Error Invariants for Fault Localization via Abstract Interpretation

Aleksandar S. Dimovski[1] [0000−0002−3601−2631]

Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia
aleksandar.dimovski@unt.edu.mk https://aleksdimovski.github.io/

Abstract. *Fault localization* aims to automatically identify the cause of an error in a program by localizing the error to a relatively small part of the program. In this paper, we present a novel technique for automated fault localization via *error invariants* inferred by abstract interpretation. An error invariant for a location in an error program over-approximates the reachable states at the given location that may produce the error, if the execution of the program is continued from that location. Error invariants can be used for *statement-wise semantic slicing* of error programs and for obtaining concise error explanations. We use an iterative refinement sequence of backward-forward static analyses by abstract interpretation to compute error invariants, which are designed to explain why an error program violates a particular assertion. We demonstrate the effectiveness of our approach to localize errors in realistic C programs.

Keywords: Fault localization · Error invariants · Abstract interpretation · Statement-wise semantic slicing.

1 Introduction

Static program analyzers [6,8,22,27,36] are today often applied to find errors in real-world programs. They usually return an error report, which shows how an assertion can be violated. However, the programmers still need to process the error report, in order to isolate the cause of an error to a manageable number of statements and variables that are relevant for the error. Using this information, they can subsequently repair the given program either manually or automatically by running specialized program repair tools [31,33].

In this paper, we present a novel technique for *automated fault localization*, which automatically generates concise error explanations in the form of statements

relevant for a given error that describe the essence of why the error occurred. In particular, we describe a fault localization technique based on so-called *error invariants* inferred via abstract interpretation. An error invariant for a given location in a program captures states that may produce the error, that is, there may be executions of the program continued from that location violating a given assertion. We observe that the same error invariants that hold for consecutive locations characterize statements in the program that are irrelevant for the error. A statement that is enclosed by the same error invariant does not change the nature of the error. Hence, error invariants can be used to find only relevant statements and information about reachable states that helps to explain the cause of an error. They also identify the relevant variables whose values should be tracked when executing the program. The obtained relevant statements constitute the so-called *statement-wise semantic slicing* of the error program, which can be changed (repaired) to make the entire program correct.

Abstract interpretation [7,29] is a general theory for approximating the semantics of programs. It has been successfully applied to deriving computable and approximated static analysis that infer dynamic properties of programs, due to its soundness guarantee (all confirmative answers are indeed correct) and scalability (with a good trade-off between precision and cost). In this paper, we focus on applying abstract interpretation to automate fault localization via inferring error invariants. More specifically, we use a combination of backward and forward refining analyses based on abstract interpretation to infer error invariants from an error program. Each next iteration of backward-forward analyses produces more refined error invariants than the previous iteration. Finally, the error invariants found in the last iteration are used to compute a slice of the error program that contains only relevant statements for the error.

The backward (over-approximating) numerical analysis is used for computing the necessary preconditions of violating the target assertion, thus reducing the input space that needs further exploration. Error invariants are constructed by going backwards step-by-step starting at the property violation, i.e. by propagating the negated assertion backwards. The negated assertion represents an error state space. When there is a precision loss caused by merging the branches of an `if` statement, we collect in a set of predicates the branching condition of that conditional. Subsequently, the forward (over-approximating) numerical analysis of a program with reduced abstract sub-input is employed to refine error invariants in all locations, thus also refining (reducing) the error state space. Based on the inferred error invariants, we can find the relevant statements and relevant variables for the assertion violation. Initially, in the first iteration, both analyses are performed on a base abstract domain (e.g., intervals, octagons, polyhedra). In the subsequent iterations, we use the set of predicates generated by the previous backward analysis to design a binary decision diagram (BDD) abstract domain functor, which can express disjunctive properties with respect to the given set of predicates. A decision node in a BDD abstract element stores a predicate, and each leaf node stores an abstract value from a base abstract domain under specific evaluation results of predicates. When the obtained set of predicates as

well as the (abstract) error state sub-space stay the same over two iterations, the iterative process stops and reports the inferred error invariants. Otherwise, the refinement process continues by performing backward-forward analyses on a BDD abstract domain based on the refined set of predicates as well as on a reduced error state sub-space. The BDD abstract domain and the reduced error state sub-space enable our analyses in the subsequent iterations to focus on smaller state sub-spaces, i.e. partitionings of the total state space, in each of which the program may involve fewer disjunctive or non-linear behaviours and thus the analyses may produce more precise results.

We have implemented our abstract interpretation-based approach for fault localization of C programs in a prototype tool. It takes as input a C program with an assertion, and returns a set of statements whose replacement can eliminate the error. The tool uses the numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library [25], and the BDD abstract domains from the BDDAPRON library [24]. BDDAPRON uses any abstract domain from the APRON library for the leaf nodes. The tool also calls the Z3 SMT solver [30] to compute the error invariants from the information inferred via abstract interpretation-based analyses. We discuss a set of C programs from the literature, SV-COMP and TCAS suites that demonstrate the usefulness of our tool.

In summary, this work makes the following contributions:

(1) We define error invariants as abstract representations of the reason why the program may go wrong if it is continued from that location;
(2) We propose an iterative abstract interpretation-based analyses to compute error invariants of a program. They are used to identify statements and program variables that are relevant for the fault in the program;
(3) We implemented the approach in a prototype tool, which uses domains from APRON and BDDAPRON libraries as well as Z3 SMT solver;
(4) We evaluate our approach for fault localization on a set of C benchmarks.

2 Motivating Examples

We demonstrate our technique for fault localization using the illustrative examples in Figs. 1, 2, and 3. The first example, `program1`, in Fig. 1 shows a program code that violates the assertion $(x > y)$ for all values of the parameter x, since $y = x + 1$ holds at the end of the program. A static analysis of this program will establish the assertion violation. However, the static analysis returns a full list of invariants in all locations of the program, including details that are irrelevant for the specific error. Similarly, other verification tools will also report many irrelevant details for the error (e.g. full execution paths).

Our technique works as follows. We begin with the first iteration of the backward-forward analyses. The backward analysis defined over the Polyhedra domain starts with the negated assertion $(x \leq y)$ at loc. ④. By propagating it backwards, it infers the preconditions: $(x \leq y)$ at loc. ③, $(x \leq z)$ at loc. ②, and $\top$ at loc. ①. The subsequent forward analysis starts with invariant $\top$ at loc. ①, and then infers invariants: $(z = x + 1)$ at loc. ②, $(z = x + 1 \wedge y = x + 1)$ at

```
void main(int x){
⊤
① int z := x + 1;
z = x + 1
② int y := z;
y = x + 1
③ z := z+1;
y = x + 1
④ assert (x > y);
```

Fig. 1. program1.

```
void main(int input){
input ≤ 41
① int x := 1;
input ≤ 41
② int y := input − 42;
B ∧ input ≤ 41 ∧ y = input − 42
③ if (y<0) then
B ∧ input ≤ 41 ∧ y = input − 42
④       x := 0;
B ∧ input ≤ 41 ∧ y = input − 42 ∧ x = 0
⑤ else
⊥
⑥ endif
B ∧ input ≤ 41 ∧ y = input − 42 ∧ x = 0
⑦ assert (x>0);
```

Fig. 2. program2 ($B \equiv (y<0)$).

```
void main(int n){
n ≥ 11
① int x := 6;
n ≥ 11 ∧ x = 6
② int y := n;
n ≥ 11 ∧ x = 6
③ while (x < n) do
n ≥ 11 ∧ n ≥ x + 1
④       x := x + 1;
n ≥ 11 ∧ n ≥ x
⑤       y := y + 1;
n ≥ 11 ∧ n ≥ x
⑥ od;
n ≥ 11 ∧ n = x
⑦ assert (x ≤ 10);
```

Fig. 3. program3.

loc. ③, and $(z = x + 2 \wedge y = x + 1)$ at loc. ④. Note that we use boxed code, such as $\boxed{z = x + 1}$, to highlight the inferred error invariants by our technique. The computed error invariants after one iteration of backward-forward analysis, shown in Fig. 1, are: $\boxed{\top}$, $\boxed{z = x + 1}$, $\boxed{y = x + 1}$, $\boxed{y = x + 1}$ in locs. ① to ④, respectively. Note how the results of backward analysis are refined using the forward analysis to compute more precise error invariants. For example, the error invariant at loc. ③, $\boxed{y = x + 1}$, is obtained by refining the backward precondition $(x \leq y)$ using the forward invariant $(z = x+1 \wedge y = x+1)$ at loc. ③. By analyzing the inferred error invariants, we get a set of relevant statements that are potential indicators of the error. Since the error invariants at locs. ③ and ④ are the same, the statement at loc. ③, $z := z + 1$, is dropped from the resulting program slice. That is, the program will remain erroneous even if $z := z + 1$ is removed from the program. Hence, this statement is irrelevant for the error. In effect, the computed program slice of relevant statements for the error consists of statements at locs. ① and ②. A fix of the error program would be to change some of those statements. Error invariants also provide information about which variables are responsible for the error. After executing the statement at loc. ②, $y := z$, we can see from the error invariant $\boxed{y = x + 1}$ that z is no longer relevant and only x and y has to be considered to the end of the program. For the purpose of comparison with the state-of-the-art, we also ran the BugAssist tool [26] on this example. BugAssist returns only the statement at loc. ① as potential bug.

