Extending Constraint-Only Representation of Polyhedra with Boolean Constraints

Alexey Bakhirkin, David Monniaux

Extending Constraint-Only Representation of Polyhedra with Boolean Constraints

Alexey Bakhirkin and David Monniaux

Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, 38000 Grenoble, France

Abstract. We propose a new relational abstract domain for analysing programs with numeric and Boolean variables. The main idea is to represent an abstract state as a set of linear constraints over numeric variables, with every constraint being enabled by a formula over Boolean variables. This allows us, unlike in some existing approaches, to avoid duplicating linear constraints shared by multiple Boolean formulas. To perform domain operations, we adapt algorithms from constraint-only representation of convex polyhedra, most importantly Fourier-Motzkin elimination and projection-based convex hull. We made a prototype implementation of the new domain in our abstract interpreter for Horn clauses. Our initial experiments are, in our opinion, promising and show directions for future improvement.

1 Introduction and related work

Static program analysis by abstract interpretation over-approximates the set of reachable states of a program by a set with a simple description, for instance, by attaching one interval to each program variable at every location in the program. Intervals however cannot express relationships between variables, so a richer approach is to attach to every location a set of valid linear inequalities, which geometrically is, a *convex polyhedron* [12,17].

Convex polyhedra are already quite formidable objects to compute with efficiently, yet they are insufficient for expressing certain invariants, and what is often needed is a *disjunction* of convex polyhedra. For instance, the strongest invariant of the following loop: `for(int i=0; i < n; i++){ }` is $(n < 0) \vee (0 \leq i \leq n)$. Note how the disjunction arises from the partition of executions into those that execute the loop at least once and those that do not. Better analysis precision may often be achieved by partitioning executions according to an abstraction of the control flow [32], or by partitioning abstract states with respect to conditions extracted from the program [11], etc. Some analyses of programs operating over arrays and maps abstract properties over these objects onto disjunctive relations between the scalar variables of the program and the values in the array cells [26,28]. For instance, a loop that fills an array `for(int i=0; i < n; i++){ a[i]=42; }` can be proved correct using an invariant $\forall k\ (0 \leq i \leq n) \wedge (0 \leq k < i \rightarrow a[k] = 42)$, with a disjunction between the cases $k < i$ (filled) and $k \geq i$ (unfilled).[1]

* Institute of Engineering Univ. Grenoble Alpes

[1] $\rightarrow$ denotes logical implication.

In all cases, the analysis needs to efficiently represent sets of convex polyhedra, possibly (but not necessarily [4]) tagged by elements of a finite set T (abstract traces, Boolean vectors, etc). Earlier works proposed to represent an abstract element by an explicit map from T to convex polyhedra, either as an array of pairs (T_i, P_i) where $T_i \subseteq T$ and P_i are polyhedra, or as a decision tree or DAG with polyhedra at the leaves. Both approaches are implemented in B. Jeannet's BddApron library [19].

One issue with this approach is that the possible number of abstract partitions is often exponential in some parameter (length of the recorded trace, number of Boolean variables) and thus every operation (post-condition, convex hull) is potentially repeated for each of the exponentially many polyhedra. At the same time, the polyhedra in different abstract partitions often share most of the constraints and only differ in few that are related to the partitioning criterion. Thus, it is tempting to store a set of polyhedra in some structure that does not require duplicating shared constraints, and to use symbolic algorithms that, as much as possible, avoid enumerating abstract partitions individually.

One approach to this is offered by different kinds of decision diagrams over linear constraints. A notable example is Linear Decision Diagrams (LDD) developed by S. Chaki, A. Gurfinkel, and O. Strichman [10]. An LDD is a DAG, where internal nodes are labelled with linear constraints, and the two leaves are *true* and *false*, thus a path through an LDD corresponds to a convex polyhedron. Based on the LDD algorithms, the same authors later developed an abstract domain of boxes [15], that only allows to have a comparison of a variable with a constant in an interior node.

Theoretical Contribution In this paper, we propose an alternative approach: to represent an abstract state as a set of implications $\{B_i \rightarrow c_i\}_{i=0..k}$, where B_i are arbitrary Boolean formulas, and c_i are linear constraints. This way, an abstract element can still be seen as an implicit map from a partition of $\mathbb{B}^m$ to convex polyhedra (similar to a BddApron element), but we do not have to duplicate storage and computations for constraints shared by multiple partitions. Another appeal of this approach is that some operations on constraint-only polyhedra can be naturally adapted to sets of implications of the form $B_i \rightarrow c_i$. The algorithms in this paper are based on Fourier-Motzkin elimination [30] and the reduction of convex hull to projection of F. Benoy, A. King, and F. Mesnard [7,33]. Whether it is possible to also adapt the more recent algorithms based on parametric linear programming and raytracing by A. Maréchal, D. Monniaux, and M. Périn [23,22,21] is a question for future work.

The Boolean variables occurring in the formulas B_i may be program variables (from small enumerated types) but may also be *observers*, partitioning according to trace history or calling context. This solves one issue with untagged disjunctions of polyhedra: when applying the widening operator to $\bigcup_i P_i$ and $\bigcup_j Q_j$, how does one "match" the P_i's and Q_j's to perform conventional widening over polyhedra [4]? Similarly, for the "join" operation $(\bigcup_i P_i) \sqcup (\bigcup_j Q_j)$, does one simply concatenate the two unions while removing duplicates (thus creating longer and longer lists), or does one "match" some P_i and Q_j for convex hull, and if so under which criteria? In our case, widening and join are guided by the Boolean variables: the polyhedra associated to the same Boolean choice are matched together.

Experimental Evaluation We made a prototype implementation of the proposed abstract domain in our abstract interpreter for Horn clauses [6,5].

2 Notation

We consider programs with Boolean and rational variables: a concrete program state is a tuple $(\mathbf{b}, \mathbf{x}) \in \mathbb{B}^m \times \mathbb{Q}^n$. We use bold lowercase symbols $\mathbf{b} \in \mathbb{B}^m$ and $\mathbf{x} \in \mathbb{Q}^n$ to denote valuations of Boolean and numeric variables respectively. We use lowercase Italic symbols b and x respectively to denote vectors of Boolean and rational variables. We refer to the j-th variable in x as $x_{(j)}$. We use other lowercase Italic symbols, e.g., a, d, to denote vector and scalar coefficients, and their meaning will be clear within their context.

Without loss of generality, we make a number of assumptions on the syntactic form of linear constraints. We assume that there exists the unique unsatisfiable linear constraint *cfalse*, i.e. we will not distinguish logically equivalent, but syntactically different falsities: $0 < 0$, $1 \geq 2$, etc. We assume that every linear constraint c_i is written as a greater-or-equal constraint with integer coefficients, i.e. $c_i = a_i x \succ d_i$, where $a_i \in \mathbb{N}^n$, $d_i \in \mathbb{N}$, and $\succ \in \{=, \geq, >\}$.

