Journal of Computer Languages 80 (2024) 101288

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Fault localization by abstract interpretation and its applications

Aleksandar S. Dimovski

Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia

ARTICLE INFO

Keywords:
Fault localization
Error invariants
Abstract interpretation
Statement-wise semantic slicing
Mutation-based Program Repair

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.

Furthermore, we present a practical application of the fault localization technique for automatic repair of programs. Given an erroneous program, we first use the fault localization to automatically identify statements relevant for the error, and then repeatedly mutate the expressions in those relevant statements until a correct program that satisfies all assertions is found. All other statements classified by the fault localization as irrelevant for the error are not mutated in the program repair process. This way, we significantly reduce the search space of mutated programs without losing any potentially correct program, and so locate a repaired program much faster than a program repair without fault localization.

We have developed a prototype tool for automatic fault localization and repair of C programs. We demonstrate the effectiveness of our approach to localize errors in realistic C programs, and to subsequently repair them. Moreover, we show that our approach based on combining fault localization and code mutations is significantly faster that the previous program repair approach without fault localization.

1. Introduction

Software programs play an important role in any aspect of our lives today. They are employed in many security and safety-critical systems in industries such as medicine, aeronautics, and automotive. Software errors in such systems have already caused serious consequences, including fatal accidents, shut down of vital systems, and loss of money. As a result, debugging [1] has become one of the most expensive and time consuming tasks of software development. Debugging consists of three steps: detecting the error in a program; localizing the error; and repairing the error. Many static analyzers and verification tools [2–5] 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 and whose change eliminates the error. Using this information, they can subsequently repair the given program either manually or automatically by running specialized program repair tools [6,7]. Therefore, fault localization [7–12] is an important step in debugging, and the automated fault localization can significantly improve manual debugging and the usability of program verification tools.

Some of the most successful formal approaches for fault localization are based on logic formulae [8,10,12]. They represent an error trace using an SMT formula and analyze it to find suspicious locations. Hence, they assume the existence of error traces on input. The error traces are usually obtained either from failing test cases or from counterexamples produced by external verification tools. These formula-based approaches include using error invariants calculated by Craig interpolants and SMT queries [8,10], maximum satisfiability [12], and weakest preconditions [9]. The above approaches reason about loops by unrolling them and so are very sensitive to the degree of unrollment; they can handle only properties expressed as SMT formulae; and the phases of error detection and error localization are completely separated. In order to address these problems of formula-based approaches, we employ abstract interpretation [13,14] to define a fault localization approach possessing the following distinctive characteristics: full automation, support for unbounded loops and infinite number of

E-mail address: aleksandar.dimovski@unt.edu.mk.

https://doi.org/10.1016/j.cola.2024.101288
Received 18 March 2024; Received in revised form 7 July 2024; Accepted 10 July 2024
Available online 17 July 2024

execution paths, parametrization by a choice of an abstract domain, a possibility to scale up to large programs, and a full integration of error detection and error localization since the approach is directly applied to programs and there is no need for error traces extracted from other tools.

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 [13,14] 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 state space at the initial location 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 from the end of the program. The negated assertion represents an error state space at the assertion (final) location. 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 input state space at the initial location is employed to refine error invariants in all locations, thus also refining (reducing) the abstract error state space at the assertion (final) location. 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 at the assertion (final) location. 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 behaviors and thus the analyses may produce more precise results.

Finally, we present an interesting application of our approach for fault localization to automatic program repair. Program repair is defined to be a *code transformation*, where the repaired transformed program satisfies a given specification (e.g., assertion). This represents an important problem since even if an error is identified in the verification phase, the manual repair is a difficult time-consuming task that requires the close knowledge of the program. Automated program repair has gained popularity due to its potential to reduce the manual effort by automatically suggesting repairs for given errors. Recently, several successful tools for automated program repair were proposed [6,15–17] by using various formal techniques ranging from symbolic execution to deductive reasoning. They are extensively examined in software engineering as a way to efficiently maintain software systems. However, the high computational complexity of automatically generated repairs remains a great barrier to the adoption of this technology in practice.

In this paper, we combine our technique for fault localization and the mutation-based program repair [7,16]. Given an erroneous program, our fault localization approach suggests statements in the program that are relevant for the given error, and the mutation-based program repair algorithm [7,16] then attempts to change (mutate) those statements in order to eliminate the error. The mutation-based repair algorithm [7,16] generates mutated programs (mutants) by using a predefined set of syntactic changes (mutations) applied to original program code. Thus, we can apply some syntactic changes to arithmetic operators (e.g., replacing `+` by `−`), Boolean operators (e.g., replacing $\wedge$ by $\vee$), relational operators (e.g., replacing $<$ by $\leq$), etc. The repair algorithm goes through an iterative generate-and-verify procedure, where the generate phase produces a mutant and the verify phase checks if the mutant is correct. The iterative procedure is implemented in an incremental manner, where the learned information in one iteration is re-used in the following one. By applying only mutations to statements relevant for the error as identified by the fault localization, we significantly narrow down the search space of all possible mutants, thus speeding up the original repair algorithm without any precision loss.

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 [18], and the BDD abstract domains from the BDDAPRON library [19]. BDDAPRON uses any abstract domain from the APRON library for the leaf nodes. The tool also calls the Z3 SMT solver [20] to compute the error invariants from the information inferred via abstract interpretation-based analyses. Moreover, we have integrated our fault localization tool with the AllRepair tool [7,16] for repairing C programs. We discuss a set of C programs from the literature, SV-COMP and TCAS suites that demonstrate the usefulness of our tools to discover the precise cause of the error and to repair them.

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 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;

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

Fig. 1. program1.

(3) We show how our fault localization technique can be incorporated into a mutation-based repair algorithm for reducing its mutant search space;

(4) We have implemented the proposed approaches for fault localization and program repairs in prototype tools;

(5) We evaluate our approaches for fault localization and program repairs on a set of C benchmarks.

This work extends and revises the SAS conference article [21]. Additional material that did not appear in the conference version include: (1) an application of our fault localization approach to automatic program repair; (2) proof sketches for the main formal results in the work (Propositions 1 and 5); (3) additional illustrations, explanations, and examples; (4) more evaluation results including experiments for our program repair approach, more benchmarks, and more performance results. The paper proceeds with motivating examples that illustrate our new approach for automated fault localization and its applications to program repair. The programming language, its concrete and abstract (forward and backward) analyses are introduced in Section 3. Section 4 presents our novel algorithms based on iterative abstract analyses for inferring error invariants. In Section 5, we combine our fault localization approach and the mutation-based program repair algorithm. Section 6 describes the implementation and the evaluation of our approaches on benchmarks taken from the literature, SV-COMP and TCAS suites. Finally, we discuss related work and conclude.