Consider program2 in Fig. 2 taken from [4]. The assertion is violated if main is called with a value less than 42 for the parameter *input*. In this case, the assignment at loc. ④ is executed and the assigned value 0 to x makes the assertion fail. The backward analysis in the first iteration of our procedure starts with the negated assertion $(x \leq 0)$, and it infers that the precondition of then branch is $\top$

and the precondition of `else` branch is $(x \leq 0)$. Their join $\top$ is the precondition before the `if` statement at loc. ③, which is then propagated back to loc. ①. Hence, there is a precision loss in analyzing the `if` statement, so we record the `if` condition $(y < 0)$, denoted as B, in the set of predicates $\mathbb{P}$. As a result of this precision loss, the error invariants inferred after first iteration are: $\boxed{\top}$ for locs. ① - ④, $\boxed{(x = 1)}$ for locs. ⑤, ⑦, and $\boxed{\bot}$ for loc. ⑥. In effect, we would drop the statement at loc. ②, `y :=` $input - 42$, as irrelevant for the error. However, the second iteration is performed on the refined BDD abstract domain defined over the Polyhedra leaf domain and the set $\mathbb{P} = \{B \equiv (y < 0)\}$ of predicates for decision nodes. Thus, we can analyze the `if` statement more precisely and obtain more precise analysis results. From the obtained error invariants, shown in Fig. 2, we can see that statement at loc. ② is now relevant for the error, while statement at loc. ①, $x := 1$, is encompassed with the same invariants, so it can be dropped from the resulting program slice as irrelevant for the error. Consider a variant of `program2`, denoted `program2-a`, where the assertion in loc. ⑦ is changed to $(x \leq 0)$. The single backward analysis infers very imprecise results by reporting that all statements are relevant for the error. However, our approach finds more precise results inferring that the whole `if` statement (locs. ③ - ⑥) is irrelevant for the error since it cannot set x to a positive value that contradicts the assertion. `BugAssist` also gives less precise results. It reports locs. ② and ③ as potential bugs for both `program2` and `program2-a`.

Finally, consider `program3` in Fig. 3. The error due to the violation of the assertion occurs when `main` is called with a value greater than 10 for the parameter n. In this case, at the end of the `while` loop, the value of x becomes equal to n, thus conflicting the given assertion. Our technique for fault localization works as follows. In the first iteration, the backward analysis starts with the invariant $(x \geq 11)$ at the assertion location ⑦. After computing the error invariants at the end of the first backward-forward iteration, we infer the more precise invariant $(n \geq 11 \wedge x = n)$ at loc. ⑦. We also obtain the error invariant $\top$ for locs. ④, ⑤, and ⑥, which would make the body of `while` irrelevant for the error. Since the error state space at loc. ⑦ is refined from $(x \geq 11)$ to $(n \geq 11 \wedge x = n)$ after the first iteration, we continue with the second iteration on the reduced error sub-space. Therefore, the second iteration starts with the invariant $(n \geq 11 \wedge x = n)$ at loc. ⑦. It infers the error invariants shown in Fig. 3. We can see that statements at locs. ② and ⑤ are redundant and can be eliminated as irrelevant. Moreover, the error invariants imply that variables n and x are relevant, while y is completely irrelevant for the assertion violation. On the other hand, `BugAssist` fails to report any potential bug locations for `program3` by default. In particular, `BugAssist` builds on the `CBMC` bounded model checker [6] for construction of the logic formula, which is then analyzed using a `MAX-SAT` solver. Hence, it reasons about loops by unrolling them, making it very sensitive to the degree of unrolling. If the loop is unrolled 15 times, `BugAssist` reports some potential bug locations. Still, it wrongly returns loc. ⑤ as potential bug, and misses to return loc. ① as potential bug.

3 The Language and its Semantics

Syntax. We consider a simple C-like sequential non-deterministic programming language. The program variables *Var* are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. The control locations before and after each statement are associated to unique syntactic labels $l \in \mathbb{L}$.

$$
\begin{array}{l}
s\,(s \in \mathit{Stm}) ::= \mathtt{skip} \mid \mathtt{x}{=}ae \mid s; s \mid \mathtt{if}\,(be)\,\mathtt{then}\,s\,\mathtt{else}\,s \mid \mathtt{while}\,(be)\,\mathtt{do}\,s \mid \mathtt{assert}(be) \\
ae\,(ae \in \mathit{AExp}) ::= n \mid [n, n'] \mid \mathtt{x} \in \mathit{Var} \mid ae \oplus ae, \\
be\,(be \in \mathit{BExp}) ::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be
\end{array}
$$

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, $\mathtt{x}$ over program variables *Var*, and $\oplus \in \{+, -, *, /\}$, $\bowtie \in \{<, \leq, =, \neq\}$. Without loss of generality, we assume that a program is a sequence of statements followed by a single assertion. That is, a program $p \in \mathit{Prog}$ is of the form: ${}^{l_{in}:}s;^{l_{ass}:}\,\mathtt{assert}(be^{ass})$.

Concrete Semantics. A *store* $\sigma : \Sigma = \mathit{Var} \to \mathbb{Z}$ is a mapping from program variables to values. The semantics of arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$ (resp., boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\mathrm{true}, \mathrm{false}\})$) is the set of possible integer (resp., boolean) values for expression ae (resp., be) in a store σ. E.g.,

$$
\begin{array}{l}
[\![n]\!]\sigma = \{n\},\ [\![[n, n']]\!]\sigma = \{n, \ldots, n'\},\ [\![\mathtt{x}]\!]\sigma = \{\sigma(\mathtt{x})\}, \\
[\![ae_0 \bowtie ae_1]\!]\sigma = \{n_0 \bowtie n_1 \mid n_0 \in [\![ae_0]\!]\sigma, n_1 \in [\![ae_1]\!]\sigma\} \\
[\![\neg be]\!]\sigma = \{\neg t \mid t \in [\![be]\!]\}
\end{array}
$$

We define a *necessary precondition* (backward) semantics and an *invariance* (forward) semantics on the complete lattice $\langle \mathbb{L} \mapsto \mathcal{P}(\Sigma), \dot{\subseteq}, \dot{\cup}, \dot{\cap}, \lambda l.\emptyset, \lambda l.\Sigma \rangle$ by induction on the syntax of programs. The dotted operators $\dot{\subseteq}, \dot{\cup}, \dot{\cap}$ defined on $\mathbb{L} \mapsto \mathcal{P}(\Sigma)$ are obtained by point-wise lifting of the corresponding operators $\subseteq, \cup, \cap$ defined on $\mathcal{P}(\Sigma)$. The above semantics work on functions from labels to sets of stores. The necessary precondition semantics backtracks from an user-supplied property to its origin [1], so it associates to each label $l \in \mathbb{L}$ a necessary precondition in the form of a set of possible stores $S \in \mathcal{P}(\Sigma)$ that may lead to the execution of the user-supplied property. The stores resulting from the necessary precondition semantics $\overleftarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ are built backwards: each function $\overleftarrow{[\![s]\!]}$ takes as input a set of stores S at the final label of s and outputs a set of possible stores before s from which stores from S may be reached after executing s. The invariance semantics [7] associates to each label $l \in \mathbb{L}$ an invariant in the form of a set of possible stores $S \in \mathcal{P}(\Sigma)$ that may arise each time the execution reaches the label l from some initial store. The stores resulting from the invariance semantics $\overrightarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ are built forward: each function $\overrightarrow{[\![s]\!]}$ takes as input a set of stores S at the initial label of s and outputs a set of possible stores reached after executing s from S. The complete definitions of functions $\overleftarrow{[\![s]\!]}$ and $\overrightarrow{[\![s]\!]}$ are given in Fig. 4.

In this way, we can collect the set of possible stores denoting necessary preconditions, written $\mathtt{Cond}_{\mathcal{F}}$, and invariants, written $\mathtt{Inv}_{\mathcal{I}}$, at each label $l \in \mathbb{L}$

$$\overleftarrow{[\![\texttt{skip}]\!]}S = S$$
$$\overleftarrow{[\![\texttt{x := } ae]\!]}S = \{\sigma \in \Sigma \mid \exists n \in [\![ae]\!]\sigma, \sigma[\texttt{x} \mapsto n] \in S\}$$
$$\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]}S = \overleftarrow{[\![s_1]\!]}(\overleftarrow{[\![s_2]\!]}S)$$
$$\overleftarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}S = \{\sigma \in \overleftarrow{[\![s_1]\!]}S \mid \text{true} \in [\![be]\!]\sigma\} \cup \{\sigma \in \overleftarrow{[\![s_2]\!]}S \mid \text{false} \in [\![be]\!]\sigma\}$$
$$\overleftarrow{[\![\texttt{while } be \texttt{ do } s]\!]}S = \texttt{lfp } \overleftarrow{\phi}$$
$$\overleftarrow{\phi}(X) = \{\sigma \in S \mid \text{false} \in [\![be]\!]\sigma\} \cup \{\sigma \in \overleftarrow{[\![s]\!]}X \mid \text{true} \in [\![be]\!]\sigma\}$$

$$\overrightarrow{[\![\texttt{skip}]\!]}S = S$$
$$\overrightarrow{[\![\texttt{x := } ae]\!]}S = \{\sigma[\texttt{x} \mapsto n] \mid \sigma \in S, n \in [\![ae]\!]\sigma\}$$
$$\overrightarrow{[\![s_1 \texttt{ ; } s_2]\!]}S = \overrightarrow{[\![s_2]\!]}(\overrightarrow{[\![s_1]\!]}S)$$
$$\overrightarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}S = \overrightarrow{[\![s_1]\!]}\{\sigma \in S \mid \text{true} \in [\![be]\!]\sigma\} \cup \overrightarrow{[\![s_2]\!]}\{\sigma \in S \mid \text{false} \in [\![be]\!]\sigma\}$$
$$\overrightarrow{[\![\texttt{while } be \texttt{ do } s]\!]}S = \{\sigma \in \texttt{lfp } \overrightarrow{\phi} \mid \text{false} \in [\![be]\!]\sigma\}$$
$$\overrightarrow{\phi}(X) = S \cup \overrightarrow{[\![s]\!]}\{\sigma \in X \mid \text{true} \in [\![be]\!]\sigma\}$$

Fig. 4. Necessary precondition (above) and invariance (below) semantics.