We sometimes write a Boolean or a linear constraint as $B[b]$ or $c[x]$ to emphasize that free variables in B and c come from vectors b and x respectively. We use the $[/]$ notation to denote substitution. For example, $B[b_{(j)}/true]$ denotes the result of substituting in B the variable $b_{(j)}$ with *true*. As a shortcut, we write $B[\mathbf{b}]$ to denote the result of substituting every free variable in B with its valuation given by $\mathbf{b}$.

3 Abstract Domain of Boolean and Linear Constraints

We propose to represent an abstract state as a set of implications:

$$S = \{B_i \rightarrow c_i\}_{i=0..k}$$

where B_i is a propositional formula over Boolean variables, and c_i is a linear constraint (equality or inequality) over numeric variables. We do *not* want B_i to be a partition of $\mathbb{B}^m$. Our intention is to never duplicate linear constraints that are shared by multiple valuations of Boolean variables.

An abstract state S represents the set of concrete states:

$$\gamma(S) = \left\{(\mathbf{b}, \mathbf{x}) \;\middle|\; \bigwedge_{i=0}^{k} B_i[\mathbf{b}] \rightarrow c_i[\mathbf{x}]\right\}$$

Alternatively, one can see an abstract state as a function that maps every valuation of Boolean variables to a convex polyhedron that describes the possible values of numeric variables. This is captured by the partial concretization γ_{b}:

$$\gamma_{\mathrm{b}}(S) = \left\{\mathbf{b} \mapsto \bigwedge_{B_i[\mathbf{b}]} c_i \;\middle|\; \mathbf{b} \in \mathbb{B}^m\right\}$$

$$\mathit{eliminateR}(x_{(j)}, S) \equiv E_0 \cup \{\, \mathit{combine}(j, B_+ \to c_+, B_- \to c_-) \mid B^+ \to c_+ \in E_+, \\ B_- \to c_- \in E_- \,\},$$

where

$$E_0 = \{B_i \to a_i x \succ d_i \in S \mid a_i(j) = 0\}$$
$$E_+ = \{B_i \to a_i x \succ d_i \in S \mid a_i(j) > 0\}$$
$$E_- = \{B_i \to a_i x \succ d_i \in S \mid a_i(j) < 0\}$$
$$\mathit{combine}(j, B_1 \to c_1, B_2 \to c_2) = B_1 \wedge B_2 \to \lambda_1 c_1 + \lambda_2 c_2, \text{ s.t. } \lambda_1, \lambda_2 > 0, \text{ and} \\ \text{in } \lambda_1 c_1 + \lambda_2 c_2, x_{(j)} \text{ appears with coefficient } 0$$

Fig. 1. Elimination of the variable $x_{(j)}$. Assuming that every equality that contains $x_{(j)}$ was replaced by a pair of inequalities.

The notion of partial concretization is useful when we want to show that we correctly lift operations on sets of constraints (e.g., projection) from linear constraints to implications $B_i \to c_i$. We normally want a operation to commute with γ_b, i.e., $\gamma_\mathrm{b}(f_\mathrm{lifted}(S))(\mathbf{b}) = f_\mathrm{original}(\gamma_\mathrm{b}(S)(\mathbf{b}))$, which would mean that there is no loss of precision on the Boolean level.

Without loss of generality, we assume that in every abstract state, the 0-th constraint has the form $B_0 \to \mathit{cfalse}$, and no other constraint has *cfalse* on the right-hand side. In particular, the empty polyhedron is represented by $\bot = \{\mathit{true} \to \mathit{cfalse}\}$ and the universal polyhedron is represented by $\top = \{\mathit{false} \to \mathit{cfalse}\}$.

Example 1. The abstract state where $x_{(0)}$ is always non-negative, and in addition, if $b_{(0)}$ holds, $x_{(0)}$ is not greater than 1 can be represented as

$$\{\mathit{false} \to \mathit{cfalse},\ \mathit{true} \to x_{(0)} \geq 0,\ b_{(0)} \to x_{(0)} \leq 1\}$$

3.1 Elimination of a Rational Variable

In constraint-only representation, existential quantifier elimination (projection) is the main operation on polyhedra, and most other operations are expressed using projection.

We can naturally adapt Fourier-Motzkin elimination [14,30,18] to our abstract domain in the following way. Let an abstract element S be $\{B_i \to c_i\}_{i=0..k}$ and let $x_{(j)}$ be the variable to eliminate. First, we split every equality where $x_{(j)}$ appears with nonzero coefficient into a pair of inequalities. Then, we partition the constraints into three sets:

1. E_0, where in the linear part $x_{(j)}$ appears with coefficient 0;
2. E_+, where in the linear part $x_{(j)}$ appears with a positive coefficient;
3. E^-, where in the linear part $x_{(j)}$ appears with a negative coefficient.

The constraints from E_0 we keep as-is, and from every pair of constraints in E_+ and E_-, we produce a positive combination, in which $x_{(j)}$ has coefficient 0. The difference from the original Fourier-Motzkin algorithm is that when we combine two constraints, we conjoin their Boolean parts. This is summarized in Fig. 1.

Example 2. Let

$$S = \{true \to x_{(0)} - x_{(1)} = 0, b_{(0)} \to x_{(0)} \geq 0, b_{(1)} \to -x_{(1)} \geq -1\}$$

and let us apply the Fourier-Motzkin-based elimination to the variable $x_{(0)}$. First, we partition the constraints into the three sets:

$$\begin{aligned} E_0 &= \{b_{(1)} \to -x_{(1)} \geq -1\} \\ E_+ &= \{true \to x_{(0)} - x_{(1)} \geq 0,\ b_{(0)} \to x_{(0)} \geq 0\} \\ E_- &= \{true \to -x_{(0)} + x_{(1)} \geq 0\} \end{aligned}$$

We keep the elements of E_0 and combine the elements of E_+ and E_-, producing the set

$$\begin{aligned} &\{b_{(1)} \to -x_{(1)} \geq -1,\ true \to 0 \geq 0,\ b_{(0)} \to x_{(1)} \geq 0\} = \\ &\{b_{(1)} \to -x_{(1)} \geq -1,\ b_{(0)} \to x_{(1)} \geq 0\} \end{aligned}$$

In this case, we only need to eliminate the trivially valid constraint $true \to 0 \geq 0$; in general Fourier-Motzkin elimination can produce constraints that are non-trivially redundant.

Lemma 1. *For every abstract state S, rational variable $x_{(j)}$, and $\mathbf{b} \in \mathbb{B}^m$,*

$$\gamma_{\mathrm{b}}(\mathit{eliminateR}(x_{(j)}, S))(\mathbf{b}) \leftrightarrow \exists x_{(j)}.\ \gamma_{\mathrm{b}}(S)(\mathbf{b})$$

Proof Idea. To prove Lemma 1, we can pick an arbitrary $\mathbf{b} \in \mathbb{B}^m$ and show that the set of linear constraints $\{c_i \mid B_i \to c_i \in \mathit{eliminateR}(j, S) \wedge B_i[\mathbf{b}]\}$ is the same as the set of constraints produced by applying standard Fourier-Motzkin elimination to $\gamma_{\mathrm{b}}(S)(\mathbf{b})$.

To eliminate multiple rational variables, we apply *eliminateR* iteratively. The standard heuristic is to pick and eliminate in every step a variable that minimizes $|E_+||E_-| - |E_+| - |E_-|$, which is the upper bound on the growth of the number of constraints.

