International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-024-00761-2

GENERAL

Speeding up static analysis with the split operator

Vincenzo Arceri[1] · **Greta Dolcetti**[2] · **Enea Zaffanella**[1]

Accepted: 14 August 2024

Abstract
In the context of abstract interpretation-based static analysis, we propose a new abstract operator modeling the *split* of control flow paths: the goal of the operator is to enable a more efficient analysis when using abstract domains that are computationally expensive, having no negative effect on precision, and occasionally resulting in a more precise analysis. We focus on the case of conditional branches guarded by numeric linear constraints, including implicit numerical branches. We provide an experimental evaluation of real-world test cases, showing that by using the split operator we can achieve significant efficiency improvements with respect to the classical approach for a static analysis based on the domain of convex polyhedra. We also briefly discuss the applicability of this new operator to different, possibly non-numeric abstract domains.

Keywords Abstract interpretation · Static analysis · Abstract operators

1 Introduction

In static analysis tools based on abstract interpretation (AI) it is common to define a neat separation between the AI framework and the abstract domains: the two components communicate by invoking abstract domain operators, such as meet, join, widening and narrowing, whose results are combined to correctly characterize the abstract semantics of the program. Among the operations needed during static analysis, one allows to *filter* an abstract element $A \in \mathbb{A}$ over a given predicate $p \in \mathrm{Pred}$, returning a refined abstract element A_p approximating those states of A that satisfy p:

$$\mathrm{filter}\colon \mathbb{A} \times \mathrm{Pred} \to \mathbb{A}.$$

This operator goes by different names depending on the context. Abstract domains adopt a rather specific terminology, which often highlights the kind of predicates that are supported; for instance, the numeric domains in Apron [25] provide `meet_lincons_array` for filtering on a set of linear constraints; similarly, we can find `add_cons` for the polyhedra domains in the PPLite library [10] and `add_congruences` for the grid domain [4] in the PPL library [6]. Static analysis tools usually adopt a more generic name, sometimes promoting the operator to an abstract instruction in their program representation: for instance, we have `Comparison` in IKOS AR [12], `S_assume` in MOPSA [28] and `assume` in both Clam/Crab [21] and LiSA [16, 31]. The filter operator can be used to enforce properties that are known to hold at a specific program point: for instance, after a call to a mock library function, we can filter the abstract state on the post-condition of the function.

The same operator is also commonly used to model the conditional split of the control flow of the program. For instance, the code fragment

```
if (i < 50)
  do_this(i);
else
  do_that(i);
```

can be modeled as shown in Fig. 1: the input abstract element A is cloned to obtain two copies; these are then filtered by the predicate $p = (i < 50)$ for the `then` branch and on its complement $\neg p = (i \geq 50)$ for the `else` branch.

While this implementation approach is both correct and precise, in some contexts it could incur avoidable inefficiencies, as it replicates most of the work done to evaluate the complementary predicates in the two program branches. The overhead is probably negligible, hence acceptable, as long as considering the most efficient abstract domains, such as the domain of intervals [14]. Things might be different when

✉ V. Arceri
vincenzo.arceri@unipr.it

G. Dolcetti
greta.dolcetti@unive.it

E. Zaffanella
enea.zaffanella@unipr.it

[1] University of Parma, Parco Area Delle Scienze, 53/A, Parma, Italy

[2] Ca' Foscari University of Venice, Via Torino 155, 30172 Mestre, Venice, Italy

Published online: 27 August 2024

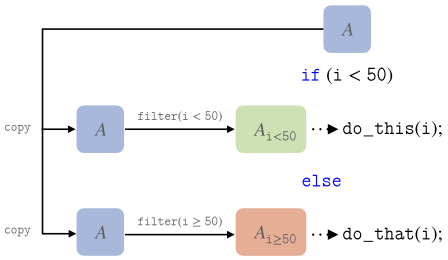

Fig. 1 Modeling a branch using the filter operator

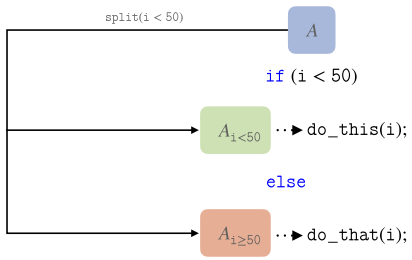

Fig. 2 Modeling a branch using the split operator

adopting more expensive and precise abstract domains, such as the domain of convex polyhedra [15]; in these cases, it might be worth exploring alternative implementation strategies to better preserve efficiency. In particular, in Fig. 2 we show an alternative approach where the conditional branch is modeled using the *split* operator:

$$\text{split}\colon \mathbb{A} \times \text{Pred} \longrightarrow \mathbb{A} \times \mathbb{A}.$$

This new abstract domain operator, initially proposed in [9], filters the input abstract element on a predicate p and on its complement $\neg p$ *at the same time*, allowing for the factorization of any replicated computational effort. Note that, if no optimization is possible, the abstract domain can just resort to the default implementation, which clones the abstract element and invokes (twice) the filter operator.

While this idea is both simple and intuitively promising, to the best of our knowledge its effectiveness has never been evaluated in the context of classical AI-based static program analysis:[1] the goal of this paper is to provide such an evaluation. To this end, we focus on the numerical context considering the abstract domain of convex polyhedra [15], a precise but computationally expensive relational domain. In particular, we define the split operator on linear predicates, describing how it behaves when the predicate deals with rational or integral variables. Then, we describe the design changes that are required, both at the interface and at the implementation levels, in order to accommodate the new split operator into an existing static analysis tool. Finally, we show the results of our experimental evaluation, assessing the effectiveness of the split operator with respect to the classical filter operator for the abstract domain of convex polyhedra.

It is important to highlight that, in principle, the high-level specification of the split operator allows for a general adoption, independently from the considered abstract domain. However, since its end goal is to enable efficiency improvements (while preserving correctness and leaving precision mostly unaffected), its actual effectiveness necessarily depends on *profitability* considerations: these should take into proper account the computational cost of filtering on a specific class of predicates, the frequency of these operations in a typical analysis, and the efficiency gains that could be obtained by the addition of a specialized split operator. Hence, we will also briefly discuss the applicability of the approach in more general contexts.

Paper structure In Sect. 2, we consider the implementation of the filter and split operators on the domain of topologically closed convex polyhedra. In Sect. 2.1 we provide a high-level description of the rational filter operator, identifying those implementation steps that are computationally more expensive; then, in Sect. 2.2 we describe the rational split operator proposed in [10], highlighting the efficiency gains that can be obtained by adopting a specialized implementation. In Sect. 2.3, we discuss how the approach can be extended to the case of integral splits, i.e., those splits defined on a linear inequality predicate whose variables can only assume integral values; in Sect. 2.4 we also consider the special case of integral splits defined on a linear equality constraint. In Sect. 3 we briefly describe the static analysis tool chosen for the experimental evaluation. In particular, we will focus on the changes that have to be applied to the tool in order to accommodate the new abstract operator, affecting both the abstract domain component and the fixpoint approximation engine. The results of the experimental evaluation are described in Sect. 4. We conclude in Sect. 5, also discussing the extension of the approach to different abstract domains and different static analysis tools.

This paper is a revised and extended version of [3]. The main extension is in Sect. 2, where we provide a rather detailed description of the algorithms implementing several variants of the split operator on the domain of convex polyhedra. The prototype implementation described in [3] has been revised and extended: the PPLite library has been enhanced to include the optimized version of all variants of the integral split operator, as described in Sect. 2; also, the modified static analysis tool is now able to apply the split operator to some implicit control flow splits that were ignored before. Moreover, we extended the experimental evaluation in Sect. 4 by considering a larger set of real-world test cases.

[1] The experimental evaluation reported in [9] targets the analysis of a particular class of hybrid systems.

2 Splitting polyhedra on linear constraints

In this section, we discuss how a split operator defined on a numerical predicate can be efficiently implemented on the domain of convex polyhedra [15]. While assuming some familiarity with the basic notions of lattice theory [11], for clarity of exposition we start by introducing some terminology and notation.

A non-trivial, non-strict linear inequality constraint β defines a closed half-space $\mathrm{con}(\{\beta\})$ of the vector space $\mathbb{R}^n$; we write $\neg\beta$ to denote the complement of β, i.e., the open half-space $\mathrm{con}(\{\neg\beta\}) = \mathbb{R}^n \setminus \mathrm{con}(\{\beta\})$; we will also write $\mathrm{cl}(\neg\beta)$ to denote the topologically closed (i.e., non-strict) complement of constraint β.

A not necessarily topologically closed (NNC) convex polyhedron $\phi = \mathrm{con}(C) \subseteq \mathbb{R}^n$ is defined as the set of solutions of a finite system C of (strict or non-strict) linear inequality constraints; equivalently, $\phi = \mathrm{gen}(\mathcal{G})$ can be defined as the set obtained by suitably combining the elements (lines, rays, points and closure points) of a generator system $\mathcal{G} = \langle L, R, P, C\rangle$. The Double Description framework [29] exploits both representations; we write $\phi \equiv (C, \mathcal{G})$ to denote that $\phi = \mathrm{con}(C) = \mathrm{gen}(\mathcal{G})$. The set $\mathbb{P}_n$ of all NNC polyhedra on $\mathbb{R}^n$, partially ordered by set inclusion, is a lattice $\langle \mathbb{P}_n, \subseteq, \emptyset, \mathbb{R}^n, \cap, \uplus\rangle$, where the emptyset and $\mathbb{R}^n$ are the bottom and top elements, the binary meet operator is set intersection and the binary join operator '$\uplus$' is the convex polyhedral hull. Readers interested in more details on the algorithms defined on the abstract domain of NNC convex polyhedra are referred to [7, 8, 10].

