Generation of Violation Witnesses by Under-Approximating Abstract Interpretation

Marco Milanese[1][0000−0002−6215−7359] and Antoine Miné[1][0000−0002−6375−3179]

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
firstname.lastname@lip6.fr

Abstract. This works studies abstract backward semantics to infer sufficient program preconditions, based on an idea first proposed in previous work [38]. This analysis exploits under-approximated domain operators, demonstrated in [38] for the polyhedra domain, to under-approximate Dijkstras liberal precondition. The results of the analysis were implemented into a static analysis tool for a toy language. In this paper we address some limitations that hinder its applicability to C-like programs. In particular, we focus on two improvements: handling of user input and integer wrapping. For this, we extend the semantic and design sound and effective abstractions. Furthermore, to improve the precision, we explore an under-approximated version of the power-set construction. This in particular helps handling arbitrary union that is difficult to implement with under-approximated domains. The improved analysis is implemented and its performance is compared with other static analysis tools in SV-COMP23 using a selected subset of benchmarks.

Keywords: Abstract interpretation · Software verification · Program analysis · Bug catching · Under-approximation.

1 Introduction

The focus of static analysis by abstract interpretation [16,17] has traditionally been on the assurance of program *correctness*. However, the dual problem of verifying the *presence* of bugs is equally intriguing, since in practice sound static analysis tools generate many false positives and checking them manually can be time-consuming. This calls for a novel kind of abstract semantics where domains and operators are under-approximated rather than over-approximated. A preliminary study in this area was done in our previous work [38], where we investigated an abstract backward semantic to infer *sufficient* program preconditions. This analysis builds on conventional abstract domains, but in this analysis they represent an *under*-approximation of the concrete invariant. To achieve this analysis, novel under-approximation domain operators are required, and in [38] we have shown how they can be designed for the polyhedra domain [21]. The results were implemented in a static analysis tool targeting a toy language.

```
unsigned int i = 10;

while (i >= 10) {
        i++;
}

i += input();
assert(i != 5);
```

(a) Program with integer overflow and input.

```
int i = input();
int j;
assume(i >= 0 && i <= 2);

if (i == 0) j = 0;
else if (i == 1) j = 1;
else if (i == 2) j = 0;

assert(j == 1);
```

(b) Simple disjunctive program.

Fig. 1: Programs that show the limits of the semantics of [38].

Motivation. Consider the program of Fig. 1a. The while loop starts with $\texttt{i} = \texttt{10}$ and terminates when `i` overflows to `0`. Therefore, if `input`() returns 5 the assertion will fail. Unfortunately the semantic proposed in [38] can not detect the overflow as it only supports mathematical numbers (i.e., numbers with infinite precision). Moreover in this semantic there is no built-in encoding for user input and thus the preconditions that it can find concern only program's arguments. Consequently, it can not find the sufficient precondition, $\texttt{input}() = \texttt{5}$, for the assertion to fail.

Additionally, under-approximated operators were studied only for the polyhedra domain, but as for conventional abstract interpretation, other domains can be considered and the choice of the domain boils down to a precision-efficiency trade-off. As an example, the polyhedra domain fails to find the sufficient precondition, $\texttt{input}() = \texttt{1}$, for the correctness of the program of Fig 1b as the invariant before the assertion, $(i \in \{0, 2\} \wedge j = 0) \vee (i = 1 \wedge j = 1)$, is not polyhedral.

Termination. The preconditions found by this analysis under-approximate Dijkstra's weakest liberal precondition, ensuring either divergence or termination within the post-condition. The former case can be problematic, such as when the analysis is used to find preconditions for bugs, as a non-empty precondition may be a symptom of an infinite loop rather than an actual bug. However, non-termination can be ruled out with various techniques, including termination checking with a ranking function and modern works on its synthesis have been quite successful [15,14]. For the sake of simplicity we do not implement those techniques and instead opt for a simple approach of checking termination by experimentally executing the program (with a time limit).

Related Works. Lately there has been an increase in interest on under-approximations following the seminal work of Peter O'Hearn on Reverse Hoare Logic/Incorrectness Logic [24,41]. Compared to our approach, this stream of works focuses on *forward*, not backward, analyses, thus they do not study preconditions for bugs but instead they find *post-conditions* for them. Moreover, as logic

methods, they can *prove* that a post-condition is a valid under-approximation of the reachable states. Some hints on how post-conditions can be inferred were discussed in [41, Sect. 6], but they handle loops by unrolling, thus limiting the analysis to some loop bound. On the contrary, our approach can *infer* preconditions and loops are handled with widening operators, so that unrolling is not needed and unbounded loops can be handled. Incorrectness logic was made memory aware by Raad et al. [42] using ideas from separation logic [43]. In comparison, our work focuses exclusively on numeric programs and abstract domains for handling memory properties are left as a future work. Finally, reasoning with incorrectness logic can be made automatic with theorem provers as in [33,42], whereas in our work, reasoning occurs with abstract domains. This makes the analysis more scalable.

Counter-examples generation is also possible with several instances of model checking, e.g., symbolic execution [32,5] and CEGAR [13], where the state exploration is handled using SMT solvers (CEGAR is guided by counter-examples and utilizes other techniques besides SMT for the refinement phase, e.g., interpolants).

Traditional backward analyses based on abstract interpretation [18,19,12] focus on inferring *necessary* preconditions P, that is conditions such that no execution starting from $\neg P$ can succeed. For example Cousot et al. [20] propose a backward precondition analysis for code contracts. They differ from us in the handling of non-determinism as they keep states that succeed at least for one non-deterministic program path, whereas we keep states that succeed for all non-deterministic paths. Moreover, unlike us, they use symbolic reasoning, not numeric domains.

In this work we focus on sufficient preconditions P, that is conditions such that all executions from P must succeed: these require *under-approximated* operators. Designing under-approximation domains featuring optimal operators can be challenging [3] at least partially explaining why they are rarer than over-approximation ones. Several high-order constructions have been proposed in which conventional domains are used to construct under-approximation ones. Lev-Ami et al. [34] propose to use set-complements of abstract domains, but this yields shapes that are rarely interesting. Other methods based on existential quantification [44] and disjunctive completions [40] were proposed, but they incur in a too high complexity and are difficult to abstract away.

Under-approximations were used also in the work of Urban et al. [47], namely for the co-domain. However, the results are difficult to compare in theory due to different abstractions and different concrete semantics.

Contribution. In this paper we extend upon the backward sufficient preconditions analysis, addressing some of the limitations that hinder its applicability to C-like programs, namely: handling of user input and integer wrapping. To improve the precision of the polyhedra domain, we consider the well-known power-set construction and derive sound under-approximation operators for it.

We then proceed to implement the improved analysis in a static analysis tool and add support for extracting a violation witness in SV-COMP's format [8].

$$
\begin{aligned}
a &::= [x, y] \mid v \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 / a_2 \\
b &::= a_1 = a_2 \mid a_1 \neq a_2 \mid a_1 \leq a_2 \mid a_1 < a_2 \mid a_1 > a_2 \mid a_1 \geq a_2 \\
&\quad \mid \mathtt{t} \mid \mathtt{f} \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \\
s &::= \textbf{skip}() \mid v := a \mid \textbf{assume}(b) \\
&\quad \mid s_1; s_2 \mid \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \mid \textbf{while } b \textbf{ do } s \textbf{ done}
\end{aligned}
$$

Fig. 2: While language.

To the best of our knowledge, this is the first abstract interpretation based tool that can certify program incorrectness (at least among the ones participating in SV-COMP). We compare its performance with that of other static analysis tools participating in SV-COMP23 [7] on a selected subset of the competition's benchmarks.

2 Semantics

The semantic studied in [38] is limited to numeric variables and properties and is given for a toy language with mathematical numbers. In this paper, we address how it can be adapted to support fixed-precision integers and user input. Handling more advanced features of C-like languages, such as pointers, arrays, structures, dynamic memory allocation, remains a future work. Floating-point arithmetic is out of the scope of this work but it should not be a large problem as [38] shows that expression evaluation can be over-approximated and still leads to sound under-approximated statements, thus rounding errors can be abstracted as small non-deterministic error intervals, as in forward analysis, and easily supported in a polyhedral analysis.

This section is organized as follows: in 2.2 we recall the semantics of [38], then in 2.3 we extend it with user input and in 2.4 with finite-precision integers.

2.1 Notation

Given a set X, we denote with $\mathcal{P}(X)$ the set of all subsets of X. If f is a function, then $\mathrm{dom}(f)$ denotes its domain. If X is a poset and $f : X \to X$, we denote with $\mathrm{gfp}_R\, f$ the greatest fix-point of f less or equal than R.