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 $(y < n)$ for all values of the parameter n, since $y = n + 1$ 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 fault localization technique, denoted `FaultLoc`, works as follows. We begin with the first iteration of the backward–forward abstract analyses. The backward analysis defined over the Polyhedra domain starts with the negated assertion $(y \geq n)$ at loc. ④. By propagating it backwards, it infers the preconditions: $(y \geq n)$ at loc. ③, $(z \geq n)$ at loc. ②, and $\top$ at loc. ①. The subsequent forward analysis starts with invariant $\top$ at loc. ①, and then infers invariants: $(z = n+1)$ at loc. ②, $(z = n+1 \wedge y = n+1)$ at loc. ③, and $(z = n+2 \wedge y = n+1)$ at loc. ④. Note that we use boxed code, such as $\boxed{z = n+1}$, 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: $\boxed{\top}$, $\boxed{z = n+1}$, $\boxed{y = n+1}$, $\boxed{y = n+1}$ 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, $\boxed{y = n+1}$, is obtained by refining the backward precondition $(y \geq n)$ using the forward invariant $(z = n+1 \wedge y = n+1)$ 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 $\boxed{y = n+1}$ that z is no longer relevant and only n and y has to be considered to the end of the program.

```
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.

Consider `program2` in Fig. 2 taken from [8]. 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 backward–forward 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 `FaultLoc` based on the refined BDD abstract domain 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.

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 state 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.

Finally, we illustrate how the inferred results by our `FaultLoc` approach can be applied to automatic program repair. One of the most successful automatic program repair tools, called `AllRepair` [16], applies a predefined set of syntactic mutations to all program expressions found in statements that can be changed in the program (e.g., assertions cannot be changed), and then explores the search space of all mutants to find a correct one. However, the space of mutants grows very rapidly as the number of statements that can be changed grows. We want to use our approach in order to reduce the search space of mutants without dropping possibly correct programs.

Consider `program1` in Fig. 1. Our approach has identified the statement $z := z{+}1$ at loc. ③ as irrelevant for the error, while the statements $z := n{+}1$ at loc. ① and $y = z$ at loc. ② are relevant for the error. Therefore, the mutation-based program repair tool will mutate only program expressions (which are right-hand sides (RHSs) of assignments) $n{+}1$ and z found in the relevant statements, whereas the program expression $z{+}1$ in the irrelevant statement will not be mutated at all. We use the following pre-defined mutations: a variable/-parameter can be replaced by any other parameter in the program; an arithmetic operator from the set $\{+, -, *, \%, \div\}$ can be replaced by any other operator from the same set; an integer constant can be increased and decreased by one; and a relational operator from the set $\{<, \leq, >, \geq\}$ can be replaced by any other operator from the same set. This way, if we mutate expressions $n{+}1$, z, and $z{+}1$, we would obtain $2^2 * 3^2 * 5^2 = 900$ mutants in the original `AllRepair` tool. However, if we combine fault localization with the `AllRepair`, we will mutate only expressions $n{+}1$ and z, thus reducing the space of mutants to $2 * 3 * 5 = 30$. Note that `AllRepair` does not explore the space of all possible mutants in general. When a possible solution is found, then all supersets of the found solutions (i.e., mutants that contain at least the mutations in the solutions) are not further explored. Thus, the explored search space and the running time have also been reduced from 546 and 3.645 s using `AllRepair` to 24 and 0.005 + 0.367 = 0.372 s using `FaultLoc+AllRepair`. Both approaches generate the same repaired programs, such as the one obtained by replacing statement $z := n{+}1$ at loc. ① with $z := n{-}1$.

Consider `program2-a` in Fig. 2, where the assertion is ($x \leq 0$). We have established that statements at loc. ① and ② are relevant for the error so the corresponding expressions 1 and *input*−42 will be mutated, whereas the `if` statement is irrelevant so it will not be mutated. In effect, the total space of mutants (resp., the explored space of mutants and the running time) is reduced from 3240 (resp., 1536 and 8.187 s) for `AllRepair` to 45 (resp., 31 and 0.022 + 0.474 = 0.496 s) for `FaultLoc+AllRepair`. Both approaches generate the same repaired program, where the statement $x := 1$ at loc. ① is changed to $x := 0$.

Consider `program3` in Fig. 3. Our approach has classified statements at locs. ② and ⑤ as irrelevant for the error, so they will not be mutated by the program repair algorithm. In effect, the explored/total mutant search space and running time will be reduced from 1051/21600 and 22.546 s for `AllRepair` to 65/720 and 1.461 s for `FaultLoc+AllRepair`. Both approaches repair the program by replacing ($x < n$) at loc. ③ with ($x > n$), and $x := x + 1$ at loc. ④ with $x := x - 1$.

3. The programming language: Syntax and semantics

In this section, we describe a small language that will be used to illustrate our work as well as its concrete and abstract semantics. Moreover, we show how to compute error invariants from a program's semantics as a way to rule out irrelevant statements for the error in a program.

3.1. 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 $l \in \mathbb{L}$.

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

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, 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 $p \in Prog$ is of the form: $^{l_{in}}\!: s;^{l_{ass}}\!: \texttt{assert}(be^{ass})$.

3.2. Concrete semantics

A *store* $\sigma \in \Sigma = Var \to \mathbb{Z}$ is a mapping from program variables to values. The semantics of arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$ (resp., Boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\text{true}, \text{false}\})$) is the set of possible

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

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

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

integer (resp., Boolean) values for expression ae (resp., be) in a store σ. Thus,

$[\![n]\!]\sigma = \{n\},\ [\![[n, n']]\!]\sigma = \{n, \dots, n'\},\ [\![\texttt{x}]\!]\sigma = \{\sigma(\texttt{x})\},$

$[\![ae_0 \oplus ae_1]\!]\sigma = \{n_0 \oplus n_1 \mid n_0 \in [\![ae_0]\!]\sigma, n_1 \in [\![ae_1]\!]\sigma\}$

$[\![ae_0 \bowtie ae_1]\!]\sigma = \{n_0 \bowtie n_1 \mid n_0 \in [\![ae_0]\!]\sigma, n_1 \in [\![ae_1]\!]\sigma\}$

$[\![\neg be]\!]\sigma = \{\neg t \mid t \in [\![be]\!]\}$

$[\![be_0 \wedge be_1]\!]\sigma = \{t_0 \wedge t_1 \mid t_0 \in [\![be_0]\!]\sigma, t_1 \in [\![be_1]\!]\sigma\}$

$[\![be_0 \vee be_1]\!]\sigma = \{t_0 \vee t_1 \mid t_0 \in [\![be_0]\!]\sigma, t_1 \in [\![be_1]\!]\sigma\}$

We define a *necessary precondition* (backward) semantics and an *invariance* (forward) semantics on 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 [22], 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 in S may be reached after executing s. The invariance semantics [13] 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. Note that a `while` statement is given in a standard fixed-point formulation [13] using the fix-point operator `lfp`, where the fixed-point functionals $\overleftarrow{\phi}, \overrightarrow{\phi} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ accumulate possible stores after another `while` iteration from a set of stores X going in a backward and forward direction, respectively.

In this way, we can collect the set of possible stores denoting necessary preconditions, written $\texttt{Cond}_{\mathcal{F}}$, and invariants, written $\texttt{Inv}_{\mathcal{I}}$, at each label $l \in \mathbb{L}$ 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 initial label l (resp., final label l'), are:

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

In the case of `while` statement $^{l:}\texttt{while}\,(be)\,\texttt{do}\,\{^{l_b:}s;^{l'_b}\}^{l'}$, necessary precondition at final label of the loop body l'_b is: $\texttt{Cond}_{\mathcal{F}}(l'_b) = \{\sigma \in \texttt{Cond}_{\mathcal{F}}(l') \mid \text{false} \in [\![be]\!]\sigma\}$, whereas invariants at labels l and l_b are: $\texttt{Inv}_{\mathcal{I}}(l) = \texttt{lfp}\,\overrightarrow{\phi}$ and $\texttt{Inv}_{\mathcal{I}}(l_b) = \{\sigma \in \texttt{Inv}_{\mathcal{I}}(l) \mid \text{true} \in [\![be]\!]\sigma\}$.

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 $CVar_S = Var \backslash UVar_S$. That is, a variable $x \in Var$ is *constrained* in S if it cannot take any value from $\mathbb{Z}$ in the stores of S in any context (i.e., for any assignment of values to other variables). 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:

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

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 the 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 $\texttt{ErrInv}_{\mathcal{F}}$ and $\texttt{ErrInv}_{\mathcal{F}}{}'$ be the error invariants for the programs $p[^{l_1:}s^{l_2}]$[1] and $p[^{l_1:}\texttt{skip}^{l_2}]$, respectively. If $\texttt{ErrInv}_{\mathcal{F}}(l_1) = \texttt{ErrInv}_{\mathcal{F}}(l_2)$, then $\texttt{ErrInv}_{\mathcal{F}}(l) = \texttt{ErrInv}_{\mathcal{F}}{}'(l)$ for all $l \in \mathbb{L}$.*

Proof Sketch. The proof is by structural induction on statements s that can be inserted in the hole $-$ of $p[-]$. We consider the case of assignment $\texttt{x} := ae$. Let $\texttt{ErrInv}_{\mathcal{F}}$ be the error invariants inferred

[1] $p[^{l_1:}s^{l_2}]$ is a complete program in which statement s is inserted at the place of hole.

for $p[^{l_1}\!:s^{l_2}]$. Since $\texttt{ErrInv}_{\mathcal{F}}(l_1) = \texttt{ErrInv}_{\mathcal{F}}(l_2)$, the variable x must be unconstrained in $\texttt{Cond}_{\mathcal{F}}(l_1)$ and $\texttt{Cond}_{\mathcal{F}}(l_2)$ due to the definition of $\texttt{ErrInv}_{\mathcal{F}}(l)$ (see Eq. (1)). Therefore, the assignment $\texttt{x} := ae$ has no effect on $\texttt{Cond}_{\mathcal{F}}$ and $\texttt{ErrInv}_{\mathcal{F}}$ due to the definition of $[\![\texttt{x} := ae]\!]$ (see Fig. 4). Note that $[\![\texttt{x} := ae]\!]\, S = S$ when x is unconstrained in S. That is, $\texttt{Cond}_{\mathcal{F}}(l_1) = \texttt{Cond}_{\mathcal{F}}(l_2)$. Hence, we will obtain the same $\texttt{Cond}_{\mathcal{F}}$ and $\texttt{ErrInv}_{\mathcal{F}}$ for $p[^{l_1}\!:\texttt{x} = ae\,^{l_2}]$ and $p[^{l_1}\!:\texttt{skip}\,^{l_2}]$. Similarly, we handle the other cases. □

However, the necessary precondition semantics $\overleftarrow{[\![s]\!]}$ and $\texttt{Cond}_{\mathcal{F}}$, the invariance semantics $\overrightarrow{[\![s]\!]}$ and $\texttt{Inv}_{\mathcal{I}}$, as well as the error invariants $\texttt{ErrInv}_{\mathcal{F}}$ are not computable since our language is Turing complete. In the following, we present their sound decidable abstractions by means of abstract domains.

3.3. Abstract semantics

We now present computable abstract analyses that over-approximate the concrete semantics [13,14]. We consider an abstract domain $(\mathbb{D}, \sqsubseteq_{\mathbb{D}})$, such that a concretization-based abstraction $\langle \mathcal{P}(\Sigma), \subseteq \rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$[2] exists that works only with a (monotonic) concretization function $\gamma_{\mathbb{D}} : \mathbb{D} \to \mathcal{P}(\Sigma)$ expressing the meaning of abstract elements from $\mathbb{D}$ in terms of concrete elements from $\mathcal{P}(\Sigma)$. 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 $\vartriangle_{\mathbb{D}}$, as well as sound transfer functions for assignments $\texttt{ASSIGN}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$, tests $\texttt{FILTER}_{\mathbb{D}} : \mathit{BExp} \times \mathbb{D} \to \mathbb{D}$, and backward assignments $\texttt{B}-\texttt{ASSIGN}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$. We let $\texttt{lfp}^{\#}$ denote an abstract fix-point operator, which is derived using widening $\triangledown_{\mathbb{D}}$ and narrowing $\vartriangle_{\mathbb{D}}$ operators [14]. They enforce convergence of the abstract fix-point operator $\texttt{lfp}^{\#}$, such that it over-approximates the concrete lfp. Finally, the concrete domain 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 $\overleftarrow{[\![s]\!]}^{\#}$ and its abstract invariance semantics $\overrightarrow{[\![s]\!]}^{\#}$ are defined as mappings $\mathbb{D} \mapsto \mathbb{D}$. The complete definitions of functions $\overleftarrow{[\![s]\!]}^{\#}$ and $\overrightarrow{[\![s]\!]}^{\#}$ are given in Fig. 5. For a while loop, $\texttt{lfp}^{\#}\, \overleftarrow{\phi^{\#}}$ and $\texttt{lfp}^{\#}\, \overrightarrow{\phi^{\#}}$ are the limits of the increasing chains:

$y_0 = \texttt{FILTER}_{\mathbb{D}}(\neg be, d),\ y_{n+1} = y_n \triangledown_{\mathbb{D}} \overleftarrow{\phi^{\#}}(y_n)$ for backward analysis

$y_0 = d,\ y_{n+1} = y_n \triangledown_{\mathbb{D}} \overrightarrow{\phi^{\#}}(y_n)$ for forward analysis

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, $\texttt{Cond}^{\#}_{d_{\mathcal{F}}}(l_{ass}) = d_{\mathcal{F}}$ and $\texttt{Inv}^{\#}_{d_{\mathcal{I}}}(l_{in}) = d_{\mathcal{I}}$. For a statement $^{l:}s^{l'}$, abstract element $\texttt{Cond}^{\#}_{d_{\mathcal{F}}}(l)$ (resp., $\texttt{Inv}^{\#}_{d_{\mathcal{I}}}(l')$) of necessary preconditions (resp., invariants) at initial label l (resp., final label l'), are:

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

In the case of while statement $^{l:}\texttt{while}\,(be)\,\texttt{do}\,\{^{l_b:}s;^{l'_b}\}\,^{l'}$, necessary precondition at label l'_b is: $\texttt{Cond}^{\#}_{d_{\mathcal{F}}}(l'_b) = \texttt{FILTER}_{\mathbb{D}}(\neg be, \texttt{Cond}^{\#}_{d_{\mathcal{F}}}(l'))$, whereas invariants at labels l and l_b are: $\texttt{Inv}^{\#}_{d_{\mathcal{I}}}(l) = \texttt{lfp}^{\#}\, \overrightarrow{\phi^{\#}}$, $\texttt{Inv}^{\#}_{d_{\mathcal{I}}}(l_b) = \texttt{FILTER}_{\mathbb{D}}(be, \texttt{Inv}^{\#}_{d_{\mathcal{I}}}(l))$.

The soundness of abstract semantics follows from the soundness of the abstract domain $\mathbb{D}$ [13,14].

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:* $\texttt{Cond}_{\mathcal{F}}(l) \subseteq \gamma_{\mathbb{D}}(\texttt{Cond}^{\#}_{d_{\mathcal{F}}}(l))$ *and* $\texttt{Inv}_{\mathcal{I}}(l) \subseteq \gamma_{\mathbb{D}}(\texttt{Inv}^{\#}_{d_{\mathcal{I}}}(l))$.

[2] Concretization-based abstraction is a relaxation of the known Galois connection abstraction that is equipped with two monotonic functions: abstraction α and concretization γ. However, some abstract domains often used in practice (e.g., Polyhedra domain [23]) do not enjoy Galois connection, but we can still reason about their soundness through the concretization function γ and the relaxed concretization-based abstraction [14].