The set $\mathbb{CP}_n$ of topologically closed polyhedra on the vector space $\mathbb{R}^n$ is a sublattice of $\mathbb{P}_n$. While being in general less precise, topologically closed polyhedra are preferred when considering the case of variables that can only assume integral values, since in such a context any strict inequality constraint can be safely refined to obtain a more precise non-strict inequality; hence, we can describe a topologically closed polyhedron $\phi = \mathrm{gen}(\mathcal{G})$ using a simpler generator system $\mathcal{G} = \langle L, R, P\rangle$, having no closure point component C.

The algorithms defined to work on the domain of topologically closed polyhedra, having to deal with fewer special cases, happen to be simpler to design, implement and justify. Hence, for exposition purposes, in the following we will focus our explanations on this simpler abstract domain. Moreover, we will only provide high level sketches of the actual implementations, ignoring the handling of corner cases (e.g., the detection of empty results), as well as some auxiliary steps such as the computation of minimal forms for the systems of constraints. We stress that our sketchy descriptions of the algorithms are only meant to highlight the more expensive computational steps to hint at the corresponding optimizations that can be obtained by adopting the new abstract operator. Readers interested in the low-level details are referred to the implementation made available in the open source library PPLite [7, 8, 10].[2]

Pseudocode 1 (Filter operator)

```
1:  function FILTER(φ, β)
2:      let φ ≡ ⟨C, G⟩ and S = sat_cons(C, G);
3:      P ← scalar_products(G, β);
4:      (G+, G0, G−) ← partition(G, P);
5:      G⋆ ← COMBINE-ADJ(G+, G−, P, S);
6:      C1 ← C ∪ {β};                          ▷ build result
7:      if is_equality(β) then
8:          G1 ← G0 ∪ G⋆;
9:      else
10:         G1 ← G+ ∪ G0 ∪ G⋆;
11:     end if
12:     let φ1 ≡ ⟨C1, G1⟩;
13:     return φ1;
14: end function
```

2.1 Filtering on a linear constraint

In Pseudocode 1, we show a function adding a linear equality or non-strict inequality constraint β to (the constraint system C defining) the topologically closed polyhedron $\phi \in \mathbb{CP}_n$, thereby implementing the filter operator. In the Double Description framework, this actually corresponds to the incremental Chernikova conversion procedure [13].

The most expensive computational steps of this function are those in lines 3–5; after computing the scalar products $\mathcal{P}$ of all the generators in $\mathcal{G}$ with the input constraint β, on line 4 the generator system is partitioned into $\mathcal{G}^+$, $\mathcal{G}^0$ and $\mathcal{G}^-$ using the sign of each scalar product; then, on line 5, the generators in $\mathcal{G}^+$ and $\mathcal{G}^-$ are combined so as to produce $\mathcal{G}^\star$. Lines 6–12 compose the Double Description for the resulting polyhedron: note that the generators in $\mathcal{G}^-$ are discarded as they violate constraint β; the same happens for those in $\mathcal{G}^+$, if β is an equality constraint.

Function COMBINE-ADJ, computing $\mathcal{G}^\star$, is described in Pseudocode 2: here each generator $g^+ \in \mathcal{G}^+$ (a generator strictly satisfying the constraint β) is linearly combined with each generator $g^- \in \mathcal{G}^-$ (a generator violating the constraint) using the previously computed scalar products p^+ and p^-, so as to produce a new generator $g^\star$ that saturates the constraint β (i.e., $g^\star$ satisfies the corresponding equality constraint and hence intuitively becomes a new element of $\mathcal{G}^0$). Note that, on line 5, predicate adjacent$(g^+, g^-, \mathcal{S})$ checks that the two considered generators are geometrically adjacent in polyhedron ϕ, so as to eagerly discard those pairs whose combination would necessarily produce a redundant generator. This adjacency test, which combines some theoretical results and implementation techniques proposed in

[2] https://github.com/ezaffanella/PPLite.

Pseudocode 2 (Combining adjacent generators)

```
1:  function COMBINE-ADJ(G+, G−, P, S)
2:      G* ← ∅;
3:      for all g+ ∈ G+ do
4:          for all g− ∈ G− do
5:              if adjacent(g+, g−, S) then
6:                  let p+ = P[g+] and p− = P[g−];
7:                  g* ← combine(g+, g−, p+, p−);
8:                  G* ← G* ∪ {g*};
9:              end if
10:         end for
11:     end for
12:     return G*;
13: end function
```

[27] and [19], turns out to be critical for the efficiency of the whole procedure; it uses the saturation matrix $\mathcal{S}$, which is cached in the representation of polyhedron ϕ and records, for each generator $g \in \mathcal{G}$, the set of constraints in C that are saturated by g. Note that the update of the saturation matrix for the result ϕ_1 is left implicit in Pseudocode 1.

Example 1
In Fig. 3 (adapted from [10]), we show an example of application of the filter operator. The polyhedron $\phi = \text{gen}(\mathcal{G}) \in \mathbb{CP}_2$ on the upper part of the figure is described by generator system $\mathcal{G} = \langle L, R, P \rangle$, where $L = R = \emptyset$ and $P = \{p_0, p_1, p_2, p_3, p_4\}$. (Since there are no lines or rays, for simplicity in the following we will only refer to the point component P of the generator system $\mathcal{G}$.) When filtering ϕ using constraint β, the generator system is partitioned in

$$\mathcal{G}^+ = \{p_0, p_2, p_3, p_4\},$$
$$\mathcal{G}^0 = \emptyset,$$
$$\mathcal{G}^- = \{p_1\}.$$

Then, we compute

$$\mathcal{G}^\star = \text{COMBINE-ADJ}(\mathcal{G}^+, \mathcal{G}^-, \mathcal{P}, \mathcal{S}) = \{p_5, p_6\},$$

where p_5 and p_6 (which are shown in red in the lower part of the figure) are obtained by combining, respectively, $p_0, p_2 \in \mathcal{G}^+$ with $p_1 \in \mathcal{G}^-$. Note that there is no need to linearly combine $p_3, p_4 \in \mathcal{G}^+$ with $p_1 \in \mathcal{G}^-$, because these pairs of generators are not adjacent: their combination would result in redundant points (which are shown in gray in the lower part of the figure). Hence, the new generator system is

$$\begin{aligned}\mathcal{G}' &= \mathcal{G}^+ \cup \mathcal{G}^0 \cup \mathcal{G}^\star \\ &= \{p_0, p_2, p_3, p_4\} \cup \emptyset \cup \{p_5, p_6\} \\ &= \{p_0, p_2, p_3, p_4, p_5, p_6\},\end{aligned}$$

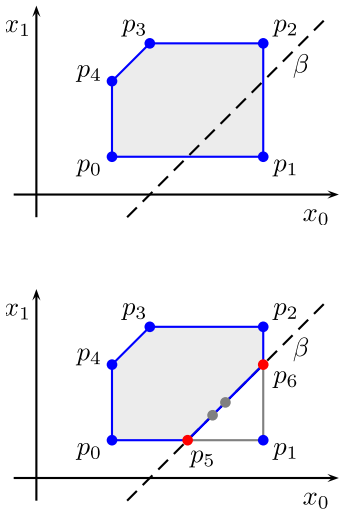

Fig. 3 Filtering on a linear constraint

which represents the filtered polyhedron

$$\phi' = \text{FILTER}(\phi, \beta) = \text{gen}(\mathcal{G}').$$

2.2 Rational split on a linear inequality

The split operator proposed in [9, Sect. 5] was focusing on the case of *rational splits* on linear (strict or non-strict) inequalities, i.e., splits on predicates to be interpreted on an abstract domain modeling non-integral valued variables; the operator was defined for both convex polyhedra domains $\mathbb{P}_n$ and $\mathbb{CP}_n$. In the context of the verification of hybrid systems [17], rational splits can be used to dynamically partition the abstract element describing the states reaching an automaton location to better approximate a continuous flow relation that is not piecewise constant. A similar application of the rational split operator is found in the abstract solving of a non-linear geometric CSP (Constraint Satisfaction Problem) [32, 34], where the current search space is partitioned into subdomains so as to refine the following constraint propagation steps. Rational splits are also useful as helper operators when implementing powerset abstract domains: for instance, given two finite sets S_1, S_2 of polyhedra, the algorithm checking whether S_1 is geometrically covered by S_2, i.e., whether or not the subset inclusion $(\cup S_1) \subseteq (\cup S_2)$ holds, typically requires the splitting of those polyhedra in S_1 that are not included in a polyhedron in S_2.

As discussed in [9], in order to obtain precise approximations, the rational split operator should be defined on the abstract domain $\mathbb{P}_n$ of NNC polyhedra, which provides full support for strict inequalities. In this case, it is possible to implement the split operator so that for each $\phi \in \mathbb{P}_n$ and β a non-strict linear inequality constraint, $\text{split}(\phi, \beta) = (\phi_1, \phi_2)$ will satisfy

Pseudocode 3 (Filter-based rational split)

```
1: function FILTER-BASED-Q-SPLIT(φ, β)
2:     φ' ← φ;                                  ▷ deep copy
3:     β' ← cl(¬β);                             ▷ non-strict complement
4:     φ1 ← FILTER(φ, β);                       ▷ then branch
5:     φ2 ← FILTER(φ', β');                     ▷ else branch
6:     return φ1, φ2;
7: end function
```

- $\phi_1 = \phi \cap \text{con}(\{\beta\})$;
- $\phi_2 = \phi \cap \text{con}(\{\neg\beta\})$;
- $\phi_1 \cup \phi_2 = \phi$; and
- $\phi_1 \cap \phi_2 = \emptyset$.