Gaussian Elimination When an abstract element contains an equality $true \to ax = d$, where $a_{(j)} \neq 0$, this equality can be used as a definition of the variable $x_{(j)}$. Then, to eliminate the $x_{(j)}$ from an abstract element, we can replace it with this definition in every remaining constraint, instead of performing Fourier-Motzkin elimination. This is useful for eliminating, e.g, temporary variables that an analysis may introduce when pre-processing the program.

Example 3. Let

$$S = \{true \to x_{(0)} - x_{(1)} = 0, b_{(0)} \to x_{(0)} \geq 0, b_{(1)} \to -x_{(1)} \geq -1\}$$

and let us apply the Gaussian elimination to the variable $x_{(0)}$, using the equality $true \to x_{(0)} - x_{(1)} = 0$. That is, we replace $x_{(0)}$ with $x_{(1)}$ in the two remaining constraints, getting

$$\{b_{(0)} \to x_{(1)} \geq 0,\ b_{(1)} \to -x_{(1)} \geq -1\}$$

This can be generalized to the case when the abstract element contains a subset of equalities $\{B_j \to a_j x = d_j\}_{j=1..m} \subseteq S$, s.t. $\bigvee_{j=1}^{m} B_j = true$, as shown in Fig. 2

$$gaussEliminateR(x_{(j)}, S) \equiv \{combine(j, B_i \to c_i, B_= \to c_=) \mid B_i \to c_i \in S, B_= \to c_= \in E_=\},$$

where

$$E_= = \{B_j \to a_j x = d_j\}_{j=1..m} \in S, \text{ s.t. } a_{j_{(x)}} \neq 0, \bigvee_{j=1}^{m} B_j = true,$$

$$combine(j, B_1 \to c_1, B_2 \to c_2) = B_1 \wedge B_2 \to \lambda_1 c_1 + \lambda_2 c_2, \text{ s.t. } \lambda_1 > 0, \text{ and}$$
$$\text{in } \lambda_1 c_1 + \lambda_2 c_2, x_{(j)} \text{ appears with coefficient } 0$$

Fig. 2. Generalization of Gaussian elimination of the variable $x_{(j)}$.

3.2 Equivalent and Redundant Constraints

When working with constraint-only representation of polyhedra, one of the big challenges is eliminating redundant constraints. As shown above, every round of Fourier-Motzkin elimination creates a quadratic number of new constraints, an most of them are usually redundant. When eliminating multiple variables (notably, during join computation, see Section 3.3), redundant constraints have to be eliminated regularly, otherwise their number might grow in a double-exponential way (while McMullen's upper bound theorem [25] implies that the number of non-redundant constraints cannot grow more than exponentially with the number of projected dimensions). In his work on constraint-only representation of polyhedra [13], A. Fouilhé argues for redundancy elimination after eliminating every variable. The conventional approach to redundancy elimination in a list of n constraints is to go over every constrain and use linear programming test whether it is redundant with respect to the $n-1$ other ones (some other criteria [20,18] cannot eliminate all redundancies. "Raytracing" [24] is a fast method to identify redundancies, but it degenerates into the conventional linear programming approach in the worst case). We adapt that approach to the Boolean setting: to check whether a constraint is redundant, we call an SMT solver. We also implement a number of less costly redundancy checks.

Pairwise Redundancy Checks There is a number of reductions that can be implemented without necessarily calling an SMT solver.

First, we can combine constraints with identical linear part:

$$\{B_1 \to c, B_2 \to c\} \equiv \{B_1 \vee B_2 \to c\}$$

This is an important step that allows to not duplicate linear constraints; duplication would be amplified by Fourier-Motzkin elimination.

Second, we can eliminate a constraint if it is implied by another constraint:

$$\text{if } B_2 \to B_1 \wedge c_1 \to c_2, \text{ then}$$
$$\{B_1 \to c_1, B_2 \to c_2\} \equiv \{B_1 \to c_1\}$$

This requires a procedure to efficiently check implication between Boolean formulas, which is available, e.g., if they are represented as BDDs. Implication between a pair of linear constraints is a straightforward syntactic check.

Pairwise reduction checks reduce the number of SMT calls, which are costly. This is especially important in lower dimensions and when few constraints are relational. In these cases, most of redundant constraints can be eliminated with pairwise checks.

SMT-Based Redundancy Check Let $S = \{B_i \to c_i\}_{i=0..k}$. Then the j-th constraint is redundant, if its negation is unsatisfiable with respect to the other constraints:

$$\mathit{isRedundant}(j, S) \equiv B_j \wedge \neg c_j \wedge \bigwedge_{i=0..k, i \neq j} (B_i \to c_i) \text{ is UNSAT}$$

An SMT-based redundancy check is an expensive operation, but it has to be performed regularly to limit the growth of the number of constraints.

In general, for a given abstract state S, there may be no unique smallest set of non-redundant constraints. Currently, we implement a greedy strategy: we successively check every constraint and if it is redundant, immediately eliminate it, before checking other constraints. We can artificially make this procedure deterministic by ordering the constraints; in particular it is beneficial to first attempt to remove constraints with larger absolute values of coefficients, both for human-readable output and for performance of an SMT solver.

Example 4. Let

$$S = \{\mathit{true} \to x_{(0)} \geq 0,\ b_{(0)} \to x_{(0)} \geq -1,\ b_{(1)} \to x_{(1)} \geq 0,\ b_{(1)} \to x_{(0)} + x_{(1)} \geq 0\}$$

Let us remove redundant constraints from this system. First, we note that $(\mathit{true} \to x_{(0)} \geq 0) \to (b_{(0)} \to x_{(0)} \geq -1)$, since $b_{(0)} \to \mathit{true}$ and $x_{(0)} \geq 0 \to x_{(0)} \geq -1$, thus the latter constraint is redundant. Second, we note that:

$$(\mathit{true} \to x_{(0)} \geq 0) \wedge (b_{(1)} \to x_{(1)} \geq 0) \wedge b_{(1)} \wedge x_{(0)} + x_{(1)} < 0 \text{ is UNSAT}$$

Thus, the remaining non-redundant constraints are:

$$\{\mathit{true} \to x_{(0)} \geq 0,\ b_{(1)} \to x_{(1)} \geq 0\}$$

3.3 Join

To perform join of two abstract elements, we adapt the projection-based convex hull computation of F. Benoy et al [7,33]. The original algorithm is based on the observation that every point in the convex hull is a convex combination of a pair of points from the original polyhedra. Fig. 3 expresses this more formally and adapted to our setting. Given the two abstract elements S_1 and S_2, first we construct the set of constraints S_{12}. The variables λ_1, λ_2 are the scaling coefficients of the two points in S_1 and S_2 respectively, s.t. $\lambda_1, \lambda_2 \geq 0$ and $\lambda_1 + \lambda_2 = 1$; y_1 an y_2 are the vectors of coordinates of the two points, pre-multiplied by scaling coefficients, and thus should satisfy the pre-multiplied constraints of S_1 and S_2; finally x is the vector of coordinates of a point in the convex hull, and thus $x = y_1 + y_2$. Eliminating y_1, y_2, λ_1, and λ_2 from S_{12} produces the closure of the convex hull. With some extra bookkeeping, it is then possible to express the resulting linear constraints in terms of the original constraints [13] and turn some closed constraints back into open (a positive combination a set of constraints, where at least one constraints is open, is also open).

