Variability Fault Localization by Abstract Interpretation and its Application to SPL Repair

Aleksandar S. Dimovski
aleksandar.dimovski@unt.edu.mk
Mother Teresa University
Skopje, North Macedonia

Abstract

Fault localization is an important step in software debugging that aims to isolate and localize the bugs (errors) to a small part of the program. This becomes more challenging in Software Product Lines (SPLs) due to the variable nature of bugs (so-called variability bugs). This paper introduces a novel variability fault localization algorithm for SPLs. Moreover, we present its practical application for automatic repair of variability bugs in SPLs.

Given a buggy SPL (program family) and an assertion to be satisfied, our variability fault localization algorithm infers so-called lifted (variability-specific) error invariants in all locations, which over-approximate the reachable states at the given locations that may produce the bug if the execution continues from those locations in some particular variants. The lifted error invariants are used for statement-wise lifted semantic slicing of buggy SPLs, which removes the statements irrelevant for assertion violation in certain variants containing some combinations of features. Finally, the obtained slice family is applied for designing an efficient SPL repair algorithm, which repeatedly mutates the (feature and program) expressions in the statements identified as relevant for the variability bug until a correct SPL is found.

We have implemented an experimental tool for variability fault localization and repair of `#ifdef`-annotated SPLs written in C. We have evaluated our approach on a set of benchmarks, and the experimental results confirm the effectiveness of our approach.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; *Software creation and management*; • **Theory of computation** → Semantics and reasoning.

Keywords: Variability Fault Localization, SPL repair, Lifted Error Invariants, Code Mutations.

ACM Reference Format:
Aleksandar S. Dimovski. 2025. Variability Fault Localization by Abstract Interpretation and its Application to SPL Repair. In *18th ACM SIGPLAN International Conference on Software Language Engineering (SLE), 12-13 June, 2025, Koblenz, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579027.3608976

1 Introduction

Software Product Line Engineering (SPLE) [10] represents an efficient method for building families of similar programs. Implementations of such *program families* use Boolean variables called *features* (or statically configured options) to organize the variable functionality. Family members, called *variants*, are specified in terms of features selected/enabled for that particular variant. The reuse of code common to multiple variants is thus maximized. Although the concept of program families (SPLs) has been theoretically introduced in 1970's by Parnas [38], their popularity has grown in the last two decades, especially in some domains, such as cars, avionics, health, etc [10].

Various verification and static analysis techniques [9, 12] are today often used to detect bugs in real-world single programs. In the last decade, many of those techniques have been lifted to the SPL systems by designing specialized so-called lifted (variability-aware) algorithms [3, 5, 15, 26, 41], which allow simultaneous verification/analysis of all variants of a program family in a single run by exploiting the commonalities between the variants. However, the verification tools usually return an error trace, which shows how the assertion is violated. The programmers still need to process it by using some debugging techniques in order to localize the bug to a relatively small part of the code and subsequently to repair the given program/program family. Therefore, automated fault localization [8, 27, 32, 40] and program repair [35, 37, 39] are important problems in program debugging, which can significantly improve the software development cycle. Still, the standard fault localization and repair algorithms cannot be directly applied to program families as they can only analyze pre-processed single programs. Moreover, automatic debugging in program families is harder than in single programs, because the so-called *variability bugs* [36] may only be exposed in certain variants under some selections of features (feature expressions), but not in the other variants. Therefore, statements can be relevant for a variability bug only in some variants, but not in the others. This variability property is the source of substantial difficulties for localizing and repairing variability bugs in SPLs.

In this work, we lift an existing single-program fault localization algorithm [17, 20] based on error invariants inferred via abstract interpretation [11, 34] to program families, by adapting it to the SPL settings. The fault localization algorithm for single programs [17, 20] uses an iterative refining sequence of backward-forward abstract analyses to infer *error invariants*, which capture memory stores in given locations that may produce the bug in a program. The backward abstract analysis is used for inferring the necessary preconditions (i.e., error invariants) of violating the target assertion, while the subsequent forward abstract analysis is employed to refine those error invariants in all locations. In the first iteration, the analyses are performed on a base (numerical) abstract domain (e.g., intervals, octagons, polyhedra). When there is a precision loss in the backward analysis caused by merging the branches of an `if` statement, the guard of that `if` statement is used as a predicate (decision node) in a BDD abstract domain, which is employed in the analyses of the next iteration. Abstract elements from the BDD domain can express disjunctive properties with respect to the set of predicates used in their decision nodes. This way, the subsequent iterations continue by performing backward-forward analyses on refined BDD abstract domains, which enable them to produce more precise (refined) error invariants.