When adopting the abstract domain $\mathbb{CP}_n$, we necessarily have to consider a non-strict variant of the rational split operator; by using the non-strict complement of β, we obtain that $\text{split}(\phi, \beta) = (\phi_1, \phi'_2)$ satisfies

- $\phi_1 = \phi \cap \text{con}(\{\beta\})$;
- $\phi'_2 = \phi \cap \text{con}(\{\text{cl}(\neg\beta)\})$; and
- $\phi_1 \cup \phi'_2 = \phi$,

but in general ϕ_1 and ϕ'_2 may intersect.

The classical, filter-based implementation of a rational split on the domain $\mathbb{CP}_n$ is shown in Pseudocode 3, which can be seen to closely follow the informal specification given when describing Fig. 1: the filter operator is invoked twice, using the non-strict inequality β and its non-strict complement $\beta' = \text{cl}(\neg\beta)$ on two identical copies of the input polyhedron.[3] In order to fully appreciate the intrinsic overheads incurred by this simple approach, one should consider the code that would be obtained by inlining the source code for the two function calls $\text{FILTER}(\phi, \beta)$ and $\text{FILTER}(\phi', \beta')$ at lines 4–5 of Pseudocode 3. The key observations are that $\phi = \phi'$ and the filtering constraints β and β' only differ in the *sign* of their coefficients. As a consequence, in the second call to function FILTER:

1. The scalar products $\mathcal{P}$ computed in line 3 of Pseudocode 1 can be obtained by negating those computed in the first call of the function;
2. The partitioning of the generator system in line 4 of Pseudocode 1 can be easily obtained by the one computed in the first call, by simply swapping the roles of $\mathcal{G}^+$ and $\mathcal{G}^-$;
3. Since the adjacency test and the linear combination helper functions are both symmetric in their arguments,[4] the previously computed set $\mathcal{G}^\star$ can be reused as it is, in the second call.

[3] While in our pseudocode we adopt a functional programming style, the actual implementation uses destructive updates whenever possible; the use of ϕ' and the comment in line 2 are meant to highlight that, in such an implementation, a deep copy of the input polyhedron is required.

[4] Namely, both $\text{adjacent}(g^+, g^-, \mathcal{S}) = \text{adjacent}(g^-, g^+, \mathcal{S})$ and $\text{combine}(g^+, g^-, p^+, p^-) = \text{combine}(g^-, g^+, p^-, p^+)$ always hold.

Pseudocode 4 (Rational split)

```
1: function Q-SPLIT(φ, β)
2:     let φ ≡ ⟨C, G⟩ and S = sat_cons(C, G);
3:     P ← scalar_products(G, β);
4:     (G+, G0, G−) ← partition(G, P);
5:     G⋆ ← COMBINE-ADJ(G+, G−, P, S);
6:     C1 ← C ∪ {β};                            ▷ then branch
7:     G1 ← G+ ∪ G0 ∪ G⋆;
8:     let φ1 ≡ ⟨C1, G1⟩;
9:     C2 ← C ∪ {cl(¬β)};                       ▷ else branch
10:    G2 ← G− ∪ G0 ∪ G⋆;
11:    let φ2 ≡ ⟨C2, G2⟩;
12:    return φ1, φ2;
13: end function
```

Hence, it is possible to design an optimized version of the split operator where, intuitively, the result of the second call of the FILTER function is obtained by reusing the intermediate results of the first call. In particular, no additional scalar product, adjacency test and linear combination needs to be computed. This is shown in Pseudocode 4, whose only differences with respect to Pseudocode 1 are in lines 9–11. The experimental evaluation reported in [9, Table 6], which is based on version 0.4 of the PPLite library, confirms that an implementation of the rational split operator based on Pseudocode 4 is able to systematically halve the computational cost incurred by the classical implementation based on Pseudocode 3.

2.3 Integral split on a linear inequality

It is often the case that the predicate guarding a conditional branch is defined on program variables having an integral datatype: in these cases, the use of the rational split operator to model the conditional branch would produce a safe result (i.e., a correct over-approximation), but it would typically incur a precision loss. Therefore, we now turn our attention to the specification of *integral splits*. Consider first the case of a split based on a (strict or non-strict) linear inequality constraint.

Example 2
Let $P = \text{con}(\{0 \le x_0 \le 3, 0 \le x_1 \le 3\})$ be a polyhedron in $\mathbb{CP}_2$ and consider a conditional branch guarded by constraint $c \equiv (2x_0 + 2x_1 \le 5)$. When applying the rational split operator, we would compute $(P_1, P_2) = \text{Q-SPLIT}(P, c)$, where

$$P_1 = \text{con}(\{0 \le x_0, 0 \le x_1, 2x_0 + 2x_1 \le 5\}),$$
$$P_2 = \text{con}(\{0 \le x_0 \le 4, 0 \le x_1 \le 4, 2x_0 + 2x_1 \ge 5\}).$$

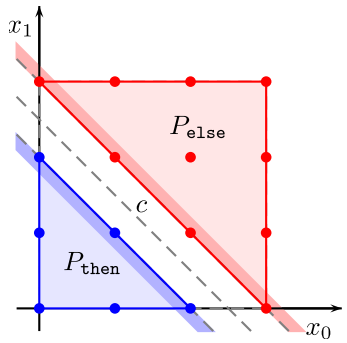

Fig. 4 Integral split on $c \equiv (2x_0 + 2x_1 \leq 5)$

If the variables are known to be integral, the guard constraint c can be refined to obtain

$$c_{\texttt{then}} = \text{Z\_refine}(c) \equiv (x_0 + x_1 \leq 2);$$

from this, we obtain the corresponding integral complement

$$c_{\texttt{else}} = \text{Z\_complement}(c_{\texttt{then}}) \equiv (-x_0 - x_1 \leq -3).$$

Hence, by implementing the split operation using two calls of the filter operator, we would obtain the more precise results

$$\begin{aligned} P_{\texttt{then}} &= \textsc{filter}(P, c_{\texttt{then}}) \\ &= \text{con}(\{0 \leq x_0, 0 \leq x_1, x_0 + x_1 \leq 2\}), \\ P_{\texttt{else}} &= \textsc{filter}(P, c_{\texttt{else}}) \\ &= \text{con}(\{x_0 \leq 3, x_1 \leq 3, x_0 + x_1 \geq 3\}), \end{aligned}$$

shown in Fig. 4.

The implementation approach informally described in the example above is the one usually adopted when modeling integral splits. The filter-based integral split function (Pseudocode 5) differs with respect to its rational variant (Pseudocode 3) in lines 3–4, which compute the integral refinement β_1 and complement β_2; in the integral case, we have $\beta_1 \neq \beta$ or $\beta_2 \neq \text{cl}(\neg\beta)$ or both (as was the case in Example 2); also, β_1 and β_2 are non-strict inequality constraints, even when the splitting constraint β is a strict inequality. Note that the described integral refinement process is generally incomplete: there are cases where one of the branches has no integral solution, but the abstract domain of convex polyhedra is unable to detect this integral inconsistency due to the intrinsic real relaxation (unless performing further checks, which are usually avoided since they are rather expensive).

As was the case for the rational split operator, an improved implementation strategy for the integral split should try to identify those computations that are uselessly replicated in the filter-based approach of Pseudocode 5. However, since in this case the two integral complementary constraints β_1 and β_2 are *not* complementary when considered in the real relaxation $\mathbb{R}^n$, such an implementation cannot be as simple and effective as that for the rational case.

Pseudocode 5 (Filter-based integral split on inequality)

```
1: function FILTER-BASED-Z-SPLIT-ON-INEQUALITY(φ, β)
2:     φ' ← φ;                                   ▷ deep copy
3:     β1 ← Z_refine(β);
4:     β2 ← Z_complement(β1);
5:     φ1 ← FILTER(φ, β1);                       ▷ then branch
6:     φ2 ← FILTER(φ', β2);                      ▷ else branch
7:     return φ1, φ2;
8: end function
```

Pseudocode 6 (Integral split on inequality)

```
1: function Z-SPLIT-ON-INEQUALITY(φ, β)
2:     let φ ≡ ⟨C, G⟩ and S = sat_cons(C, G);
3:     β1 ← Z_refine(β);
4:     β2 ← Z_complement(β1);
5:     P1 ← scalar_products(G, β1);
6:     P2 ← adjust_products(P1, G, β1, β2);
7:     return Z-SPLIT-AUX(C, G, S, β1, P1, β2, P2);
8: end function
```

Nonetheless, it can be observed that the two filtering constraints β_1 and β_2 still happen to have the same *slope*: namely, while having a different inhomogeneous term (e.g., in Example 2, we have 2 in $c_{\texttt{then}}$ and -3 in $c_{\texttt{else}}$), the homogeneous coefficients of the variables only differ in their sign, as was the case in the rational case. This property is suitably exploited when defining the Z-SPLIT-ON-INEQUALITY function in Pseudocode 6: after computing, in line 5, the scalar products $\mathcal{P}_1$ using constraint β_1, in line 6 the corresponding scalar products $\mathcal{P}_2$ for β_2 are obtained by "adjusting" those in $\mathcal{P}_1$: in practice, each scalar product in $\mathcal{P}_2$ is computed, starting from the corresponding scalar product in $\mathcal{P}_1$, by a constant number of arbitrary precision arithmetic operations. In contrast, the number of operations required for each full (i.e., non-adjusted) scalar product computation is linear in the dimension n of the vector space; this is one of the overheads incurred by calling $\textsc{filter}(\phi', \beta_2)$ in Pseudocode 5.