$$
\begin{aligned}
&\mathit{convexHull}(S_1, S_2) \equiv \mathit{eliminateR}(\{y_1, y_2, \lambda_1, \lambda_2\}, S_{12}), \text{ where}\\
&S_{12} = \{B_i^1 \to a_i^1 y_1 \geq d_i^1 \lambda_1 \mid B_i^1 \to a_i^1 x \succ d_i^1 \in S_1\} \cup\\
&\qquad \{B_i^2 \to a_i^2 y_2 \geq d_i^2 \lambda_2 \mid B_i^2 \to a_i^2 x \succ d_i^2 \in S_2\} \cup\\
&\qquad \{\mathit{true} \to \lambda_1 \geq 0,\ \mathit{true} \to \lambda_2 \geq 0,\ \mathit{true} \to \lambda_1 + \lambda_2 = 1\} \cup\\
&\qquad \{\mathit{true} \to x_{(j)} = y_1(j) + y_2(j) \mid j = 1..n\} \cup\\
&\qquad \{B_0^1 \to \lambda_1 = 0,\ B_0^2 \to \lambda_2 = 0 \mid B_0^1 \to \mathit{cfalse} \in S_1,\ B_0^2 \to \mathit{cfalse} \in S_2\}
\end{aligned}
$$

Fig. 3. Convex hull of two abstract states. The sign $\geq$ stands for the closed version of the corresponding sign $\succ$. When $\succ$ is $>$, $\geq$ is $\geq$; otherwise $\geq$ is the same as $\succ$.

Lemma 2. *For every pair of abstract states S_1, S_2 and every $\mathbf{b} \in \mathbb{B}^m$,*

$$\gamma_{\mathrm{b}}(\mathit{join}(S_1, S_2))(\mathbf{b}) = \gamma_{\mathrm{b}}(S_1)(\mathbf{b}) \sqcup \gamma_{\mathrm{b}}(S_2)\mathbf{b}$$

Proof Idea. To prove Lemma 2, similarly to Lemma 1, we can pick an arbitrary $\mathbf{b} \in \mathbb{B}^m$ and show that the set of constraints S_{12} in Fig. 3 is the same as the set of constraints generated by F. Benoy's convex hull applied to $\gamma_{\mathrm{b}}(S_1)(\mathbf{b})$ and $\gamma_{\mathrm{b}}(S_2)\mathbf{b}$.

Join of Elements with Disjoint Pure Boolean Constraints Let S_1, S_2 be a pair of abstract elements:

$$S_1 = \{B_0^1 \to \mathit{cfalse}\} \cup \{B_i^1 \to c_i^1\}_{i=1..n} \qquad S_2 = \{B_0^2 \to \mathit{cfalse}\} \cup \{B_j^2 \to c_j^2\}_{j=1..m},$$

where $\neg B_0^1 \wedge \neg B_0^2 = \mathit{false}$, i.e., their pure Boolean constraints are disjoint, and for a given valuation of Boolean variables $\mathbf{b}$, at least one of the polyhedra $\gamma_{\mathrm{b}}(S_1)(\mathbf{b})$, $\gamma_{\mathrm{b}}(S_2)(\mathbf{b})$ is empty. In this case, S_1 and S_2 can be joined exactly and without computing the convex hull as follows:

$$
\begin{aligned}
\mathit{boolDisjointJoin}(S_1, S_2) \equiv \{B_i^1 \wedge \neg B_0^1 \to c_i^1\}_{i=1..n} \cup \{B_j^2 \wedge \neg B_0^2 \to c_j^2\}_{j=1..m} \cup\\
\{B_0^1 \wedge B_0^2 \to \mathit{cfalse}\}
\end{aligned}
$$

As we later show, this optimization is important for efficient elimination of Boolean variables. Soundness can be shown by writing down the disjunction of logical formulas corresponding to S_1 and S_2, distributing the disjunction over the conjunctions and applying equivalences that follow from $\neg B_0^1 \wedge \neg B_0^2 = \mathit{false}$. Let S_1, S_2 be a pair of abstract elements:

$$S_1 = \{B_0^1 \to \mathit{cfalse}\} \cup \{B_i^1 \to c_i^1\}_{i=1..n} \qquad S_2 = \{B_0^2 \to \mathit{cfalse}\} \cup \{B_j^2 \to c_j^2\}_{j=1..m},$$

where $\neg B_0^1 \wedge \neg B_0^2 = \mathit{false}$. Let us observe the disjunction of their corresponding logical characterizations:

$$\big((B_0^1 \to \mathit{cfalse}) \wedge (\bigwedge_{i=1}^{n} B_i^1 \to c_i^1)\big) \vee \big((B_0^2 \to \mathit{cfalse}) \wedge (\bigwedge_{j=1}^{m} B_j^2 \to c_j^2)\big)$$

Conjoining the pure boolean constraints to numeric constraints

$$=\big((B_0^1 \to \mathit{cfalse}) \wedge (\bigwedge_{i=1}^{n} B_i^1 \wedge \neg B_0^1 \to c_i^1)\big) \vee \big((B_0^2 \to \mathit{cfalse}) \wedge (\bigwedge_{j=1}^{m} B_j^2 \wedge \neg B_0^2 \to c_j^2)\big) =$$

Distributing the disjunction

$$= \bigwedge_{i=1..n, j=1..m} (\neg B_i^1 \vee B_0^1 \vee c_i^1 \vee \neg B_j^2 \vee B_0^2 \vee c_j^2) \wedge$$
$$\bigwedge_{i=1}^{n} (\neg B_i^1 \vee B_0^1 \vee c_i^1 \vee \neg B_0^2 \vee \mathit{cfalse}) \wedge$$
$$\bigwedge_{j=1}^{m} (\neg B_j^2 \vee B_0^2 \vee c_j^2 \vee \neg B_0^1 \vee \mathit{cfalse}) \wedge$$
$$(\neg B_0^1 \vee \mathit{cfalse} \vee \neg B_0^2 \vee \mathit{cfalse})$$

From $\neg B_0^1 \wedge \neg B_0^2 = \mathit{false}$ it follows that $B_0^1 \vee B_0^2 = \mathit{true}$, $\neg B_0^1 \to B_0^2$, and $\neg B_0^2 \to B_0^1$

$$= \bigwedge_{i=1}^{n} (B_i^1 \wedge \neg B_0^1 \to c_i^1) \wedge \bigwedge_{j=1}^{m} (B_j^2 \wedge \neg B_0^2 \to c_j^2) \wedge (B_0^1 \wedge B_0^2 \to \mathit{cfalse})$$

Which is the logical characterization of $\mathit{boolDisjointJoin}(S_1, S_2)$.