of a program ${}^{l_{in}:}s;^{l_{ass}:}\texttt{assert}(be^{ass})$. We assume that at the assertion label l_{ass} the possible stores are $\mathcal{F} \in \mathcal{P}(\Sigma)$, whereas at the initial label l_{in} are $\mathcal{I} \in \mathcal{P}(\Sigma)$. That is, $\texttt{Cond}_{\mathcal{F}}(l_{ass}) = \mathcal{F}$ and $\texttt{Inv}_{\mathcal{I}}(l_{in}) = \mathcal{I}$. For each statement ${}^{l:}s^{l'}$, the set $\texttt{Cond}_{\mathcal{F}}(l) \in \mathcal{P}(\Sigma)$ (resp., $\texttt{Inv}_{\mathcal{I}}(l') \in \mathcal{P}(\Sigma)$) of possible necessary preconditions (resp., invariants) at the initial label l (resp., final label l'), are:

$$\texttt{Cond}_{\mathcal{F}}(l) = \overleftarrow{[\![s]\!]}\ \texttt{Cond}_{\mathcal{F}}(l'), \qquad \texttt{Inv}_{\mathcal{I}}(l') = \overrightarrow{[\![s]\!]}\ \texttt{Inv}_{\mathcal{I}}(l)$$

We now define the error invariant map $\texttt{ErrInv}_{\mathcal{F}} : \mathbb{L} \to \mathcal{P}(\Sigma)$ as follows. Let $\mathcal{F} = \{\sigma \in \Sigma \mid [\![be^{ass}]\!]\sigma = \text{false}\}$ be a set of stores in which $\texttt{assert}(be^{ass})$ is not valid. Given a set of stores $S \in \mathcal{P}(\Sigma)$, we define the set of *unconstrained variables* in S as $UVar_S = \{x \in Var \mid \forall y \in Var \backslash \{x\}, (\exists n \in \mathbb{Z}. \exists \sigma[y \mapsto n] \in S \implies \forall n' \in \mathbb{Z}. \exists \sigma'[x \mapsto n'][y \mapsto n] \in S)\}$. The set of *constrained variables* of S is $CVar_S = Var \backslash UVar_S$. Given a store $\sigma \in \Sigma$, let $\sigma|_{Var'}$ denote the restriction of σ to the sub-domain $Var' \subseteq Var$, such that $\sigma|_{Var'}(x) = \sigma(x)$ for all $x \in Var'$. Given a set of stores $S \in \mathcal{P}(\Sigma)$, define $S|_{Var'} = \{\sigma|_{Var'} \mid \sigma \in S\}$. Then, we define:

$$\texttt{ErrInv}_{\mathcal{F}}(l) = \{\sigma \in \texttt{Cond}_{\mathcal{F}}(l) \mid \sigma|_{CVar_{\texttt{Cond}_{\mathcal{F}}(l)}} \in \texttt{Inv}_{\texttt{Cond}_{\mathcal{F}}(l_{in})}(l)|_{CVar_{\texttt{Cond}_{\mathcal{F}}(l)}}\} \quad (1)$$

That is, the error invariants at label l is the set of necessary preconditions $\texttt{Cond}_{\mathcal{F}}(l)$ restricted with respect to the constrained variables from $CVar_{\texttt{Cond}_{\mathcal{F}}(l)}$ with the values they obtain in the set $\texttt{Inv}_{\texttt{Cond}_{\mathcal{F}}(l_{in})}(l)$ of invariants at l obtained by taking as the initial set of stores $\texttt{Cond}_{\mathcal{F}}(l_{in})$. For example, let $Var = \{x, y, z\}$, $\texttt{Cond}_{\mathcal{F}}(l) = \{\sigma \in \Sigma \mid \sigma(x) = \sigma(y)\}$, and $\texttt{Inv}_{\texttt{Cond}_{\mathcal{F}}(l_{in})}(l) = \{[x \mapsto 2, y \mapsto 2, z \mapsto 3]\}$. Then, we have $CVar_{\texttt{Cond}_{\mathcal{F}}(l)} = \{x, y\}$, $UVar_{\texttt{Cond}_{\mathcal{F}}(l)} = \{z\}$, and $\texttt{ErrInv}_{\mathcal{F}}(l) = \{[x \mapsto 2, y \mapsto 2, z \mapsto n] \mid n \in \mathbb{Z}\}$.

We now show that the error invariants enable us to locate irrelevant statements for the error as ones that do not change the occurrence of the error if they are replaced by `skip`.

Proposition 1. *Let $p[^{l_1:}\_^{l_2}]$ be a program with a missing hole that represents a statement between labels l_1 and l_2, and let $\mathcal{F}$ be a set of final error states. Let $\mathit{ErrInv}_{\mathcal{F}}$ and $\mathit{ErrInv}'_{\mathcal{F}}$ be the error invariants for the programs $p[^{l_1:}s^{l_2}]$ [1] and $p[^{l_1:}\mathtt{skip}^{\,l_2}]$, respectively. If $\mathit{ErrInv}_{\mathcal{F}}(l_1) = \mathit{ErrInv}_{\mathcal{F}}(l_2)$, then $\mathit{ErrInv}_{\mathcal{F}}(l) = \mathit{ErrInv}'_{\mathcal{F}}(l)$ for all $l \in \mathbb{L}$.*

Proof. The proof is by structural induction on statements s that can be inserted in the hole $\_$ in $p[\_]$. We consider the case of assignment $\mathtt{x}{=}ae$. Let $\mathtt{ErrInv}_{\mathcal{F}}$ be the error invariants inferred for $p[^{l_1:}s^{l_2}]$. Since $\mathtt{ErrInv}_{\mathcal{F}}(l_1) = \mathtt{ErrInv}_{\mathcal{F}}(l_2)$, the variable $\mathtt{x}$ must be unconstrained in $\mathtt{Cond}_{\mathcal{F}}(l_1)$ and $\mathtt{Cond}_{\mathcal{F}}(l_2)$ due to the definition of $\mathtt{ErrInv}_{\mathcal{F}}(l)$ (see Eqn. 1). Therefore, the assignment $\mathtt{x}{=}ae$ has no effect on $\mathtt{Cond}_{\mathcal{F}}$ and $\mathtt{ErrInv}_{\mathcal{F}}$ due to the definition of $\overleftarrow{[\![\mathtt{x := } ae]\!]}$ (see Fig. 4). Note that $\overleftarrow{[\![\mathtt{x := } ae]\!]}S = S$ when $\mathtt{x}$ is unconstrained in S. That is, $\mathtt{Cond}_{\mathcal{F}}(l_1) = \mathtt{Cond}_{\mathcal{F}}(l_2)$. Hence, we will obtain the same $\mathtt{Cond}_{\mathcal{F}}$ and $\mathtt{ErrInv}_{\mathcal{F}}$ for $p[^{l_1:}\mathtt{x}{=}ae^{\,l_2}]$ and $p[^{l_1:}\mathtt{skip}^{\,l_2}]$. Similarly, we handle the other cases. □

However, the necessary precondition semantics $\overleftarrow{[\![s]\!]}$ and $\mathtt{Cond}_{\mathcal{F}}$, the invariance semantics $\overrightarrow{[\![s]\!]}$ and $\mathtt{Inv}_{\mathcal{I}}$, as well as the error invariants $\mathtt{ErrInv}_{\mathcal{F}}$ are not computable since our language is Turing complete.

Abstract Semantics. We now present computable abstract analyses that over-approximate the concrete semantics. We consider an abstract domain $(\mathbb{D}, \sqsubseteq_{\mathbb{D}})$, such that there exist a concretization-based abstraction $\langle \mathcal{P}(\Sigma), \subseteq \rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$. [2] We assume that the abstract domain $\mathbb{D}$ is equipped with sound operators for ordering $\sqsubseteq_{\mathbb{D}}$, least upper bound (join) $\sqcup_{\mathbb{D}}$, greatest lower bound (meet) $\sqcap_{\mathbb{D}}$, bottom $\bot_{\mathbb{D}}$, top $\top_{\mathbb{D}}$, widening $\triangledown_{\mathbb{D}}$, and narrowing $\triangle_{\mathbb{D}}$, as well as sound transfer functions for assignments $\mathtt{ASSIGN}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$, tests $\mathtt{FILTER}_{\mathbb{D}} : \mathit{BExp} \times \mathbb{D} \to \mathbb{D}$, and backward assignments $\mathtt{B\text{-}ASSIGN}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$. We let $\mathtt{lfp}^{\#}$ denote an abstract fix-point operator, derived using widening $\triangledown_{\mathbb{D}}$ and narrowing $\triangle_{\mathbb{D}}$, that over-approximates the concrete $\mathtt{lfp}$. Finally, the concrete domain $(\mathbb{L} \to \mathcal{P}(\Sigma), \dot{\subseteq})$ is abstracted using $\langle \mathbb{L} \to \mathcal{P}(\Sigma), \dot{\subseteq} \rangle \xleftarrow{\dot{\gamma}_{\mathbb{D}}} \langle \mathbb{L} \to \mathbb{D}, \dot{\sqsubseteq} \rangle$.