Having obtained $\mathcal{P}_1$ and $\mathcal{P}_2$, the procedure goes on by calling the auxiliary function Z-SPLIT-AUX, described in Pseudocode 7. Here, the partitions of the input generator system $\mathcal{G}$ corresponding to $\mathcal{P}_1$ and $\mathcal{P}_2$ are computed; note that, in general, we obtain different sets (even when taking into account the change of sign). For instance, when considering Example 2, the four points $p_0 = (0,0)$, $p_1 = (0,3)$, $p_2 = (3,3)$ and $p_3 = (3,0)$ of the input generator system $\mathcal{G}$ are partitioned as follows:

- $\mathcal{G}_1^+ = \{p_0\}$ and $\mathcal{G}_2^- = \{p_0\}$;
- $\mathcal{G}_1^0 = \emptyset$ and $\mathcal{G}_2^0 = \{p_1, p_3\}$;

Pseudocode 7 (Integral split helper)

```
1:  function Z-SPLIT-AUX(C, G, S, β1, P1, β2, P2)
2:      (G1+, G10, G1−) ← partition(G, P1);
3:      (G2+, G20, G2−) ← partition(G, P2);
4:      (G1⋆, G2⋆) ← Z-COMBINE(G1+, G1−, P1, G2+, G2−, P2, S);
5:      C1 ← C ∪ {β1};                                  ▷ then branch
6:      if is_equality(β1) then
7:          G1 ← G10 ∪ G1⋆;
8:      else
9:          G1 ← G1+ ∪ G10 ∪ G1⋆;
10:     end if
11:     let φ1 ≡ ⟨C1, G1⟩;
12:     C2 ← C ∪ {β2};                                  ▷ else branch
13:     G2 ← G2+ ∪ G20 ∪ G2⋆;
14:     let φ2 ≡ ⟨C2, G2⟩;
15:     return φ1, φ2;
16: end function
```

Pseudocode 8 (Integral combination of generators)

```
1:  function Z-COMBINE(G1+, G1−, P1, G2+, G2−, P2, S)
2:      G1* ← ∅;
3:      G2* ← ∅;
4:      for all g+ ∈ G1+ ∪ G2− do
5:          for all g− ∈ G1− ∪ G2+ do
6:              if adjacent(g+, g−, S) then
7:                  if g+ ∈ G1+ and g− ∈ G1− then
8:                      let p1+ = P1[g+] and p1− = P1[g−];
9:                      g1⋆ ← combine(g+, g−, p1+, p1−);
10:                     G1⋆ ← G1⋆ ∪ {g1⋆};
11:                 end if
12:                 if g+ ∈ G2− and g− ∈ G2+ then
13:                     let p2+ = P2[g+] and p2− = P2[g−];
14:                     g2⋆ ← combine(g+, g−, p2+, p2−);
15:                     G2⋆ ← G2⋆ ∪ {g2⋆};
16:                 end if
17:             end if
18:         end for
19:     end for
20:     return G1⋆, G2⋆;
21: end function
```

– $\mathcal{G}_1^- = \{p_1, p_2, p_3\}$ and $\mathcal{G}_2^+ = \{p_2\}$.

Namely, we obtain $\mathcal{G}_1^+ = \mathcal{G}_2^-$ (as in the rational case), but $\mathcal{G}_1^- \neq \mathcal{G}_2^+$.

In most cases, the sets $\mathcal{G}_1^+$ and $\mathcal{G}_2^-$ (resp., $\mathcal{G}_1^-$ and $\mathcal{G}_2^+$) have many generators in common, meaning that in Pseudocode 5 most pairs of generators are processed twice. The helper function Z-COMBINE called on line 4 of Pseudocode 12 implements an *ad hoc* version of the function combining adjacent generators (i.e., function COMBINE-ADJ of Pseudocode 2), whose goal is to compute both $\mathcal{G}_1^\star$ and $\mathcal{G}_2^\star$ at the same time in an attempt to avoid redundant computations as much as possible. To this end, Pseudocode 8 uses a single pair of nested loops so as to check each $g^+ \in \mathcal{G}_1^+ \cup \mathcal{G}_2^-$ with respect to each $g^- \in \mathcal{G}_1^- \cup \mathcal{G}_2^+$: in particular, the adjacency test on line 6 is computed only *once* for each pair (g^+, g^-); if the test succeeds, then any required linear combination is computed and sets $\mathcal{G}_1^*$ and $\mathcal{G}_2^*$ are updated accordingly.

When exiting from the call to Z-COMBINE, in lines 5–14 of Pseudocode 7, the auxiliary integral split function concludes its work by composing the Double Description representations of its results ϕ_1 and ϕ_2.[5]

2.4 Integral split on a linear equality

We now consider the case of an integral split based on a linear (dis-) equality constraint. This case turns out to be trickier, since the domain of convex polyhedra, like most numerical domains based on convex approximations, is generally unable to precisely filter on a disequality constraint.

[5] The reason for adding the **if-then-else** on lines 6–10 is that the same auxiliary function is used in the next section to compute integral splits under equality constraints.

Example 3
For polyhedron P of Example 2, consider a branch guarded by constraint $c \equiv (x_0 = 2)$. We can be precise on the equality branch, computing

$$\begin{aligned} P_{\texttt{then}} &= \text{FILTER}(P, x_0 = 2) \\ &= \text{con}(\{x_0 = 2, 0 \leq x_1 \leq 3\}). \end{aligned}$$

On the other branch, we may try to lower the disequality $c^{\neq} \equiv (x_0 \neq 2)$ into the pair of inequalities

$$\begin{aligned} (c^<, c^>) &= \text{Z\_complement\_eq}(c) \\ &\equiv ((x_0 \leq 1), (x_0 \geq 3)); \end{aligned}$$

hence, we would intuitively compute

$$\begin{aligned} P^<_{\texttt{else}} &= \text{FILTER}(P, c^<) \\ &= \text{FILTER}(P, x_0 \leq 1) \\ &= \text{con}(\{0 \leq x_0 \leq 1, 0 \leq x_1 \leq 3\}), \\ P^>_{\texttt{else}} &= \text{FILTER}(P, c^>) \\ &= \text{FILTER}(P, x_0 \geq 3) \\ &= \text{con}(\{x_0 = 3, 0 \leq x_1 \leq 3\}), \end{aligned}$$

as shown in Fig. 5. Unfortunately, this effort towards precision is later made useless by the join computation $P_{\texttt{else}} = P^<_{\texttt{else}} \uplus P^>_{\texttt{else}} = P$, whose result is the same as the input polyhedron (so that $P_{\texttt{then}} \subseteq P_{\texttt{else}}$).

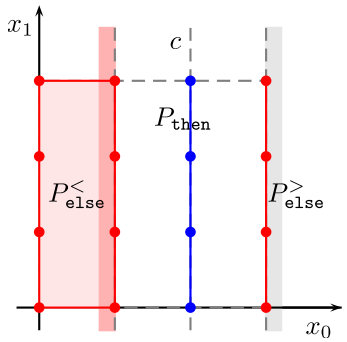

Fig. 5 Integral split on $c \equiv (x_0 = 2)$

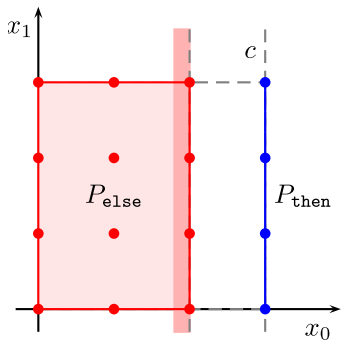

Fig. 6 Integral split on $c \equiv (x_0 = 3)$

Some attempts can be made to identify those lucky cases where a disequality can be successfully refined into an inequality constraint, as in the following example.

Example 4
When splitting polyhedron P of Example 2 on constraint $c \equiv (x_0 = 3)$, we obtain

$$\begin{aligned}(c^<, c^>) &= \text{Z\_complement\_eq}(c)\\ &\equiv ((x_0 \leq 2), (x_0 \geq 4))\end{aligned}$$

so that

$$\begin{aligned}P_{\texttt{then}} &= \textsc{filter}(P, x_0 = 3)\\ &= \text{con}(\{x_0 = 3, 0 \leq x_1 \leq 3\}),\\ P^{<}_{\texttt{else}} &= \textsc{filter}(P, c^<)\\ &= \textsc{filter}(P, x_0 \leq 2)\\ &= \text{con}(\{0 \leq x_0 \leq 2, 0 \leq x_1 \leq 3\}),\\ P^{>}_{\texttt{else}} &= \textsc{filter}(P, c^>)\\ &= \textsc{filter}(P, x_0 \geq 4) = \bot.\end{aligned}$$

Hence, as shown in Fig. 6, we obtain the precise result

$$P_{\texttt{else}} = P^{<}_{\texttt{else}} \uplus P^{>}_{\texttt{else}} = P^{<}_{\texttt{else}} \uplus \bot = P^{<}_{\texttt{else}}.$$

Pseudocode 9 (Filter-based integral split on equality)

```
1:  function FILTER-BASED-Z-SPLIT-ON-EQUALITY(φ, β)
2:      φ1 ← FILTER(φ, β);
3:      (β<, β>) ← Z_complement_eq(β);
4:      φ< ← FILTER(φ, β<);
5:      φ> ← FILTER(φ, β>);
6:      if φ< = ⊥ then
7:          φ2 ← φ>;                          ▷ lucky case
8:      else if φ> = ⊥ then
9:          φ2 ← φ<;                          ▷ lucky case
10:     else
11:         φ2 ← φ;                           ▷ unlucky case
12:     end if
13:     return φ1, φ2;
14: end function
```

In Pseudocode 9 we sketch a possible user-level implementation of the integral split for an equality constraint β that tries to identify the lucky case described in Example 4, based on a repeated use of the filter operator.