In this paper, we introduce a variability fault localization algorithm for SPL debugging that operates on the entire program family in one single pass and infers so-called lifted (variability-specific) error invariants that hold only under some combinations of features. The key for the efficiency of the lifted algorithm is the use of a lifted analysis domain that enables an explicit interaction (sharing) between error invariants corresponding to different variants. That is, the elements of the lifted analysis domain are *lifted BDDs*, in which the (first-level) decision nodes are (Boolean) feature variables and the leaf nodes are program properties (error invariants) from base abstract domains. Thus, an error invariant (represented in a leaf node) holds only for variants specified by the values of features along the path leading to the given leaf. However, after the first iteration, we work with so-called lifted two-level BDDs, in which the first level decision nodes are feature variables, the second-level decision nodes are predicates defined over program variables (i.e., guards of `if`-s in which the analysis precision loss is observed), and the leaf nodes are program properties (error invariants). In fact, the first-level decision nodes recursively partition the space of all possible valid variants $\mathcal{K}$, while the second-level decision nodes recursively partition the memory store space. The inferred lifted error invariants can be used to detect features that need to be selected on/off together to make some statements relevant for a variability bug. The obtained variability-specific statements relevant for a bug constitute the so-called *(semantic) slice family* of the buggy SPL, which can be changed to make the SPL correct.

Additionally, we leverage the variability fault localization algorithm to automatically repair program families. SPL repair is defined to be a code transformation, where the transformed repaired SPL satisfies the given assertion. The SPL repair is a difficult problem, since the repair of one variant may cause new bugs to appear in other variants due to the feature interaction in the given SPL [2]. In this paper, we combine our variability fault localization algorithm and the mutation-based SPL repair [21]. Given a buggy SPL, our variability fault localization algorithm first suggests variability-specific statements that are relevant for the given variability bug, and the mutation-based SPL repair algorithm then attempts to change (mutate) feature and program expressions only in those relevant statements in order to eliminate the bug. The mutation-based SPL repair algorithm [21] generates mutated program families (mutants) by using a predefined set of syntactic changes (mutations) applied to original program code. For example, we can apply some syntactic changes to features (e.g., by applying negation $\neg$), arithmetic operators (e.g., by replacing `+` by `-`), Boolean operators (e.g., by replacing $\land$ by $\lor$), 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. By applying only mutations to variability-specific statements relevant for the bug as identified by the variability fault localization, we significantly narrow down the search space of all possible mutants, thus speeding up the SPL repair algorithm in [21] without any precision loss.

We have implemented our approach for variability fault localization of C program families, called `SPLFaultLoc`, which is built on top of the fault localizer [17, 20]. The tool uses the numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library [31], and the BDD abstract domains from the BDDAPRON library [30]. The tool also calls the Z3 SMT solver [14] for computing (lifted) error invariants. Moreover, we have integrated our variability fault localization tool with the SPLAllRepair tool [21] for repairing C program families. Finally, we evaluate our approach on a selected set of benchmarks from SV-COMP suite.

In summary, we make several contributions in this work:

(1) We present a novel algorithm for localizing variability bugs in SPLs;
(2) We show how our variability fault localization can be incorporated into a mutation-based SPL repair algorithm for reducing its mutant's search space;
(3) We implement the proposed approaches for fault localization and repair of SPLs in prototype tools and evaluate them on `#ifdef`-based C benchmarks.

```
void main (int n){
(A ∧ ⊤) ∨ (¬A ∧ ⊥)
① int z = n + 1;
(A ∧ (z=n+1)) ∨ (¬A ∧ ⊥)
② int y = n − 1;
(A ∧ (z=n+1)) ∨ (¬A ∧ ⊥)
③ #if (A) y = z; #endif
(A ∧ (y=n+1)) ∨ (¬A ∧ ⊥)
④ #if (¬B ∧ C) z = z + 1; #endif
(A ∧ (y=n+1)) ∨ (¬A ∧ ⊥)
⑤ assert (y < n);
```

Figure 1. intro.c.

```
A = [0, 1];
B = [0, 1];
C = [0, 1];
if (A) then z = n + 1;
if (¬A) then z = n + 1;
y = n − 1;
if (A) then y = z;
if (¬B ∧ C) then z = z + 1;
assert (y < n);
```