3.4 Other Operations

Intersection with a Constraint To intersect an abstract state S with a constraint $B \to c$, we add $B \to c$ to S. To intersect an abstract state S with a linear constraint c, we add $\mathit{true} \to c$ to S. To intersect an abstract state S with a Boolean constraint B, we add $\neg B \to \mathit{cfalse}$ to S.

Linear Assignment The general way to apply a linear assignment $x_{(j)} := ax + d$ is by renaming and elimination. We introduce a fresh variable $x'_{(j)}$ that denotes the value of $x_{(j)}$ after or before the assignment, relate it to $x_{(j)}$, eliminate $x_{(j)}$ and then rename $x'_{(j)}$ into $x_{(j)}$:

$$\mathit{post}(x_{(j)} := ax + d, S) \equiv \mathit{eliminateR}(x_{(j)}, S \cup \{x'_{(j)} = ax + d\})[x'_{(j)}/x_{(j)}]$$
$$\mathit{pre}(x_{(j)} := ax + d, S) \equiv \mathit{eliminateR}(x_{(j)}, S \cup \{x_{(j)} = (ax + d)[x_{(j)}/x'_{(j)}]\})[x'_{(j)}/x_{(j)}]$$

This applies to both invertible (where in $ax + d$, $x_{(j)}$ has a non-zero coefficient) and non-invertible assignments. Invertible assignments (e.g. $a := 2a+1$) can also be implemented by substituting the inverted expressions (e.g. $a \mapsto (a - 1)/2$) into the constraints [12, section 4.2.2.1].

Elimination of a Boolean Variable We use the equivalence $\exists b \in \mathbb{B}.\, \varphi = \varphi[b/\mathit{true}] \vee \varphi[b/\mathit{false}]$ over-approximate logical disjunction with the join operation:

$$\mathit{eliminateB}(b_{(j)}, S) \equiv \mathit{join}(S[b_{(j)}/\mathit{true}], S[b_{(j)}/\mathit{false}])$$

Example 5. Let

$$S = \{b_{(0)} \to x_{(0)} = 0,\ b_{(0)} \to x_{(1)} = 0,\ \neg b_{(0)} \to x_{(0)} = 1,\ \neg b_{(0)} \to x_{(0)} = 1\}$$

That is, when $b_{(0)}$ is *true*, $x_{(0)} = x_{(1)} = 0$, and when $b_{(0)}$ is *false*, $x_{(0)} = x_{(1)} = 0$. To eliminate the single Boolean variable $b_{(0)}$, we take the join of the two abstract elements:

$$S[b_{(0)}/true] = \{true \to x_{(0)} = 0,\ true \to x_{(1)} = 0\}$$
$$S[b_{(0)}/false] = \{true \to x_{(0)} = 1,\ true \to x_{(1)} = 1\}$$

One possible representation of the result is

$$eliminateB(b_{(0)}, S) = \{true \to x_{(0)} \geq 0,\ true \to -x_{(0)} \geq -1,\ true \to x_{(0)} - x_{(1)} = 0\}$$

Example 6. For an example of a join of two Boolean-disjoint abstract states, let us consider the abstract state

$$S = \{true \to x_{(0)} \geq 0,\ b_{(0)} \to -x_{(0)} \geq -1,\ b_{(0)} \neq b_{(1)} \to false\}$$

and let us eliminate the variable $b_{(0)}$ from it. Notice that this abstract state asserts that $b_{(0)} = b_{(1)}$, and thus we expect that the elimination will result in substituting $b_{(0)}$ with $b_{(1)}$ in every constraint. First, we compute

$$S[b_{(0)}/true] = \{true \to x_{(0)} \geq 0,\ true \to -x_{(0)} \geq -1,\ \neg b_{(1)} \to false\}$$
$$S[b_{(0)}/false] = \{true \to x_{(0)} \geq 0,\ b_{(1)} \to false\}$$

Then, we observe that these abstract states are Boolean-disjoint, since $\neg\neg b_{(1)} \wedge \neg b_{(1)} = false$, i.e., we can apply the specialized version on join and, as expected, get

$$\begin{aligned} &boolDisjointJoin(S[b_{(0)}/true], S[b_{(0)}/false]) \\ &= \{b_{(1)} \to x_{(0)} \geq 0,\ b_{(1)} \to -x_{(0)} \geq -1,\ \neg b_{(1)} \to x_{(0)} \geq 0\} \\ &= \{true \to x_{(0)} \geq 0,\ b_{(1)} \to -x_{(0)} \geq -1\} \end{aligned}$$

This example demonstrates a common scenario when eliminating temporary Boolean variables. The eliminated variable may be introduced using an explicit equality, like in this example, or in some similar way that makes it so that restricting this variable to *true* and *false* respectively produces Boolean-disjoint elements. Having a specialized join operation for Boolean-disjoint abstract states is important when an analysis may transform the input program and introduce such variables.

Boolean Assignment An assignment of the form $b_{(j)} := B$ we implement, similarly to the linear case, using renaming and elimination:

$$post(b_{(j)} := B, S) \equiv eliminateB(b_{(j)}, S \cup \{\neg(b'_{(j)} \leftrightarrow B) \to cfalse\})[b'_{(j)}/b_{(j)}]$$
$$pre(b_{(j)} := B, S) \equiv eliminateB(b_{(j)}, S \cup \{\neg(b_{(j)} \leftrightarrow B[b_{(j)}/b'_{(j)}]) \to cfalse\})[b'_{(j)}/b_{(j)}]$$

Linear to Boolean Assignment In some cases, during the analysis we want to introduce an observer variable – a Boolean variable that the stores truth value of some linear constraint at some point of program execution. When c is an inequality (not an equality), the assignment $b_{(j)} := c$ is straightforward to implement, since the equivalence $b \leftrightarrow c$ can be represented as a pair of constraints: $b \rightarrow c, \neg b \rightarrow \neg c$. That is,

$$post(b_{(j)} := c, S) \equiv eliminateB(b_{(j)}, S \cup \{b'_{(j)} \rightarrow c, \neg b'_{(j)} \rightarrow \neg c\})[b'_{(j)}/b_{(j)}]$$

and similarly for *pre*. For an equality $ax = d$, though, we cannot assign its truth value to a single Boolean variable. Instead, we have to use two Boolean variables to separately assign to them the truth values of $ax \geq d$ and $-ax \geq -d$.