It should be noted that, in some implementations of the abstract domain of convex polyhedra, it is possible to check whether a polyhedron ϕ is disjoint from (the polyhedron implicitly defined by) a linear constraint β using a more efficient helper predicate, rather than invoking the filter operator. As an example, predicate relation_with(ϕ, β), available in both the Parma Polyhedra Library [6] and PPLite [10], can establish the relation (disjointness, inclusion, proper intersection) of a polyhedron with a linear constraint β by examining the sign of the scalar products of the generators of ϕ with β. However, such a helper predicate and the corresponding optimizations are not generally available (e.g., they are not part of the generic abstract domain interface provided by the Apron library [25]).

As a consequence, the actual user-level implementation of this variant of the split operator may vary depending on the considered static analysis tool, in an attempt to obtain a generic implementation characterized by a reasonable precision/efficiency tradeoff. For instance, in order to check if ϕ is disjoint from an inequality constraint β, the static analysis tool Crab first rewrites the constraint as $\beta \equiv (x_i \bowtie expr)$ and then uses another helper function to evaluate and compare the ranges of values that variable x_i and expression $expr$ can assume in polyhedron ϕ; this check is repeated for each variable x_i occurring in constraint β (implicitly exploiting the fact that in the Crab Intermediate Representation, which resembles 3-address code, all branching predicates are defined on two variables at most). What can be generally observed is that, in the lack of an *ad hoc* operator for the integral split, the analysis tool may be forced to perform several calls to lower-level abstract operators (entailment checks, evaluations of the value range of a linear expression, etc.), with a corresponding multiplication of the overheads that are inherently incurred when interfacing the static analysis tool with a

Pseudocode 10 (Integral split on equality)

```
1:  function Z-SPLIT-ON-EQUALITY(φ, β)
2:      if is_integral_inconsistent(β) then
3:          return ⊥, φ;                                  ▷ φ disjoint from β
4:      end if
5:      let φ ≡ ⟨C, G⟩ and S = sat_cons(C, G);
6:      P ← scalar_products(G, β);
7:      if IS-INCLUDED-IN-EQUALITY(G, P) then
8:          return φ, ⊥;                                  ▷ φ satisfies β
9:      end if
10:     if IS-DISJOINT-FROM-EQUALITY(G, P) then
11:         return ⊥, φ;                                  ▷ φ disjoint from β
12:     end if
13:     (β^<, β^>) ← Z_complement_eq(β);
14:     P^< ← adjust_products(P, G, β, β^<);
15:     P^> ← adjust_products(P, G, β^<, β^>);
16:     if IS-DISJOINT-FROM-INEQUALITY(G, P^<) then
17:                                                       ▷ φ disjoint from β^<
18:         return Z-SPLIT-AUX(C, G, S, β, P, β^>, P^>);
19:     end if
20:     if IS-DISJOINT-FROM-INEQUALITY(G, P^>) then
21:                                                       ▷ φ disjoint from β^>
22:         return Z-SPLIT-AUX(C, G, S, β, P, β^<, P^<);
23:     end if
24:     φ_2 ← φ;                                          ▷ unlucky case: deep copy
25:     φ_1 ← FILTER(φ, β);
26:     return φ_1, φ_2;
27: end function
```

Pseudocode 11 (Helper predicates)

```
1:  function IS-INCLUDED-IN-EQUALITY(G, P)
2:      (G^+, G^0, G^-) ← partition(G, P);
3:      return G = G^0;
4:  end function

5:  function IS-DISJOINT-FROM-EQUALITY(G, P)
6:      (G^+, G^0, G^-) ← partition(G, P);
7:      let ⟨L, R, P⟩ = G;
8:      return (L ⊆ G^0 and R ⊆ G^0 ∪ G^- and P ⊆ G^-)
9:          or (L ⊆ G^0 and R ⊆ G^0 ∪ G^+ and P ⊆ G^+);
10: end function

11: function IS-DISJOINT-FROM-INEQUALITY(G, P)
12:     (G^+, G^0, G^-) ← partition(G, P);
13:     let ⟨L, R, P⟩ = G;
14:     return L ⊆ G^0 and R ⊆ G^0 ∪ G^- and P ⊆ G^-;
15: end function
```

generic abstract domain component; hence, the more precise and expensive domains are likely to witness a degradation in their overall efficiency.

The integral split of a polyhedron $\phi \in \mathbb{CP}_n$ on an equality constraint β is specified in Pseudocode 10. In lines 2-4, the procedure starts by filtering out the trivial case of an equality constraint that, while being satisfiable on rationals, has no integral solution at all; as an example, consider the equality $\beta \equiv (2x_0 + 2x_1 = 5)$. After obtaining the scalar products $\mathcal{P}$ using the equality constraint β (line 6), the procedure checks for other special cases in lines 7-12: namely, when the input polyhedron ϕ is included in the hyperplane defined by β, the 'else' branch ϕ_2 is known to be empty; similarly, when the input polyhedron ϕ is disjoint from the hyperplane defined by β, the 'then' branch ϕ_1 is known to be empty; the helper predicates efficiently checking these conditions, based on the sign of the already computed scalar products, are shown in Pseudocode 11. The procedure then tries to identify lucky cases such as the one described in Example 4: to this end, in lines 13–15 it computes the constraints $\beta^<$ and $\beta^>$ and the corresponding scalar products $\mathcal{P}^<$ and $\mathcal{P}^>$ (which are efficiently obtained from $\mathcal{P}$, as discussed before). This attempt succeeds when ϕ is disjoint from $\beta^<$ (resp., $\beta^>$): if this happens, then $\phi_2^< = \bot$ and $\phi_2 = \phi_2^>$ (resp., $\phi_2^> = \bot$ and $\phi_2 = \phi_2^<$), so that the procedure can conclude its work by calling the helper function Z-SPLIT-AUX of Pseudocode 7 in line 18 (resp., line 22). Lines 24–26 are meant to handle the unlucky case when, as in Example 3, both $\phi_2^<$ and $\phi_2^>$ are not empty; hence, ϕ_2 is obtained by just copying the input polyhedron ϕ, while ϕ_1 is obtained by calling (only once) the filter operator.

Having defined both the linear inequality and equality variants of the integral split operator, we can easily combine them, as shown in Pseudocode 12, to obtain a user-friendly operator Z-SPLIT.

Pseudocode 12 (Integral split)

```
1:  function Z-SPLIT(φ, β)
2:      if is_equality(β) then
3:          return Z-SPLIT-ON-EQUALITY(φ, β);
4:      else
5:          return Z-SPLIT-ON-INEQUALITY(φ, β);
6:      end if
7:  end function
```

3 Enabling splits in a static analysis

For our experiments we have chosen Clam/Crab, which is the Abstract Interpretation engine included in the SeaHorn framework [22]. The Clam component uses Clang/LLVM to obtain the LLVM bitcode of the program under analysis and then generates the corresponding CrabIR representation [21]; this is processed by the Crab component, which computes the abstract semantics according to the chosen analysis configuration. The latter includes, among many other parameters, the choice of the abstract domain: Crab supports many (combinations of) abstract domains and includes interfaces towards

the abstract domains provided by libraries Apron [25] and ELINA [33].

Our prototype analyzer was obtained by modifying the `dev14` branch of Clam/Crab,[6] which is based on LLVM 14. The addition of a new abstract operator requires that suitable changes are applied to both the abstract domain and the fixpoint engine components of the considered static analysis tool.

3.1 Changes in the abstract domain component

The new split operators for the domain of convex polyhedra have been implemented in the PPLite library [7, 8, 10]. As previously said, the rational split operator, implementing the algorithm Q-SPLIT of Pseudocode 4, was proposed and experimentally evaluated in [9]; it is available since version 0.4 of the library, for both abstract domains $\mathbb{P}_n$ and $\mathbb{CP}_n$. A variant of the integral split operator for the domain $\mathbb{CP}_n$, implementing the algorithm Z-SPLIT-ON-INEQUALITY of Pseudocode 6 but *without* the optimizations provided by function Z-COMBINE of Pseudocode 8, has been available since version 0.10.1 of the library;[7] the fully optimized version of the integral split operator will be available starting from release 0.12 of the PPLite library.

The new split operators have been integrated in the Apron interface wrapper for PPLite; clearly, since the generic Apron interface is missing suitable entry points for these new abstract domain operators, they have been added as non-generic, *ad hoc* functions. The use of the Apron wrapper was essential to simplify the integration of the PPLite domains in Clam/Crab; we also extended the generic abstract domain interface in Crab by adding a new method for the integral split operator: this invokes the new abstract operator when the interface is instantiated with a PPLite domain, while resorting to the unoptimized implementation (based on function FILTER-BASED-Z-SPLIT of Pseudocode 5) when it is instantiated with alternative abstract domains.