2.2 Background on Sufficient Preconditions Semantic

We recall here the semantics from [38], on top of which we construct our new analysis. The analysis is given both in equational form (where the program is represented as a control flow graph) and in big-step form (where the program is represented with an inductive language), for our purposes we only consider the latter.

$$
\begin{aligned}
\overleftarrow{\tau}[\![\mathbf{skip}()]\!]S &\triangleq S \\
\overleftarrow{\tau}[\![v := a]\!]S &\triangleq \{\rho \mid \forall x \in E[\![a]\!]\rho.\ \rho[v \mapsto x] \in S\} \\
\overleftarrow{\tau}[\![\mathbf{assume}(b)]\!]S &\triangleq S \cup \{\rho \mid B[\![b]\!]\rho = \{\mathtt{f}\}\} \\
\overleftarrow{\tau}[\![s_1; s_2]\!]S &\triangleq (\overleftarrow{\tau}[\![s_1]\!] \circ \overleftarrow{\tau}[\![s_2]\!])S \\
\overleftarrow{\tau}[\![\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\!]S &\triangleq (\overleftarrow{\tau}[\![\mathbf{assume}(b)]\!] \circ \overleftarrow{\tau}[\![s_1]\!])S \cap (\overleftarrow{\tau}[\![\mathbf{assume}(\neg b)]\!] \circ \overleftarrow{\tau}[\![s_2]\!])S \\
\overleftarrow{\tau}[\![\mathbf{while}\ b\ \mathbf{do}\ s\ \mathbf{done}]\!]S &\triangleq \mathrm{gfp}_{\overleftarrow{\tau}[\![\neg b]\!]}(\lambda X.\ X \cap (\overleftarrow{\tau}[\![\mathbf{assume}(b)]\!] \circ \overleftarrow{\tau}[\![s]\!])X)
\end{aligned}
$$

Fig. 3: Backward semantic of statements.

Language and Forward Semantics. We assume a simple While programming language with **assume**(b), assignments and **skip**() atomic statements and sequencing, if-then-else and while loops inductive statements (see Fig. 2). The set of variables is denoted with $\mathcal{V}$ and it is assumed to be fixed. Variables are of mathematic integer type, hence program stores (or environments) are in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$. Arithmetic and boolean expressions are interpreted respectively by $E[\![a]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathbb{Z})$ and $B[\![b]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\{\mathtt{t}, \mathtt{f}\})$, whereas statements by $\tau[\![s]\!] : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$. For more details we refer the reader to [38,39].

Backward Semantic. Conventional backward analyses focus on inferring *necessary* preconditions for some post-condition. In particular, given a program $s \in$ While and a post-condition $S \in \mathcal{P}(\mathcal{E})$, they infer an over-approximation (an over-approximation of a necessary precondition is again a necessary precondition) of $P_n \triangleq \{\rho \mid \exists \rho' \in \tau[\![s]\!]\{\rho\}.\ \rho' \in S\}$. Notice that if $\tau[\![s]\!]\{\rho\} \in S$ then $\rho \in P_n$, i.e., if a store ρ transitions to S, then it is contained in P_n.

On the contrary, the backward analysis proposed in [38] infers *sufficient* preconditions. In particular, it infers an under-approximation (an under-approximation of a sufficient precondition is again a sufficient precondition) of $P_s \triangleq \{\rho \mid \forall \rho' \in \tau[\![s]\!]\{\rho\}.\ \rho' \in S\}$. Notice that necessary and sufficient preconditions can differ in the presence of non-determinism as demonstrated in the following example.

Example 1. Consider the program $s \equiv x := x + [-1, 1]$ with post-condition $S \triangleq [0, 5]$[1]. The strongest necessary precondition is $P_n = [-1, 6]$ as for any $x \in P_n$ there *exists* a trace leading to the post-condition ($\forall x \in P_n.\ \tau[\![s]\!]\{x\} \cap S \neq \varnothing$). The weakest sufficient precondition is $P_s = [1, 4]$ as for any $x \in P_s$ *all* traces lead to the post-condition ($\forall x \in P_n.\ \tau[\![s]\!]\{x\} \subseteq S$). □

Let $f : \mathcal{P}(A) \to \mathcal{P}(B)$ be a function, we define the *backward* version of f, denoted $\overleftarrow{f}$, as

$$\overleftarrow{f}(B) \triangleq \{a \in A \mid f(\{a\}) \subseteq B\}.$$

In particular, letting $f \equiv \tau[\![s]\!]$ yields $\overleftarrow{\tau}[\![s]\!]$ that computes the *sufficient* precondition of s. Backward versions of functions enjoy several properties and in

[1] With an abuse of notation we confuse the store $[x \mapsto z]$ with z.

particular we can exploit them to compute $\overleftarrow{\tau}[\![s]\!]$ by induction on the syntax of s. We report the resulting backward semantic in Fig. 3 and refer the reader to Theorems 2 and 3 of [38] for further details (in particular, the soundness of this construction).

Abstract Semantic. As usual in abstract interpretation, we represent program properties with *abstract domains*. An abstract domain is a tuple $\langle D^\sharp, \gamma^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \nabla^\sharp, \tau^\sharp[\![s]\!] \rangle$ where $\gamma^\sharp : D^\sharp \to \mathcal{P}(\mathcal{E})$ is a monotonic map, $\sqsubseteq^\sharp$: $D^\sharp \times D^\sharp$ is a partial order relation, $\sqcup^\sharp$, $\sqcap^\sharp$, are sound over-approximations of $\cup$, $\cap$, $\nabla^\sharp$ is a widening operator and $\tau^\sharp[\![s]\!]$ is a sound over-approximation of $\tau[\![s]\!]$ for atomic statements (compound statements are handled by induction on the syntax).

Whereas conventional reachability analysis is *sound* when it *over-approximates* concrete invariants, the sufficient precondition analysis is sound when it *under-approximates* them. For this reason, an abstraction of the concrete semantic of Fig. 3 can not be obtained by simply replacing the concrete operators with the abstract ones (as typically done in abstract interpretation), instead they must be replaced with a new special set of abstract operators that guarantee an under-approximation of the concrete computation. In particular, we need operators $\underline{\sqcup}^\sharp$, $\underline{\sqcap}^\sharp$, that under-approximate respectively $\cup$, $\cap$, a lower widening[2] $\underline{\nabla}^\sharp$ and $\underline{\overleftarrow{\tau}}^\sharp[\![s]\!]$ that under-approximates $\overleftarrow{\tau}[\![s]\!]$ for atomic statements. As an example, in [38] it is shown how to design such operators for the polyhedra domain, with the exception of $\underline{\sqcup}^\sharp$. However a simple, yet imprecise, definition for $\underline{\sqcup}^\sharp$ is to just return one of its arguments. A more precise operator will be presented in Sect. 3, exploiting the powerset domain. Consequently, a sound abstraction of the backward semantic can be obtained by leveraging the under-approximated versions of domain operators.

Additionally, even though the concrete backward semantics depends solely on the post-condition, to design an abstract transfer function, it can be useful to have an over-approximation of the precondition (e.g., to linearize arithmetic expressions as in [37]). Fortunately this over-approximation can be easily computed (through a traditional forward reachability analysis) and stored for later usage in the backward pass. Hence, for the rest of this paper, we will assume the availability of an over-approximation of the result of each backward operator.

Correctness and Incorrectness. Program specifications can be modeled in the language with **assert**(b) statements. Their semantic changes depending on the goal of the analysis, whether it is for preconditions for program correctness or incorrectness. We can see this with an example.

Example 2. Consider the programs of Figs. 4a, 4b, 4c. Program 4a contains no assertion, thus it is trivially always correct (and never incorrect). In Program 4b, to compute a sufficient precondition for correctness we collect $x \geq 50$ from the

[2] A lower (or dual) widening $\underline{\nabla}^\sharp : D^\sharp \times D^\sharp \to D^\sharp$ is a binary operator such that: 1. for all $d_1^\sharp, d_2^\sharp \in D^\sharp$, $\gamma^\sharp(d_1^\sharp \mathbin{\underline{\nabla}^\sharp} d_2^\sharp) \subseteq \gamma^\sharp(d_1^\sharp) \cap \gamma^\sharp(d_2^\sharp)$; 2. for any sequence $(x_i^\sharp)_{i \in \mathbb{N}}$, the sequence defined as $y_0^\sharp \triangleq x_0^\sharp$ and $y_{i+1}^\sharp \triangleq y_i^\sharp \mathbin{\underline{\nabla}^\sharp} x_{i+1}^\sharp$ becomes stable in a finite number of iterations.

```
1: x := x + 10
```
(4a)

```
1: x := x + 10
2: assert(x ≥ 50)
```
(4b)

```
1: x := x + 10
2: assert(x ≥ 50)
3: x := x − 10
4: assert(x ≤ 50)
```
(4c)

assertion and then subtract 10, which yields $x \geq 40$ (likewise, for incorrectness we obtain $x < 40$). In Program 4c the reasoning for correctness is the same as the previous program: we proceed backwards and for each assertion encountered we retain only the states satisfying its guard. This yields $x \in [40, 50]$. On the other hand, to find preconditions for program incorrectness we have two possibilities: the failure of the assertion at line 2 or 4. For the one at line 2 we proceed as in Program 4b (which yields $x < 40$). For the one at line 4 we collect $x > 50$ and proceed backwards. As soon as we encounter the assertion at line 2, we retain the states satisfying its guard (as for correctness), as in order to reach line 4, an execution must satisfy the assertion at line 2. Combining the preconditions yields $x \notin [40, 50]$. □

The previous example suggests that our semantics can infer preconditions for both correctness and incorrectness. In the former case the analysis has to start from $\top$ (or some other post-condition of interest) and compute $\overleftarrow{\tau}[\![\mathbf{assert}(b)]\!]S$ as $S \cap [b]$ where $[b]$ denotes the set of states satisfying b. In the latter it has to start from $\bot$ and compute $\overleftarrow{\tau}[\![\mathbf{assert}(b)]\!]S$ as $(S \cap [b]) \cup [\neg b]$.