Widening Widening in convex polyhedra is based on the idea of keeping the constraints of the previous approximation that are also satisfied by the new approximation [17,3]. In our setting, we want, for every linear constraint from the previous approximation, to find for which values of Boolean variables it is implied by the new approximation. To find for which values of Booleans an inequality c is implied by an abstract state S, we can conjoin $true \rightarrow \neg c$ to S and then eliminate all the rational variables. This produces an abstract state of the form $\{B \rightarrow cfalse\}$ which is interpreted as: when B holds, $\neg c$ is unsatisfiable in S and thus $B \rightarrow c$ is implied by S. Thus, assuming that every equality is first split into a pair of inequalities and that $S_1 \sqsubseteq S_2$, we get:

$$widen(S_1, S_2) \equiv \left\{ B_i^3 \rightarrow c_i^1 \;\middle|\; \begin{array}{l} B_i^1 \rightarrow c_i^1 \in S_1 \wedge \\ eliminateR(x, S_2 \cup \{true \rightarrow \neg c_i^1\}) = \{B_i^3 \rightarrow cfalse\} \end{array} \right\} \cup \{B_0^2 \rightarrow cfalse \mid B_0^2 \rightarrow cfalse \in S_2\}$$

Inclusion Test To check for inclusion between abstract states, we currently use an SMT solver. Let $S_1 = \{B_i^1 \rightarrow c_i^1\}_{i=0..k_1}$ and $S_2 = \{B_j^2 \rightarrow c_j^2\}_{j=0..k_2}$. Then

$$S_1 \sqsubseteq S_2 \equiv \Big(\bigwedge_{i=0}^{k_1} B_i^1 \rightarrow c_i^1 \Big) \wedge \neg \Big(\bigwedge_{j=0}^{k_2} B_j^2 \rightarrow c_j^2 \Big) \text{ is UNSAT}$$

Checking, whether an abstract state S is empty, i.e., whether $S \sqsubseteq \bot$ also requires an SMT solver call.

3.5 Implementation Details

Representing Boolean Formulas We currently propose to represent Boolean formulas with BDDs, the main reason being that BDDs allow to represent formulas in a canonical way and avoid unbounded syntactic growth, when formulas are repeatedly conjoined (during elimination) and disjoined (when combining constraints with coinciding linear part).

Constraints over Integer Variables To achieve additional precision, we can rewrite linear constraints when every variable with a non-zero coefficient is integer. In this case, a strict inequality can be rewritten as non-strict:

$$ax > d \equiv ax \geq d + 1$$

For an inequality over integer variables, we can divide the coefficients of a constraint over integer variables by the GCD of the variable coefficients, rounding the free coefficient towards 0:

$$ax \geq d \equiv (a/g)x \geq round(d/g), \text{ where } g = \gcd a$$

For an equality over integer variables, the free coefficient has to be divisible by the GCD of the variable coefficients, otherwise the equality is unsatisfiable.

4 Implementation and Experiments

We implemented the proposed abstract domain in our abstract interpreter for Horn clauses [6,5]. Our tool can find models of systems of constrained Horn clauses [9] with predicates over numeric and Boolean variables. It is based on the technique of path focusing [27] and uses an SMT solver (Z3) to iterate over relevant disjuncts of the direct consequence relation. As the abstract domain, it supports BddApron [19] and now also the abstract domain that we propose in this paper. The tool is implemented in OCaml.

4.1 Example

Fig. 4 shows an example of a kind of a program that we are interested in. Fig. 4 is a typical result of instrumenting a program with Boolean observer variables that record which branches were taken during an execution. At every step, this program non-deterministically chooses whether to assume a constraint on a numeric variable $x_{(i)}$, and the choice is recorder in a Boolean variable $b_{(i)}$. At this point, we do not care how exactly this program was obtained, and we are interested in efficiently computing invariants in a way that allows to relate Boolean and numeric variables. Our original motivation though comes from using observer variables for trace partitioning in array-manipulating programs [29], where different branches correspond to different relations between array indices.

Fig. 5 encodes the example program as a system of Horn clauses that can be processed by our tool. In this system, predicates $P_0, \cdots, P_3$ denote the invariants of the four program locations, and every Horn clause corresponds to one transition (in general, a clause may encode multiple sequences of statements). The smallest model of the system in Fig. 5 is the collecting semantics of the program in Fig. 4.

The original implementation of our tool used the BddApron abstract domain, and the invariant that it infers for the predicate P_3 (the final program location) consists of 8

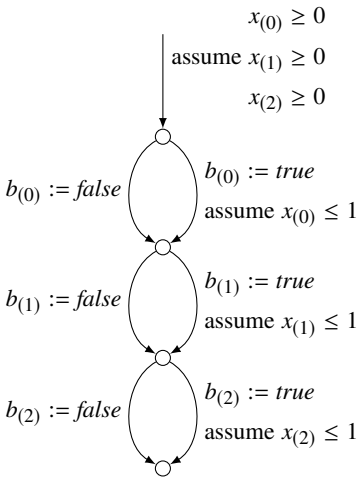

Fig. 4. An example of a program instrumented with observer variables.

$$x_{(0)} \geq 0 \wedge x_{(1)} \geq 0 \wedge x_{(2)} \geq 0 \rightarrow P_0(x, b)$$
$$P_0(x, b) \wedge \neg b_{(0)} \rightarrow P_1(x, b)$$
$$P_0(x, b) \wedge b_{(0)} \wedge x_{(0)} \leq 1 \rightarrow P_1(x, b)$$
$$P_1(x, b) \wedge \neg b_{(1)} \rightarrow P_2(x, b)$$
$$P_1(x, b) \wedge b_{(1)} \wedge x_{(1)} \leq 1 \rightarrow P_2(x, b)$$
$$P_2(x, b) \wedge \neg b_{(2)} \rightarrow P_3(x, b)$$
$$P_2(x, b) \wedge b_{(2)} \wedge x_{(2)} \leq 1 \rightarrow P_3(x, b)$$

Fig. 5. An encoding of the program in Fig. 4 into Horn clauses for our tool.

polyhedra, one for every valuation of Boolean variables:

$$\begin{aligned}
&(\neg b_{(0)} \neg b_{(1)} \neg b_{(2)} \wedge x_{(0)} \geq 0 \wedge x_{(1)} \geq 0 \wedge x_{(2)} \geq 0) \vee \\
&(\neg b_{(0)} \neg b_{(1)} b_{(2)} \wedge x_{(0)} \geq 0 \wedge x_{(1)} \geq 0 \wedge 1 \geq x_{(2)} \geq 0) \vee \\
&\cdots \vee \\
&(b_{(0)} b_{(1)} b_{(2)} \wedge 1 \geq x_{(0)} \geq 0 \wedge 1 \geq x_{(1)} \geq 0 \wedge 1 \geq x_{(2)} \geq 0)
\end{aligned}$$

In a larger program, such an invariant would be propagated further, with every post-condition computation begin essentially repeated for each of the eight polyhedra (i.e., exponentially many times in the number of Boolean variables).

The implementation of the domain that we propose in this paper allows to represent P_3 in a much more compact form:

$$\begin{aligned}
&\{\, \mathit{true} \rightarrow x_{(0)} \geq 0,\ \mathit{true} \rightarrow x_{(1)} \geq 0,\ \mathit{true} \rightarrow x_{(2)} \geq 0, \\
&\ \ b_{(0)} \rightarrow -x_{(0)} \geq -1,\ b_{(1)} \rightarrow -x_{(1)} \geq -1,\ b_{(2)} \rightarrow -x_{(2)} \geq -1 \,\}
\end{aligned}$$

4.2 Experiments

We evaluate the performance of the implementation using two sets of programs. For both sets, we measure the total time it took to run on every program a single forward analysis with narrowing. We summarize the results in Table 1. Time figures were obtained on a PC with a Core i7-3630QM CPU and 8GB RAM.