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

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

$\texttt{Cond}^{\#}_{d_{err}}, \mathbb{P} := \overleftarrow{[\![s]\!]}^{\#}\, d_{err}$;
$d_{in} := \texttt{Cond}^{\#}_{d_{err}}(l_{in})$;
if $(d_{in} = \bot_{\mathbb{D}})$ **then** $\{\texttt{ErrInv}^{\#}_{d_{err}} := \texttt{Cond}^{\#}_{d_{err}};\ d'_{err} := \bot_{\mathbb{D}};\ \mathbb{P} := \emptyset\}$;
if $(d_{in} \neq \bot_{\mathbb{D}})$ **then**
  $\texttt{Inv}^{\#}_{d_{in}} := \overrightarrow{[\![s]\!]}^{\#}\, d_{in}$;
  $\texttt{ErrInv}^{\#}_{d_{err}} := \textsc{MinSupport}(\texttt{Inv}^{\#}_{d_{in}} \sqcap_{\mathbb{D}} \texttt{Cond}^{\#}_{d_{err}}, \texttt{Cond}^{\#}_{d_{err}})$;
  $d'_{err} := \texttt{ErrInv}^{\#}_{d_{err}}(l_{ass})$
return $\texttt{ErrInv}^{\#}_{d_{err}}, d'_{err}, \mathbb{P}$;

Polyhedra abstract domain. The abstract domain $\mathbb{D}$ can be instantiated with different numerical abstract domains including Intervals [13], Octagons [14], and Polyhedra [23]. They differ in precision and computational complexity. Each domain employs data structures and algorithms specific to the shape of properties it represents and manipulates. In this work, we will mainly use the Polyhedra domain due to its precision. The *Polyhedra domain* [23], denoted as $\langle P, \sqsubseteq_P \rangle$, is a fully relational numerical abstract domain, which allows manipulating conjunctions of linear inequalities (constraints) of the form $\alpha_1 \texttt{x}_1 + \cdots + \alpha_n \texttt{x}_n \geq \beta$, where $\texttt{x}_1$, …, $\texttt{x}_n$ are variables and $\alpha_i, \beta \in \mathbb{Q}$ (rationals). The abstract operations are defined in [23].

A property element is represented as a conjunction of linear constraints given in the matrix form $(\!|\mathbf{A}, \vec{\mathbf{b}}|\!)$ which consists of a matrix $\mathbf{A} \in \mathbb{Q}^{m \times n}$ and a vector $\vec{\mathbf{b}} \in \mathbb{Q}^m$, where n is the number of variables and m is the number of constraints. We present some operations. The concretization function is:

$$\gamma_P((\!|\mathbf{A}, \vec{\mathbf{b}}|\!)) = \{\vec{\mathbf{v}} \in \mathbb{Q}^n \mid \mathbf{A} \cdot \vec{\mathbf{v}} \geq \vec{\mathbf{b}}\}$$

The meet $\sqcap_P$ and the widening $\triangledown_P$ are defined as:

$$(\!|\mathbf{A}_1, \vec{\mathbf{b}}_1|\!) \sqcap_P (\!|\mathbf{A}_2, \vec{\mathbf{b}}_2|\!) = (\!|\begin{pmatrix}\mathbf{A}_1\\ \mathbf{A}_2\end{pmatrix}, \begin{pmatrix}\vec{\mathbf{b}}_1\\ \vec{\mathbf{b}}_2\end{pmatrix}|\!)$$

$$(\!|\mathbf{A}_1, \vec{\mathbf{b}}_1|\!) \triangledown_P (\!|\mathbf{A}_2, \vec{\mathbf{b}}_2|\!) = \{c \in (\!|\mathbf{A}_1, \vec{\mathbf{b}}_1|\!) \mid (\!|\mathbf{A}_2, \vec{\mathbf{b}}_2|\!) \sqsubseteq_P \{c\}\}$$

where c represents one constraint from $(\!|\mathbf{A}_1, \vec{\mathbf{b}}_1|\!)$. $\texttt{FILTER}_P$ handles precisely affine inequality tests by adding them to the input polyhedra.

$$\texttt{FILTER}_P(\sum_i \alpha_i \texttt{x}_i \geq \beta, (\!|\mathbf{A}, \vec{\mathbf{b}}|\!)) = (\!|\begin{pmatrix}\mathbf{A}\\ \alpha_1 \ldots \alpha_n\end{pmatrix}, \begin{pmatrix}\vec{\mathbf{b}}\\ \beta\end{pmatrix}|\!)$$

In all other cases, $\texttt{FILTER}_P$ performs the sound identity operation. Likewise, $\texttt{ASSIGN}_P$ handles exactly affine assignments, and it performs the sound non-deterministic assignment for all other (non-affine) cases. The soundness of the abstract semantics based on Polyhedra is proved in [23].

4. Abstract error invariants

In this section, we formalize our idea for using iterative abstract analysis to infer abstract error invariants. First, we present one iteration of the backward–forward analysis for generating abstract error invariants. Then, we introduce our procedure for iterative abstract analysis. Finally, we present the BDD abstract domain that is used to refine inferred error invariants.

4.1. Abstract analysis

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

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

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

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 $\mathbb{D}$ specifying the considered approximated properties $d \in \mathbb{D}$. 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 $\texttt{AbsAnalysis}(s, d_{err}, \mathbb{D})$ procedure is given in Algorithm 1. It takes as input a statement s, a target abstract error state d_{err}, and a chosen abstract domain $\mathbb{D}$. First, we call a backward abstract analysis $\overleftarrow{[\![s]\!]^\sharp}\, d_{err}$ (Line 1), which computes the necessary preconditions $\texttt{Cond}^\sharp_{d_{err}}$ of statement s. 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 $\overrightarrow{[\![s]\!]^\sharp}\, d_{in}$ 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 statement s 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): for any $l \in \mathbb{L}$

$$\texttt{ErrInv}^\sharp_{d_{err}}(l) = \textsc{MinSupport}(\texttt{Inv}^\sharp_{d_{in}}(l) \sqcap_{\mathbb{D}} \texttt{Cond}^\sharp_{d_{err}}(l), \texttt{Cond}^\sharp_{d_{err}}(l)) \tag{2}$$

where MinSupport is minimal support set. 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 MinSupport is computed [24].

Definition 3. Let $P = \{p_1, \ldots, p_n\}$ and $p'_1 \wedge \cdots \wedge p'_k$ be linear constraint formulas over program variables, such that $p_1 \wedge \cdots \wedge p_n \vDash p'_1 \wedge \cdots \wedge p'_k$. A subset $P' \subseteq P$ *supports* the inference $\bigwedge_{p_i \in P} p_i \vDash p'_1 \wedge \cdots \wedge p'_k$ iff $\bigwedge_{p_j \in P'} p_j \vDash p'_1 \wedge \cdots \wedge p'_k$. A support set P' is minimal iff no proper subset of P' can support the inference.

For $P \vDash p'_1 \wedge \cdots \wedge p'_k$, let the $\textsc{MinSupport}(P, p'_1 \wedge \cdots \wedge p'_k)$ denote the set of minimal supporting conjuncts in P that imply $p'_1 \wedge \cdots \wedge p'_k$. An implementation of MinSupport through unsatisfiability cores is available in existing SMT solvers (e.g., Z3 [20]) for many theories such as linear arithmetic. That is, we ask the SMT solver to find the unsatisfiability core of $\bigwedge_{p_i \in P} p_i \wedge \neg(p'_1 \wedge \cdots \wedge p'_k)$ (which is negation of $\bigwedge_{p_i \in P} p_i \implies (p'_1 \wedge \cdots \wedge p'_k)$). The conjuncts in P that are part of this unsatisfiability core represent a minimal support set for $(p'_1 \wedge \cdots \wedge p'_k)$.

Example 4. 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 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 $\texttt{Inv}^\sharp_{d_{in}}(l) \sqcap_{\mathbb{D}} \texttt{Cond}^\sharp_{d_{err}}(l)$. We can now show the soundness of $\texttt{ErrInv}^\sharp_{d_{err}}$.

Proposition 5. *Let* $\mathcal{F} = \gamma_{\mathbb{D}}(d_{\mathcal{F}})$. *For* $l \in \mathbb{L}$, $\texttt{ErrInv}_{\mathcal{F}}(l) \subseteq \gamma_{\mathbb{D}}(\texttt{ErrInv}^\sharp_{d_{\mathcal{F}}}(l))$.

Proof Sketch. By definitions in Eqs. (1) and (2), $\texttt{ErrInv}_{\mathcal{F}}$ and $\texttt{ErrInv}^\sharp_{d_{\mathcal{F}}}$ are obtained by refining $\texttt{Cond}_{\mathcal{F}}$ and $\texttt{Cond}^\sharp_{d_{\mathcal{F}}}$ using results from $\texttt{Inv}_{\mathcal{I}}$ and $\texttt{Inv}^\sharp_{d_{\mathcal{I}}}$ (where $\mathcal{I} = \texttt{Cond}_{\mathcal{F}}(d_{in})$ and $\mathcal{I} = \gamma_{\mathbb{D}}(d_{\mathcal{I}})$), respectively. By using the soundness of $\texttt{Cond}^\sharp_{d_{\mathcal{F}}}$ (Proposition 2), we have $\texttt{ErrInv}_{\mathcal{F}}(l) \subseteq \texttt{Cond}_{\mathcal{F}}(l) \subseteq \gamma_{\mathbb{D}}(\texttt{Cond}^\sharp_{d_{\mathcal{F}}}(l))$ for $l \in \mathbb{L}$. This means that $\texttt{Cond}^\sharp_{d_{\mathcal{F}}}$ can be used as sound error invariants albeit very imprecise (see the procedure `FaultLoc_Single` in Section 6). By using the soundness of $\texttt{Inv}^\sharp_{d_{\mathcal{I}}}$ (Proposition 2), that is $\texttt{Inv}_{\mathcal{I}}(l) \subseteq \gamma_{\mathbb{D}}(\texttt{Inv}^\sharp_{d_{\mathcal{I}}}(l))$, and definitions of $\texttt{ErrInv}_{\mathcal{F}}$ in Eq. (1) and $\texttt{ErrInv}^\sharp_{d_{\mathcal{F}}}$ in Eq. (2), we have $\texttt{ErrInv}_{\mathcal{F}}(l) \subseteq \gamma_{\mathbb{D}}(\texttt{ErrInv}^\sharp_{d_{\mathcal{F}}}(l))$ for $l \in \mathbb{L}$. □

4.2. Iterative abstract analysis

The $\texttt{AbsAnalysis}(s, d_{err}, \mathbb{D})$ 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. These domains can only express a conjunction of linear

Algorithm 2: $\texttt{FaultLoc}(p, \mathbb{D})$

Input: Program $p \equiv {}^{l_{in}:}\, s; {}^{l_{ass}:}\, \texttt{assert}(be^{ass})$, 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 $\quad \texttt{ErrInv}^{\sharp}_{d_{err}}, d'_{err}, \mathbb{P}' := \texttt{AbsAnalysis}(s, d_{err}, \mathbb{D})$;
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** $Slice(s, \texttt{ErrInv}^{\sharp}_{d_{err}})$;

constraints over program variables. 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 state d_{err}. In particular, we use a BDD abstract domain functor [25–27], 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 $\texttt{FaultLoc}(p, \mathbb{D})$ procedure is shown in Algorithm 2. The procedure is called with the following parameters: a program p of the form ${}^{l_{in}:}\, s; {}^{l_{ass}:}\, \texttt{assert}(be^{ass})$, and a base abstract domain $\mathbb{D}$. Initially, we call $\texttt{AbsAnalysis}(s, \texttt{FILTER}_{\mathbb{D}}(\neg be^{ass}, \top_{\mathbb{D}}), \mathbb{D})$ 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} 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 $\texttt{Slice}(s, \texttt{ErrInv}^{\sharp}_{d_{err}})$ (Line 8) returns a slice of program ${}^{l_{in}:}\, s; {}^{l_{ass}:}\, \texttt{assert}(be^{ass})$ 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 [25–27], denoted $\mathbb{BD}(\mathbb{P}, \mathbb{A})$, plays an important role in the iterative abstract analysis procedure. We exploit the well-known efficiency of BDDs, introduced by Bryant [28] for representing Boolean functions [29], and adapt them to represent disjunctive analysis properties. More specifically, 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) [25–27]. 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 $\langle\!|a|\!\rangle$ 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: $P_1 < \cdots < P_n$. The left and right subtrees are either both leafs or both rooted at decision nodes labeled with the same predicate.

However, BDTs contain some redundancy. There are three optimizations we can apply to BDTs in order to reduce their representation [28]: (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) $b \in \mathbb{BD}(\mathbb{P}, \mathbb{A})$ [25–27]. 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. Furthermore, by applying the abstract operations and transfer functions on BDDs in canonical forms, we obtain as result BDDs in canonical forms.

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 $a \in \mathbb{A}$. More formally, $\gamma_{\mathbb{BD}}(b) = \overline{\gamma}_{\mathbb{BD}}[\text{true}](b)$, where function $\overline{\gamma}_{\mathbb{BD}}$ is defined as:

$$\overline{\gamma}_{\mathbb{BD}}[C](\langle\!|a|\!\rangle) = (\bigvee_{\mu \in \texttt{Eval}(\mathbb{P}), \mu \vDash C} \mu) \wedge \gamma_{\mathbb{A}}(a),$$
$$\overline{\gamma}_{\mathbb{BD}}[C]([\![P : tl, tr]\!]) = \overline{\gamma}_{\mathbb{BD}}[C \wedge P](tl) \vee \overline{\gamma}_{\mathbb{BD}}[C \wedge \neg P](tr)$$

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 [25–27] 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* [25]. Let $b \in \mathbb{BD}(\mathbb{P}, \mathbb{A})$ be a BDD obtained after application of the assignment transfer function. The reconstruction on leaves procedure proceeds as follows:

- Find all leaves in b: say $L = \{a_1, \ldots, a_n\}$.
- Given a top-down path of predicates $P_1 \cdot \ldots \cdot P_k$ in the BDD b, we calculate $a'_i = a_i \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}}(P_1 \wedge \cdots \wedge P_k)$ for all leafs in L. The path $P_1 \cdot \ldots \cdot P_k$ now leads to the new updated leaf: $a'_1 \sqcup_{\mathbb{A}} \ldots \sqcup_{\mathbb{A}} a'_n$.

Example 6. Suppose we have a BDD $b = [\![(\mathtt{x} \leq 0) : \langle\!|\mathtt{x} = 0|\!\rangle, \langle\!|1 \leq \mathtt{x} \leq 10|\!\rangle]\!]$ and an assignment $\mathtt{x} := \mathtt{x} - 1$. Note that the left leaf $\langle\!|\mathtt{x} = 0|\!\rangle$ satisfies the decision node $(\mathtt{x} \leq 0)$, while the right leaf $\langle\!|1 \leq \mathtt{x} \leq 10|\!\rangle$ satisfies its negation. After performing the (forward) assignment transfer function without reconstruction on leaves, we obtain: $[\![(\mathtt{x} \leq 0) : \langle\!|\mathtt{x} = -1|\!\rangle, \langle\!|0 \leq \mathtt{x} \leq 9|\!\rangle]\!]$. Hence, the right leaf node $(0 \leq \mathtt{x} \leq 9)$ does not satisfy the predicate leading to it: $\neg(\mathtt{x} \leq 0)$. However, after the reconstruction on leaves, we obtain: $[\![(\mathtt{x} \leq 0) : \langle\!| -1 \leq \mathtt{x} \leq 0|\!\rangle, \langle\!|1 \leq \mathtt{x} \leq 9|\!\rangle]\!]$. This is shown graphically in Fig. 7.

The efficiency of BDDs comes from the opportunity to share equal sub-trees, in case some properties are independent from the value of some Boolean predicates.

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

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

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

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

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

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 $\texttt{ErrInv}^{\sharp}_{d_{err}}$.

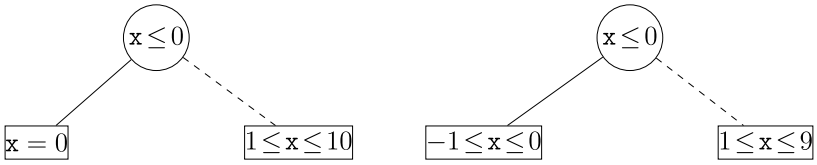

Fig. 7. BDDs before (left) and after (right) application of $\overleftarrow{[\![\texttt{x := x} - 1]\!]^{\sharp}}$ (solid edges = true, dashed edges = false).

5. Applications to program repair

In this section, we show how our fault localization algorithm can be combined with an existing mutation-based program repair approach. First, we give an overview how programs are translated to SMT formulas, and then we present the efficient algorithm for automatic program repair with fault localization.

5.1. Translating programs to formulas

We now describe the process of translating programs into SMT formulas using the CBMC bounded model checker [2]. Three transformations of the input program are employed. First, we *simplify* the program by substituting all `if` and `while` guards (conditions) with new Boolean variables. Second, we *unroll* the program with bound b, such that all `while`-s in the given program are unrolled b-times. That is, the `while`-body is duplicated b times, where each copy is guarded by an `if` with the same guard as the original `while`-guard. To block all paths going through the `while` longer than the bound b, we use `assume`$(\neg g)$ in the innermost duplicated `while`-body, where g is equal to the `while`-guard. For example, the statement "`while` (be) `do` s" after unwinding with $b = 2$ will be transformed to:

`int` g`:=`be`; if` (g) `then` $\{s;$ g`:=`be`; if` (g) `then` $\{s;$ g`:=`be`; assume`$(\neg g)$`; } }`

Third, we transform the program to the *SSA (Static Single Assignment)-form*, such that any assignment to a variable x is changed into an unique assignment to a new variable $\texttt{x}_i$ ($i \geq 0$). To decide which definition of a given variable reaches a particular use after an `if` with the guard g, we insert the Φ-assignment $\texttt{x}_k := g?\texttt{x}_i : \texttt{x}_j$ after the `if`. This means that Φ selects $\texttt{x}_i$ if control reaches the Φ-assignment via the path on which g is `true`; otherwise Φ selects $\texttt{x}_j$. Thus, all uses of x after the Φ-assignment $\texttt{x}_k := g?\texttt{x}_i : \texttt{x}_j$ become uses of $\texttt{x}_k$ until the next assignment of x. After the above three transformations, in the obtained simplified program all original program expressions are right-hand sides (RHSs) of assignments, loops are replaced with `if`-s, and each variable is assigned once.

The generated transformed program p is translated to a set of SMT formulas S_p as follows. We translate: an assignment `x` $:=e$ to equation formula `x` $=e$; a Φ-assignment `x2 := be?x0 : x1` to formula `x2=ite(be,x0,x1)` (meaning $(be \wedge \texttt{x2} = \texttt{x0}) \vee (\neg be \wedge \texttt{x2} = \texttt{x1})$); an `assume`$(be)$ to formula be; and an `assert`(be) to formula $\neg be$. In effect, we obtain that one formula in S_p encodes a single statement in the input program p. The obtained set of formulas S_p is partitioned into three subsets: $S_p^{\texttt{soft,rel}}$ that includes formulas corresponding to statements (assignments) containing original program expressions that are classified by `FaultLoc` as relevant for the error, $S_p^{\texttt{soft,irrel}}$ that includes formulas corresponding to statements (assignments) containing original program expressions that are classified by `FaultLoc` as irrelevant for the error, and $S_p^{\texttt{hard}}$ that includes the other formulas corresponding to assertions, assumptions, and Φ-assignments. For example, SMT formulas for `program1`, `program2-a`, and `program3` (with bound $b = 1$) from Section 2 are as follows:

$S^{\texttt{soft,rel}}_{\texttt{program1}} = \{z0 = n0+1, y0 = z0\}$,
$S^{\texttt{soft,irrel}}_{\texttt{program1}} = \{z1 = z0+1\}$, $S^{\texttt{hard}}_{\texttt{program1}} = \{\neg(y0 < n0)\}$
$S^{\texttt{soft,rel}}_{\texttt{program2-a}} = \{x0 = 1, y0 = input0 - 42\}$,
$S^{\texttt{soft,irrel}}_{\texttt{program2-a}} = \{g0 = (y0 < 0), x1 = 0\}$,
$S^{\texttt{hard}}_{\texttt{program2-a}} = \{\neg(x2 \leq 0), x2 = \texttt{ite}(g0, x1, x0)\}$
$S^{\texttt{soft,rel}}_{\texttt{program3}} = \{x0 = 6, g0 = (x0 < n0), x1 = x0+1\}$,
$S^{\texttt{soft,irrel}}_{\texttt{program3}} = \{y0 = n0, y1 = y0+1\}$
$S^{\texttt{hard}}_{\texttt{program3}} = \{x2 = \texttt{ite}(g0, x1, x0), \neg(g1 = (x1 < n0)), \neg(x2 \leq 10)\}$

The formula φ_p^b for program p is the conjunction of formulas in S_p, thus encoding all possible b-bounded paths in program p. We say that a program p is *b-correct* if all assertions in it are valid in all b-bounded paths of p.

Proposition 7 ([2]). *A program p is b-correct iff φ_p^b is unsatisfiable.*

The formula $\varphi_p^b = \phi_1 \wedge \cdots \wedge \phi_n$ for a program p is instrumented with Boolean variables called *guard variables*, thus obtaining $\texttt{inst}(\varphi_p^b) =$

Algorithm 3: `FaultLoc+AllRepair`(p, b)

Input: Program p, unwinding bound b
Output: Set of solutions Sol

1 $(S_p^{\texttt{hard}}, S_p^{\texttt{soft,rel}}, S_p^{\texttt{soft,irrel}}) := \texttt{CBMC}(p, \texttt{FaultLoc}(p, \mathbb{D}), b)$;
2 $(S_1, \dots, S_n) := \texttt{Mutate}(S_p^{\texttt{soft,rel}})$;
3 $(S'_1, \dots, S'_n, V_1, \dots, V_n, V_{\texttt{orig}}) := \texttt{InstGuardVar}(S_1, \dots, S_n)$;
4 $\varphi_p^b := (\wedge_{s \in S_p^{\texttt{hard}} \cup S_p^{\texttt{soft,irrel}}} s) \wedge (\wedge_{s \in S'_1 \cup \dots \cup S'_n} s)$;
5 $k := 1$; $Sol := \emptyset$; $\varphi := \wedge_{i=1}^{n} \texttt{Exact}(V_i, 1)$;
6 **while** $(k \leq n)$ **do**
7 $\varphi_k := \varphi \wedge \texttt{AtLeast}(V_{\texttt{orig}}, n-k)$;
8 $satres, V := \texttt{SAT}(\varphi_k)$;
9 **if** $(satres)$ **then**
10 $smtres := \texttt{SMT}(\varphi_p^b \wedge \wedge_{v \in V} v)$;
11 **if** $(\neg smtres)$ **then**
12 $Sol := Sol \cup V$;
13 $\varphi_k := \varphi_k \wedge (\vee_{v \in V \setminus (V_{\texttt{orig}})} \neg v)$;
14 **else**
15 $\varphi_k := \varphi_k \wedge (\vee_{v \in V} \neg v)$;
16 **else**
17 $k := k + 1$;
18 **return** Sol;

$(x_1 \Longrightarrow \phi_1) \wedge \dots \wedge (x_n \Longrightarrow \phi_n)$, where $x_1, \dots, x_n$ are new guard variables. The formula $(x_i \Longrightarrow \phi_i)$ can be satisfied either when x_i is set to false or when x_i is set to true and ϕ_i is satisfied. In our algorithm, some guard variables called *assumptions* are conjuncted with $\texttt{inst}(\varphi_p^b)$ and passed to an incremental SMT solver, which needs to check only the satisfiability of ϕ_i formulas corresponding to assumptions. This way, we enable incremental SMT solving, where the information learned in one iteration is re-used in the next iteration.

We will use formulas $\texttt{AtLeast}(\{l_1, \dots, l_n\}, k)$ and $\texttt{Exact}(\{l_1, \dots, l_n\}, k)$ to require that at least k and exactly k, respectively, of the literals $l_1, \dots, l_n$ are true. They are called Boolean cardinality formulas and we will use the MiniCard SAT-solver [30] to check their satisfiability.

5.2. Algorithm

We integrate the `AllRepair` procedure of [16] with the `FaultLoc` procedure in Algorithm 2, whose goal is to reduce the mutant's search space by not mutating locations (statements) that are found by `FaultLoc` as irrelevant for the error. The novel procedure, called `FaultLoc+AllRepair`, is shown in Algorithm 3. The input program p is translated to formulas $(S_p^{\texttt{hard}}, S_p^{\texttt{soft,rel}}, S_p^{\texttt{soft,irrel}})$ by using the CBMC model checker, where the information of the `FaultLoc` procedure is used for constructing subsets $S_p^{\texttt{soft,rel}}$ and $S_p^{\texttt{soft,irrel}}$. Next, we use the `Mutate` procedure to generate all possible mutations $S_1, \dots, S_n$ of formulas in $S_p^{\texttt{soft,rel}}$, where S_i is a set of formulas obtained by mutating some $\phi_i \in S_p^{\texttt{soft,rel}}$. Hence, $S_1, \dots, S_n$ correspond to n program locations where statements relevant for the error in p may occur. A *mutation* is a replacement of an program expression of the RHS of an assignment with another expression. We have a fixed set of mutations for each type of expressions: arithmetic operators $\{+, -, *, \%, \div\}$; relational operators $\{<, \leq, >, \geq\}$ and $\{==, !=\}$; logical operators $\{\&\&, \|\}$; integer constants $\{n, n+1, n-1\}$; and program variables $\{x, y \in Param(p)\}$, where $Param(p)$ are the parameters used in program p. For arithmetic operators, this means that any operator from $\{+, -, *, \%, \div\}$ can be substituted with any other operator from the same set; for integer constants $n \in \mathbb{Z}$, this means that they can be increased by one or decreased by one; whereas for program variables $x \in Var$, this means that they can be replaced by any parameter $Param(p)$.

The `InstGuardVars` procedure instruments all formulas in $S_1, \dots, S_n$ by new guard variables $V_1, \dots, V_n$, thus producing instrumented formulas $S'_1, \dots, S'_n$ (Line 3). The set $V_{\texttt{orig}}$ contains guard variables corresponding to original formulas in $S_p^{\texttt{soft,rel}}$. The formula φ_p^b is initialized to be the conjunction of formulas from $S^{\texttt{hard}}$ and $S_p^{\texttt{soft,irrel}}$, and all instrumented formulas from $S'_1 \cup \dots \cup S'_n$ (Line 4). Then, we search the space of all mutated formulas using an iterative generate-and-verify procedure (Lines 6–17). In the *generate step*, we create Boolean formula φ_k expressing that at most k guard variables are not original (i.e., at least $n-k$ are original by $\texttt{AtLeast}(V_{\texttt{orig}}, n-k)$) and there is exactly one guard variable selected for any of n relevant statements (Line 7). An SAT solver checks the satisfiability of φ_k, such that any satisfying assignment V of φ_k corresponds to one mutated program (Line 8). In the *verify step*, the formula corresponding to the mutated program, which is φ_p^b with assumptions in V, is checked by the Z3 SMT solver (Line 10). If the formula is found unsatisfiable, we report the mutant corresponding to V as possible solution and block all supersets of V for further exploration (line 13); otherwise we block the current mutant V for exploration (line 15) and continue with the search.

6. Evaluation

We have implemented a prototype tool based on our approach for fault localization via inferring error invariants and for its application to program repair. We now evaluate our tool.

6.1. Implementation and experimental setup

Our tool is based on the APRON library [18], which includes the abstract domains of intervals, octagons, and polyhedra, and the BDD-APRON library [19]. It also calls the Z3 SMT solver [20] 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 no support for arrays, pointers, `struct` and `union` types. We have also implemented our program repair algorithm that combines our tool for fault localization and the `AllRepair` tool [7,16] for repairing C programs.

For the aim of evaluating our fault localization approach, we ran: (1) our approach, denoted `FaultLoc`; (2) an approach where we use single-iteration procedure and backward abstract analysis $\texttt{Cond}_d^\sharp$ is only employed to compute abstract error invariants, denoted `FaultLoc_Single`; and (3) the logic formula-based fault localization tool `BugAssist` [12].[3] Given an error program, `BugAssist` uses the CBMC bounded model checker [2] to generate an error trace as well as to construct the corresponding trace formula, which is then analyzed by a `MAX-SAT` solver. Moreover, we also evaluate the benefits that our fault localization approach brings to the program repair. For this purpose, we compare: (1) the tool for program repair with no fault localization, called `AllRepair`; and (2) the approach that combines our `FaultLoc` and `AllRepair`, called `FaultLoc+AllRepair`. We use a set of numerical benchmarks taken from the literature [8], different folders of SV-COMP (https://sv-comp.sosy-lab.org/) and TCAS [31].

Experiments are run on 64-bit Intel®Core™ i7-1165G7 CPU@2.80 GHz, VM LUbuntu 20.10, with 8 GB memory, and we use a timeout value of 300 s. All times are reported as average over thirty independent executions, which is considered large enough size of statistical samples [32]. The observed speed-up/slow-down of the average execution time is reported as a global measure of the acceleration/deceleration of the execution time when comparing performances of two approaches [32]. 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: https://doi.org/10.5281/zenodo.8167960.

[3] The other known logic formula-based tool [8,10] is not available online.

6.2. Examples

We now present several of our benchmarks in detail.

Warming-up example. Consider the warming-up example, denoted `warming`, given in Fig. 8 taken from [11]. Our approach `FaultLoc` needs three iterations to terminate by using the BDD domain with predicates $B1 \equiv (\texttt{input1} > 0)$ and $B2 \equiv (\texttt{input2} > 0)$ and the Polyhedra leaf domain. Some of the inferred error invariants are shown in Fig. 8. The first error invariant $B1 \wedge B2$ shows that the error occurs if the input values of `input1` and `input2` are greater than 0. The next shown error invariant $B1 \wedge B2 \wedge x = 1 \wedge y = 1$ incorporates the effect of assignments at locs. ①–③. From this invariant, we realize that to the end of the program, the variable z is irrelevant and the values of variables x and y should be tracked. Furthermore, it states that only `then`-branches of both `if` statements are important for the error. In summary, `FaultLoc` infers that z is an irrelevant variable, so all statements referring to z are rendered as irrelevant (locs. ③, ⑦, Ⓒ) as well as the statement $x := 10$ in the `else` branch of the first `if` statement. If we use a single backward analysis `FaultLoc_Single`, we obtain less precise results. In particular, the statement $x := 10$ at loc. ⑧ is now relevant. Moreover, due to the precision loss that we have after analyzing the first `if` statement, the statement $x := 1$ at loc. ① is rendered as irrelevant and dropped from the reported program slice. It is interesting to note that the slices reported by both approaches, `FaultLoc` and `FaultLoc_Single`, contain the same number of statements (see Table 1, column `RLOC`). Still, the second approach `FaultLoc_Single` based on single program analysis loses precision in two places: one statement is wrongly reported as relevant, and one statement is wrongly reported as irrelevant. Therefore, we conclude that the precision of `FaultLoc_Single` is 87% (see Table 1, column `Prec`).

We have also analyzed a slightly changed warming-up example, denoted `warming-a`, where the assertion is $(x > 6) \vee (y > 7)$. We establish that assertion violation may happen when $B1 \wedge \neg B2$ holds at the initial location. Due to the precise BDD abstract domain we obtain very precise analysis results using `FaultLoc` that enable us to eliminate the second `if` statement completely (locs. ⑨–Ⓒ). Indeed, the execution of this `if` statement does not influence the violation of the assertion. Moreover, statements at locs. ③, ⑦, and ⑧ are also redundant. On the other hand, the single backward analysis `FaultLoc_Single` reports as relevant the second `if` statement and the statement at loc. ⑧. Again, it reports as irrelevant the statement at loc. ①. This is due to the precision loss that occurred during the analysis of `if` statements. In effect, the precision of `FaultLoc_Single` drops to 56% in this case (see Table 1, column `Prec`).

We have also applied the tool `BugAssist` to these examples. For instance, `BugAssist` reports statements at locs. ⑧ and Ⓒ as potential bugs, while statements at locs. ① and ② as irrelevant for both `warming` and `warming-a`. The precision of `BugAssist` is 75% for `warming` and 43% for `warming-a`.

SV-COMP *examples.* Consider the program in Fig. 9. 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 `FaultLoc`, 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 `FaultLoc` are shown in Fig. 9. 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 $\neg B \wedge y \geq 0$. 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.