3.2 Changes in the fixpoint approximation engine

The adaptation of the fixpoint engine required more work than expected. This is mainly due to a CrabIR language design choice (inherited from IKOS AR form [12]) whereby all conditional branches are expressed in a declarative way, combining a non-deterministic branch with the addition, in each target of the branch, of `assume` abstract statements encoding the branch condition. As an example, the CrabIR CFG representation for the following simple function

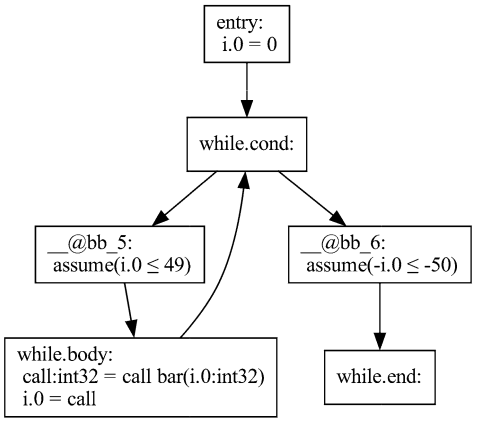

Fig. 7 Original CrabIR CFG, using `assume` statements

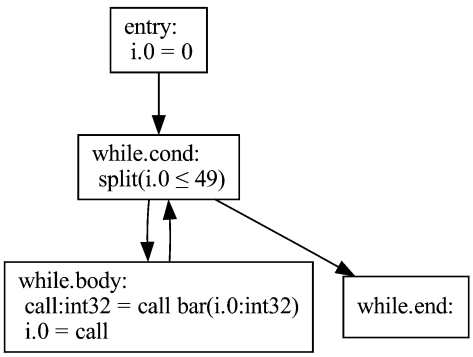

Fig. 8 Modified CrabIR CFG, using `split` statement

```
void foo() {
  int i = 0;
  while(i < 50)
    i = bar(i);
}
```

is shown in Fig. 7; roughly speaking, the non-optimized implementation of the conditional branch is *hard coded* in the CrabIR representation, preventing the application of the split operator.

As a workaround, we enriched CrabIR by adding a new abstract statement, called `split`, and then modified Clam to generate this new statement whenever translating a numerical conditional branch (see the CFG in Fig. 8). Branches based on Boolean and pointer tests are not affected and hence maintain the declarative encoding.

The other main change to the fixpoint computation engine regards the choice of where to store the invariants computed during the analysis. Exploiting the declarative encoding of conditional branches, by default Crab annotates each node in the CFG with the pairs $\langle pre, post \rangle$ of invariants that are valid at the start (pre) and at the end ($post$) of the node. However, when the CFG is modified by introducing `split` statements, this approach is no longer adequate, because we would need to store two different invariants ($post_{\texttt{then}}$ and $post_{\texttt{else}}$) at the

[6] https://github.com/seahorn/clam/tree/dev14.

[7] The experimental evaluation in [3] was based on this variant of the algorithm.

exit of those nodes ending with a `split` statement. Therefore, we modified the engine so as to store an invariant along each *edge* of the CFG, as well as storing an invariant on those nodes where a widening/narrowing is computed. At the end of the analysis, in a *finalization phase*, the edge invariants are used to compute the $\langle pre, post \rangle$ pairs of each CFG node, which requires computing many joins; while this simplistic approach incurs avoidable inefficiencies, it preserves the expectations of the other Clam/Crab components that have not been modified.

It is worth noting that our prototype is considering a *subset* of all the numerical branches that, in principle, could benefit from the split operator. For instance, Clam/Crab provides limited support for floating-point datatypes and safely ignores conditional branches whose predicates are defined on floating-point variables, yielding a pure non-deterministic branch; as a consequence, our prototype cannot trigger the invocation of the rational variant of the split operator.

Similarly, Clam/Crab safely ignores integral splits when the corresponding predicates are linear inequalities defined on *unsigned* variables. The reason is that, even though the LLVM bitcode representation of integer values does not encode their signedness, in the Crab IR representation these are always interpreted as signed integer values (so that the biggest unsigned values are mapped to negative signed values). Hence, when faced with an unsigned comparison, the Crab IR representation would need to reinterpret the (signed) arguments: the default approach avoids the corresponding cost by safely ignoring the branch comparison. Note that it is possible to instruct Clam/Crab to modify its IR representation so as to correctly encode an unsigned integer linear predicate into a Boolean combination of signed integer linear predicates; however, this program transformation step is implemented by first introducing new Boolean variables, which store the result of the signed comparisons, and then branching (via the declarative filter approach) on those Boolean variables, thereby escaping from our optimizations that only target numerical branches. As a consequence, we conjecture that, in our experimental evaluation, the efficiency improvements obtainable thanks to the split operator are underestimated.

Note that the prototype static analysis tool described in [3] was also disregarding those *implicit* numerical branches that in CrabIR are encoded using the `select` statement, which implements a conditional assignment. As an example, the conditional assignment[8]

```
x = ite(y <= 0, 1, 2);
```

implicitly encodes the control flow

```
if (y <= 0)
  x = 1;
else
  x = 2;
```

Our new implementation applies the split operator also when abstractly evaluating these `select` statements, provided the corresponding linear predicate is defined on signed integral variables.

4 Experimental evaluation

In our experimental evaluation, we analyzed programs coming from two different sources. First, we considered 39 C source files, distributed with PAGAI [24], which are variants of benchmarks taken from the SNU real-time benchmark suite for WCET (worst-case execution time) analysis; note that, while considering all of them, in the following we only show the results for the five tests whose analysis time is greater than a second. Then, we enriched our benchmark suite by also considering 25 Linux drivers from the SVCOMP repository;[9] we applied no specific selection criterion and randomly picked drivers having an analysis time greater than a second and less than five minutes.

In our experiments, we tried to apply minimal changes to the default configuration of the analyzer: in particular, we instructed LLVM to systematically inline function calls, so as to improve the call context sensitivity of the analysis. Note that, by default, Clam/Crab instructs LLVM to lower all switch statements, which are thus translated to chains of conditional branches and hence can benefit of the split optimization.

Time efficiency comparison Table 1 reports the timing results obtained in our experimental evaluation: the 5 PAGAI tests are shown in the top half of the table, the Linux drivers in the lower half. The 2nd column in Table 1 shows the baseline for the efficiency evaluation, i.e., the overall analysis time in seconds when using `F_Poly`, the Cartesian factored [23] convex polyhedra domain of PPLite; note that we include time spent in pre-analysis phases (e.g., parsing, LLVM bitcode generation and Clam preprocessing steps), while excluding post-analysis phases (e.g., assertion checks based on the results of the analysis).[10] In the 3rd column, we show the time obtained when using the Cartesian factored convex polyhedra domain of ELINA [33]: this is done to highlight that our starting point for the efficiency comparison is in line with what is considered the most efficient

[8] In Crab, the `select` statement is pretty printed using the ternary `ite` operator.

[9] https://github.com/sosy-lab/sv-benchmarks/tree/master/c/ldv-linux-4.2-rc1.

[10] Experiments have been performed on a MacBook Pro with Apple M1 Pro CPU, 16 GB of RAM and running macOS Ventura (13.5.2).

Table 1 Overall static analysis time (in seconds) without/with splits

(Abridged) test name	Without splits		PPLite with split		Time ratio
	PPLite	ELINA	No opt	Opt	
adpcm	42.8	(⋆) 0.7	41.2	42.0	1.02
prog9000	15.9	118.0	19.5	22.1	0.72
nsichneu	13.2	17.7	12.4	8.5	1.56
decompress	2.2	2.1	2.2	0.8	2.69
filter	1.1	0.9	1.2	1.1	1.00
mmc-host	219.4	167.9	179.7	67.5	3.25
rbd	175.9	71.0	123.2	78.5	2.24
9xxx	170.0	149.3	152.6	22.0	7.72
wl12xx	162.3	211.0	188.0	41.0	3.96
mdc	78.2	(⋆) 64.5	96.5	7.6	10.30
firewire	48.0	(⋆) 44.5	52.0	4.8	10.00
w83781d	47.5	37.1	64.2	61.2	0.78
snic	30.3	29.1	29.1	10.0	3.01
hwmon-abituguru3	29.8	37.3	29.8	17.5	1.70
btcoexist	26.5	21.1	23.6	11.5	2.30
hdpvr	24.6	21.7	25.9	4.2	5.79
udlfb	23.2	21.4	21.0	15.3	1.52
lnet-selftest	11.8	(⋆) 12.3	11.1	7.0	1.69
media-usb-tm6000	10.1	9.1	13.2	3.6	2.80
libcomposite	8.6	8.0	8.7	3.6	2.39
pcmcia_rsrc	5.7	7.6	5.0	8.3	0.68
r8152	5.7	5.6	6.1	3.6	1.55
power-bq2415x	5.1	4.8	6.8	5.4	0.94
media-pci-ttpci	5.0	5.1	5.0	2.7	1.84
uas	4.9	5.0	4.8	1.0	4.82
prism54	3.7	4.8	3.5	2.7	1.39
hid-usb	3.2	4.2	3.3	2.1	1.53
cx25840	2.7	2.9	2.6	2.2	1.22
vfio-pci	2.0	(⋆) 2.0	2.2	1.5	1.34
cpia2	1.5	1.7	1.5	1.1	1.28

library for convex polyhedra. Note, however, that ELINA cannot be used as a proper baseline, as it is known that by using machine integers (rather than the arbitrary precision integers adopted by PPLite), it sometimes raises overflow exceptions, after which it returns an over-approximation of the actual result. When this happens, both the efficiency and the precision of the analysis are deeply affected; these cases are highlighted in the table using the symbol (⋆).