2.3 User Input

A crucial aspect of programming is I/O. The language we studied has a limited support for I/O in the form of input arguments and return values, but this is often not enough as real-world programs can perform I/O operations at arbitrary execution points. In particular, as we focus on finding preconditions, we are mostly interested in the effect of user *input*.

To address this issue, a new statement, $v := \mathbf{input}()$, is added to the language. When this statement is executed, the machine reads a value from an external source and stores it in v. As different input statements may read from different sources, we assume that finitely many sources are available, each identified with an index, and annotate each input statement with the index identifying the source, e.g., $v := \mathbf{input}_n()$ for an input from the *nth* source.

Remark 1 (Input versus non-determinism). It might appear that the following two programs

$x := \mathbf{input}_1()$	$x := [-\infty, +\infty]$
$\mathbf{assert}(x > 100)$	$\mathbf{assert}(x > 100)$

have the same semantics, but this is not the case: intuitively, $\mathbf{input}_1()$ differs from the non-deterministic interval $[-\infty, +\infty]$ in that the former depicts an user-controllable, input to the program, while the latter depicts an "internal"

uncontrollable form of input. Consequently, the (correctness) precondition for the first program involves not only the value of x but also the value returned by $\mathbf{input}_1()$. Vice versa, the one of the second program only concerns the value of x. In the first case, the precondition is: $\mathbf{input}_1 > 100$. Indeed, if this condition holds the program satisfies the assertion. In the second one, the precondition is $\varnothing$. Indeed, there is no set of stores that for *any* non-deterministic execution ensures that the assertion is satisfied. □

Concrete Semantic In order to model user input, our representation of states as program stores is not sufficient anymore. Inspired by the input representation proposed in [25], we model external inputs as *streams*, i.e., pairs of an infinite sequence of integers and an index, where the index is used to store the position of the next number to be read from the sequence. The set of streams is denoted with $\mathcal{S} \triangleq \mathbb{Z}^\omega \times \mathbb{N}$ and $\text{get}(s)$ indicates the current value of the stream s. We further assume that p different input sources are available, modeled as *multi-streams*, i.e., vectors of p streams, denoted with $\mathcal{M} \triangleq \mathcal{S}^p$. Therefore program states are modeled as pairs of a store (environment) and a multi-stream, $\mathcal{E}' \triangleq \mathcal{E} \times \mathcal{M}$. We further denote with $\text{incr}_n(m)$ the multi-stream m in which the nth stream is equal to m_n but with its index incremented and the other streams left unchanged.

For non-input statements, the semantic $\tau[\![s]\!] : \mathcal{P}(\mathcal{E}') \to \mathcal{P}(\mathcal{E}')$ operates on the store as before, while leaving the streams untouched, e.g.,

$$\tau[\![v := a]\!]P \triangleq \{(\rho[v \mapsto x], m) \mid (\rho, m) \in P, x \in E[\![a]\!]\rho\}.$$

On the contrary, the semantic of $v := \mathbf{input}_n()$ stores in v the current value of the nth stream and increments its index:

$$\tau[\![v := \mathbf{input}_n()]\!]P \triangleq \{(\rho[v \mapsto \text{get}(m_n)], \text{incr}_n(m)) \mid (\rho, m) \in P\}.$$

Time-invariant Stream Abstraction In the concrete semantics, user inputs are modeled as reads from an infinite sequence, but since sequences are not directly representable by conventional numeric abstract domains, some further abstraction is necessary. Rather than providing directly an encoding of concrete states into numeric domains, we propose an intermediate abstraction allowing later an easier representation in numeric domains.

Abstraction. Input streams can be abstracted in several different ways, e.g., by retaining a finite prefix, or with an automaton, etc. In this work, we consider an abstraction that classifies streams as either time-invariant (e.g., $(111..., 0) \in \mathcal{S}$) or time-dependent (e.g, $(123..., 0) \in \mathcal{S}$). The set of time-invariant streams is denoted with $\mathcal{S}_i$ and the set of time-dependent streams with $\mathcal{S}_t$. In this abstraction, in the former case we track the value that is repeated in the stream, while in the latter all the information is discarded. In both cases the information regarding the current position on the stream (the index) is not preserved.

Example 3. Consider the following example.

$i, j := 0$
for $x = 1$ to 10 **do**
  $i := i + \mathbf{input}_1()$
  $j := j + \mathbf{input}_2()$
end for
assert$(i = j)$

There are several streams that can render the assertion true, for instance if $\mathbf{input}_1()$ returns 10 at the first iteration and 0 for other iterations and $\mathbf{input}_2()$ always returns 1. In this case the stream for $\mathbf{input}_1()$ is time-dependent and the one for $\mathbf{input}_2()$ is time-invariant.

Notice that the abstraction tracks sets of states, maintaining the value of both program variables and time-invariant streams, and thus it is able to express relationships between them. For instance, the set of preconditions where both streams are time-invariant, with the same value in $[0, 100]$. It can also infer relations between stream values and programs variables. □

To formalize this abstraction we use a partial map from *p stream variables* to $\mathbb{Z}$. We denote the *nth* stream variable with v_n^s and with $\mathcal{V}_s$ the set of stream variables. If v_n^s is defined in the map, then the corresponding stream is time-invariant with value matching the variable's value, otherwise it is time-dependent. More formally:

Definition 1 (Time-invariant stream abstraction). *Let $\mathcal{E}'$ be a set of states. We define $\widehat{\mathcal{E}}' \triangleq (\mathcal{V} \cup \mathcal{V}_s) \rightharpoonup \mathbb{Z}$, the concretization $\widehat{\gamma} : \mathcal{P}(\widehat{\mathcal{E}}') \to \mathcal{P}(\mathcal{E}')$ and abstraction $\widehat{\alpha} : \mathcal{P}(\mathcal{E}') \to \mathcal{P}(\widehat{\mathcal{E}}')$ functions as follows:*

- $\widehat{\gamma}(R) \triangleq \{(\rho, m) \mid \exists \widehat{\rho} \in R.\, \widehat{\rho}(v_k)\big|_{\mathcal{V}} = \rho(v_k)\big|_{\mathcal{V}},\ \text{matchStream}(\widehat{\rho}, m)\}$
- $\widehat{\alpha}(R) \triangleq \{\widehat{\rho} \mid \exists (\rho, m) \in R.\, \widehat{\rho}(v_k)\big|_{\mathcal{V}} = \rho(v_k)\big|_{\mathcal{V}},\ \text{matchStream}(\widehat{\rho}, m)\}$

where:

$$\text{matchStream}(\widehat{\rho}, m) \Leftrightarrow \forall n = 1, .., p.\ \widehat{\rho}(v_n^s) = \begin{cases} \text{get}(m_n) & \text{if } m_n \in S_i \\ undef & \text{if } m_n \in S_t \end{cases}$$ □

Theorem 1. *The following Galois Connection holds:* $(\mathcal{P}(\mathcal{E}'), \subseteq) \underset{\widehat{\alpha}}{\overset{\widehat{\gamma}}{\leftrightarrows}} (\mathcal{P}(\widehat{\mathcal{E}}'), \subseteq)$.