Figure 2. Simulator of intro.c.

```
void main (int n){
(A ∧ ¬P ∧ n ≥ 42) ∨ (A ∧ P ∧ ⊥) ∨ (¬A ∧ ⊥)
① int #if (A) x = 1; #else x = 0; #endif;
(A ∧ ¬P ∧ n ≥ 42 ∧ x = 1) ∨ (A ∧ P ∧ ⊥) ∨ (¬A ∧ ⊥)
② int #if (A) y = n − 42; #endif;
(A ∧ ¬P ∧ n ≥ 42 ∧ x = 1 ∧ y = n − 42) ∨ (A ∧ P ∧ ⊥) ∨ (¬A ∧ ⊥)
③ if (y<0) then
④ (⊥) x = 0; (⊥)
⑤ else skip;
⑥ endif
(A ∧ ¬P ∧ x = 1) ∨ (A ∧ P ∧ ⊥) ∨ (¬A ∧ ⊥)
⑦ assert (x ≤ 0);
```

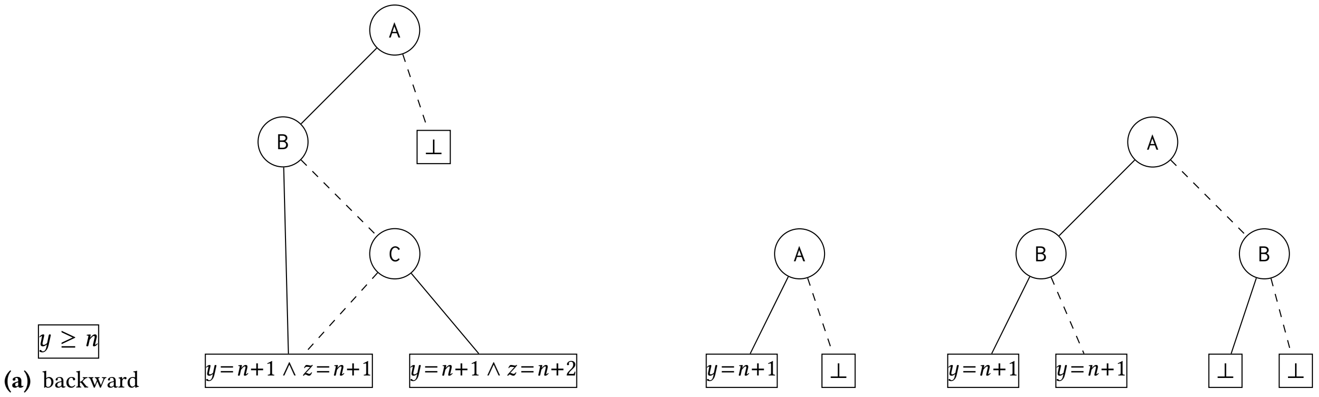
Figure 3. cond.c ($P \equiv (y<0)$).

2 Motivating Example

To illustrate the potential of our approach, we now present two motivating examples. The program family intro.c, shown in Fig. 1, consists of four sequentially composed assignments and #if-s followed by an assertion at loc. ⑤. The set of Boolean (optional) features is $\mathcal{F} = \{A, B, C\}$ and the set of valid configurations is $\mathcal{K} = \{A \land B \land C, \neg A \land B \land C, A \land \neg B \land C, \neg A \land \neg B \land C, A \land B \land \neg C, \neg A \land B \land \neg C, A \land \neg B \land \neg C, \neg A \land \neg B \land \neg C\}$. There is one variant (single program) corresponding to each configuration from $\mathcal{K}$, which is generated by appropriately resolving #if directives. For example, the variant for configuration $(A \land B \land C)$ will have all features A, B and C selected on (set to *true*) so that statements at locs. ③ and ④ become: $y = z$ and skip, whereas the variant for $(\neg A \land \neg B \land C)$ will have both features A and B selected off (set to *false*) and C set to *true* so that statements at locs. ③ and ④ become: skip and $z = z + 1$. In such program families, it may happen that bugs (e.g., assertion violation) are variable (variability bugs), i.e. they may occur in some variants but not in the others. In the case of intro.c, the variants $(A \land B \land C)$, $(A \land \neg B \land C)$, $(A \land B \land \neg C)$ and $(A \land \neg B \land \neg C)$ are buggy since the invariant $y = n + 1$ holds at loc. ⑤ and so the assertion $(y < n)$ is violated. On the other hand, the variants $(\neg A \land B \land C)$, $(\neg A \land \neg B \land C)$, $(\neg A \land B \land \neg C)$ and $(\neg A \land \neg B \land \neg C)$ are correct since the invariant $y = n - 1$ holds at loc. ⑤ and so the assertion $(y < n)$ is always valid. Therefore, our aim is to find the root cause of the variability bug in the form of features and statements relevant for it, and then generate a correct fix.