SV-COMP Programs For the first set of experiments, we selected a number of programs from "loop" and "recursive" categories of the Competition on Software Verification SV-COMP [1] and translated them into Horn clauses (the input language of our tool) with the

tool SeaHorn [16] using two different Clang optimization levels `-O3` and `-O0` (SeaHorn operates on LLVM bytecode). This way we obtained 123 systems of Horn clauses. By default, SeaHorn uses a version of large block encoding [8] and produces programs with relatively few locations, but with complicated transition relations and a large number of temporary Boolean and numeric variables; even a simple C program can produce a good benchmark for the implementation of an abstract domain. In Appendix A, we show an example of a C program and the corresponding system of Horn clauses produced by SeaHorn. On SV-COMP programs, the implementation of the proposed domain is 2-10 times slower than BddApron; about 5 times slower on average.

Hand-Crafted Programs For the second set, we selected 10 hand-crafted programs coming from different sources: array-manipulating programs encoded using array abstraction of L. Gonnord and D. Monniaux [28], other programs that use trace partitioning with observer variables, etc. With hand-crafted examples, we noticed that some of SMT queries that test constraint for redundancy cause the solver (Z3 4.5.0) to reach timeout, which we set at 10 seconds. This does not make the analysis unsound; a timeout of a redundancy check only causes the analysis to keep a redundant constraint in an abstract element. We have not yet found a workaround, and we display the hand-crafted programs in two rows: all programs (10) and programs that do not cause solver timeouts (8). On hand-crafted programs without solver timeout, the implementation of the proposed domain is 2-10 times slower than BddApron; about 7 times slower on average.

Conclusion On average, the current implementation of the proposed abstract domain is about 5-7 times slower than BddApron. We find this result promising (given that this is our initial prototype implementation) and it shows directions for future improvement. In particular, much of the analysis time is spent in SMT solver calls in order to detect redundant constraints. These calls are costly, but have to be performed regularly. We are going to address the performance of eliminating redundant constraints in future work.

5 Conclusion and Future Work

In this paper, we propose a new relational abstract domain for analysing programs with numeric and Boolean variables. The main idea is to represent an abstract state as a set of linear constraints over numeric variables, with every constraint being enabled by a formula over Boolean variables. This allows, unlike in some existing approaches, avoiding the duplication of linear constraints shared by multiple Boolean formulas. Currently,

Table 1. Experimental results

Program set	#	Total time, s	
		BddApron	This paper
SV-COMP	123	9.2	52
Hand-crafted, no solver timeout	8	0.9	6.8
Hand-crafted, all	10	1.6	113.5

we use the simple formulations of Fourier-Motzkin elimination [30] and projection-based convex hull [7,33], and we rely on an SMT solver for redundancy elimination and inclusion checks (the counterpart of systematically using linear programming). Our experiments have shown that this is a worthy combination, which avoids some of the inefficiencies of earlier works.

The main direction for future work is to improve the performance of eliminating redundant constraints. There may be multiple ways to do this.

First, we may find additional heuristics that will reduce the number calls to a complete elimination procedure (that now calls an SMT solver). For example, A. Maréchal and M. Périn propose a fast incomplete procedure to detect non-redundant constraints based on raytracing [23], and there may be a way to adapt it (or a similar heuristic) to our setting.

Second, we may replace the SMT calls with a specialized procedure that combines LP-based and BDD-based reasoning. In particular, the observation is that a constraint is non-redundant, if it is non-redundant for at least one valuation of Boolean variables. While, an abstract element in the worst case describes exponentially many (in the number of constraints) convex polyhedra, there may be a way to not enumerate all of them during the redundancy check, at least in the average case.

Third, we may attempt to adapt to our setting the state-of-the art algorithms for constraint-only polyhedra, but this is not straightforward. For example, we cannot immediately adapt algorithms that require an interior point, such as those on parametric linear programming (for projection and convex hull) and ray-tracing (for redundancy elimination), by A. Maréchal, D. Monniaux, and M. Périn [23,22,21]: different Boolean assignments may have different interior points (in case of polyhedra with empty interior, we consider the interior relative to the affine span; but again this depends on the affine span). This is unfortunate, since much time is currently spent inside the SMT solver for checking for redundancy, and parametric linear programming is more efficient than Fourier-Motzkin elimination. A possible workaround, to be explored, is to partition the Boolean space according to affine span and point in the relative interior.

Regardless of the issues related to redundancy, presence of Boolean constraints often prevents us from using some standard approaches to representing polyhedra. In computations over convex polyhedra, one usually maintains, in addition to a system of inequalities, a system of linear inequalities that defines the affine span of the polyhedron. Given an ordering over the dimensions, this system of equalities may be echelonized and used to eliminate variables from the system of inequalities. The resulting system of equalities and non-redundant, normalized inequalities is canonical [2]. In our case, the affine span may depend on the Boolean part, thus it is impossible to canonicalize the inequalities uniformly with respect to the Booleans. We intend to investigate partitioning the Boolean space according to the affine span.

---

[2] In the case of polyhedra with nonempty interior, an non-redundant system of normalized inequalities canonically describes a polyhedron: each inequality corresponds to a face. This is not true in the general case: both $x \leq y \wedge y \leq x \wedge 0 \leq x \wedge x \leq 1$ and $x \leq y \wedge y \leq x \wedge 0 \leq x \wedge y \leq 1$ are non-redundant systems defining the same polyhedron. Its affine span is defined by $x = y$, then one can rewrite the inequalities using this equality and obtain $x = y \wedge 0 \leq y \wedge y \leq 1$, which is canonical.

References

1. Competition on software verification (SV-COMP), http://sv-comp.sosy-lab.org/, last accessed in April 2018.
2. Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA. IEEE (2009)
3. Bagnara, R., Hill, P.M., Ricci, E., Zaffanella, E.: Precise widening operators for convex polyhedra. Sci. Comput. Program. 58(1-2), 28–56 (2005)
4. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. STTT 9(3-4), 413–414 (2007)
5. Bakhirkin, A.: HCAI, a path focusing abstract interpreter for Horn clauses, https://gitlab.com/abakhirkin/hcai, last accessed in April 2018
6. Bakhirkin, A., Monniaux, D.: Combining forward and backward abstract interpretation of Horn clauses. In: Ranzato [31], pp. 23–45
7. Benoy, F., King, A., Mesnard, F.: Computing convex hulls with a linear solver. TPLP 5(1-2), 259–271 (2005)
8. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA [2], pp. 25–32
9. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. LNCS, vol. 9300, pp. 24–51. Springer (2015)
10. Chaki, S., Gurfinkel, A., Strichman, O.: Decision diagrams for linear arithmetic. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA [2], pp. 53–60
11. Chen, J., Cousot, P.: A binary decision tree abstract domain functor. In: Blazy, S., Jensen, T. (eds.) Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9291, pp. 36–53. Springer (2015)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978)
13. Fouilhé, A.: Revisiting the abstract domain of polyhedra : constraints-only representation and formal proof. (Le domaine abstrait des polyèdres revisité : représentation par contraintes et preuve formelle). Ph.D. thesis, Université Grenoble Alpes, France (2015)
14. Fourier, J.: Note, second extrait. Histoire de l'Académie pour 1824, p. XLVII, vol. 2, pp. 325–328. Gauthier-Villars, Paris (1890), http://gallica.bnf.fr/ark:/12148/bpt6k33707/f330
15. Gurfinkel, A., Chaki, S.: Boxes: A symbolic abstract domain of boxes. In: Cousot, R., Martel, M. (eds.) Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6337, pp. 287–303. Springer (2010)
16. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification (CAV). LNCS, vol. 9206, pp. 343–361. Springer (2015)
17. Halbwachs, N.: Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Ph.D. thesis, Université Scientifique et Médicale de Grenoble & Institut