For each statement s, its abstract necessary precondition semantics $\overleftarrow{[\![s]\!]}^{\sharp}$ and its abstract invariance semantics $\overrightarrow{[\![s]\!]}^{\sharp}$ are defined as mappings $\mathbb{D} \mapsto \mathbb{D}$. The complete definitions of functions $\overleftarrow{[\![s]\!]}^{\sharp}$ and $\overrightarrow{[\![s]\!]}^{\sharp}$ are given in Fig. 5. Suppose that at the assertion label l_{ass} the abstract element is $d_{\mathcal{F}} \in \mathbb{D}$, whereas at the initial label l_{in} is $d_{\mathcal{I}} \in \mathbb{D}$. Thus, $\mathtt{Cond}^{\sharp}_{d_{\mathcal{F}}}(l_{ass}) = d_{\mathcal{F}}$ and $\mathtt{Inv}^{\sharp}_{d_{\mathcal{I}}}(l_{in}) = d_{\mathcal{I}}$. For a statement $^{l:}s^{l'}$, abstract element $\mathtt{Cond}^{\sharp}_{d_{\mathcal{F}}}(l)$ (resp., $\mathtt{Inv}^{\sharp}_{d_{\mathcal{I}}}(l')$) of necessary preconditions (resp., invariants) at initial label l (resp., final label l'), are:

$$\mathtt{Cond}^{\sharp}_{d_{\mathcal{F}}}(l) = \overleftarrow{[\![s]\!]}^{\sharp}\ \mathtt{Cond}^{\sharp}_{d_{\mathcal{F}}}(l'), \qquad \mathtt{Inv}^{\sharp}_{d_{\mathcal{I}}}(l') = \overrightarrow{[\![s]\!]}^{\sharp}\ \mathtt{Inv}^{\sharp}_{d_{\mathcal{I}}}(l)$$

The soundness of abstract semantics follows from the soundness of $\mathbb{D}$ [7,29].

[1] $p[^{l_1:}s^{l_2}]$ is a complete program in which statement s is inserted at the place of hole.
[2] Concretization-based abstraction is a relaxation of the known Galois connection abstraction, which is more used in practice (e.g., Polyhedra domain).

$$\overleftarrow{[\![\texttt{skip}]\!]}^\sharp d = d$$
$$\overleftarrow{[\![\texttt{x := } ae]\!]}^\sharp d = \texttt{B-ASSIGN}_{\mathbb{D}}(\texttt{x := } ae, d)$$
$$\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]}^\sharp d = \overleftarrow{[\![s_1]\!]}^\sharp(\overleftarrow{[\![s_2]\!]}^\sharp d)$$
$$\overleftarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}^\sharp d = \texttt{FILTER}_{\mathbb{D}}(be, \overleftarrow{[\![s_1]\!]}^\sharp d) \sqcup_{\mathbb{D}} \texttt{FILTER}_{\mathbb{D}}(\neg be, \overleftarrow{[\![s_2]\!]}^\sharp d)$$
$$\overleftarrow{[\![\texttt{while } be \texttt{ do } s]\!]}^\sharp d = \texttt{lfp}^\sharp \overleftarrow{\phi}^\sharp$$
$$\overleftarrow{\phi}^\sharp(x) = \texttt{FILTER}_{\mathbb{D}}(\neg be, d) \sqcup_{\mathbb{D}} \texttt{FILTER}_{\mathbb{D}}(be, \overleftarrow{[\![s]\!]}^\sharp x)$$

$$\overrightarrow{[\![\texttt{skip}]\!]}^\sharp d = d$$
$$\overrightarrow{[\![\texttt{x := } ae]\!]}^\sharp d = \texttt{ASSIGN}_{\mathbb{D}}(\texttt{x := } ae, d)$$
$$\overrightarrow{[\![s_1 \texttt{ ; } s_2]\!]}^\sharp d = \overrightarrow{[\![s_2]\!]}^\sharp(\overrightarrow{[\![s_1]\!]}^\sharp d)$$
$$\overrightarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}^\sharp d = \overrightarrow{[\![s_1]\!]}^\sharp(\texttt{FILTER}_{\mathbb{D}}(be, d)) \sqcup_{\mathbb{D}} \overrightarrow{[\![s_2]\!]}^\sharp(\texttt{FILTER}_{\mathbb{D}}(\neg be, d))$$
$$\overrightarrow{[\![\texttt{while } be \texttt{ do } s]\!]}^\sharp d = \texttt{FILTER}_{\mathbb{D}}(\neg be, \texttt{lfp}^\sharp \overrightarrow{\phi}^\sharp)$$
$$\overrightarrow{\phi}^\sharp(x) = d \sqcup_{\mathbb{D}} \overrightarrow{[\![s]\!]}^\sharp(\texttt{FILTER}_{\mathbb{D}}(be, x))$$

Fig. 5. Abstract necessary precondition (above) and invariance (below) semantics.

Algorithm 1: $\texttt{AbsAnalysis}(p, d_{err}, \mathbb{D})$

Input: Program p, error state d_{err}, abstract domain $\mathbb{D}$
Output: Error invariants $\texttt{ErrInv}^\sharp_{d_{err}}$, refined error state d'_{err}, predicates set $\mathbb{P}$

1 $\texttt{Cond}^\sharp_{d_{err}}, \mathbb{P} := \overleftarrow{[\![p]\!]}^\sharp\, d_{err}$;
2 $d_{in} := \texttt{Cond}^\sharp_{d_{err}}(l_{in})$;
3 **if** $(d_{in} = \bot_{\mathbb{D}})$ **then** $\{\texttt{ErrInv}^\sharp_{d_{err}} := \texttt{Cond}^\sharp_{d_{err}};\ d'_{err} := \bot_{\mathbb{D}};\ \mathbb{P} := \emptyset\ \}$;
4 **if** $(d_{in} \neq \bot_{\mathbb{D}})$ **then**
5 $\texttt{Inv}^\sharp_{d_{in}} := \overrightarrow{[\![p]\!]}^\sharp\, d_{in}$;
6 $\texttt{ErrInv}^\sharp_{d_{err}} := \textsc{MinSupport}(\texttt{Inv}^\sharp_{d_{in}} \sqcap_{\mathbb{D}} \texttt{Cond}^\sharp_{d_{err}}, \texttt{Cond}^\sharp_{d_{err}})$;
7 $d'_{err} := \texttt{ErrInv}^\sharp_{d_{err}}(l_{ass})$
8 **return** $\texttt{ErrInv}^\sharp_{d_{err}}, d'_{err}, \mathbb{P}$;

Proposition 2. *Let $\mathcal{F} = \gamma_{\mathbb{D}}(d_{\mathcal{F}})$ and $\mathcal{I} = \gamma_{\mathbb{D}}(d_{\mathcal{I}})$. For any $l \in \mathbb{L}$, we have: $\mathit{Cond}_{\mathcal{F}}(l) \subseteq \gamma_{\mathbb{D}}(\mathit{Cond}^\sharp_{d_{\mathcal{F}}}(l))$ and $\mathit{Inv}_{\mathcal{I}}(l) \subseteq \gamma_{\mathbb{D}}(\mathit{Inv}^\sharp_{d_{\mathcal{I}}}(l))$.*

4 Inferring Abstract Error Invariants

In this section, we first present one iteration of the backward-forward analysis for generating abstract error invariants. Then, we introduce our procedure for iterative abstract analysis and the BDD abstract domain functor we use.

4.1 Abstract Analysis

Sound abstract error invariants can be computed automatically by backward abstract interpretation, $\texttt{Cond}^\sharp_d$, and forward abstract interpretation, $\texttt{Inv}^\sharp_d$. Both

analyses are parameterized by an abstract domain $\mathbb{D}$. In this work, we combine backward and forward abstract analyses to generate for each label the constraints, called *(abstract) error invariants*, that describe states which are reachable from the input and may cause the target assertion fail.

The $\texttt{AbsAnalysis}(p, d_{err}, \mathbb{D})$ procedure is given in Algorithm 1. It takes as input a program p, a target abstract error state d_{err}, and a chosen abstract domain $\mathbb{D}$. First, we call a backward abstract analysis $\overleftarrow{[\![p]\!]}^\sharp\, d_{err}$ (Line 1), which computes the necessary preconditions $\texttt{Cond}^\sharp_{d_{err}}$ of program p. Additionally, the backward analysis computes a set of predicates $\mathbb{P}$ by selecting branch conditions of `if` statements, where precision loss is observed. Given a conditional ${}^{l:}\texttt{if}\,(be)\,\texttt{then}\,{}^{l_{tt}:}s_{tt}\,\texttt{else}\,{}^{l_{ff}:}s_{ff}$, the precondition in l is obtained by joining the preconditions found in the `then` l_{tt} and `else` branches l_{ff}. If those preconditions are not equal, that is $\texttt{Cond}^\sharp_{d_{err}}(l_{tt}) \neq \texttt{Cond}^\sharp_{d_{err}}(l_{ff})$, then we collect the corresponding branch condition be in $\mathbb{P}$ since some precision loss occurs. Subsequently, we check the precondition found at the initial label $d_{in} = \texttt{Cond}^\sharp_{d_{err}}(l_{in})$ (Lines 2,3,4). If $d_{in} = \bot_{\mathbb{D}}$ (which means there is no concrete input state that violates the assertion), the assertion must be valid and the procedure terminates with no further computations (Line 3). Otherwise, a forward analysis $\overrightarrow{[\![p]\!]}^\sharp\, d_{in}$ is started to refine the inferred $\texttt{Cond}^\sharp_{d_{err}}(l)$ and the abstract error state d_{err} (Line 5). It takes as input the program p and the input abstract state d_{in}, and computes invariants $\texttt{Inv}^\sharp_{d_{in}}$ in all labels. The (abstract) error invariants map $\texttt{ErrInv}^\sharp_{d_{err}}$ is then generated using $\texttt{Cond}^\sharp_{d_{err}}$ and $\texttt{Inv}^\sharp_{d_{in}}$ as follows (Line 6):

$$\texttt{ErrInv}^\sharp_{d_{err}}(l) = \textsc{MinSupport}(\texttt{Inv}^\sharp_{d_{in}}(l) \sqcap_{\mathbb{D}} \texttt{Cond}^\sharp_{d_{err}}(l), \texttt{Cond}^\sharp_{d_{err}}(l)),\ \text{for } l \in \mathbb{L}$$

where MinSupport is minimal support set. Finally, the procedure returns as outputs $\texttt{ErrInv}^\sharp_{d_{err}}$, refined abstract error state $\texttt{ErrInv}^\sharp_{d_{err}}(l_{ass})$, and set $\mathbb{P}$.