The 4th and 5th columns of the table report the overall analysis time obtained when applying the CFG transformation that inserts the new `split` statements in the CrabIR representation; as discussed before, this implies that the program invariants are stored on the CFG edges and the analyzer has to invoke the invariant finalization phase. The 4th column, labeled 'no opt', shows the time obtained when still using the classical, filter-based implementation of the `split` statement; the 5th column, labeled 'opt', shows the time obtained when actually enabling the newly implemented split operator. Finally, the last column shows the speedup obtained by the optimized split implementation (5th column) with respect to the baseline (2nd column).

On the considered tests, we are able to obtain significant speedups, sometimes beyond our own expectations; in a few cases, we also obtain minor slowdowns. Overall, the geometric mean of the speedups is 1.25 for the PAGAI tests and 2.27 for the Linux drivers.

By comparing the values of the 4th and 5th columns of Table 1, we can confirm that the speedups obtained are mainly caused by the use of the optimized split operator; namely, when only modifying the CFG to introduce the `split` statement and storing program invariants at the CFG edges, we do somehow affect the efficiency of the analysis, but we

Table 2 Number of nodes, splits, selects and maximum RSS (in KB)

(Abridged) test name	Without splits		Split	Select	With splits		Mem ratio
	Nodes	Mem			Nodes	Mem	
adpcm	146	141,120	34	32	93	125,712	1.12
prog9000	1491	1,721,472	275	12	947	1,410,304	1.22
nsichneu	2004	1,657,584	625	0	754	621,952	2.67
decompress	1032	175,792	266	4	638	43,232	4.07
filter	1121	244,368	187	372	809	230,480	1.06
mmc-host	26,254	8,835,840	3772	119	19,701	1,158,688	7.63
rbd	62,327	7,004,544	6437	231	50,735	2,476,112	2.83
9xxx	13,127	70,552.16	2301	5	8900	397,072	0.18
wl12xx	18,267	8,076,960	3781	20	13,307	1,096,768	7.36
mdc	16,089	7,084,048	2402	30	11,837	320,512	22.10
firewire	9925	243,680	2038	77	6842	153,712	1.59
w83781d	41,614	1,554,992	5367	1328	31,092	1,150,192	1.35
snic	20,055	3,087,776	2258	132	15,710	531,168	5.81
hwmon-abituguru3	2653	297,312	521	10	2088	119,072	2.50
btcoexist	41,104	4,456,576	8959	0	23,544	924,656	4.82
hdpvr	10,551	3,624,400	1714	20	8022	159,616	22.71
udlfb	9129	603,312	1806	43	5743	234,656	2.57
lnet-selftest	11,391	6,638,096	1588	70	8442	277,184	23.95
media-usb-tm6000	15,447	3,751,312	1453	21	12,886	203,168	18.46
libcomposite	6798	546,288	876	48	5360	225,344	2.42
pcmcia_rsrc	3055	249,104	381	60	2375	138,528	1.80
r8152	24,969	592,400	3907	6	17,789	247,088	2.40
power-bq2415x	23,763	384,864	1985	1746	20,518	254,832	1.51
media-pci-ttpci	9057	125,360	2003	16	5610	97,440	1.29
uas	2792	572,384	514	7	1919	48,800	11.73
prism54	6430	38,080	1065	29	4541	37,984	1.00
hid-usb	7303	177,856	1478	45	4704	99,120	1.79
cx25840	11,977	287,712	2736	21	6556	234,560	1.23
vfio-pci	3024	125,360	564	16	2224	97,440	1.29
cpia2	7178	111,424	1335	23	4803	73,840	1.51

typically obtain minor speedups or even slowdowns. As an example, for the test `nsichneu`, by just storing the program invariants at the CFG edges, the baseline analysis time (13.2 seconds) is reduced by 0.8 seconds; when enabling the optimized split operator, we obtain a more significant decrease of 4.7 seconds.

A deeper investigation has shown that the few slowdowns are mainly caused by the unoptimized program invariant finalization phase described in the previous section. For instance, this finalization phase is responsible of 25%–30% of the overall analysis time for the tests `prog9000`, `w83781d` and `power-bq2415x`, thereby hiding any efficiency improvement coming from the split operator. As explained in the previous section, we believe that this finalization phase could be improved, but this would require some changes in the post-analysis phases of the Clam/Crab tool.

Memory efficiency comparison A remarkable side effect of the CFG transformation that inserts the numerical `split` statements is a significant reduction in peak memory usage. As intuitively suggested by the CFGs in Figs. 7 and 8, the introduction of the `split` statements causes a decrease in the number of nodes of the CFG, with a consequential reduction of the number of program invariants that need to be stored during the analysis.

This is highlighted by the data shown in Table 2; the 2nd and 6th columns of the table show the number of nodes in the CrabIR CFGs generated without and with the CFG transformation (note that, in both cases, we refer to the CFGs *after* the LLVM inlining phase); the 4th and 5th columns report the number of `split` and `select` statements occurring in

each test, respectively;[11] in the 3rd and 7th columns, we provide an estimation of the peak memory usage by reporting the maximum RSS (Resident Set Size) for the two aforementioned configurations (in KB); the last column of the table shows the ratio of the two memory measurements. Overall, the geometric mean of the memory reduction ratio is 1.73 for the PAGAI tests and 3.13 for the Linux drivers.

A recent paper [26] proposes a technique to reduce the memory footprint of the static analysis tool IKOS, which shares with Crab the main design of the fixpoint approximation engine. Since the technique adopted in [26] is completely independent from the split optimization, we conjecture that the two optimizations can be applied together, combining their improvements.

Precision comparison When defining the split operator, our main goal was to obtain an efficiency improvement: in principle, no effect should be observed on the precision of the analysis. As a matter of fact, the split operator as implemented in [3] was obtaining the same precision of the filter-based approach on all considered tests, as confirmed by systematically comparing all the computed program invariants using the `clam-diff` helper tool.

However, the improved integral split operator for equality constraints, as described in Sect. 2.4, can sometimes obtain a minor precision improvement with respect to the filter-based approach implemented in Clam/Crab. As discussed in Sect. 2, these changes in precision are due to small differences in the handling of variable integrality: in principle, the accuracy of the integral split operator could range from the less precise rational split, yielding no integral refinement at all, to the very precise and very expensive case of an implementation computing the best possible integral refinement. As a consequence, in the current experimental evaluation, we observed several cases (15 out of 25 Linux driver tests) where the program invariants computed when using the split operator were slightly more precise than those of the baseline analysis.

Note that occasionally a minor perturbation in the precision of the analysis can significantly affect its efficiency: this explains the biggest slowdown (0.68) reported in Table 1, for the Linux driver `pcmcia_rsrc`. When analyzing this test, the precision improvements obtained by the split operator trigger the computation of a longer increasing chain in the fixpoint approximation engine: the widening operator is called 796 times (it is called only 337 times in the unoptimized case), leading to the efficiency loss.

[11] The number of `select` statements is the same with or without the CFG transformation introducing splits; we report their number to highlight that the explicit control flow splits are usually much more frequent than the implicit ones encoded by `select` statements.

5 Conclusions

This paper proposes a new abstract domain operator that is able to speed up the static analysis when splitting the control flow path on a predicate and its complement. Our prototype, built modifying Clam/Crab, is able to obtain important memory and time improvements on several tests, including both synthetic benchmarks and real-world programs.

Future work can investigate several directions. First of all, the current prototype can be extended to enable the optimization on more kinds of numerical `split` statements. As an example, one could modify the rewriting approach currently adopted by Clam/Crab for the unsigned integral comparisons, so as to avoid introducing Boolean variables and directly branch on the signed integral comparisons.

Second, we plan to evaluate the applicability of the approach to other abstract domains, taking into account that its effectiveness strongly depends on profitability considerations.

In [3], we suggested the addition of a split operator for numerical abstract domains based on the LDDs [20] and RDDs [18] (Linear and Range Decision Diagrams) data structures, such as the `Boxes` domain. A Linear Decision Diagram is a BDD-like data structure where non-terminal nodes are labeled by constraints on numerical variables; suitable semantic functions allow for the implementation of all the abstract domain operators, including the filter operator. Hence, we made an attempt to extend the `Boxes` domain by defining a corresponding split operator;[12] unfortunately, in our tests we were unable to obtain significant efficiency improvements with respect to the baseline implementation.

We still believe that some of the abstract domains directly supporting the representation of disjunctive information could benefit from an optimized split operator. Things seem promising when considering the finite powerset of convex polyhedra [5]. Note that when using a disjunctive domain, it is clearly possible to workaround the limitations of convex over-approximations and thus improve the precision of splits. For instance, in Fig. 9 we consider the split of the powerset $S = \{P, P'\}$ on the equality constraint c, where P and c are those described in Example 3. By avoiding the convex polyhedral hull approximation, we can obtain $S_{\texttt{then}} = \{P_{\texttt{then}}\}$ and $S_{\texttt{else}} = \{P^{<}_{\texttt{else}}, P^{>}_{\texttt{else}}, P'\}$; note that P', which is not directly affected by the split operator, can be simply "moved" into $S_{\texttt{else}}$, avoiding useless and costly copy operations. The PPLite library already includes an implementation of the split operator for the finite powerset domain of convex polyhedra; since this domain is very precise but also rather expensive, it will probably benefit from the corresponding efficiency improvements. Note that, if necessary,

[12] The authors would like to thank Matteo Boroni Grazioli for his help in the implementation and evaluation of this operator.