National Polytechnique de Grenoble (Mar 1979), https://tel.archives-ouvertes.fr/tel-00288805

18. Imbert, J.: Fourier's elimination: Which to choose? In: PPCP. pp. 117–129 (1993)
19. Jeannet, B.: Bddapron, http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/, last accessed in April 2018. To our knowledge, there is no corresponding publication.
20. Kohler, D.: Projections of convex polyhedral sets. Ph.D. thesis, University of California, Berkeley (1967)
21. Maréchal, A.: New Algorithmics for Polyhedral Calculus via Parametric Linear Programming. (Nouvelle Algorithmique pour le Calcul Polyédral via Programmation Linéaire Paramétrique). Ph.D. thesis, Université Grenoble Alpes, France (2017)
22. Maréchal, A., Monniaux, D., Périn, M.: Scalable minimizing-operators on polyhedra via parametric linear programming. In: Ranzato [31], pp. 212–231
23. Maréchal, A., Périn, M.: Efficient elimination of redundancies in polyhedra by raytracing. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10145, pp. 367–385. Springer (2017)
24. Maréchal, A., Périn, M.: Efficient elimination of redundancies in polyhedra by raytracing. In: Bouajjani, A., Monniaux, D. (eds.) Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10145, pp. 367–385. Springer (2017)
25. McMullen, P.: The maximum numbers of faces of a convex polytope. Mathematika 17, 179–184 (1970)
26. Monniaux, D., Alberti, F.: A simple abstraction of arrays and maps by program translation. In: Static analysis (SAS). Lecture Notes in Computer Science, vol. 9291, pp. 217–234. Springer Verlag (2015)
27. Monniaux, D., Gonnord, L.: Using bounded model checking to focus fixpoint iterations. In: Yahav, E. (ed.) Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6887, pp. 369–385. Springer (2011)
28. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free Horn clauses. In: Static analysis. Lecture Notes in Computer Science, vol. 9837. Springer Verlag (Sep 2016)
29. Monniaux, D., Gonnord, L.: Cell morphing: From array programs to array-free Horn clauses. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 361–382. Springer (2016)
30. Motzkin, T.S.: Beiträge zur Theorie der Linearen Ungleichungen. Ph.D. thesis, Universität Zürich (1936)
31. Ranzato, F. (ed.): Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings, Lecture Notes in Computer Science, vol. 10422. Springer (2017)
32. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5), 26 (2007)
33. Simon, A., King, A.: Exploiting sparsity in polyhedral analysis. In: Hankin, C., Siveroni, I. (eds.) Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3672, pp. 336–351. Springer (2005)

A Input Example

Fig. 6 shows an example of a C program. Fig. 7 in the next page shows the corresponding system of Horn clauses produced by SeaHorn.

```
int main() {
  int x = 0, y = 0;
  while (x <= 99) {
    ++x;
    ++y;
  }
  assert(x == 100 && y == 100);
}
```

Fig. 6. An example of a C program

```
(rule (verifier.error false false false))
(rule (verifier.error false true true))
(rule (verifier.error true false true))
(rule (verifier.error true true true))
(rule main@entry)
(rule (=>
  (and main@entry
    true
    (=> main@_bb_0 (and main@_bb_0 main@entry_0))
    main@_bb_0
    (=> (and main@_bb_0 main@entry_0) (= main@%y.0.i2_0 0))
    (=> (and main@_bb_0 main@entry_0) (= main@%x.0.i1_0 0))
    (=> (and main@_bb_0 main@entry_0) (= main@%y.0.i2_1 main@%y.0.i2_0))
    (=> (and main@_bb_0 main@entry_0) (= main@%x.0.i1_1 main@%x.0.i1_0)))
  (main@_bb main@%x.0.i1_1 main@%y.0.i2_1)))
(rule (=>
  (and (main@_bb main@%x.0.i1_0 main@%y.0.i2_0)
    true
    (= main@%_1_0 (+ main@%x.0.i1_0 1))
    (= main@%_2_0 (+ main@%y.0.i2_0 1))
    (= main@%_3_0 (<= main@%_1_0 99))
    (=> main@_bb_1 (and main@_bb_1 main@_bb_0))
    main@_bb_1
    (=> (and main@_bb_1 main@_bb_0) main@%_3_0)
    (=> (and main@_bb_1 main@_bb_0) (= main@%y.0.i2_1 main@%_2_0))
    (=> (and main@_bb_1 main@_bb_0) (= main@%x.0.i1_1 main@%_1_0))
    (=> (and main@_bb_1 main@_bb_0) (= main@%y.0.i2_2 main@%y.0.i2_1))
    (=> (and main@_bb_1 main@_bb_0) (= main@%x.0.i1_2 main@%x.0.i1_1)))
  (main@_bb main@%x.0.i1_2 main@%y.0.i2_2)))
(rule (let ((a!1
  (and
    (main@_bb main@%x.0.i1_0 main@%y.0.i2_0)
    true
    (= main@%_1_0 (+ main@%x.0.i1_0 1))
    (= main@%_2_0 (+ main@%y.0.i2_0 1))
    (= main@%_3_0 (<= main@%_1_0 99))
    (=> main@verifier.error_0
        (and main@verifier.error_0 main@_bb_0))
    (=> (and main@verifier.error_0 main@_bb_0) (not main@%_3_0))
    (=> (and main@verifier.error_0 main@_bb_0)
        (= main@%.lcssa5_0 main@%_2_0))
    (=> (and main@verifier.error_0 main@_bb_0)
        (= main@%.lcssa_0 main@%_1_0))
    (=> (and main@verifier.error_0 main@_bb_0)
        (= main@%.lcssa5_1 main@%.lcssa5_0))
    (=> (and main@verifier.error_0 main@_bb_0)
        (= main@%.lcssa_1 main@%.lcssa_0))
    (=> main@verifier.error_0 (= main@%_4_0 (= main@%.lcssa_1 100)))
    (=> main@verifier.error_0
        (= main@%_5_0 (= main@%.lcssa5_1 100)))
    (=> main@verifier.error_0
        (= main@%or.cond.i_0 (and main@%_4_0 main@%_5_0)))
    (=> main@verifier.error_0 (not main@%or.cond.i_0))
    (=> main@verifier.error.split_0
        (and main@verifier.error.split_0 main@verifier.error_0))
    main@verifier.error.split_0)))
  (=> a!1 main@verifier.error.split)))
(query main@verifier.error.split)
```

Fig. 7. System of Horn clauses produced by SeaHorn for the program in Fig. 6.