We now show how the *minimal support set* MinSupport is computed [23].

Definition 1. *Let $P = \{p_1, \ldots, p_n\}$ and $p'_1 \wedge \ldots \wedge p'_k$ be linear constraint formulas over program variables, such that $p_1 \wedge \ldots \wedge p_n \models p'_1 \wedge \ldots \wedge p'_k$. A subset $P' \subseteq P$* supports *the inference $\bigwedge_{p_i \in P} p_i \models p'_1 \wedge \ldots \wedge p'_k$ iff $\bigwedge_{p_j \in P'} p_j \models p'_1 \wedge \ldots \wedge p'_k$. A support set P' is minimal iff no proper subset of P' can support the inference.*

For $P \models p'_1 \wedge \ldots \wedge p'_k$, let the $\textsc{MinSupport}(P, p'_1 \wedge \ldots \wedge p'_k)$ denote the set of minimal supporting conjuncts in P that imply $p'_1 \wedge \ldots \wedge p'_k$. An implementation of MinSupport through unsatisfiability cores is available in existing SMT solvers (e.g., Z3 [30]) for many theories such as linear arithmetic. That is, we ask the SMT solver to find the unsatisfiability core of $\bigwedge_{p_i \in P} p_i \wedge \neg(p'_1 \wedge \ldots \wedge p'_k)$ (which is negation of $\bigwedge_{p_i \in P} p_i \implies (p'_1 \wedge \ldots \wedge p'_k)$). The conjuncts in P that are part of this unsatisfiability core represent a minimal support set for $(p'_1 \wedge \ldots \wedge p'_k)$.

Example 1. Suppose that at a given program location the precondition $(x \geq 0)$ is inferred by the backward analysis, while the invariant $(x = 1 \wedge z = y + 1)$ is inferred using the refined forward analysis. The formulas $p_1 : x = 1, p_2 : z = y + 1$ together imply the formula $p' : x \geq 0$. By checking the unsatisfiability core of the

Algorithm 2: $\texttt{IterativeAbsAnalysis}(p, be^{ass}, \mathbb{D})$

Input: Program p, assertion be^{ass}, abstract domain $\mathbb{D}$
Output: Program slice p'

1 $\mathbb{P} := \emptyset$; $\mathbb{P}' := \emptyset$; $d_{err} := \bot_{\mathbb{D}}$; $d'_{err} := \texttt{FILTER}_{\mathbb{D}}(\neg be^{ass}, \top_{\mathbb{D}})$; $\mathbb{A} := \mathbb{D}$;
2 **while** $(\mathbb{P} \neq \mathbb{P}')$ *or* $(d_{err} \neq d'_{err})$ **do**
3 $\quad \mathbb{P} := \mathbb{P}'$; $\quad d_{err} := d'_{err}$;
4 $\quad \texttt{ErrInv}^{\sharp}_{d_{err}}, d'_{err}, \mathbb{P}' := \texttt{AbsAnalysis}(p, d_{err}, \mathbb{D})$;
5 $\quad$ **if** $(d'_{err} = \bot_{\mathbb{D}})$ **then return** `skip` ;
6 $\quad$ **if** $(d'_{err} \neq \bot_{\mathbb{D}})$ **then** $\mathbb{D} := \mathbb{BD}(\mathbb{P}', \mathbb{A})$;
7 $\quad$ **if** $(Timeout)$ **then break** ;
8 **return** $Slice(p, \texttt{ErrInv}^{\sharp}_{d_{err}})$;

formula $p_1 \wedge p_2 \wedge \neg p'$, we can find that the subset $\{p_1\}$ suffices to establish p', and thus $\{p_1 : x = 1\}$ represents a minimal support set. □

We assume that the elements of the abstract domain are finite conjunctions of linear constraints over program variables. The application of MinSupport removes the redundant conjuncts from the invariants in $\texttt{Inv}^{\sharp}_{d_{in}}(l) \sqcap_{\mathbb{D}} \texttt{Cond}^{\sharp}_{d_{err}}(l)$. By using Proposition 2 and definition of $\texttt{ErrInv}^{\sharp}_{d_{err}}$, we can show the following.

Proposition 3. *Let* $\mathcal{F} = \gamma_{\mathbb{D}}(d_{\mathcal{F}})$. *For any* $l \in \mathbb{L}$, $\mathit{ErrInv}_{\mathcal{F}}(l) \subseteq \gamma_{\mathbb{D}}(\mathit{ErrInv}^{\sharp}_{d_{\mathcal{F}}}(l))$.

4.2 Iterative Abstract Analysis

The $\texttt{AbsAnalysis}(p, d_{err}, \mathbb{D})$ procedure may produce very imprecise (abstract) error invariants due to the over-approximation. One of the major sources of imprecision is that the most commonly used base abstract domains $\mathbb{D}$ (intervals, octagons, polyhedra) have limitations in expressing disjunctive and non-linear properties, which are common in programs. To address these issues, we propose an iterative abstract analysis, wherein the refinement process makes use of the predicates $\mathbb{P}$ inferred at the joins of `if` statements as well as the reduced abstract error sub-space d_{err}. In particular, we use a BDD abstract domain functor, denoted as $\mathbb{BD}(\mathbb{P}, \mathbb{A})$, which can characterize disjunctions of elements from domain $\mathbb{A}$. A decision node in the BDD abstract domain stores a predicate from $\mathbb{P}$, and a leaf node stores an abstract element from the base abstract domain $\mathbb{A}$ under specific evaluation results of predicates found in decision nodes up to the given leaf. We refer to Section 4.3 for detailed description of the BDD domain.

The overall $\texttt{IterativeAbsAnalysis}(p, be^{ass}, \mathbb{D})$ procedure is shown in Algorithm 2. The procedure is called with the following parameters: a program p, an assertion be^{ass} to be checked, and a base abstract domain $\mathbb{D}$. Initially, we call $\texttt{AbsAnalysis}(p, \texttt{FILTER}_{\mathbb{D}}(\neg be^{ass}, \top_{\mathbb{D}}), \mathbb{D})$, given in Algorithm 1, with the negated assertion $\neg be^{ass}$ as error state space d_{err} in order to infer error invariants $\texttt{ErrInv}^{\sharp}_{d_{err}}$, a refined abstract error sub-space d'_{err}, and predicate set $\mathbb{P}'$ (Line 4). If the refinement process is enabled, that is, the newly obtained $\mathbb{P}'$ and d'_{err}

$$\texttt{Slice}(\texttt{skip}, \texttt{ErrInv}^\sharp_{d_{err}}) = \texttt{skip}$$

$$\texttt{Slice}(\texttt{x := } ae, \texttt{ErrInv}^\sharp_{d_{err}}) = \begin{cases} \texttt{skip}, & \text{if } \texttt{ErrInv}^\sharp_{d_{err}}(l) = \texttt{ErrInv}^\sharp_{d_{err}}(l') \\ \texttt{x := } ae, & \text{otherwise} \end{cases}$$

$$\texttt{Slice}(s_1 \texttt{ ; } s_2, \texttt{ErrInv}^\sharp_{d_{err}}) = \begin{cases} \texttt{skip}, & \text{if } \texttt{ErrInv}^\sharp_{d_{err}}(l) = \texttt{ErrInv}^\sharp_{d_{err}}(l') \\ \texttt{Slice}(s_1, \texttt{ErrInv}^\sharp_{d_{err}})\texttt{;}\texttt{Slice}(s_2, \texttt{ErrInv}^\sharp_{d_{err}}), & \text{otherwise} \end{cases}$$

$$\texttt{Slice}(\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2, \texttt{ErrInv}^\sharp_{d_{err}}) =$$
$$\begin{cases} \texttt{skip}, & \text{if } \texttt{ErrInv}^\sharp_{d_{err}}(l) = \texttt{ErrInv}^\sharp_{d_{err}}(l') \\ \texttt{if } be \texttt{ then } \texttt{Slice}(s_1, \texttt{ErrInv}^\sharp_{d_{err}}) \texttt{ else } \texttt{Slice}(s_2, \texttt{ErrInv}^\sharp_{d_{err}}), & \text{otherwise} \end{cases}$$

$$\texttt{Slice}(\texttt{while } (be) \texttt{ do } s, \texttt{ErrInv}^\sharp_{d_{err}}) = \begin{cases} \texttt{skip}, & \text{if } \texttt{ErrInv}^\sharp_{d_{err}}(l) = \texttt{ErrInv}^\sharp_{d_{err}}(l') \\ \texttt{while } (be) \texttt{ do } \texttt{Slice}(s, \texttt{ErrInv}^\sharp_{d_{err}}), & \text{otherwise} \end{cases}$$

Fig. 6. Definition of $\texttt{Slice}(s, \texttt{ErrInv}^\sharp_{d_{err}})$, where "$l : s; l'$" is a statement in program whose error invariants map is $\texttt{ErrInv}^\sharp_{d_{err}}$.

are not the same as $\mathbb{P}$ and d_{err} from the previous iteration (Line 2), the call to `AbsAnalysis` is repeated again with refined parameters d'_{err} and $\mathbb{BD}(\mathbb{P}', \mathbb{A})$ (Line 6), where $\mathbb{A}$ is the input base domain $\mathbb{D}$. Note that, if $\mathbb{P}' = \emptyset$ then $\mathbb{BD}(\mathbb{P}', \mathbb{A})$ is simply $\mathbb{A}$. The procedure terminates when either the refinement is no longer enabled (Line 2), or the assertion is proved true when $d'_{err} = \bot_{\mathbb{D}}$ (which means there is no concrete error state, so we return the program slice "`skip`" since no statement is relevant for the error) (Line 5), or a time limit is reached (Line 7). The procedure $\texttt{Slice}(p, \texttt{ErrInv}^\sharp_{d_{err}})$ (Line 8) returns a slice of program p containing only the statements *relevant* for the assertion failure. Given a statement ${}^{l:}s^{\,l'}$, `Slice` replaces statement s with `skip` if $\texttt{ErrInv}^\sharp_{d_{err}}(l) = \texttt{ErrInv}^\sharp_{d_{err}}(l')$. In this case, we say s is *irrelevant* for the error. That is, the statements for which we can find an encompassing error invariant are not needed to reproduce the error and can be dropped. Otherwise, `Slice` recursively pre-process all sub-statements of compound statements or returns basic statements. The complete definition of $\texttt{Slice}(s, \texttt{ErrInv}^\sharp_{d_{err}})$ is given in Fig. 6.