```
void main(int input1, int input2) {
B1 ∧ B2
1 int x := 1;
2 int y := 1;
3 int z := 1;
B1 ∧ B2 ∧ x = 1 ∧ y = 1
4 if (input1>0) then{
5   x := x + 5;
6   y := y + 6;
7   z := z + 4; }
8 else ⊥ x := 10; ⊥
B1 ∧ B2 ∧ x = 6 ∧ y = 7
9 if (input2>0) then{
A   x := x + 6;
B   y := y + 5;
C   z := z + 4; }
B1 ∧ B2 ∧ x = 12 ∧ y = 12
D assert ((x < 10)||(y < 10)); // assert ((x > 6)||(y > 7));
```

Fig. 8. `Warming-up` ($B1 \equiv \texttt{input1} > 0, B2 \equiv \texttt{input2} > 0$).

If we analyze this program via single backward analysis `FaultLoc_Single`, we do not consider separately `then` and `else` branches of the `if` statement as in `FaultLoc`. 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 obtain 53% precision for `FaultLoc_Single` (see Table 1, column `Prec`).

Consider the program `Mysore-1` from SV-COMP given in Fig. 10. The inferred error invariants by `FaultLoc` are shown in Fig. 10. 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 `FaultLoc_Single`, then the loop invariant is $(c \leq 0 \wedge x + c \leq -1)$, and thus no statement is eliminated as irrelevant. As a result, `FaultLoc_Single` 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. The `BugAssist` reasons about loops by unrolling them, which makes it very sensitive to the degree of unrolling. 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) [31], which represents an aircraft collision detection system used by all US commercial aircrafts. An extract from the error implementation is shown in Fig. 11. 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

