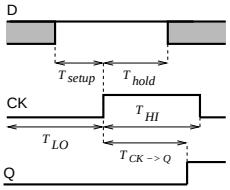
Figure 16(a) depicts a D flip-flop [27]. Briefly stated, a D flip-flop is a 1-bit register. It stores the data value in signal D whenever there is a rising edge in the clock signal CK. The output Q of the circuit is the value which was stored in the last clock rising edge. We would like to characterize the behavior of this circuit in terms of the internal gate delays. The flip-flop has to be characterized with respect to three parameters (see Figure 16(b)):

- *Setup time*, noted as T_{setup}, is the amount of time that D should remain stable before a clock rising edge.
- *Hold time*, noted as T_{hold}, is the amount of time that D should remain stable after a clock rising edge.
- *Delay* or *clock-to-output time*, noted as $T_{CK \to Q}$, is the amount of time required by the latch to propagate a change in the input D to the output Q.

The timing analysis algorithm is capable of deriving a set of sufficient linear constraints that guarantee the correctness of the circuit's behavior. This behavior will be correct if the output Q matches the value of D in the last clock rising edge. Formally, this property can be stated as:



(a)



(b)

$$T_{CK \to Q} \leq D_2 + D_3 + D_4$$
$$T_{setup} > D_1 + D_2 - d_2$$
$$T_{hold} > D_2 + D_3$$
$$T_{HI} > D_2 + D_3 + D_4$$
$$T_{HI} > T_{hold}$$
$$T_{LO} > T_{setup}$$
$$d_1 > D_2$$

(c)

Fig. 16. (a) Implementation of a D flip-flop [27], (b) description of variables that characterize any D flip-flop and (c) sufficient constraints for correctness for any delay of the gates.

The value of Q after a delay $T_{CK \to Q}$ from CK's rising edge must be equal to the value of D CK's rising edge.

Any behavior not fulfilling this property is considered to be a failure. Fig. 16(c) reports the set of sufficient timing constraints derived by the algorithm. Each gate g_i has a symbolic delay in the interval $[d_i, D_i]$. Notice that the timing constraints are unit inequalities.

Timing verification has been performed on several asynchronous circuits from the literature. This verification can be seen as the analysis of a set of clock variables, and the underlying timing behavior can be modeled as assignments and guards on these variables [7]. The analysis of clock variables has been performed using two different numerical abstractions: convex polyhedra and octahedra. The implementation of polyhedra uses the New Polka polyhedra library [26], while the library of OhDD is implemented on top of the CUDD package [32]. Table 2 shows a comparison of the experimental results for some examples. All these examples were verified successfully using both octahedra and polyhedra, as all relevant constraints were unit linear inequalities. For all these cases, the execution time of convex polyhedra and octahedra is comparable, while the memory usage for octahedra is lower. For each example, we provide the number of different states (configurations) of the circuit, the number of clock and delay variables of the abstractions and the execution time required by the analysis with each abstraction.

The difference in memory usage is quantified in the next example, an asynchronous pipeline with different number of stages and an environment running at a fixed frequency. The processing time required by each stage i has a processing time bounded by an interval, with unknown upper and lower bound $[d_i, D_i]$. Whenever a stage finishes its computation, it sends the result to the next stage if it is empty. The safety property being verified in this case was *"the environment will never have to wait before sending new data to the pipeline"*, i.e. whenever the environment sends new data to the pipeline, the first stage

Table 2
Experimental results using convex polyhedra and octahedra.

Example	# of States	# of variables	Polyhedra		OhDD	
			CPU	Memory	CPU	Memory
nowick	60	30	0.5s	82Mb	0.1s	9Mb
sbuf-read-ctl	74	31	1.2s	83Mb	1.4s	10Mb
rcv-setup	72	27	2.1s	83Mb	8.3s	21Mb
alloc-outbound	82	39	1.3s	83Mb	0.2s	10Mb
ebergen	83	27	1.3s	83Mb	1.7s	11Mb
mp-forward-pkt	194	29	1.9s	85Mb	3.8s	20Mb
chu133	288	26	1.3s	85Mb	1.0s	13Mb

is empty. Fig.17 shows the pipeline, with an example of a correct and incorrect behavior. The tool discovers that correct behavior can be ensured if the following holds:

$$d_{IN} > D_1 \wedge \ldots \wedge d_{IN} > D_N \wedge d_{IN} > D_{OUT}$$

where D_i is the delay of stage i, and d_{IN} and D_{OUT} refer to environment delays. This property is equivalent to:

$$d_{IN} > max(D_1, \ldots, D_N, D_{OUT})$$

Therefore, the pipeline is correct if the environment is slower than the slowest stage of the pipeline. Both the polyhedra and octahedra abstract domain are able to discover this property. This example is interesting because it exhibits a very high degree of concurrency. The verification times and memory usage for different lengths of the pipeline can be found in Table 3. Notice that the memory consumption of OhDD is lower than that of convex polyhedra. This reduction in memory usage is sufficient to verify larger pipelines (n = 6 stages) not verifiable with our convex polyhedra implementation. Using convex polyhedra, the verification tool runs out of memory. Memory usage remains relatively low during the execution (about 450Mb) but reaches a peak of more than 1.7Gb after 8 hours of CPU time; the peak in memory usage occurs during one of the transformations between dual representations of polyhedra. However, the reduction in memory usage achieved by octahedra comes at the expense of an increase in the execution time, which is spent minimizing the decision diagrams.

Table 3
Experimental results for the pipeline example.

# of stages	# of States	# of variables	Polyhedra		OhDD	
			CPU Time	Memory	CPU Time	Memory
2	36	20	0.6s	64Mb	1.0s	5Mb
3	108	24	2.0s	67Mb	17.0s	8Mb
4	324	28	13.5s	79Mb	4m 9.0s	39Mb
5	972	32	4m 19.2s	147Mb	1h 6m 14.0s	57Mb
6	2916	36	–	–	39h 44m 18.0s	83Mb

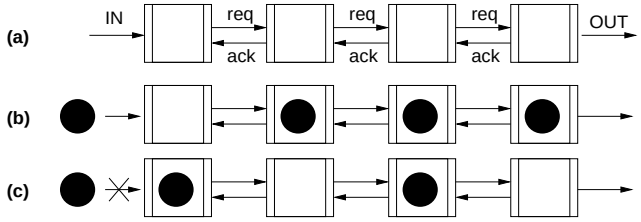


Fig. 17. (a) Asynchronous pipeline with N=3 stages, (b) correct behavior of the pipeline and (c) incorrect behavior. Dots represent data elements.

5.2 Additional experiments

The experimental results presented so far have been computed with OhDD using the reduction rules for non-negative variables (NNV). Also, a predefined top-down variable order has been used. The order for variables has been chosen manually, where the variables which are likely to appear in many constraints are placed near the top of the diagram. In this section, we illustrate the perfomance of OhDD with different settings. The example used to perform the comparison is the asynchronous pipeline depicted in Figure 17.

For the first comparison, we have implemented the reduction rules for unconstrained variables (UV) discussed in Section 4.2. This change affects a very small part of the implementation, namely the methods *GetCofactors* and *CombineCofactors* which have been presented in Figure 11. Intuitively, these reduction rules should have a worse performance in problems where variables may be non-negative, as they cannot avoid representing constraints like $(x_1 \geq 0)$ or $(x_1 + x_2 \geq 0)$ explicitly. Table 4 shows the experimental results with these alternative reduction rules. These results confirm the intuition: the decision diagrams using the UV rules are larger, using more memory and requiring more CPU time to be traversed. The difference becomes more noticeable as the number of variables becomes larger, because the number of redundant inequalities grows exponentially with the number of variables. For instance, in a system with three non-negative variables that satisfy $(x_1 - x_2 \geq 3)$ the follow-

Table 4
Comparison between reduction rules for unconstrained and non-negative variables.

# of stages	# of States	# of variables	Unconstrained (UV)		Non-negative (NNV)	
			CPU Time	Memory	CPU Time	Memory
2	36	20	1.0s	9Mb	1.0s	5Mb
3	108	24	23.8s	15Mb	17.0s	8Mb
4	324	28	8m 35.4s	54Mb	4m 9.0s	39Mb
5	972	32	8h 48m 8.9s	87Mb	1h 6m 14.0s	57Mb
6	2916	36	>48h	Time-out	39h 44m 18.0s	83Mb

ing inequalities cannot be simplified by the UV reduction rules even though they are redundant: $(x_1 \geq 3)$, $(x_2 \geq 0)$, $(x_3 \geq 0)$, $(x_1 + x_2 \geq 3)$, $(x_1 + x_3 \geq 3)$, $(x_2 + x_3 \geq 0)$, $(x_1 - x_2 + x_3 \geq 3)$ and $(x_1 + x_2 + x_3 \geq 3$.

Therefore, the conclusion is that *whenever variables are known to be non-negative, the NNV reduction rules should be preferred* to the UV rules. Nevertheless, in some problems variables may have negative values; in those cases, we can use the more general UV rules.