Semantic. The semantic of statements different from $v := \mathbf{input}_n()$ coincides with the concrete one as those statements only operate on the store part of the state (not on the streams), and that part is not abstracted; for example

$$\widehat{\tau}[\![v := a]\!]P \triangleq \{\widehat{\rho}[v \mapsto x] \mid \widehat{\rho} \in P, x \in E[\![a]\!]\widehat{\rho}\big|_{\mathcal{V}}\}.$$

On the other hand, $\widehat{\tau}[\![v := \mathbf{input}_n()]\!]$ affects the stream. In particular, if the stream is time-invariant (thus v_n^s is defined in $\widehat{\rho}$) then $\widehat{\tau}[\![v := \mathbf{input}_n()]\!]$ copies

v_n^s (which is equal to the stream's value) to v. Otherwise, if the stream is time-dependent, v gets $[-\infty, +\infty]$ as no information is retained in the abstraction and $[-\infty, +\infty]$ is always a sound choice. More formally:

$$\widehat{\tau}[\![v := \mathbf{input}_n()]\!]P \triangleq \{\widehat{\rho}[v \mapsto x] \mid \widehat{\rho} \in P, x \in \mathbb{Z}.\ (v_n^s \in \mathrm{dom}(\widehat{\rho}) \Rightarrow x = \widehat{\rho}(v_n^s))\}.$$

As in the forward semantic, the backward semantic of non-input statements can be easily derived from the concrete semantic; for example

$$\overleftarrow{\widehat{\tau}}[\![v := a]\!]S = \{\widehat{\rho} \mid \forall x \in E[\![a]\!]\widehat{\rho}.\ \widehat{\rho}[v \mapsto x] \in S\}.$$

The backward semantic of input statements ensures that if the stream is time-dependent (v_n^s undefined) then for *all* substitutions of v in $\widehat{\rho}$ the resulting store is in the post-condition. Otherwise if the stream is time-independent, then $\widehat{\rho}[v \mapsto \widehat{\rho}(v_n^s)]$ must be in the post-condition. More formally we have:

$$\begin{aligned}\overleftarrow{\widehat{\tau}}[\![v := \mathbf{input}_n()]\!]S = \{\widehat{\rho} \mid \forall x \in \mathbb{Z}.\ &(v_n^s \notin \mathrm{dom}(\widehat{\rho}) \wedge \widehat{\rho}[v \mapsto x] \in S) \vee \\ &(v_n^s \in \mathrm{dom}(\widehat{\rho}) \wedge (x = \widehat{\rho}(v_n^s) \Leftrightarrow \widehat{\rho}[v \mapsto x] \in S))\}\end{aligned}$$

Theorem 2. *The semantic of statements, both forward and backward, is sound:*

$$\tau[\![s]\!]\widehat{\gamma}(R) \subseteq \widehat{\gamma}(\widehat{\tau}[\![s]\!]R) \qquad\qquad \widehat{\gamma}(\overleftarrow{\widehat{\tau}}[\![s]\!]S) \subseteq \overleftarrow{\tau}[\![s]\!]\widehat{\gamma}(S).$$

Abstract Semantic In the previous section, we demonstrated an abstraction of input streams into environments where some variables can be defined (time-invariant streams) or not (time-dependent streams). The usual numeric domains can not be used directly as they assume that *all* variables are defined. This issue was already studied in the context of abstracting *heterogeneous* environments (i.e., environments where some variables are optional). One simple approach is to partition the environments according to the defined variables, but this scales poorly as there can be an exponential number of partitions in the worst case.

We adopt instead the method proposed in [31], though in a simplified version. This approach lifts a numeric domain $D^\sharp$, to a domain $\widehat{D}^\sharp$ consisting of pairs $\langle d^\sharp, l\rangle$, where $\mathcal{V} \subseteq l \subseteq \mathcal{V} \cup \mathcal{V}_s$ and $d^\sharp \in D^\sharp$. The element $d^\sharp$ is defined on $\mathcal{V} \cup \mathcal{V}_s$ and the concretization of $\langle d^\sharp, l\rangle$ yields states with domain subsuming l and satisfying the constraints of $d^\sharp$. In the original approach [31], the elements of $\widehat{D}^\sharp$ contained an additional set $u \supseteq l$ representing an upper bound for the domain of the states (here $u = \mathcal{V} \cup \mathcal{V}_s$), but in our case this additional flexibility is not needed since stream variables can not be added or removed explicitly (e.g., with ad-hoc statements) but they can only be *added* as a side-effect of input statements, hence only the lower bound can vary.

More formally the concretization is defined as $\widehat{\gamma}^\sharp(\langle d^\sharp, l\rangle) \triangleq \{\widehat{\rho} \mid \exists \widehat{\rho}' \in \gamma^\sharp(d^\sharp), \mathcal{V} \subseteq l \subseteq \mathrm{dom}(\widehat{\rho}) \subseteq \mathcal{V} \cup \mathcal{V}_s, \widehat{\rho} = \widehat{\rho}'\big|_{\mathrm{dom}(\widehat{\rho})}\}$. Details on the construction of over-approximation domain operators can be found in [31]. Here instead we focus on under-approximation operators. If $\sqcap^\sharp$, $\sqcup^\sharp$, $\underline{\nabla}^\sharp$ are under-approximation operators for the base domain $D^\sharp$, then they can be lifted to $\widehat{D}^\sharp$:

- *Join:* $\langle d_1^\sharp, l_1\rangle \mathbin{\widehat{\sqcup}^\sharp} \langle d_2^\sharp, l_2\rangle \triangleq \langle d_1^\sharp \mathbin{\sqcup^\sharp} d_2^\sharp, l_1 \cup l_2\rangle$;
- *Meet:* $\langle d_1^\sharp, l_1\rangle \mathbin{\widehat{\sqcap}^\sharp} \langle d_2^\sharp, l_2\rangle \triangleq \langle d_1^\sharp \mathbin{\sqcap^\sharp} d_2^\sharp, l_1 \cup l_2\rangle$;
- *Widening:* $\langle d_1^\sharp, l_1\rangle \mathbin{\underline{\widehat{\nabla}}^\sharp} \langle d_2^\sharp, l_2\rangle \triangleq \begin{cases} \langle d_1^\sharp \mathbin{\underline{\nabla}^\sharp} d_2^\sharp, l_1\rangle & \text{if } l_2 \subseteq l_1 \\ \langle d_1^\sharp \mathbin{\sqcap^\sharp} d_2^\sharp, l_2\rangle & \text{if } l_1 \subset l_2 \end{cases}$.

Proposition 1. *$\mathbin{\widehat{\sqcup}^\sharp}$, $\mathbin{\widehat{\sqcap}^\sharp}$ are sound under-approximations of $\cup$, $\cap$ and $\mathbin{\underline{\widehat{\nabla}}^\sharp}$ is a lower widening.*

Semantic. The semantic of input statements can be handled as follows:

$$\widehat{\tau}^\sharp[\![v := \mathbf{input}_n()]\!]\langle D^\sharp, l\rangle \triangleq \begin{cases} \langle \tau^\sharp[\![v := [-\infty, \infty]]\!] D^\sharp, l\rangle & \text{if } v_n^s \notin l \\ \langle \tau^\sharp[\![v := v_n^s]\!] D^\sharp, l\rangle & \text{if } v_n^s \in l \end{cases}$$

Indeed, if $v_n^s \notin l$ then the concretization contains *both* time-invariant and time-dependent streams: for the latter no information is stored, thus v gets $\top$. If instead $v_n^s \in l$ then the concretization contains *only* time-invariant streams and thus the assignment copies the value from the stream variable.

The backward semantic is computed as:

$$\widehat{\overleftarrow{\tau}}^\sharp[\![v := \mathbf{input}_n()]\!]\langle D^\sharp, l\rangle \triangleq \begin{cases} \langle D^\sharp, l\rangle & \text{if } v_n^s \notin l \wedge \overleftarrow{\tau}^\sharp[\![v := [-\infty, \infty]]\!] D^\sharp = D^\sharp \\ \langle \overleftarrow{\tau}^\sharp[\![v := v_n^s]\!] D^\sharp, l \cup \{v_n^s\}\rangle & \text{otherwise} \end{cases}$$

Indeed, we can distinguish three cases:

1. If $v_n^s \in l$ then we only have time-invariant streams in the post-condition. In this case the forward transfer function performs the assignment $v := v_n^s$, thus the backward precondition can simply invert this assignment;
2. If $v_n^s \notin l$ and $\overleftarrow{\tau}^\sharp[\![v := [-\infty, \infty]]\!] D^\sharp = D^\sharp$ then we have *both* time-invariant and time-dependent streams. In addition, as the backward projection leaves $D^\sharp$ unmodified, for *all* states $\widehat{\rho} \in \gamma^\sharp(D^\sharp)$ and $x \in \mathbb{Z}$, $\widehat{\rho}[v \mapsto x] \in \gamma^\sharp(D^\sharp)$. Therefore the backward precondition is simply $\langle D^\sharp, l\rangle$. Notice that the condition on the backward projection is crucial to ensure the soundness of time-dependent streams: the projection over-approximates any assignment, thus [38, Theorem 2.6] ensures that $\widehat{\rho}[v \mapsto x] \in \gamma^\sharp(D^\sharp)$.
3. If $v_n^s \notin l$ and $\overleftarrow{\tau}^\sharp[\![v := [-\infty, \infty]]\!] D^\sharp \neq D^\sharp$ then, as before, we have *both* time-invariant and time-dependent streams, but, unlike the previous case, there exist $\widehat{\rho} \in \gamma^\sharp(D^\sharp)$ and $x \in \mathbb{Z}$ such that $\widehat{\rho}[v \mapsto x] \notin \gamma^\sharp(D^\sharp)$. Consequently, time-dependent streams can not be included in the precondition as they would be unsound. For this reason the precondition adds v_n^s to l (thus under-approximating the precondition) and transforms $D^\sharp$ as in the first case.