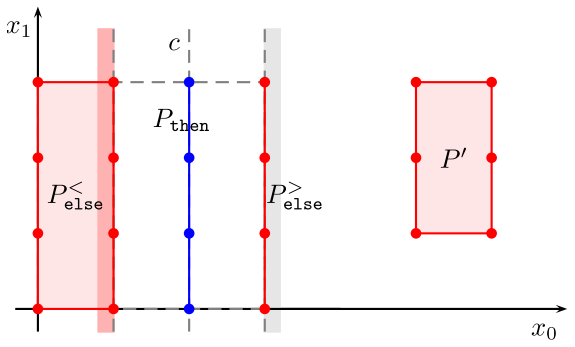

Fig. 9 Example of split on a powerset domain

one could further limit the computational cost of the powerset domain by adopting the decoupled approach of [2].

Another possibility is to investigate the profitability of implementing split operators for DFA-based abstract domains for string analysis [1, 30]; for instance, we could optimize the abstract evaluation of branches based on predicates such as `str.startsWith("prefix")`, whose default implementations on the domain of DFAs are often expensive.

Finally, it would be interesting to extend our experimental evaluation to different static analysis tools; while we conjecture that similar results can be obtained for other tools analyzing the low-level program representation (e.g., IKOS [12] and PAGAI [24]), it is more difficult to predict the effectiveness of the optimization for those tools targeting the AST-based high-level representations (e.g., MOPSA [28]).

Acknowledgements The authors would like to express their gratitude to Jorge A. Navas for his help in getting them acquainted with the inner workings of Clam/Crab and for developing helper tool `clam-diff` to compare the precision of different analyses.

Funding Open access funding provided by Università degli Studi di Parma within the CRUI-CARE Agreement. This work was partially supported by Bando di Ateneo per la ricerca 2022, founded by University of Parma (MUR_DM737_2022_FIL_PROGETTI_B_ARCERI_COFIN).

References

1. Arceri, V., Mastroeni, I., Xu, S.: Static analysis for ecmascript string manipulation programs. Appl. Sci. **10**, 3525 (2020). https://doi.org/10.3390/app10103525
2. Arceri, V., Mastroeni, I., Zaffanella, E.: Decoupling the ascending and descending phases in abstract interpretation. In: Sergey, I. (ed.) Proceedings, Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022. Lecture Notes in Computer Science, vol. 13658, pp. 25–44. Springer (2022). https://doi.org/10.1007/978-3-031-21037-2_2
3. Arceri, V., Dolcetti, G., Zaffanella, E.: Speeding up static analysis with the split operator. In: Ferrara, P., Hadarean, L. (eds.) Proceedings of the 12th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023, pp. 14–19. ACM (2023). https://doi.org/10.1145/3589250.3596141
4. Bagnara, R., Dobson, K.L., Hill, P.M., Mundell, M., Zaffanella, E.: Grids: a domain for analyzing the distribution of numerical values. In: Puebla, G. (ed.) Revised Selected Papers, Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12–14, 2006. Lecture Notes in Computer Science, vol. 4407, pp. 219–235. Springer (2006). https://doi.org/10.1007/978-3-540-71410-1_16
5. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. Int. J. Softw. Tools Technol. Transf. **9**(3–4), 413–414 (2007). https://doi.org/10.1007/s10009-007-0029-y
6. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008). https://doi.org/10.1016/j.scico.2007.08.001
7. Becchi, A., Zaffanella, E.: A direct encoding for NNC polyhedra. In: Chockler, H., Weissenbacher, G. (eds.) Proceedings, Part I, Computer Aided Verification - 30th International Conference, CAV 2018, Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018. Lecture Notes in Computer Science, vol. 10981, pp. 230–248. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_13
8. Becchi, A., Zaffanella, E.: An efficient abstract domain for not necessarily closed polyhedra. In: Podelski, A. (ed.) Proceedings, Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018. Lecture Notes in Computer Science, vol. 11002, pp. 146–165. Springer (2018). https://doi.org/10.1007/978-3-319-99725-4_11
9. Becchi, A., Zaffanella, E.: Revisiting polyhedral analysis for hybrid systems. In: Chang, B.E. (ed.) Proceedings, Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019. Lecture Notes in Computer Science, vol. 11822, pp. 183–202. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_10
10. Becchi, A., Zaffanella, E.: PPLite: zero-overhead encoding of NNC polyhedra. Inf. Comput. **275**, 104620 (2020) https://doi.org/10.1016/j.ic.2020.104620
11. Birkhoff, G.: Lattice Theory, Colloquium Publications, vol. XXV, 3rd edn. Am. Math. Soc., Providence (1967)
12. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: a framework for static analysis based on abstract interpretation. In: Proceedings, Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Lecture Notes in Computer Science, vol. 8702, pp. 271–277. Springer (2014). https://doi.org/10.1007/978-3-319-10431-7_20
13. Chernikova, N.V.: Algorithm for discovering the set of all solutions of a linear programming problem. USSR Comput. Math. Math. Phys. **8**(6), 282–293 (1968)
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973.

15. 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). https://doi.org/10.1145/512760.512770
16. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: Do, L.N.Q., Urban, C. (eds.) SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021, pp. 1–6. ACM (2021). https://doi.org/10.1145/3460946.3464316
17. Frehse, G.: Phaver: algorithmic verification of hybrid systems past hytech. Int. J. Softw. Tools Technol. Transf. **10**(3), 263–279 (2008). https://doi.org/10.1007/s10009-007-0062-x
18. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Disjunctive interval analysis. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Proceedings, Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021. Lecture Notes in Computer Science, vol. 12913, pp. 144–165. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_7
19. Genov, B.: The convex hull problem in practice: Improving the running time of the double description method. PhD thesis, University of Bremen, Germany (2014)
20. Gurfinkel, A., Chaki, S.: Boxes: a symbolic abstract domain of boxes. In: Cousot, R., Martel, M. (eds.) Proceedings, Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Lecture Notes in Computer Science, vol. 6337, pp. 287–303. Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_18
21. Gurfinkel, A., Navas, J.A.: Abstract interpretation of LLVM with a region-based memory model. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13124, pp. 122–144. Springer (2021). https://doi.org/10.1007/978-3-030-95561-8_8.
22. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Kroening, D., Pasareanu, C.S. (eds.) Proceedings, Part I, Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015. Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
23. Halbwachs, N., Merchat, D., Gonnord, L.: Some ways to reduce the space dimension in polyhedra computations. Form. Methods Syst. Des. **29**(1), 79–95 (2006). https://doi.org/10.1007/s10703-006-0013-2
24. Henry, J., Monniaux, D., Moy, M.: PAGAI: a path sensitive static analyser. In: Third Workshop on Tools for Automatic Program Analysis, TAPAS 2012, Deauville, France, September 14, 2012. Electronic Notes in Theoretical Computer Science, vol. 289, pp. 15–25. Elsevier (2012). https://doi.org/10.1016/j.entcs.2012.11.003
25. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) Proceedings, Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Lecture Notes in Computer Science, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52
26. Kim, S.K., Venet, A.J., Thakur, A.V.: Memory-efficient fixpoint computation. In: Pichardie, D., Sighireanu, M. (eds.) Proceedings, Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020. Lecture Notes in Computer Science, vol. 12389, pp. 35–64. Springer (2020). https://doi.org/10.1007/978-3-030-65474-0_3
27. Le Verge, H.: A note on Chernikova's algorithm. Publication interne 635, IRISA, Campus de Beaulieu, Rennes, France (1992)
28. Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native C extensions. In: Dragoi, C., Mukherjee, S., Namjoshi, K.S. (eds.) Proceedings, Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021. Lecture Notes in Computer Science, vol. 12913, pp. 323–345. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_16
29. Motzkin, T.S., Raiffa, H., Thompson, G.L., Thrall, R.M.: The double description method. In: Kuhn, H.W., Tucker, A.W. (eds.) Contributions to the Theory of Games – Volume II, Annals of Mathematics Studies, vol. 28, pp. 51–73. Princeton University Press, Princeton (1953)
30. Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Twinning automata and regular expressions for string static analysis. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Proceedings, Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021. Lecture Notes in Computer Science, vol. 12597, pp. 267–290. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_13
31. Negrini, L., Ferrara, P., Arceri, V., Cortesi, A.: Lisa: a generic framework for multilanguage static analysis. In: Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M. (eds.) Challenges of Software Verification, pp. 19–42. Springer Nature Singapore, Singapore (2023). https://doi.org/10.1007/978-981-19-9601-6_2
32. Pelleau, M., Miné, A., Truchet, C., Benhamou, F.: A constraint solver based on abstract domains. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Proceedings, Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22. Lecture Notes in Computer Science, vol. 7737, pp. 434–454. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_26
33. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, pp. 46–59. ACM (2017). https://doi.org/10.1145/3009837.3009885
34. Ziat, G., Maréchal, A., Pelleau, M., Miné, A., Truchet, C.: Combination of boxes and polyhedra abstractions for constraint solving. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J.C., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) Revised Selected Papers, Part II, Formal Methods. FM 2019 International Workshops, Porto, Portugal, October 7-11, 2019. Lecture Notes in Computer Science, vol. 12233, pp. 119–135. Springer (2019). https://doi.org/10.1007/978-3-030-54997-8_8