4.3 BDD Abstract Domain Functor

The binary decision diagram (BDD) abstract domain functor, denoted $\mathbb{BD}(\mathbb{P}, \mathbb{A})$, plays an important role in the iterative abstract analysis procedure. The abstract elements of the domain $\mathbb{BD}(\mathbb{P}, \mathbb{A})$ are disjunctions of leaf nodes that belong to an existing base abstract domain $\mathbb{A}$, which are separated by the values of Boolean predicates from the set $\mathbb{P}$ organized in decision nodes. Therefore, the state space $\mathcal{P}(\Sigma)$ is partitioned with respect to the set of predicates $\mathbb{P}$, such that each top-down path of a BDD abstract element represents one or several partitionings of $\mathcal{P}(\Sigma)$, and we store in the leaf node the property inferred for those partitionings.

We first consider a simpler form of binary decision diagrams called *binary decision trees* (BDTs). A *binary decision tree* (BDT) $t \in \mathbb{BT}(\mathbb{P}, \mathbb{A})$ over the set $\mathbb{P}$ of predicates and the leaf abstract domain $\mathbb{A}$ is either a leaf node $\ll a \gg$ with $a \in \mathbb{A}$

and $\mathbb{P} = \emptyset$, or $[\![P : tl, tr]\!]$, where P is *the smallest element* of $\mathbb{P}$ with respect to its ordering, tl is the left subtree of t representing its true branch, and tr is the right subtree of t representing its false branch, such that $tl, tr \in \mathbb{BT}(\mathbb{P}\backslash\{P\}, \mathbb{A})$. Note that, $\mathbb{P} = \{P_1, \ldots, P_n\}$ is a totally ordered set with ordering: $P_1 < \ldots < P_n$.

However, BDTs contain some redundancy. There are three optimizations we can apply to BDTs in order to reduce their representation [2]: (1) Removal of duplicate leaves; (2) Removal of redundant tests; (3) Removal of duplicate non-leaves. If we apply reductions (**1**)-(**3**) to a BDT $t \in \mathbb{BT}(\mathbb{P}, \mathbb{A})$ until no further reductions are possible, and moreover if the ordering on the Boolean predicates from $\mathbb{P}$ occurring on any path is fixed to the ordered list $[P_1, \ldots, P_n]$, then we obtain a *reduced ordered binary decision diagram* (or only BDD for short) $b \in \mathbb{BD}(\mathbb{P}, \mathbb{A})$. Notice that BDDs have a canonical form, so any disjunctive property from the BDT domain can be represented in an unique way by a BDD.

Given a set of predicates $\mathbb{P}$, an evaluation for $\mathbb{P}$ is a function $\mu : \mathbb{P} \to \{\text{true}, \text{false}\}$. $\texttt{Eval}(\mathbb{P})$ denotes the set of all evaluations for $\mathbb{P}$. Each evaluation $\mu \in \texttt{Eval}(\mathbb{P})$ can be represented as a formula $\bigwedge_{P \in \mathbb{P}} \nu(P)$, where $\nu(P) = P$ if $\mu(P) = \text{true}$ and $\nu(P) = \neg P$ if $\mu(P) = \text{false}$. Given a BDD $b \in \mathbb{BD}(\mathbb{P}, \mathbb{A})$, the concretization function $\gamma_{\mathbb{BD}}$ returns $\gamma_{\mathbb{A}}(a)$ for $\mu \in \texttt{Eval}(\mathbb{P})$, where μ satisfies the constraints reached along the top-down path to the leaf node $a \in \mathbb{A}$.

The abstract operations, transfer functions, and soundness of the domain $\mathbb{BD}(\mathbb{P}, \mathbb{A})$ are obtained by lifting the corresponding operations, transfer functions, and soundness of the leaf domain $\mathbb{A}$. We refer to [3,35,11] for more details. However, the assignment transfer function needs more care, since its application on a leaf node in one partitioning (i.e., one evaluation of $\mathbb{P}$) may cause its result to enter other partitionings. In such a case, the result in each partitioning is updated to be the join of all elements which belong to that partitioning after applying the transfer function to all leaf nodes of the current BDD. This procedure is known as *reconstruction on leaves* [3].

Example 2. Suppose we have a BDD $b = [\![(\texttt{x} \leq 0) : \ll \texttt{x} = 0 \gg, \ll 1 \leq \texttt{x} \leq 10 \gg]\!]$ and an assignment `x := x-1`. Note that the left leaf $\ll \texttt{x} = 0 \gg$ satisfies the decision node $(\texttt{x} \leq 0)$, while the right leaf $\ll 1 \leq \texttt{x} \leq 10 \gg$ satisfies its negation. After performing the (forward) assignment transfer function without reconstruction on leaves, we obtain: $[\![(\texttt{x} \leq 0) : \ll \texttt{x} = -1 \gg, \ll 0 \leq \texttt{x} \leq 9 \gg]\!]$. Hence, the right leaf node $(0 \leq \texttt{x} \leq 9)$ does not satisfy the predicate leading to it: $\neg(\texttt{x} \leq 0)$. However, after the reconstruction on leaves, we obtain: $[\![(\texttt{x} \leq 0) : \ll -1 \leq \texttt{x} \leq 0 \gg, \ll 1 \leq \texttt{x} \leq 9 \gg]\!]$.

5 Evaluation

We have implemented a prototype tool based on our approach for fault localization via inferring error invariants. We now evaluate our tool.

Implementation and Experimental setup. Our tool is based on the APRON library [25], which includes the abstract domains of intervals, octagons, and polyhedra, and the BDDAPRON library [24]. It also calls the Z3 SMT solver

[30] to compute minimal support sets. The tool is written in OCaml and consists of around 7K LOC. It supports a subset of the C language. The current tool provides a limited support for arrays, pointers, `struct` and `union` types.

For the aim of evaluation, we ran: (1) our tool, denoted `Full_AI`; (2) a single backward analysis, denoted `Single_AI`; and (3) the logic formula-based fault localization tool `BugAssist` [26]. [3] Given an error program, `BugAssist` uses the `CBMC` bounded model checker [6] to generate an error trace as well as to construct the corresponding trace formula, which is then analyzed by a `MAX-SAT` solver. We use a set of numerical benchmarks taken from the literature [4], different folders of SV-COMP (https://sv-comp.sosy-lab.org/) and TCAS [21].

Experiments are executed on 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM LUbuntu 20.10, with 8 GB memory, and we use a timeout value of 300 seconds. All times are reported as average over five independent executions. We report total times, measured via `real` values of the `time` command, needed for the actual tasks to be performed. For all three approaches, this includes times to parse the program, to check the assertion violation of the given program, and to identify potential error locations. The implementation is available from [14]: https://doi.org/10.5281/zenodo.8167960.

SV-COMP examples. Consider the program in Fig. 7. It represents a suitably adjusted `easy2-1` example from SV-COMP, where the `if` statement is nested inside the `while`. In the first iteration of `Full_AI`, the `if` condition $(x < 3)$ is added to the set of predicates $\mathbb{P}$, due to the analysis imprecision of the `if` statement. Hence, in the following iterations we use the BDD domain based on the predicate set $\mathbb{P} = \{B \equiv (x < 3)\}$. The inferred error invariants by `Full_AI` are shown in Fig. 7. After calling the slicing procedure `Slice`, we see that statements at locs. ③ and ⑥ are redundant, and so can be dropped. E.g., the statement at loc. ⑥ is enclosed by the error invariant $\boxed{\neg B \wedge y \geq 0}$. The computed error invariants further highlight the information about the state that is essential for the error at each location, thus indicating that variable z is completely irrelevant.

If we analyze this program using single backward analysis `Single_AI`, we do not consider separately the `then` and `else` branches of the `if` statement as in `Full_AI`. Thus, we obtain very imprecise analysis results: $\top$ for all locations inside `while` and loc. ①, as well as $(x \geq 1)$ for all other locations. In fact, we would consider as relevant only statement at loc. ① and `while` condition at loc. ④, whereas all other locations would be irrelevant. This way, we would drop statements at loc. ②, ⑤, ⑦, due to the over-approximation of abstraction, although they are relevant for the error. We thus obtain 53% precision for `Single_AI` (see Table 1, column `Prec%`).

Consider the program `Mysore-1` from SV-COMP given in Fig. 8. The inferred error invariants by `Full_AI` are shown in Fig. 8. We can see that the statements in the body of `while` are redundant and so can be eliminated from the generated program slice. On the other hand, if we use a single backward analysis `Single_AI`,

[3] The other known logic formula-based tool [20,4] is not available online.

```
void main(int n) {
⊤
① int x := 7;
¬B ∧ x = 7
② int y := 0;
¬B ∧ x = 7 ∧ y = 0
③ int z := y;
¬B ∧ x = 7 ∧ y = 0
④ while (x>0) do
(¬B ∧ x ≥ 3 ∧ y ≥ 0) ∨ (B ∧ 1 ≤ x ≤ 2 ∧ x + y ≥ 2)
⑤   if (x<3) then y := y+1;
⑥   else z := z-1;
⑦   x := x − 1;
(¬B ∧ x ≥ 3 ∧ y ≥ 0) ∨ (B ∧ 0 ≤ x ≤ 2 ∧ x + y ≥ 2)
⑧ od;
(B ∧ x = 0 ∧ y ≥ 2)
⑨ assert (y ≤ 0);
```

Fig. 7. `easy2-1` example.

```
void main(int x) {
(x ≤ −1)
① int c := 0;
(x ≤ −1 ∧ c = 0)
② while (x + c ≥ 0) do
⊥
⑤   x := x-c;
⊥
⑥   c := c+1;
⊥
⑧ od;
(c = 0)
⑨ assert (c > 0);
```

Fig. 8. `Mysore-1` example.

then the loop invariant is $(c \leq 0 \wedge x + c \leq -1)$, and thus no statement is eliminated as irrelevant. As a result, `Single_AI` gives 66% precision (see Table 1).