Theorem 3. *The abstract semantic is sound, i.e., for any $s \in$ While and $\widehat{d}^\sharp \in \widehat{D}^\sharp$ the following holds:*

$$\widehat{\tau}[\![s]\!]\widehat{\gamma}^\sharp(\widehat{d}^\sharp) \subseteq \widehat{\gamma}^\sharp(\widehat{\tau}^\sharp[\![s]\!]\widehat{d}^\sharp) \qquad\qquad \widehat{\overleftarrow{\tau}}[\![s]\!]\widehat{\gamma}^\sharp(\widehat{d}^\sharp) \supseteq \widehat{\gamma}^\sharp(\widehat{\overleftarrow{\tau}}^\sharp[\![s]\!]\widehat{d}^\sharp)$$

2.4 Integer Wrapping

In this section, we generalize our framework to support fixed precision integers (i.e., with wrap-around), typically found in C-like languages. This is important as some analyzers detect integer overflows but do not handle wrap-around: they either stop the analysis for the traces that overflow (which is not sound for programs that do wrap-around on purpose) or put the variable to the full range of their type (which is sound but imprecise). For the sake of brevity, we limit our presentation to unsigned 8-bit integers, but it is easy to generalize this framework to other types.

Arithmetic and boolean semantics are replaced with versions that operate with 8-bit unsigned integers, e.g., $E[\![x + 10]\!]\{x \mapsto 250\} = \{[x \mapsto 4]\}$. To do so, it suffices to replace the usual arithmetic operators with versions that take care of integer wrapping. Unfortunately this requires new wrap aware operators to be designed. To avoid this difficulty, we prefer a modular approach in which firstly the result is computed with infinite precision operators (i.e., the usual unwrapped ones), and then it is wrapped with a wrapping operator.

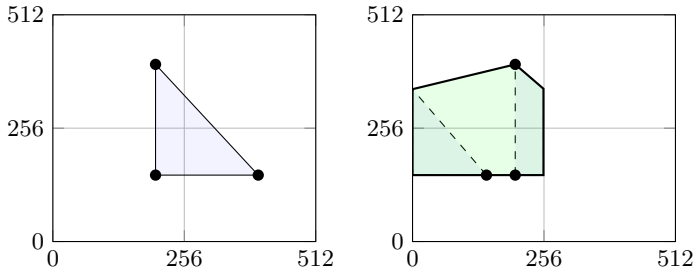
Definition 2 (Wrapping operator). *Define* $\mathrm{wrap} : \mathbb{Z} \to [0, 255]$ *as* $\mathrm{wrap}(z) \triangleq z \mod 256$*, where* $\mod$ *computes the Euclidean remainder.* □

Consequently, the abstract semantic must take into account wrapping of integers. Several approaches have been proposed in the literature to handle this problem [45,29,27,46].

Case Study: Polyhedra Domain As an example, we show how to instantiate the abstract semantic to the case of the polyhedra domain. Our work is based on the work of Simon et al. [46]: they demonstrate how to design sound abstract operators for the polyhedra domain that take into account integer wrapping. For this purpose, they presented an algorithm for computing a $\mathrm{wrap}^\sharp$ operator. It takes in input a polyhedron P, a variable v to wrap, and as result it produces a new polyhedron P' in which v lies in $[0, 255]$. Fig. 5 shows an example of the computation of $\mathrm{wrap}^\sharp$.

We extend their work (which only tackles over-approximation forward operators) to handle under-approximation backward operators. Intuitively, the backward version of $\mathrm{wrap}^\sharp$ for a polyhedron P (along a variables v) should compute a polyhedral representation of the points that, after wrapping, end up in P. This boils down to replicating P infinitely many times, where each copy is translated by a integer multiple of 256 along v, i.e., the sequence $\{P + 256k\boldsymbol{e}_v\}_{k\in\mathbb{Z}}$. Fig. 6 shows an example of this computation.

Although all the polyhedra of the sequence above are valid unwrappings, not necessarily all of them represent reachable states. In particular, if *pre* is an over-approximation of the input of $\mathrm{wrap}^\sharp$, then the valid polyhedra are only the ones intersecting *pre*. This information can be used to guide the unwrapping of a polyhedron. We present in Algorithm 1 a procedure for computing $\overleftarrow{\mathrm{wrap}}^\sharp$. The auxiliary function $quadrantIndices(pre, v)$ computes the indices of the quadrants spanned by v in *pre* (see [46, Algorithm 1]). Notice that the polyhedra of



(a) Polyhedron before wrapping. (b) Polyhedron after wrapping.

Fig. 5: Wrapping of a polyhedron along the horizontal axis. The polyhedron on the left is split in two parts: one that does not need wrapping ($x \in [0, 255]$) and another that does ($x \in [256, 511]$). To compute the result (green polyhedron), the first part is joined with the translation of the second one by -256 (along x).

the sequence are merged together with an under-approximating join, but this can incur a loss of precision if the polyhedra are separated (as in Fig. 6) since in this case the set union is not convex, and thus to under-approximate it with a polyhedral shape, only one polyhedron can be retained (hence in Fig. 6, the result must be either one of the polyhedra in green or the one in blue). A robust solution to this kind of imprecisions will be addressed in the next section. Additionally, the meet with *pre* in the algorithm excludes the polyhedra that are surely not reachable, thus increasing the odds of retaining an appropriate polyhedron.

3 Powerset Domain

As noted in [38], designing an under-approximating join for polyhedra can be challenging. This problem was sidestepped by designing such an operator only in some specific cases, namely on joins occurring in the analysis of backward filters of if-then-else statements and while loops. This simplifies the design as only under-approximations of the join of an arbitrary polyhedron with a half-space are handled. This is carried out using special heuristics, tailored to handle many practical cases. Unfortunately, they are not robust and may cause losses of precision in other cases. This is especially true for our semantic, as, unlike the one in [38], we use $\sqcup^\sharp$ to handle arbitrary joins (e.g., in $\overleftarrow{\text{wrap}}$ and later in this section for widenings).

Furthermore, even if a perfect join heuristic could be designed, the polyhedra domain would still not be precise if the concrete union is non-convex. This can often occur in real world programs (e.g., in the unwrapping of the polyhedron of Fig. 6).

3.1 Under-approximated Powerset

A robust approach to addressing this issue is to leverage the powerset [26] construction: in this construction, a base domain $D^\sharp$ is lifted to a finite set of

Algorithm 1 Calculate $\overleftarrow{\text{wrap}}^\sharp(P, v, t, pre)$

Require: Parameter $m > 0$: maximum number of copies.
$q_l, q_u \leftarrow quadrantIndices(pre, v, t)$
if $q_l = -\infty \wedge q_u = +\infty$ **then**
  return P
else if $q_l = -\infty$ **then**
  $q_l \leftarrow q_u - m$
else if $q_u = +\infty$ **then**
  $q_u \leftarrow q_l + m$
else
  {Retain at most m copies}
  $q_u \leftarrow q_l + \min(q_u - q_l, m)$
end if
$Q \leftarrow \bot^\sharp$
for $k \leftarrow q_l$ to $q_u - 1$ **do**
  $Q \leftarrow Q \sqcup^\sharp ((\tau^\sharp [\![v := v - 256k]\!] P) \sqcap^\sharp pre)$
end for
return Q

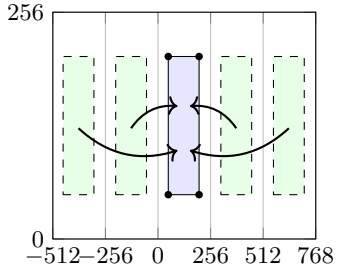


Fig. 6: Unwrapping of x. The unwrapping of the blue polyhedron produces infinitely many (here only four are shown) copies of it, separated by the integer's size. The wrapping of each polyhedron in green (depicted with the arrows) coincides with P.

abstract elements $\mathcal{P}_{finite}(D^\sharp)$. The concretization of a set yields the union of the concretizations of all its elements. Notably, the join operator becomes exact, and thus it is a sound under-approximation of $\cup$. Consequently the under-approximation join $\sqcup_p^\sharp$ can coincide with the over-approximating one $\sqcup_p^\sharp$.

Definition 3 (Powerset domain). *Let $D^\sharp$ be an abstract domain. We let $P(D^\sharp) \triangleq \mathcal{P}_{finite}(D^\sharp)$ be its powerset lifting. $P(D^\sharp)$ is partially ordered by $S_1^\sharp \sqsubseteq_p^\sharp S_2^\sharp \Leftrightarrow \forall d_1^\sharp \in S_1^\sharp.\ \exists d_2^\sharp \in S_2^\sharp.\ d_1^\sharp \sqsubseteq^\sharp d_2^\sharp$. Moreover, join and meet are respectively defined as $S_1^\sharp \sqcup_p^\sharp S_2^\sharp \triangleq S_1^\sharp \cup S_2^\sharp$ and $S_1^\sharp \sqcap_p^\sharp S_2^\sharp \triangleq \{d_1^\sharp \sqcap^\sharp d_2^\sharp \mid d_1^\sharp \in S_1^\sharp, d_2^\sharp \in S_2^\sharp\}$.* □

Additionally, if the base meet is exact (which is the case in many numeric domains, including polyhedra), also $\sqcap_p^\sharp$ is, thus it is a sound under-approximating operator as well.