```
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. 9. 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. 10. Mysore-1 example.

locations. FaultLoc needs two iterations to end by using the BDD domain with four predicates:

$B1 \equiv$ (need_upward_RA = 1) ∧ (need_downward_RA = 1)

$B2 \equiv$ (Own_Tracked_Alt < Other_Tracked_Alt)

$B3 \equiv$ (own_above_threat = 0) ∧ (Curr_Vertical_Sep ≥ MINSP)∧
(Up_Separation ≥ Positive_RA_Alt_Tresh)

$B4 \equiv$ (own_below_threat = 0)∨
(own_below_threat = 1 ∧ ¬(Down_Separation
> Positive_RA_Alt_Tresh))

The slice computed by FaultLoc approach contains 44 locations relevant for the error. Some of these statements are shown underlined in Fig. 11. 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 FaultLoc_Single 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 FaultLoc_Single, none of these locations is from the buggy Non_Crossing_Biased_Climb().

Performances. Table 1 shows the result of running our tool FaultLoc, the single backward analysis FaultLoc_Single, 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, "RLOC" is the number of potential (relevant) fault locations, and "Prec" is the precision (in percentage) of the given approach to locate the relevant statements for the error. This is the ratio of the sum of *correctly* classified relevant/irrelevant for the error locations by an approach to the total number of locations in the program. A classification of a location as relevant/irrelevant for the error given by the concrete semantics is considered correct.

We conclude that our technique FaultLoc gives more precise information about potential error statements than simply performing a backward analysis and BugAssist. On average by using FaultLoc, the number of locations to check for potential error (RLOC) is reduced to 47.6% of the total code (LOC). In fact, FaultLoc pin-pointed the correct error locations for all examples, thus achieving the precision of 100%. On the other hand, the precision of FaultLoc_Single is 70% and the precision of BugAssist is 64%, on average. Although our technique FaultLoc is the most precise, it is slower than FaultLoc_Single due to the several iterations it needs to produce the fully refined error invariants. On our benchmarks,[4] this translates to slow-downs (FaultLoc vs. FaultLoc_Single) that range from 4.3 to 38 times. However, this is an acceptable precision/cost tradeoff, since the more precise results of FaultLoc can be applied in various areas (e.g., in program repair). FaultLoc 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, FaultLoc 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 FaultLoc in all examples is still significantly smaller than our human effort required to isolate the fault (see Table 1).

6.3. Application to program repair

Table 2 shows the performance of the approaches for repairing our benchmarks. The column LOC is the total number of locations, RLOC is the number of relevant fault locations, Space is given in the form n/m where n is the explored and m is the total size of the mutant search space, and Time is the running time given in seconds.

For programs easy2-1 (resp., Mysore-1) in Fig. 9 (resp., Fig. 10), our fault localization algorithm establishes that statements at locs. ③ and ⑥ (resp., at locs. ⑤ and ⑥) are irrelevant for the error. Therefore, FaultLoc+AllRepair successfully finds repairs in 17.325 (resp., 2.112) sec by exploring 1250 (resp., 40) mutants, while AllRepair timeouts after 300 s. Similarly, we repair program2, where only statement at loc. ① is irrelevant for the error. Still, we observe that FaultLoc+AllRepair achieves 3× reduction of the explored/total search space and 2× speed-up of the running time compared to AllRepair.

To handle the obtained slices of other benchmarks (warming, warming-a, TCASv.1), we apply a simpler list of mutations: {+, −}, {∗, %, ÷}, {<, ≤}, {>, ≥}, {==, ! =}, and variables/parameters are not mutated. Still, the total mutant search space is huge: 136,048,896

[4] We do not consider TCASv.1 here since FaultLoc_Single is useless in locating the bug in this case.

```
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 Positive_RA_Alt_Tresh := 740;
 ...
int res := alt_sep_test() ; }
```

Fig. 11. An excerpt from an error TCAS implementation.

Table 1
Performance results of FaultLoc vs. FaultLoc_Single vs. BugAssist. FaultLoc and FaultLoc_Single use Polyhedra domain. All times in sec.

Bench.	LOC	FaultLoc			FaultLoc_Single			BugAssist		
		Time	RLOC	Prec	Time	RLOC	Prec	Time	RLOC	Prec
program1	6	0.058	2	100%	0.014	2	100%	0.032	1	66%
program2	9	0.187	5	100%	0.013	4	83%	0.031	2	50%
program2-a	9	0.086	2	100%	0.016	6	33%	0.031	2	66%
program3	10	0.453	3	100%	0.016	3	100%	2.954	5	71%
warming	19	0.387	12	100%	0.016	12	0.87	0.017	12	0.75
warming-a	19	0.301	7	100%	0.017	13	0.56	0.202	8	0.43
easy2-1	15	1.404	10	100%	0.012	3	41%	8.445	9	66%
Mysore-1	9	0.050	4	100%	0.014	6	66%	0.210	3	83%
TCASv.1	118	57.71	44	100%	0.224	28	86%	0.094	1	62%

for warming and 4.7×10^{24} for TCASv.1, so AllRepair timeouts even when the simpler mutations are applied. The fault localization algorithm removes 4 irrelevant locations for warming and 9 irrelevant locations for warming-a, thus FaultLoc+AllRepair terminates in 34.74 s for warming and 0.692 s for warming-a. However, FaultLoc produces 44 locations relevant for the error in TCASv.1, so the obtained mutant search space is still huge (1.3×10^{15}), thus making FaultLoc+AllRepair to timeout as well.

We can conclude that our FaultLoc+AllRepair significantly outperforms AllRepair for all benchmarks, and moreover often turns previously infeasible tasks into feasible. For feasible tasks, FaultLoc+AllRepair achieves speed-ups that range from 2.3 to 16.5 times compared to AllRepair. Still, both approaches have the same precision by reporting the same repaired programs.

7. Related work

We divide our discussion of related work into five categories: automated fault localization, program slicing, abstract interpretation-based backward analysis, decision-tree abstract domains, fault localization for program repair, and possible extensions.