Another set of experiments has evaluated the gains that can be achieved by performing dynamic reordering. Dynamic reordering [28] is a technique which explores which top-down ordering of variables provides a smaller decision diagram. The chosen ordering may be altered dynamically depending on the structure of the decision diagram, *i.e.* a good ordering at a given point in time may be a bad ordering later. In this case, setting up the experiments has been more difficult as the implementation of dynamic reordering is very complex. In order to evaluate the potential effect of dynamic reordering in OhDD, we have used the dynamic reordering methods and heuristics provided by the CUDD package. The reduction rules of OhDD have been altered in order to become compatible with the reordering procedures used by CUDD, *i.e.* using reduction rules which are more similar to those of BDDs.

Table 5 summarizes the result of this prototype implementation. The columns on the left describe the results with the BDD-like reduction rules and the dynamic reordering turned off. On the right, we present the results with the same reduction rules, but using a dynamic reordering procedure called *sifting* [28]. Studying these results, it is clear that these reduction rules achieve results which are far worse than those produced with the other reduction rules (UV and NNV) presented in this paper. Dynamic reordering manages to reduce the size of these diagrams, improving both the memory and CPU time. However, even though the CPU time is lower, dynamic reordering does not provide a drastic reduction in CPU time, something which would be required to make the analysis more practical. The CPU time still grows exponentially, reaching

Table 5
Evaluation of dynamic reordering in OhDD using the CUDD package.

# of stages	# of States	# of variables	No reordering		Dynamic reordering	
			CPU Time	Memory	CPU Time	Memory
2	36	20	1.2s	6Mb	1.0s	5Mb
3	108	24	34.7s	16Mb	25.2s	8Mb
4	324	28	13m 19.4s	55Mb	7m 1.7s	39Mb
5	972	32	$> 12h$	Time-out	7h 14m 14.0s	57Mb

unacceptable values very quickly. The reason behind the low gains achieved with dynamic reordering seems to be that the initial order chosen for the variables of the OhDD was good, due to our knowledge about the kind of constraints likely to appear in the analysis of a timed system. Nevertheless, the improvements achieved in other applications where a good ordering cannot be established a priori may be relevant enough to make dynamic reordering an essential feature of OhDD.

5.3 Other Applications

In general, the octahedron abstract domain may be interesting in any analysis problem where convex polyhedra can be used. Many times, the precision obtained with convex polyhedra is very good, but the efficiency of the analysis limits the applicability. In these scenarios, using octahedra might be adequate as long as unit linear inequalities provide sufficient information for the specific problem. If the variables involved in the analysis are positive, the suitability of octahedra becomes even more noticeable. Some examples of areas of applications are the following:

- *Static discovery of bounds in the size of asynchronous communication channels*: Many systems communicate using a non-blocking semantics, where the sender does not wait until the receiver is ready to read the message. In these systems, each channel requires a buffer to store the pending messages. Allocating these buffers statically would improve performance but it is not possible, as the amount of pending messages during execution is not known in advance. Analysis with octahedra could discover these bounds statically. The analysis of the bounds of these channels can be performed using octahedra. Furthermore, the size of a channel is always positive, so the reduction rule for non-negative variables can be used in the analysis. This problem is related to the problem of structural boundedness of a Petri Net [25], where an upper bound on the number of tokens that can be in each place of the Petri Net must be found.

- *Analysis of timed systems*: Clocks and delays are restricted to positive values in many types of models. Octahedra can be used to analyze these values and discover complex properties such as timing constraints or worst-case execution time (WCET).
- *Analysis of string length in programs* [12]: Checking the absence of buffer overflows is important in many scenarios, specially in the applications where security is critical, *e.g.* an operating system. C programs are prone to errors related to the manipulation of strings. Several useful constraints on the length of strings can be represented with octahedra. For instance, a constraint on the concatenation of two strings can be $\texttt{strlen}(\texttt{strcat}(s_1, s_2)) = \texttt{strlen}(s_1) + \texttt{strlen}(s_2)$.
- *Analysis of term size in logic programs* [33], which can be used among other things to prove termination of logic programs [31].
- *Proof of mutual exclusion and other synchronization properties among concurrent processes*: many high-level synchronization constraints can be expressed easily as properties on counter (semaphore) variables [16]. For instance, mutual exclusion among n processes can be represented with constraints like $(x_1 + \ldots + x_n \leq 1)$, where $x_i = 1$ if the process i is inside the critical section, and $x_i = 0$ otherwise.