`BugAssist` again reports less precise results by wrongly identifying as irrelevant statements at locs. ①, ② for `easy2-1` and at loc. ① for `Mysore-1`, as well as statement at loc. ⑥ for `easy2-1` as potential bug. Recall that `BugAssist` reasons about loops by unrolling them. Thus, it needs 10 unrollings of the loop for `easy2-1`. The precision of `BugAssist` is 66% for `easy2-1` and 83% for `Mysore-1`.

TCAS example. The final example is an error implementation of the Traffic Alert and Collision Avoidance System (TCAS) [21], which represents an aircraft collision detection system used by all US commercial aircrafts. An extract from the error implementation is shown in Fig. 9. The error in this TCAS implementation is caused by a wrong comparison in the function `Non_Crossing_Biased_Climb`(). On some inputs, this error causes the variable `need_upward_RA` to become 1. The effect is that the assertion will get violated. Note that the strict inequality '>' in ¬(`Down_Separation` > `Positive_RA_Alt_Tresh`) from function `Non_Crossing_Biased_Climb`() is problematic, which causes the error. It should be replaced with '≥' for the implementation to be correct.

Our tool first inlines all functions into the `main`() function, which is then analyzed statically. Thus, the complete program has 308 locations in total, and the `main`() function after inlinement contains 118 locations. `Full_AI` needs two

```
int Non_Crossing_Biased_Climb() {
 ⋮
if (own_below_threat=0)∨((own_below_threat=1)∧¬(Down_Separation>Positive_RA_Alt_Tresh))
then result := 1;
else result := 0;
 ⋮
return result;
}

int alt_sep_test() {
if (High_Confidence=1)∧(Own_Tracked_Alt_Rate ≤ OLEV)∧(Cur_Vertical_Sep>MAXALTDIFF)
then enabled := 1;
else enabled := 0;
 ⋮
if (enabled=1)∧(((tcas_equipped=1)∧(intent_not_known=1))∨(tcas_equipped=0))
then {
  if (Non_Crossing_Biased_Climb() = 1)∧(Own_Below_Threat() = 1)
    then need_upward_RA := 1;
    else need_upward_RA := 0;
    ⋮
}
assert (need_upward_RA=0)∨(Down_Separation ≥ Positive_RA_Alt_Tresh);
}

void main() {
 ⋮
int High_Confidence := [0, 1];
int Other_Tracked_Alt := ?;
 ⋮
int Down_Separation := ?;
int Other_RAC := [0, 1];
int Other_Capability := [0, 1];
int Climb_Inhibit := [0, 1];
int Positive_RA_Alt_Tresh := 740;
 ⋮
int res := alt_sep_test() ;
}
```

Fig. 9. An excerpt from an error TCAS implementation.

iterations to terminate by using the BDD domain with four predicates:

$B1 \equiv (\texttt{need\_upward\_RA=1}) \wedge (\texttt{need\_downward\_RA=1})$
$B2 \equiv (\texttt{Own\_Tracked\_Alt} < \texttt{Other\_Tracked\_Alt})$
$B3 \equiv (\texttt{own\_above\_threat=0}) \wedge (\texttt{Curr\_Vertical\_Sep} \geq \texttt{MINSP}) \wedge$
$\quad (\texttt{Up\_Separation} \geq \texttt{Positive\_RA\_Alt\_Tresh})$
$B4 \equiv (\texttt{own\_below\_threat=0}) \vee$
$\quad (\texttt{own\_below\_threat=1} \wedge \neg(\texttt{Down\_Separation} > \texttt{Positive\_RA\_Alt\_Tresh}))$

The slice computed by `Full_AI` approach contains 44 locations relevant for the error. Some of these statements are shown underlined in Fig. 9. Note that not underlined `else` branches are classified as irrelevant. The reported relevant statements are sufficient to understand the origins of the error. The generated slice depends only on 15 variables instead of 37 variables in the original program. The number of input variables is also reduced from 12 to 6. Thus, we conclude that the obtained slice significantly reduces the search space for error statements.

The single backward analysis `Single_AI` reports a slice containing only 28 locations. However, the slice does not contain any statement from the buggy `Non_Crossing_Biased_Climb`(), thus missing the real reasons for the error. On the other hand, the `BugAssist` tool reports as potential bugs only 2 locations, the condition '`if` (`enabled=1`) . . .' and the assertion, both from `alt_sep_test()` function. Similarly as in the case of `Single_AI`, none of these locations is from the buggy `Non_Crossing_Biased_Climb`().

Performances Table 1 shows the result of running our tool `Full_AI`, the single backward analysis `Single_AI`, and the `BugAssist` tool on the benchmarks considered so far. The column "`LOC#`" is the total number of locations in the program, "`Time`" shows the run-time in seconds, "`Slice#`" is the number of potential (relevant) fault locations, and "`Perc%`" is the percentage precision of the given approach to locate the relevant statements for the error. This is the ratio of the sum of *correctly* classified erroneous and non-erroneous locations by an approach to the total number of locations in the program. A classification of a location as erroneous given by the concrete semantics is considered correct.

We conclude that our technique, `Full_AI`, gives more precise information about potential error statements than simply performing a backward analysis and `BugAssist`. In fact, `Full_AI` pin-pointed the correct error locations for all examples. On average, the number of locations to check for potential error (`Slice#`) is reduced to 47.6% of the total code (`LOC#`). On the other hand, the precision of `Single_AI` is 70% and the precision of `BugAssist` is 64%, on average. Although our technique `Full_AI` is the most precise, it is slower than `Single_AI` due to the several iterations it needs to produce the fully refined error invariants. `Full_AI` and `BugAssist` have often comparable running times, except for the loop benchmarks when `BugAssist` is slower due to the need to unwind the loops. Moreover, `Full_AI` reports more fine-grained information by identifying relevant variables for the error, whereas `BugAssist` reports only potential bug locations. Finally, we should note that the run-time of our technique `Full_AI` in all examples is still significantly smaller than our human effort required to isolate the fault.

6 Related Work

Fault localization has been an active area of research in recent years [4,5,20,26,33]. The most successful approaches for fault localization are based on logic formulae [4,5,20,26,33]. They represent an error trace using an SMT formula and analyze it to find suspicious locations. Hence, they assume the existence of error traces

Table 1. Performance results of `Full_AI` vs. `Single_AI` vs. `BugAssist`. `Full_AI` and `Single_AI` use Polyhedra domain. All times in sec.

Bench.	LOC#	Full_AI			Single_AI			BugAssist		
		Time	Slice#	Perc%	Time	Slice#	Prec%	Time	Slice#	Perc%
program1	6	0.056	2	100%	0.013	2	100%	0.031	1	66%
program2	9	0.187	5	100%	0.013	4	83%	0.031	2	50%
program2-a	9	0.088	2	100%	0.015	6	33%	0.030	2	66%
program3	10	0.453	3	100%	0.016	3	100%	2.954	5	71%
easy2-1	15	2.401	10	100%	0.011	3	41%	8.441	9	66%
Mysore-1	9	0.050	4	100%	0.014	6	66%	0.210	3	83%
TCASv.1	118	57.78	44	100%	0.225	28	86%	0.095	1	62%

on input. The error traces are usually obtained either from failing test cases or from counterexamples produced by external verification tools. In contrast, our approach is directly applied on (error) programs, thus it needs no specific error traces from other tools making it more self-contained. This way, the two phases of error-detection and error-localization are integrated by our approach.

The closest to our approach for inferring error invariants applied to fault localization is the work proposed by Ermis et. al [20,4]. They use Craig interpolants and SMT queries to calculate error invariants in an error trace. Another similar approach that uses error traces and SAT queries is `BugAssist` [26]. It uses MAX-SAT based algorithm to identify a maximal subset of statements from an error trace that are not needed to prove the unsatisfiability of the logical formula representing the error trace. One limitation of `BugAssist` is that control-dependent variables and statements are not considered relevant. Moreover, `BugAssist` do not report error invariants, which can be especially useful for dense errors where the error program cannot be sliced significantly. Hence, `BugAssist` cannot identify relevancy of variables. Other logic formula-based approaches include using weakest preconditions [5], and syntactic information in the form of graphs for static and dynamic dependency relations [33] to localize the errors.

Rival [32] uses abstract interpretation static analyzer ASTREE [9] to investigate the found alarms and to classify them as true errors or false errors. It uses an refining sequence of forward-backward analyses to obtain an approximation of a subset of traces that may lead to an error. Hence, the above work aims to find a set of traces resulting in an error, thus defining so-called trace-wise semantic slicing. In contrast, our approach aims to find statements that are reasons for the error, thus defining the statement-wise semantic slicing. The under-approximated backward analysis proposed by Mine [28] infers sufficient preconditions ensuring that the target property holds for all non-deterministic choices. It would produce the under-approximations of concrete error invariants if applied to our approach. We could then combine the results of under- and over-approximating error invariants, so that if both are the same for some locations we can be certain that the corresponding statements are either error-relevant or error-irrelevant. The

work [19] also uses forward-backward analyses to estimate the probability that a target assertion os satisfied/violated.