Automated fault localization. Fault localization has been an active area of research in recent years [7–10,12]. Several approaches are based

Table 2
Performance results of `FaultLoc+AllRepair` vs. `AllRepair`. Times are in seconds.

Bench.	LOC	FaultLoc+AllRepair			AllRepair		
		RLOC	Space	Time	RLOC	Space	Time
program1	6	2	24/30	0.372	3	546/900	3.645
program2	9	5	243/1080	1.504	6	699/3240	3.403
program2-a	9	2	31/45	0.496	6	1536/3240	8.187
program3	10	5	65/720	1.488	7	1051/21,600	22.55
warming	19	12	24,350/419,904	34.74	16	59,209/136,048,896	Timeout
warming-a	19	7	492/1944	0.695	16	59,209/136,048,896	Timeout
easy2-1	15	10	1250/58,320	17.35	12	7215/874,800	Timeout
Mysore-1	9	4	40/360	2.109	6	1916/108,000	Timeout
TCASv.1	118	44	52,476/1.3×10^{15}	Timeout	115	51,876/4.7×10^{24}	Timeout

on finding differences between the failing and successful executions, by defining heuristic metric on executions to identify locations that separate them [11]. The logic formula-based fault localization approaches [8,10,12] have been very successful in practice. They represent an error trace using an SMT formula and analyze it to find suspicious locations. 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. We refer to our technique for whole programs and all inputs as a *full fault localization*. The closest to our approach for inferring error invariants applied to fault localization is the work proposed by Ermis et al. [8,10]. 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` [12]. 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 for the fault. Other logic formula-based approaches include using weakest preconditions [9], and syntactic information in the form of graphs for static and dynamic dependency relations [7] to localize the errors.

Rival [33] uses abstract interpretation static analyzer ASTREE [3] 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 [34] 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 [35] also uses forward–backward analyses to estimate the probability that a target assertion is satisfied/violated.

Program slicing. Program slicing [36] is a well-known technique that extracts from programs the statements relevant to a given behavior. It has been successfully applied in various areas of software engineering including program debugging. The program slice introduced by Weiser [36] is defined as an executable subset of program statements that preserves the behavior of the original program at a particular program label l for a subset of program variables V, called the slicing criterion $\langle l, V \rangle$. Various program slicing techniques have been defined [36–38] by using the notion of Program Dependency Graphs (PDGs), which make explicit both the data and control dependences for each operation in a program that are based on syntactic presence of a variable in the definition of another variable or on a conditional expression. By introducing the notions of semantic data and control dependences [39], more precise semantics-based PDGs are computed that do not contain false dependences from the traditional syntactic PDGs. For example, although the expression $e = x - x$ syntactically depends on x, semantically there is no dependence. The semantics dependency can be lifted to an abstract domain where dependences are computed with respect to some specific abstract properties rather than concrete values, thus giving rise to so-called abstract semantics-based slicing [39]. Fault localization aims to solve a more specific problem compared to program slicing by identifying program statements relevant for the assertion failure. This way, we can employ more precise program analysis techniques that are specifically tailored for the given problem, and so obtain more precise results/slices.

Decision-tree abstract domains. Decision-tree domains have been used in abstract interpretation community recently [25,26,40]. Segmented decision tree abstract domains have enabled path dependent static analysis [25,40]. Their elements contain decision nodes that are determined either by values of program variables [40] or by the `if` conditions [25], whereas the leaf nodes are numerical properties. Urban and Miné [26] use decision tree abstract domains to prove program termination. Decision nodes are labeled with linear constraints that split the memory space and leaf nodes contain affine ranking functions for proving termination. Recently, specialized decision tree lifted domains [27,41–44] have been proposed to analyze program families (or any other configurable software system - Software Product Line) [45–47]. 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 [27] 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 [43,44], as well as dynamic program families with features changing during run-time [42].

Fault localization for program repair. Automated program repair has been recently considered as a way for efficient maintenance of software systems [6,15–17]. These works aim to repair the buggy program, so that the transformed program does not exhibit any faults. Some of the most successful approaches for automated program repair use formal techniques to guide the repair process. SemFix [6] uses symbolic execution to find a repair constraint and then generates a correct fix based on it, while Maple [17] uses a formal verification system to locate buggy expressions that are replaced with templates (linear expressions of program variables with unknown coefficients) in which the unknown coefficients are determined using constraint solving. The work [15] uses a deductive synthesis framework for repairing recursive functional programs with respect to specifications expressed in the form of pre- and post-conditions. Finally, Rothenberg and Grumberg [16] have developed the AllRepair tool for automatic program repair based on code mutations.

Automated program repair has often been combined with fault localization [7,48]. Once a set of statements relevant for the error has been found, we need to replace those statements in order to fix the error. However, the above works use syntactic information in the form of graphs for static and dynamic dependency relations to localize the errors. In contrast, we use a pure semantics-based technique via abstract semantic analyses to identify statements relevant for the error. This way, we can find irrelevant statements with respect to some subtle errors.

Program repair is also related to program sketching, where a program with missing parts (holes) has to be completed in such a way that a given specification is satisfied. Recently, abstract interpretation has been successfully applied to program sketching [49–51]. 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. We can combine our approach for fault localization with the techniques for program sketches to develop novel procedures for repairing subtle semantic errors.

Possible extensions. The current implementation of `FaultLoc` supports an interesting subset of C that includes numerical programs. This is due to the fact that our tool uses numerical abstract domains from the APRON library in practice. However, our `FaultLoc` algorithm can take any abstract domain $\mathbb{D}$ as parameter. Hence, it can be integrated with various abstract domains that can handle other language constructs. For example, if we use abstract domains for bit-vectors [52] and strings [53] then we can handle bit- and string-manipulating programs, while if we use abstract domains for heaps [54] then we can handle object-oriented programs and other static programming languages like Java. Moreover, since static analysis by abstract interpretation and several abstract domains have been developed for the dynamic programming language Python [55], we can use those abstract domains to adapt our algorithm for performing fault localization of Python.

The `AllRepair` tool [16] for program repair is based on the CBMC bounded model checker [2] that translates C programs to SMT formulas. JBMC [56] is a version of the CBMC for verifying Java bytecode by translating Java bytecode to SMT formulas. This way, a new version of `AllRepair` can be built on top of the JBMC for program repair of Java.

8. 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. We have seen that the slice contains potential error statements for the error program and a fix would be to change some of them. Our evaluation demonstrates that our algorithm provides useful error explanations, and so it can help to understand the cause of an error more easily. Moreover, we have integrated our approach for fault localization with the mutation-based program repair. Namely, the fault localization returns a set of statements that are likely to be wrong, whereas the mutation-based program repair finds potential replacements to these statements that fixes the error.

CRediT authorship contribution statement

Aleksandar S. Dimovski: Writing – review & editing, Writing – original draft, Software, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data is publically available (link is in the paper)

References

[1] W.E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Trans. Softw. Eng. 42 (8) (2016) 707–740, URL https://doi.org/10.1109/TSE.2016.2521368.

[2] E.M. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: Tools and Algorithms for the Construction and Analy. of Systems, 10th International Conference, TACAS 2004, Proceedings, in: LNCS, vol. 2988, Springer, 2004, pp. 168–176, URL https://doi.org/10.1007/978-3-540-24730-2_15.

[3] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, The astreé analyzer, in: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings, in: LNCS, vol. 3444, Springer, 2005, pp. 21–30, URL https://doi.org/10.1007/978-3-540-31987-0_3.

[4] B. Yin, L. Chen, J. Liu, J. Wang, P. Cousot, Verifying numerical programs via iterative abstract testing, in: Static Analysis - 26th International Symposium, SAS 2019, Proceedings, in: LNCS, vol. 11822, Springer, 2019, pp. 247–267, URL https://doi.org/10.1007/978-3-030-32304-2_13.

[5] A.S. Dimovski, R. Lazic, Compositional software verification based on game semantics and process algebra, Int. J. Softw. Tools Technol. Transf. 9 (1) (2007) 37–51, URL https://doi.org/10.1007/s10009-006-0005-y.

[6] H.D.T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, SemFix: program repair via semantic analysis, in: 35th International Conference on Software Engineering, ICSE '13, IEEE Computer Society, 2013, pp. 772–781, URL https://doi.org/10.1109/ICSE.2013.6606623.

[7] B. Rothenberg, O. Grumberg, Must fault localization for program repair, in: Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II, in: LNCS, vol. 12225, Springer, 2020, pp. 658–680, URL https://doi.org/10.1007/978-3-030-53291-8_33.

[8] J. Christ, E. Ermis, M. Schäf, T. Wies, Flow-sensitive fault localization, in: Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013. Proceedings, in: LNCS, vol. 7737, Springer, 2013, pp. 189–208, URL https://doi.org/10.1007/978-3-642-35873-9_13.

[9] M. Christakis, M. Heizmann, M.N. Mansur, C. Schilling, V. Wüstholz, 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, in: LNCS, vol. 11427, Springer, 2019, pp. 226–243, URL https://doi.org/10.1007/978-3-030-17462-0_13.

[10] E. Ermis, M. Schäf, T. Wies, Error invariants, in: FM 2012: Formal Methods - 18th International Symposium, 2012. Proceedings, in: LNCS, vol. 7436, Springer, 2012, pp. 187–201, URL https://doi.org/10.1007/978-3-642-32759-9_17.

[11] A. Groce, S. Chaki, D. Kroening, O. Strichman, Error explanation with distance metrics, Int. J. Softw. Tools Technol. Transf. 8 (3) (2006) 229–247, URL https://doi.org/10.1007/s10009-005-0202-0.

[12] M. Jose, R. Majumdar, Cause clue clauses: error localization using maximum satisfiability, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, ACM, 2011, pp. 437–446, URL https://doi.org/10.1145/1993498.1993550.

[13] P. Cousot, R. Cousot, 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, ACM, 1977, pp. 238–252, URL http://doi.acm.org/10.1145/512950.512973.

[14] A. Miné, Tutorial on static inference of numeric invariants by abstract interpretation, Found. Trends Program. Lang. 4 (3–4) (2017) 120–372, URL https://doi.org/10.1561/2500000034.

[15] E. Kneuss, M. Koukoutos, V. Kuncak, Deductive program repair, in: Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II, in: LNCS, vol. 9207, Springer, 2015, pp. 217–233, URL https://doi.org/10.1007/978-3-319-21668-3_13.