6 Conclusions and future work

A new numerical abstract domain called octahedron has been presented. This domain can represent and manipulate constraints on the sum or difference of an arbitrary number of variables. In terms of precision, this abstraction is between octagons and convex polyhedra. Regarding complexity, the worst case complexity of octahedra operations over n variables is $O(3^n)$ in memory, and $O(3^n)$ in execution time in addition to the cost of saturation. However, worst-case performance is misleading due to the use of a decision diagram approach. For instance, BDDs have a worst-case complexity of $O(2^n)$, but they have a very good behavior in many real examples. Performance in this case depends on factors such as the ordering of the variables in the decision diagram, the effectiveness of the cache and the nature of the problem to be solved. In the experimental results of OhDD, memory consumption was shown to be smaller than that of our convex polyhedra implementation. Running time was comparable to that of convex polyhedra in small and medium-sized examples, while in more complex examples the execution time was worse. This shows that OhDD trade speed for a reduction in memory usage. Dynamic reordering can be used to improve both memory and time, but only by a small margin.

Future work in this area will try to improve the execution time of octahedra operations. An area where there is room for improvement is the current bottleneck of the representation, the saturation procedure. Another direction of

research is the design of alternative implementations of the octahedron abstract domain. For instance, as the set of coefficients is limited and small, each unit inequality can be stored efficiently using bit-vectors. Experimental results using a bit-vector implementation have been reported in [8].

Acknowledgements

The authors would like to thank the referees for their insightful comments which have enhanced the final version of this paper.

References

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

[2] D. Avis, K. Fukuda, and S. Picozzi. On canonical representations of convex polyhedra. In *Proc. Int. Conf on Mathematical Software*, pages 350–360. World Scientific, 2002.

[3] R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages*. PhD thesis, University of Pisa, 1997.

[4] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proc. International Conference on Computer Aided Verification*, pages 341–353, 1999.

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

[6] N. Chernikova. Algoritm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 6(8):282–293, 1964.

[7] R. Clarisó and J. Cortadella. Verification of timed circuits with symbolic delays. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 628–633, 2004.

[8] R. Clarisó and J. Cortadella. Verification of concurrent systems with parametric delays using octahedra. In *Proc. International Conference on Application of Concurrency to System Design*, pages 122–131. IEEE Computer Society Press, 2005.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints

among variables of a program. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[11] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 197–212. Springer-Verlag, 1989.

[12] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 155–167. ACM Press, 2003.

[13] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In *Hybrid and Real-Time Systems*, pages 346–360, Grenoble, France, 1997. Springer Verlag, LNCS 1201.

[14] F. Fernández and P. Quinton. Extension of Chernikova's algorithm for solving general mixed linear programming problems. Technical Report 437, IRISA, 1988.

[15] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[16] A. J. Gerber. Process synchronization by counter variables. *SIGOPS Operating Systems Review*, 11(4):6–17, 1977.

[17] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[18] T. A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[19] C. Mauras. Symbolic simulation of interpreted automata. In *3rd Workshop on Synchronous Programming*, 1996.

[20] P. McMullen. The maximum number of faces of a convex polytope. *Mathematica*, (17):179–184, 1970.

[21] S. Minato. Zero-supressed BDDs for set manipulation in combinatorial problems. In *Proc. ACM/IEEE Design Automation Conference*, pages 272–277. ACM Press, 1993.

[22] A. Miné. The octagon abstract domain. In *Proc. of Analysis, Slicing and Tranformation (in Working Conference on Reverse Engineering)*, pages 310–319. IEEE CS Press, 2001.

[23] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, 1999.

[24] D. Muller and W. Bartky. A theory of asynchronous circuits. In *Proc. International Symposium on the Theory of Switching*, pages 204–243. Hardware University Press, 1959.

[25] T. Murata. State equation, controllability and maximal matchings of Petri nets. *IEEE Transactions on Automatic Control*, AC-22(3):412–416, 1977.

[26] New Polka: Convex Polyhedra Library.
http://www.irisa.fr/prive/bjeannet/newpolka.html.

[27] C. Piguet et al. Memory element of the Master-Slave latch type, constructed by CMOS technology. US Patent 5,748,522, 1998.

[28] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conference on Computer-Aided Design*, pages 42–47. IEEE Computer Society Press, 1993.

[29] S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. of International Conference on Verification, Model Checking and Abstract Interpretation*, number 3385 in Lecture Notes in Computer Science, pages 21–47. Springer-Verlag, 2005.

[30] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *Proc. of Logic Based Program Development and Transformation*, LNCS 2664, pages 71–89. Springer-Verlag, 2002.

[31] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 216–226. ACM Press, 1991.

[32] F. Somenzi. CUDD: Colorado university decision diagram package. Available online at `http://vlsi.colorado.edu/~fabio/CUDD`.

[33] A. van Gelder. Deriving constraints among argument sizes in logic programs (extended abstract). In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 47–60. ACM Press, 1990.

[34] H. L. Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[35] F. Wang. Efficient data structure for fully symbolic verification of real-time software systems. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems*, pages 157–171. Springer-Verlag, 2000.

[36] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. In *Proc. International Conference on Computer Aided Verification*. Springer-Verlag, 2004.