Decision-tree domains have been used in abstract interpretation community recently [3,10,35]. Segmented decision tree abstract domains have enabled path dependent static analysis [3,10]. Their elements contain decision nodes that are determined either by values of program variables [10] or by the `if` conditions [3], whereas the leaf nodes are numerical properties. Urban and Miné [35] use decision tree abstract domains to prove program termination. Decision nodes are labelled with linear constraints that split the memory space and leaf nodes contain affine ranking functions for proving termination. Recently, specialized decision tree lifted domains have been proposed to analyze program families (or any other configurable software system) [11,16,12,18]. Decision nodes partition the configuration space of possible feature values (or statically configurable options), while leaf nodes provide analysis information of program variants (family members) corresponding to each partition. The work [11] uses lifted BDD domains to analyze program families with Boolean features. Subsequently, the lifted decision tree domain has been proposed to handle program families with both Boolean and numerical features [18], as well as dynamic program families with features changing during run-time [16].

Once a set of statements relevant for the error has been found, we need to replace those statements in order to fix the error. Recently, abstract interpretation has been successfully applied to program sketching [17,13,15]. The above works leverage a lifted (family-based) static analysis to synthesize program sketches, which represent partial programs with some missing integer holes in them [34]. We can combine our approach for fault localization with the techniques for program sketches to develop an automatic procedure for program repair.

7 Conclusion

In this work, we have proposed error invariants for reasoning about the relevancy of portions of an error program. They provide a semantic argument why certain statements are irrelevant for the cause of an error. We have presented an algorithm that infers error invariants via abstract interpretation and uses them to obtain compact slices of error programs relevant for the error. Our evaluation demonstrates that our algorithm provides useful error explanations.

References

1. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI). pp. 46–55. ACM (1993). https://doi.org/10.1145/155090.155095, https://doi.org/10.1145/155090.155095
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819, https://doi.org/10.1109/TC.1986.1676819

3. Chen, J., Cousot, P.: A binary decision tree abstract domain functor. In: Static Analysis - 22nd International Symposium, SAS 2015, Proceedings. LNCS, vol. 9291, pp. 36–53. Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_3, https://doi.org/10.1007/978-3-662-48288-9_3
4. Christ, J., Ermis, E., Schäf, M., Wies, T.: Flow-sensitive fault localization. In: Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013. Proceedings. LNCS, vol. 7737, pp. 189–208. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_13, https://doi.org/10.1007/978-3-642-35873-9_13
5. Christakis, M., Heizmann, M., Mansur, M.N., Schilling, C., Wüstholz, V.: Semantic fault localization and suspiciousness ranking. In: Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Proceedings, Part I. LNCS, vol. 11427, pp. 226–243. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_13, https://doi.org/10.1007/978-3-030-17462-0_13
6. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings. LNCS, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15, https://doi.org/10.1007/978-3-540-24730-2_15
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Record of the Fourth ACM Symposium on POPL. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, http://doi.acm.org/10.1145/512950.512973
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings. LNCS, vol. 3444, pp. 21–30. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_3, https://doi.org/10.1007/978-3-540-31987-0_3
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does astrée scale up? Formal Methods in System Design **35**(3), 229–264 (2009). https://doi.org/10.1007/s10703-009-0089-6, https://doi.org/10.1007/s10703-009-0089-6
10. Cousot, P., Cousot, R., Mauborgne, L.: A scalable segmented decision tree abstract domain. In: Time for Verification, Essays in Memory of Amir Pnueli. LNCS, vol. 6200, pp. 72–95. Springer (2010). https://doi.org/10.1007/978-3-642-13754-9_5, https://doi.org/10.1007/978-3-642-13754-9_5
11. Dimovski, A.S.: A binary decision diagram lifted domain for analyzing program families. J. Comput. Lang. **63**, 101032 (2021). https://doi.org/10.1016/j.cola.2021.101032, https://doi.org/10.1016/j.cola.2021.101032
12. Dimovski, A.S.: Lifted termination analysis by abstract interpretation and its applications. In: GPCE '21: Concepts and Experiences, Chicago, IL, USA, October, 2021. pp. 96–109. ACM (2021). https://doi.org/10.1145/3486609.3487202, https://doi.org/10.1145/3486609.3487202
13. Dimovski, A.S.: Quantitative program sketching using lifted static analysis. In: Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Proceedings. LNCS, vol. 13241, pp. 102–122. Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_6, https://doi.org/10.1007/978-3-030-99429-7_6

14. Dimovski, A.S.: Artifact for the paper "error invariants for fault localization via abstract interpretation". Zenodo (2023). https://doi.org/10.5281/zenodo.8167960, https://doi.org/10.5281/zenodo.8167960
15. Dimovski, A.S.: Quantitative program sketching using decision tree-based lifted analysis. J. Comput. Lang. **75**, 101206 (2023). https://doi.org/10.1016/j.cola.2023.101206, https://doi.org/10.1016/j.cola.2023.101206
16. Dimovski, A.S., Apel, S.: Lifted static analysis of dynamic program families by abstract interpretation. In: 35th European Conference on Object-Oriented Programming, ECOOP 2021. LIPIcs, vol. 194, pp. 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.14, https://doi.org/10.4230/LIPIcs.ECOOP.2021.14
17. Dimovski, A.S., Apel, S., Legay, A.: Program sketching using lifted analysis for numerical program families. In: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings. LNCS, vol. 12673, pp. 95–112. Springer (2021). https://doi.org/10.1007/978-3-030-76384-8_7, https://doi.org/10.1007/978-3-030-76384-8_7
18. Dimovski, A.S., Apel, S., Legay, A.: Several lifted abstract domains for static analysis of numerical program families. Sci. Comput. Program. **213**, 102725 (2022). https://doi.org/10.1016/j.scico.2021.102725, https://doi.org/10.1016/j.scico.2021.102725
19. Dimovski, A.S., Legay, A.: Computing program reliability using forward-backward precondition analysis and model counting. In: Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Proceedings. LNCS, vol. 12076, pp. 182–202. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_9, https://doi.org/10.1007/978-3-030-45234-6_9
20. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: FM 2012: Formal Methods - 18th International Symposium, 2012. Proceedings. LNCS, vol. 7436, pp. 187–201. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_17, https://doi.org/10.1007/978-3-642-32759-9_17
21. Graves, T.L., Harrold, M.J., Kim, J., Porter, A.A., Rothermel, G.: An empirical study of regression test selection techiques. ACM Trans. Softw. Eng. Methodol. **10**(2), 184–208 (2001). https://doi.org/10.1145/367008.367020, https://doi.org/10.1145/367008.367020
22. Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate taipan: Trace abstraction and abstract interpretation - (competition contribution). In: 23rd International Conference, TACAS 2017, Proceedings, Part II. LNCS, vol. 10206, pp. 399–403 (2017). https://doi.org/10.1007/978-3-662-54580-5_31, https://doi.org/10.1007/978-3-662-54580-5_31
23. Harris, W.R., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 71–82. ACM (2010). https://doi.org/10.1145/1706299.1706309, https://doi.org/10.1145/1706299.1706309
24. Jeannet, B.: Relational interprocedural verification of concurrent programs. In: Seventh IEEE Inter. Conf. on Software Engineering and Formal Methods, SEFM'09. pp. 83–92. IEEE Computer Society (2009). https://doi.org/10.1109/SEFM.2009.29, https://doi.org/10.1109/SEFM.2009.29
25. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceed-

ings. LNCS, vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52, https://doi.org/10.1007/978-3-642-02658-4_52
26. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011. pp. 437–446. ACM (2011). https://doi.org/10.1145/1993498.1993550, https://doi.org/10.1145/1993498.1993550
27. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252, https://doi.org/10.1145/360248.360252
28. Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Sci. Comput. Program. **93**, 154–182 (2014). https://doi.org/10.1016/j.scico.2013.09.014, https://doi.org/10.1016/j.scico.2013.09.014
29. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages **4**(3-4), 120–372 (2017). https://doi.org/10.1561/2500000034, https://doi.org/10.1561/2500000034
30. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24, https://doi.org/10.1007/978-3-540-78800-3_24
31. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: program repair via semantic analysis. In: 35th International Conference on Software Engineering, ICSE '13. pp. 772–781. IEEE Computer Society (2013). https://doi.org/10.1109/ICSE.2013.6606623, https://doi.org/10.1109/ICSE.2013.6606623
32. Rival, X.: Understanding the origin of alarms in astrée. In: Static Analysis, 12th International Symposium, SAS 2005, Proceedings. LNCS, vol. 3672, pp. 303–319. Springer (2005). https://doi.org/10.1007/11547662_21, https://doi.org/10.1007/11547662_21
33. Rothenberg, B., Grumberg, O.: Must fault localization for program repair. In: Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II. LNCS, vol. 12225, pp. 658–680. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_33, https://doi.org/10.1007/978-3-030-53291-8_33
34. Solar-Lezama, A.: Program sketching. STTT **15**(5-6), 475–495 (2013). https://doi.org/10.1007/s10009-012-0249-7, https://doi.org/10.1007/s10009-012-0249-7
35. Urban, C., Miné, A.: A decision tree abstract domain for proving conditional termination. In: Static Analysis - 21st International Symposium, SAS 2014. Proceedings. LNCS, vol. 8723, pp. 302–318. Springer (2014). https://doi.org/10.1007/978-3-319-10936-7_19, https://doi.org/10.1007/978-3-319-10936-7_19
36. Yin, B., Chen, L., Liu, J., Wang, J., Cousot, P.: Verifying numerical programs via iterative abstract testing. In: Static Analysis - 26th International Symposium, SAS 2019, Proceedings. LNCS, vol. 11822, pp. 247–267. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_13, https://doi.org/10.1007/978-3-030-32304-2_13