Widening. A trivial widening can be obtained by joining all elements of the powerset (for each argument) and then applying the base widening (hence the result is a singleton). Likewise to get a trivial lower widening, it is possible to apply the base lower widening on just one element of each argument and discard all the others. Unfortunately, these operators are quite imprecise, as shown in the following example.

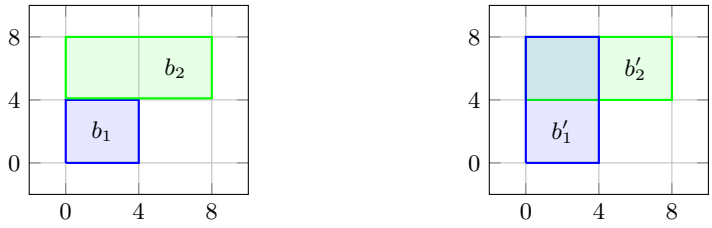


Fig. 7: Powerset refinement. The blue polyhedron (left figure) can be extended over the green one (right figure) without changing the overall concretization of the powerset.

Example 4. Consider the following example suggested by Gopan and Reps [28].

$i, j := 0$
for $i := 1$ to 100 **do**
  if $i \leq 50$ **then** $j \leftarrow j + 1$ **else** $j \leftarrow j - 1$
end for

The while loop presents two phases: one in which j is incremented (then branch), and one in which it is decremented (else branch). This induces a non-convex loop invariant, that requires at least two polyhedra to be represented precisely. It is clear that the trivial widenings can not find such a result as they yield a singleton. □

A simple, yet useful, improvement consists in retaining *stable* elements, i.e., elements that are shared in both arguments, and widen only the *remaining* (unstable) elements. More formally:

Definition 4 (Improved Powerset Widening). *Let* $S_1^\sharp = \langle d_{1,1}^\sharp, .., d_{1,n}^\sharp \rangle$ *and* $S_2^\sharp = \langle d_{2,1}^\sharp, .., d_{2,m}^\sharp \rangle$. *Define:*

$$S_1^\sharp \nabla_p^\sharp S_2^\sharp \triangleq \begin{cases} \langle d_1^\sharp \nabla^\sharp (\sqcup^\sharp S_{u_2}^\sharp) \rangle \cup (S_s^\sharp \setminus \{d_1^\sharp\}) & \text{if } S_{u_1}^\sharp = \varnothing \wedge S_{u_2}^\sharp \neq \varnothing \wedge d_1^\sharp \in S_1^\sharp \\ \langle (\sqcup^\sharp S_{u_1}^\sharp) \nabla^\sharp (\sqcup^\sharp S_{u_2}^\sharp) \rangle \cup S_s^\sharp & \text{otherwise} \end{cases}$$

$$S_1^\sharp \underline{\nabla}_p^\sharp S_2^\sharp \triangleq \begin{cases} S_s^\sharp & \text{if } S_{u_1}^\sharp = \varnothing \vee S_{u_2}^\sharp = \varnothing \\ \langle d_{u_1}^\sharp \mathbin{\underline{\nabla}^\sharp} d_{u_2}^\sharp \rangle \cup S_s^\sharp & \text{if } d_{u_1}^\sharp \in S_{u_1}^\sharp \wedge d_{u_2}^\sharp \in S_{u_2}^\sharp \end{cases}$$

where $S_s^\sharp \triangleq S_1^\sharp \cap S_2^\sharp$ *is the set of stable elements,* $S_{u_1}^\sharp \triangleq S_1^\sharp \setminus S_s^\sharp$ *the set of unstable elements of* $S_1^\sharp$ *and* $S_{u_2}^\sharp \triangleq S_2^\sharp \setminus S_s^\sharp$ *the set of unstable elements of* $S_2^\sharp$. □

Proposition 2. $\nabla_p^\sharp$ *(*$\underline{\nabla}_p^\sharp$*) is an upper (lower) widening operator for* $P(D^\sharp)$.

Refinement. Consider the program s computing $y := y + [0, 8]$ and the powerset $S \triangleq \{b_1, b_2\}$ of Fig. 7. The backward (concrete) semantic of S, computed point-wise, is $\{\varnothing\}$ as both $\overleftarrow{\tau}[\![s]\!]b_1 = \varnothing$ and $\overleftarrow{\tau}[\![s]\!]b_2 = \varnothing$. But the set of states represented by S, that is $\cup_{b \in S} b$, does admit a non-empty precondition since

Algorithm 2 Integer polyhedral refinement: adjacent constraints

Require: $d_1^\sharp, d_2^\sharp$ polyhedra in constraint representation.
Ensure: $d^\sharp$ refines $d_1^\sharp$ with $d_2^\sharp$.
  $d^\sharp \leftarrow d_1^\sharp$
  for c **matching** $\boldsymbol{a} \cdot \boldsymbol{v} \geq b$ **in** $d_1^\sharp$ **do**
    $d_{1,nc}^\sharp \leftarrow d_1^\sharp \setminus \{c\}$
    $v_1, r_1 \leftarrow sat(d_1^\sharp, c)$ $\{sat(d^\sharp, c)$ returns the vertices and rays of $d^\sharp$ saturating $c\}$
    $c' \leftarrow \boldsymbol{a} \cdot \boldsymbol{v} \geq b - 1$
    $v_2, r_2 \leftarrow sat(d_2^\sharp, c')$
    $d_m^\sharp \leftarrow gen(v_1 \cup v_2, r_1 \cap r_2)$ $\{gen(v, r)$ returns the polyhedron generated by vertices v and rays $r\}$
    if $c' \in d_m^\sharp$ **then**
      $d_{m,nc'}^\sharp \leftarrow d_m^\sharp \setminus \{c'\}$
      $d_i^\sharp \leftarrow d_{1,nc}^\sharp \cap d_{m,nc'}^\sharp \cap d_2^\sharp$
    else
      $d_i^\sharp \leftarrow d_{1,nc}^\sharp \cap d_m^\sharp \cap d_2^\sharp$
    end if
    $d_h^\sharp \leftarrow d_1^\sharp \sqcup^\sharp d_i^\sharp$
    $\{rays(\cdot)$ computes the set of rays of a polyhedron$\}$
    if $rays(d_1^\sharp) \subseteq rays(d_h^\sharp)$ **then**
      $d^\sharp \leftarrow d_h^\sharp$
      break
    end if
  end for

$\overleftarrow{\tau}[\![s]\!] \cup_{b \in S} b = [x \mapsto [0,4], y \mapsto 0] \neq \varnothing$. This is possible as the backward semantic (unlike the forward one) is not a $\cup-$morphism, but instead only the inclusion holds, i.e., $\bigcup_i \overleftarrow{\tau}[\![s]\!] S_i \subseteq \overleftarrow{\tau}[\![s]\!] \bigcup_i S_i$ for any family of states $\{S_i\}_{i \in \mathbb{N}}$ (whereas the equality holds for $\cup-$morphisms).

However, the powerset $S' \triangleq \{b'_1, b'_2\}$ admits a non-empty backward semantic as $\{\overleftarrow{\tau}[\![s]\!] b'_1, \overleftarrow{\tau}[\![s]\!] b'_2\} = \{[x \mapsto [0,4], y \mapsto 0], \varnothing\}$, even if S' represents the same states as S (the only difference is the internal composition of the powerset). For this reason the elements of the powerset domain should be kept as large as possible, so that $\overleftarrow{\tau}^\sharp[\![s]\!]$ is maximized (notice that in the forward analysis setting, we strive for the opposite goal, namely keeping the elements as small as possible). In particular, we can allow some sharing of states among elements of the set, provided that this does not affect the overall concretization of the powerset.

For this purpose we use a *refinement* operator: an under-approximation join $\sqcup^\sharp$ is a refinement operator if $d^\sharp \triangleq d_1^\sharp \sqcup^\sharp d_2^\sharp \sqsupseteq^\sharp d_1^\sharp$, meaning that $d^\sharp$ *refines* $d_1^\sharp$ with states from $d_2^\sharp$. Then, we can refine a powerset by replacing each element with its refinement with all the other elements. Additionally, refinements can help mitigate the computational cost of the powerset as, after refinement, some elements may become redundant and thus can be removed.