[16] B. Rothenberg, O. Grumberg, Sound and complete mutation-based program repair, in: FM 2016: Formal Methods - 21st International Symposium, Proceedings, in: LNCS, vol. 9995, 2016, pp. 593–611, URL https://doi.org/10.1007/978-3-319-48989-6_36.

[17] T. Nguyen, Q. Ta, W. Chin, Automatic program repair using formal verification and expression templates, in: Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Proceedings, in: LNCS, vol. 11388, Springer, 2019, pp. 70–91, URL https://doi.org/10.1007/978-3-030-11245-5_4.

[18] B. Jeannet, A. Miné, Apron: A library of numerical abstract domains for static analysis, in: Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings, in: LNCS, vol. 5643, Springer, 2009, pp. 661–667, URL https://doi.org/10.1007/978-3-642-02658-4_52.

[19] B. Jeannet, Relational interprocedural verification of concurrent programs, in: Seventh IEEE Inter. Conf. on Software Engineering and Formal Methods, SEFM'09, IEEE Computer Society, 2009, pp. 83–92, URL https://doi.org/10.1109/SEFM.2009.29.

[20] L.M. de Moura, N.B. rner, Z3: an efficient SMT solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings, in: LNCS, vol. 4963, Springer, 2008, pp. 337–340, URL https://doi.org/10.1007/978-3-540-78800-3_24.

[21] A.S. Dimovski, Error invariants for fault localization via abstract interpretation, in: Static Analysis - 30th International Symposium, SAS 2023, Proceedings, in: LNCS, vol. 14284, Springer, 2023, pp. 190–211, URL https://doi.org/10.1007/978-3-031-44245-2_10.

[22] F. Bourdoncle, Abstract debugging of higher-order imperative languages, in: Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), 1993, ACM, 1993, pp. 46–55, URL https://doi.org/10.1145/155090.155095.

[23] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Conference Record of the Fifth Annual ACM Symposium on POPL'78, ACM Press, 1978, pp. 84–96, URL https://doi.org/10.1145/512760.512770.

[24] W.R. Harris, S. Sankaranarayanan, F. Ivancic, A. Gupta, 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, ACM, 2010, pp. 71–82, URL https://doi.org/10.1145/1706299.1706309.

[25] J. Chen, P. Cousot, A binary decision tree abstract domain functor, in: Static Analysis - 22nd International Symposium, SAS 2015, Proceedings, in: LNCS, vol. 9291, Springer, 2015, pp. 36–53, URL https://doi.org/10.1007/978-3-662-48288-9_3.

[26] C. Urban, A. Miné, A decision tree abstract domain for proving conditional termination, in: Static Analysis - 21st International Symposium, SAS 2014. Proceedings, in: LNCS, vol. 8723, Springer, 2014, pp. 302–318, URL https://doi.org/10.1007/978-3-319-10936-7_19.

[27] A.S. Dimovski, Lifted static analysis using a binary decision diagram abstract domain, in: Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019, ACM, 2019, pp. 102–114, URL https://doi.org/10.1145/3357765.3359518.

[28] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. 35 (8) (1986) 677–691, URL https://doi.org/10.1109/TC.1986.1676819.

[29] A. Dimovski, D. Gligoroski, Generating highly nonlinear boolean functions using a genetic algorithm, in: 6th Int. Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Service, TELSIKS 2003, in: IEEE, vol. 2, 2003, pp. 604–607, URL https://doi.org/10.1109/TELSKS.2003.1246297.

[30] M.H. Liffiton, J.C. Maglalang, A cardinality solver: More expressive constraints for free - (poster presentation), in: Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Proceedings, in: LNCS, vol. 7317, Springer, 2012, pp. 485–486, URL https://doi.org/10.1007/978-3-642-31612-8_47.

[31] T.L. Graves, M.J. Harrold, J. Kim, A.A. Porter, G. Rothermel, An empirical study of regression test selection techiques, ACM Trans. Softw. Eng. Methodol. 10 (2) (2001) 184–208, URL https://doi.org/10.1145/367008.367020.

[32] S.A.A. Touati, J. Worms, S. Briais, The speedup-test: a statistical methodology for programme speedup analysis and computation, Concurr. Comput. Pract. Exp. 25 (10) (2013) 1410–1426, URL https://doi.org/10.1002/cpe.2939.

[33] X. Rival, Understanding the origin of alarms in astrée, in: Static Analysis, 12th International Symposium, SAS 2005, Proceedings, in: LNCS, vol. 3672, Springer, 2005, pp. 303–319, URL https://doi.org/10.1007/11547662_21.

[34] A. Miné, Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions, Sci. Comput. Program. 93 (2014) 154–182, URL https://doi.org/10.1016/j.scico.2013.09.014.

[35] A.S. Dimovski, A. Legay, Computing program reliability using forward-backward precondition analysis and model counting, in: Fundamental Approaches To Software Engineering - 23rd International Conference, FASE 2020, Proceedings, in: LNCS, vol. 12076, Springer, 2020, pp. 182–202, URL https://doi.org/10.1007/978-3-030-45234-6_9.

[36] M.D. Weiser, Program slicing, IEEE Trans. Softw. Eng. 10 (4) (1984) 352–357, URL https://doi.org/10.1109/TSE.1984.5010248.

[37] S. Horwitz, T.W. Reps, D.W. Binkley, Interprocedural slicing using dependence graphs, ACM Trans. Program. Lang. Syst. 12 (1) (1990) 26–60, URL https://doi.org/10.1145/77606.77608.

[38] S. Sinha, M.J. Harrold, G. Rothermel, System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow, in: B.W. Boehm, D. Garlan, J. Kramer (Eds.), Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, ACM, 1999, pp. 432–441, URL https://doi.org/10.1145/302405.302675.

[39] I. Mastroeni, D. Nikolic, Abstract program slicing: From theory towards an implementation, in: 12th International Conference on Formal Engineering Methods, ICFEM 2010. Proceedings, in: LNCS, vol. 6447, Springer, 2010, pp. 452–467, URL https://doi.org/10.1007/978-3-642-16901-4_30.

[40] P. Cousot, R. Cousot, L. Mauborgne, A scalable segmented decision tree abstract domain, in: Time for Verification, Essays in Memory of Amir Pnueli, in: LNCS, vol. 6200, Springer, 2010, pp. 72–95, URL https://doi.org/10.1007/978-3-642-13754-9_5.

[41] A.S. Dimovski, C. Brabrand, A. Wasowski, Finding suitable variability abstractions for lifted analysis, Formal Aspects Comput. 31 (2) (2019) 231–259, URL https://doi.org/10.1007/s00165-019-00479-y.

[42] A.S. Dimovski, S. Apel, Lifted static analysis of dynamic program families by abstract interpretation, in: 35th European Conference on Object-Oriented Programming, ECOOP 2021, in: LIPIcs, vol. 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 14:1–14:28, URL https://doi.org/10.4230/LIPIcs.ECOOP.2021.14.

[43] A.S. Dimovski, S. Apel, A. Legay, Several lifted abstract domains for static analysis of numerical program families, Sci. Comput. Program. 213 (2022) 102725, URL https://doi.org/10.1016/j.scico.2021.102725.

[44] A.S. Dimovski, Lifted termination analysis by abstract interpretation and its applications, in: GPCE '21: Concepts and Experiences, October, 2021, ACM, 2021, pp. 96–109, URL https://doi.org/10.1145/3486609.3487202.

[45] A.S. Dimovski, Symbolic game semantics for model checking program families, in: Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings, in: LNCS, vol. 9641, Springer, 2016, pp. 19–37.

[46] A.S. Dimovski, A. Wasowski, From transition systems to variability models and from lifted model checking back to UPPAAL, in: Models, Algorithms, Logics and Tools, in: LNCS, vol. 10460, Springer, 2017, pp. 249–268, URL http://dx.doi.org/10.1007/978-3-319-63121-9_13.

[47] A.S. Dimovski, CTL* family-based model checking using variability abstractions and modal transition systems, Int. J. Softw. Tools Technol. Transf. 22 (1) (2020) 35–55, URL https://doi.org/10.1007/s10009-019-00528-0.

[48] V. Debroy, W.E. Wong, Using mutation to automatically suggest fixes for faulty programs, in: Third International Conference on Software Testing, Verification and Validation, ICST 2010, IEEE Computer Society, 2010, pp. 65–74, URL https://doi.org/10.1109/ICST.2010.66.

[49] A.S. Dimovski, S. Apel, A. Legay, Program sketching using lifted analysis for numerical program families, in: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings, in: LNCS, vol. 12673, Springer, 2021, pp. 95–112, URL https://doi.org/10.1007/978-3-030-76384-8_7.

[50] A.S. Dimovski, Quantitative program sketching using decision tree-based lifted analysis, J. Comput. Lang. 75 (2023) 101206, URL https://doi.org/10.1016/j.cola.2023.101206.

[51] A.S. Dimovski, Generalized program sketching by abstract interpretation and logical abduction, in: Static Analysis - 30th International Symposium, SAS 2023, Proceedings, in: LNCS, vol. 14284, Springer, 2023, pp. 212–230, URL https://doi.org/10.1007/978-3-031-44245-2_11.

[52] Y. Yoon, W. Lee, K. Yi, Inductive program synthesis via iterative forward-backward abstract interpretation, in: PLDI '23: 44th ACM SIGPLAN International Conference on PLDI, 2023, ACM, 2023, pp. 1657–1681, URL https://doi.org/10.1145/3591288.

[53] M. Journault, A. Miné, A. Ouadjaout, Modular static analysis of string manipulations in c programs, in: Proc. of the 25th International Static Analysis Symposium, SAS'18, in: LNCS, vol. 11002, Springer, 2018, pp. 243–262, http://dx.doi.org/10.1007/978-3-319-99725-4_16.

[54] G. Balakrishnan, T.W. Reps, Recency-abstraction for heap-allocated storage, in: K. Yi (Ed.), Static Analysis, 13th International Symposium, SAS 2006, Proceedings, in: LNCS, vol. 4134, Springer, 2006, pp. 221–239, URL https://doi.org/10.1007/11823230_15.

[55] R. Monat, A. Ouadjaout, A. Miné, Static type analysis by abstract interpretation of python programs, in: Proc. of the 34th European Conference on Object-Oriented Programming, ECOOP'20, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, Dagstuhl Publishing, 2020, pp. 17:1–17:29, http://dx.doi.org/10.4230/LIPIcs.ECOOP.2020.17.

[56] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, M. Trtik, JBMC: A bounded model checking tool for verifying Java bytecode, in: Computer Aided Verification, CAV, in: LNCS, vol. 10981, Springer, 2018, pp. 183–190.