Algorithm 3 Integer polyhedral refinement: adjacent singleton variables

Require: $d_1^\sharp, d_2^\sharp$ polyhedra in constraint representation.
Ensure: $d^\sharp$ refines $d_1^\sharp$ with $d_2^\sharp$.
  $d^\sharp \leftarrow d_1^\sharp$
  for c **matching** $v = n$ **in** $d_1^\sharp$ **do**
    if $v = n + 1 \in d_2^\sharp \vee v = n - 1 \in d_2^\sharp$ **then**
      {$rays(\cdot)$ computes the set of rays of a polyhedron}
      if $rays(d_1^\sharp) = rays(d_2^\sharp)$ **then**
        $d^\sharp \leftarrow d_1^\sharp \sqcup^\sharp d_2^\sharp$
        break
      end if
    end if
  end for

3.2 Case Study: Polyhedra Refinement

To design a refinement operator, it is possible to leverage a procedure for checking if the over-approximation join is exact. Indeed, an exact join is also a valid refinement. For the polyhedra domain, this problem has been studied by Bemporad et al. [6] and Bagnara et al. [4], but they focus on polyhedra representing real-valued environments.

On the other hand, if variables are of integer type (as in our semantic), the join can be exact even if the union of the polyhedra is not convex. For example the union of the polyhedra (in constraint representation) $d_1^\sharp \triangleq \{0 \leq x \leq 1\}$ and $d_2^\sharp \triangleq \{2 \leq x \leq 3\}$ is not a convex set, but still the join is exact: $\gamma(d_1^\sharp) \cup \gamma(d_2^\sharp) = \{0, 1, 2, 3\} = \gamma(d_1^\sharp \sqcup^\sharp d_2^\sharp)$.[3] Since this kind of polyhedra appears frequently in practice (e.g., in loops incrementing variables by one unit), we propose two refinement algorithms tailored for these cases.

Consider the bi-dimensional polyhedra $d_1^\sharp \triangleq \{x \geq 0, y \geq 0, x + y \leq 4\}$ and $d_2^\sharp \triangleq \{x \leq 3, y \leq 3, x + y \geq 5\}$. It is easy to check that there exists a part of $d_2^\sharp$ that can be exactly joined with $d_1^\sharp$ (the triangle with vertices $(2, 3)$, $(3, 2)$, $(\frac{8}{3}, \frac{8}{3})$), and thus can refine $d_1^\sharp$. This is the case as the strip $4 < x + y < 5$ separating it from $d_1^\sharp$ does not contain any integer. Algorithm 2 tackles this case by scanning for constraints of this kind and if they are found, computes parts of the second argument that can refine the first.

Algorithm 3 scans for a variable in $d_1^\sharp$ and $d_2^\sharp$ that is fixed in the two polyhedra to constants differing by one unit. If such a variable is found, then the join between the two polyhedra is exact. As an example, let $d_1^\sharp \triangleq \{x = 0, 0 \leq y \leq 2\}$ and $d_2^\sharp \triangleq \{x = 1, 4 \leq y \leq 6\}$. Since the strip $0 < x < 1$ does not contain any integer, the join is exact.

[3] With an abuse of notation, we confuse $\{x\} \to \mathbb{Z}$ with $\mathbb{Z}$.

4 Implementation and Experiments

In addition to the theoretical foundations, the contribution [38] included a PoC static analyzer, Banal [1]. This analyzer targets a toy language with a semantic not compatible with the one of C (e.g., it assumes mathematical integers instead of machine integers). We extended it with the features presented in this work: an implementation of a subset of the C semantic and a frontend for a significant subset of the language, user input (Sect. 2.3), machine integers (Sect. 2.4), a powerset domain (Sect. 3) and improved operators. As a consequence, our new prototype is able to analyze benchmarks from the SV-COMP competition. As our work focuses on incorrectness, we report only the results of the analysis of incorrect programs.

Witnesses Generation. To analyze SV-COMP benchmarks, Banal translates each call site to `__VERIFIER_nondet_int()` with an input statement (each with a distinct stream). Consequently, it computes preconditions in the form of an abstract element relating the input variables. Then, as all states in the abstract element are valid sufficient preconditions for the violation of some assertion, we extract one concrete vector of values. Notice that for the purpose of SV-COMP, the quality of a violation witness [8] is measured by how much it restricts the state-space exploration. The more restricted it is, the less states the validator has to explore in order to check the witness. By picking a concrete vector (which represents only one execution path) we obtain the most precise kind of witness.

Furthermore, as previously discussed, the preconditions generated by our analysis may simply lead to an infinite loop, rather than a true bug. To rule out this possibility Banal replaces each input call site with its concrete value, compiles the benchmark and runs it with a time limit (2s). If an assertion fails, then the counter-example is confirmed.

Finally, a witness is generated in SV-COMP's `graphml` format: we make a control flow automaton resembling the control flow graph of the benchmark and specify for each input site the corresponding concrete value. Moreover, to certify the correctness of our result using independent techniques, we validate the witness with the CPA-W2T [9] validator (which is specifically tailored for checking concrete witnesses) and declare the benchmark to be successfully analyzed only if successfully validated.

Experimental Evaluation. To asses the performance of our analysis, we run our tool and three leading tools from SV-COMP23: CPAChecker [10,22], UAutomizer [30] and Veriabs [2,23] on a selected subset of the competition's benchmarks. In particular, we built our set of benchmarks from the `ReachSafety-Loops` set of the competition, as it comprises several simple numerical programs, from which we removed the `nla-digbench` and `nla-digbench-scaling` folders as they contain programs with polynomial invariants that require special analysis techniques that are out of scope of this work. Our set of benchmarks contains 63 C files (35172 LOC) corresponding to 61% (in terms of LOC) of the `ReachSafety-Loops` set.

Analyzer	Count				
	Success	Unknown	Timeout/OOM	Unsupported	Other
Banal	16	16	6	25	0
CPAChecker	44	0	18	0	1
UAutomizer	37	0	25	0	1
Veriabs	43	3	17	0	0

Table 1: Outcome of the analysis of `ReachSafety-Loops` excluding `nla-digbench` and `nla-digbench-scaling` folders.

Benchmark	Analyzer	Time [s]					
		10^1	10^2	10^3	10^4	10^5	10^6
Mono3_1	Banal	0.94	1	0.84	0.96	0.84	0.92
	CPAChecker	10	22	83	✗	✗	✗
	UAutomizer	50	✗	✗	✗	✗	✗
	Veriabs	42	47	50	92	190	✗
Mono4_1	Banal	0.79	0.81	0.8	0.82	0.85	0.85
	CPAChecker	9.7	19	63	✗	✗	✗
	UAutomizer	36	97	✗	✗	✗	✗
	Veriabs	43	43	46	64	✗	✗
Mono5_1	Banal	1.2	1.3	1.2	1.2	1.2	1.3
	CPAChecker	10	20	69	✗	✗	✗
	UAutomizer	42	✗	✗	✗	✗	✗
	Veriabs	42	43	45	78	✗	✗
Mono6_1	Banal	1.4	1.2	1.2	1.3	1.2	1.2
	CPAChecker	9.9	21	76	✗	✗	✗
	UAutomizer	75	✗	✗	✗	✗	✗
	Veriabs	41	42	46	70	✗	✗
const_1-2	Banal	0.38	0.41	0.4	0.42	0.39	0.41
	CPAChecker	9.5	19	55	✗	✗	✗
	UAutomizer	36	✗	✗	✗	✗	✗
	Veriabs	30	32	32	32	32	32

Benchmark	Analyzer	Time [s]					
		10^1	10^2	10^3	10^4	10^5	10^6
count_up_down-2	Banal	0.49	0.5	0.51	0.51	0.5	0.49
	CPAChecker	9.6	19	62	✗	✗	✗
	UAutomizer	37	✗	✗	✗	✗	✗
	Veriabs	31	32	30	28	29	28
multivar_1-2	Banal	0.61	0.65	0.71	0.67	0.63	0.61
	CPAChecker	7.7	8	7.6	7.7	7.8	7.8
	UAutomizer	28	27	27	27	27	27
	Veriabs	27	28	27	29	28	28
simple_2-2	Banal	0.31	0.33	0.32	0.31	0.34	0.34
	CPAChecker	✗	✗	✗	✗	✗	✗
	UAutomizer	✗	✗	✗	✗	✗	✗
	Veriabs	26	26	27	26	27	27
simple_nested	Banal	11	11	11	11	11	11
	CPAChecker	27	✗	✗	✗	✗	✗
	UAutomizer	83	✗	✗	✗	✗	✗
	Veriabs	39	100	✗	✗	✗	✗
assert_loop	Banal	0.46	0.41	0.38	0.41	0.46	0.41
	CPAChecker	9.8	21	73	✗	✗	✗
	UAutomizer	34	110	✗	✗	✗	✗
	Veriabs	36	35	37	88	✗	✗

Table 2: Analysis time for increasing number of loop iterations, for some selected benchmarks. ✗ denotes a timeout.

All tests were conducted on an Intel Core i7-8550U CPU with 3GiB memory limit and 300 seconds time limit using the BenchExec [11] platform. We report the results in Table 1, where *unknown* indicates an inconclusive result and *unsupported* indicates a failure due to missing support for some C features (e.g., arrays, pointers).

Despite some encouraging results, Banal performs worse than the other tools due to several imprecisions (e.g, widening failure, non-linear arithmetic) in the analysis and missing support for several C features. However, since it is based on abstract interpretation, Banal is faster than the other tools. In particular, on the successfully analyzed tasks Banal is 22x faster than CPAChecker, 50x than UAutomizer and 50x than Veriabs. This performance gap becomes even wider if we consider programs where bugs are reached after many loop iterators (so called deep bugs) as Banal uses widening operators whereas other tools often necessitate loop unrolling and thus are limited to bugs reachable in few loop iterations (so called shallow bugs). To assess this, we selected some benchmarks from the previous set and re-run the analysis fixing the number of loop iterations with different values. The results are reported in Table 2. CPAChecker and UAutomizer hit timeouts when loops require > 1000 unrollings, while Ba-

```
for (int a = 0; a < 1000; ++a) {
        for (int b = 0; b < 1000; ++b) {
            assert(a != 1000-1 || b !=
↪  1000-1);
        }
}
```

(a) `simple_nested`

```
int i = 0;

while (i < 1000) {
        i++;
        assert(i < 1000);
}
```

(b) `assert_loop`

Fig. 8: Simple programs with deep bugs.

nal's execution time is not affected. In all cases (including shallow bugs, e.g., < 10 iterations) we observe that Banal is much faster than the other tools. Interestingly, also Veriabs can scale thanks to *loop summarization* [48] techniques allowing it to replace loops with expressions summarizing their effect. However these techniques only work with special loop structures. To exhibit this, we added two synthetic benchmarks `simple_nested` and `assert_loop` (see Fig 8a, 8b) for which the summarization fails (thus forcing Veriabs to unroll the loop) but Banal succeeds.

5 Conclusion

In this article, we built on top of the preliminary work of [38], studying how to improve it to construct a more effective analysis. It supports more varied and realistic semantics (such as wrap-around) as well as classic abstract domain constructions (such as powerset domains, improved widenings, etc.), to the point where it can provide encouraging results on realistic analysis problems. Our implementation targeted C programs, but the semantics are agnostic with respect to the language and can used to analyze any language with machine integers data types.

Future work. Although this work displays promising results, much work is still needed to analyze real-world programs. Firstly, we believe that more precise abstractions are needed to analyze numeric properties (e.g., domains for constants, congruences, bit-wise operations). The semantic should be extended to handle more features of the C language (e.g., memory allocation, arrays, structs). We do support non-linear integer arithmetic, thus adding support for floating point arithmetic should not be a too large effort. Moreover, we proposed an abstraction modeling streams as returning always the same value: this may suffice in loop-free programs (as each stream is read only once), but can be imprecise in other cases. Whereas the polyhedra domain (and even its power-set) is precise, it comes with a significant computational cost. We believe that more lightweight domains (like intervals [16] or octagons [36]) and packing techniques will play a crucial role in making this analysis more scalable to real world programs.

Acknowledgments This work was supported by the SECURVAL project. The SECUREVAL project was funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-22-PECY-0005.

6 Data Availability Statement

All the software used for the experimental part of this work was released in an artifact [35]. It includes not only the source code of the Banal static analyzer, but also the benchmarks and scripts used to produce the Tables 1, 2.

References

1. The banal static analyzer prototype. http://www.di.ens.fr/~mine/banal, accessed: 2023-08-11
2. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: Veriabs: Verification by abstraction and test generation. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1138–1141. IEEE (2019)
3. Ascari, F., Bruni, R., Gori, R.: Limits and difficulties in the design of under-approximation abstract domains. In: International Conference on Foundations of Software Science and Computation Structures. pp. 21–39. Springer International Publishing Cham (2022)
4. Bagnara, R., Hill, P.M., Zaffanella, E.: Exact join detection for convex polyhedra and other numerical abstractions. Computational Geometry **43**(5), 453–473 (2010)
5. Baldoni, R., Coppa, E., Delia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)
6. Bemporad, A., Fukuda, K., Torrisi, F.D.: Convexity recognition of the union of polyhedra. Computational Geometry **18**(3), 141–154 (2001)
7. Beyer, D.: Competition on software verification and witness validation: Sv-comp 2023. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 495–522. Springer (2023)
8. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 721–733 (2015)
9. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: International Conference on Tests and Proofs. pp. 3–23. Springer (2018)
10. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. pp. 184–190. Springer (2011)
11. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer **21**, 1–29 (2019)
12. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation. pp. 46–55 (1993)

13. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) **50**(5), 752–794 (2003)
14. Colóon, M.A., Sipma, H.B.: Synthesis of linear ranking functions. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 67–81. Springer (2001)
15. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety: (tool paper). In: Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18. pp. 415–418. Springer (2006)
16. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252 (1977)
17. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 269–282 (1979)
18. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. The Journal of Logic Programming **13**(2-3), 103–179 (1992)
19. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Automated Software Engineering **6**(1), 69–95 (1999)
20. Cousot, P., Cousot, R., Logozzo, F.: Precondition inference from intermittent assertions and application to contracts on collections. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 150–168. Springer (2011)
21. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 84–96 (1978)
22. Dangl, M., Löwe, S., Wendler, P.: Cpachecker with support for recursive programs and floating-point arithmetic: (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21. pp. 423–425. Springer (2015)
23. Darke, P., Agrawal, S., Venkatesh, R.: Veriabs: A tool for scalable verification by abstraction (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27. pp. 458–462. Springer (2021)
24. De Vries, E., Koutavas, V.: Reverse hoare logic. In: International Conference on Software Engineering and Formal Methods. pp. 155–171. Springer (2011)
25. Delmas, D., Miné, A.: Analysis of software patches using numerical abstract interpretation. In: Static Analysis: 26th International Symposium, SAS 2019, Porto, Portugal, October 8–11, 2019, Proceedings 26. pp. 225–246. Springer (2019)
26. Filé, G., Ranzato, F.: The powerset operator on abstract interpretations. Theoretical Computer Science **222**(1-2), 77–111 (1999)
27. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Interval analysis and machine arithmetic: Why signedness ignorance is bliss. ACM Transactions on Programming Languages and Systems (TOPLAS) **37**(1), 1–35 (2015)
28. Gopan, D., Reps, T.: Lookahead widening. In: International Conference on Computer Aided Verification. pp. 452–466. Springer (2006)

29. Gotlieb, A., Leconte, M., Marre, B.: Constraint solving on modular integers. In: ModRef Worksop, associated to CP'2010 (2010)
30. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25. pp. 36–52. Springer (2013)
31. Journault, M., Miné, A., Ouadjaout, A.: An abstract domain for trees with numeric relations. In: European Symposium on Programming. pp. 724–751. Springer (2019)
32. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976)
33. Le, Q.L., Raad, A., Villard, J., Berdine, J., Dreyer, D., O'Hearn, P.W.: Finding real bugs in big programs with incorrectness logic. Proceedings of the ACM on Programming Languages **6**(OOPSLA1), 1–27 (2022)
34. Lev-Ami, T., Sagiv, M., Reps, T., Gulwani, S.: Backward analysis for inferring quantified preconditions. Tr-2007-12-01, Tel Aviv University (2007)
35. Marco, M., Miné, A.: Artifact of paper: "Generation of Violation Witnesses by Under-Approximating Abstract Interpretation" (Oct 2023). https://doi.org/10.5281/zenodo.8399723
36. Miné, A.: The octagon abstract domain. Higher-order and symbolic computation **19**, 31–100 (2006)
37. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 348–363. Springer (2006)
38. Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Science of Computer Programming **93**, 154–182 (2014)
39. Miné, A., et al.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends® in Programming Languages **4**(3-4), 120–372 (2017)
40. Moy, Y.: Sufficient preconditions for modular assertion checking. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 188–202. Springer (2008)
41. O'Hearn, P.W.: Incorrectness logic. Proceedings of the ACM on Programming Languages **4**(POPL), 1–32 (2019)
42. Raad, A., Berdine, J., Dang, H.H., Dreyer, D., OHearn, P., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32. pp. 225–252. Springer (2020)
43. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. IEEE (2002)
44. Schmidt, D.A.: A calculus of logical relations for over-and underapproximating static analyses. Science of Computer Programming **64**(1), 29–53 (2007)
45. Sen, R., Srikant, Y.: Executable analysis using abstract interpretation with circular linear progressions. In: 2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007). pp. 39–48. IEEE (2007)
46. Simon, A., King, A.: Taming the wrapping of integer arithmetic. In: International Static Analysis Symposium. pp. 121–136. Springer (2007)
47. Urban, C., Ueltschi, S., Müller, P.: Abstract interpretation of ctl properties. In: Static Analysis: 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings 25. pp. 402–422. Springer (2018)

48. Xie, X., Chen, B., Liu, Y., Le, W., Li, X.: Proteus: Computing disjunctive loop summary via path dependency analysis. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 61–72 (2016)