Our variability fault localization procedure, SPLFaultLoc, starts with two lifted backward-forward analyses of the intro.c family, which are performed on a lifted BDD abstract domain defined over the Polyhedra leaf domain and the Boolean features A, B, C for decision nodes. The lifted backward analysis starts by propagating the negated assertion $(y \geq n)$ from loc. ⑤ backwards, thus inferring the preconditions: $(y \geq n)$ at loc. ④, $(A \land (z \geq n)) \lor (\neg A \land (y \geq n))$ at loc. ③, $(A \land (z \geq n)) \lor (\neg A \land \bot)$ at loc. ②, and $(A \land \top) \lor (\neg A \land \bot)$ at loc. ①. The inferred lifted preconditions at all locations are over-approximations of memory stores at those locations that may lead to assertion violation $(y \geq n)$ if the program is continued from them. For example, the precondition $(A \land \top) \lor (\neg A \land \bot)$ at loc. ① states that for variants satisfying A the assertion violation can be reached from any memory store (i.e., $\top \equiv$ *true*) at loc. ①, while for variants satisfying ¬A the assertion violation cannot be reached from any memory store (i.e., $\bot \equiv$ *false*) at loc. ①. Note also that the lifted preconditions are disjunctive properties represented as BDDs. For example, the BDD $(y \geq n)$ at loc. ⑤ is shown in Fig. 4a, where the implicit decision node $true \equiv \lor_{k \in \mathcal{K}} k$ means that the fact $(y \geq n)$ holds for all variants in $\mathcal{K}$. The subsequent lifted forward analysis starts by propagating the invariant $(A \land \top) \lor (\neg A \land \bot)$ from loc. ① forward, thus inferring the invariants: $(A \land (z = n + 1)) \lor (\neg A \land \bot)$ at loc. ②, $(A \land (z = n + 1) \land (y = n - 1)) \lor (\neg A \land \bot)$ at loc. ③, $(A \land (z = n + 1) \land (y = n + 1)) \lor (\neg A \land \bot)$ at loc. ④, and $\big((A \land B) \lor (A \land \neg B \land \neg B) \land (z = n+1) \land (y = n+1)\big) \lor (A \land \neg B \land (z = n + 2) \land (y = n + 1)) \lor (\neg A \land \bot)$ at loc. ⑤ (see Fig. 4b for graphical representation). The results of backward analysis are refined using the results of forward analysis to compute more precise error invariants. For example, the backward precondition $(y \geq n)$ at loc. ⑤ is refined using the forward invariant $\big((A \land B) \lor (A \land \neg B \land \neg C) \land (z = n+1) \land (y = n+1)\big) \lor (A \land \neg B \land C \land (z = n+2) \land (y = n+1)) \lor (\neg A \land \bot)$, thus generating the error invariant $\boxed{(A \land (y = n + 1)) \lor (\neg A \land \bot)}$ for loc. ⑤ (see Fig. 4c for graphical representation). The computed lifted error invariants after the first iteration of backward-forward analyses are shown in Fig. 1 using boxed code, such as $\boxed{(A \land (y = n + 1)) \lor (\neg A \land \bot)}$ for loc. ⑤. They represent the over-approximations of the memory stores at given locations that may produce the bug.

By analyzing the lifted error invariants, we can determine the features and statements relevant for the variability bug. That is, we observe that statements enclosed by the same error invariants do not change the nature of the bug and



(a) backward anal. **(b)** forward analysis **(c)** error invariant as BDD **(d)** error invariant as BDT

Figure 4. Inferred BDDs abd BDTs at loc. ⑤ (solid edges = *true*, dashed edges = *false*).

so they are classified as irrelevant for the bug. Since the variants satisfying ¬A are correct (the error invariants ⊥ are inferred at all locations), we can conclude that all statements with feature A selected off are irrelevant for the variability bug and so the resulting program slice for ¬A is empty. For variants satisfying A, the error invariants before and after the statements at locs. ② and ④ are the same (see Fig. 1), so they can be dropped from the resulting program slice for A. Finally, the obtained slice family containing features and statements relevant for the variability bug in `intro.c` is:

① `#if (A)` $z = n + 1$; `#endif` ③ `#if (A)` $y = z$; `#endif`

In addition, we want to use the obtained slice family of relevant variability-specific statements for the bug to perform an automatic SPL repair. The `SPLAllRepair` tool [21] applies a pre-defined set of syntactic mutations to all (feature and program) expressions found in statements that can be changed (e.g., assertions cannot be changed), and explores the search space of all mutants. However, the space of mutants grows very rapidly as the number of statements that can be changed grows. Thus, we want to reduce the search space of mutants by only mutating expressions in statements relevant for the bug. For `intro.c`, this translates to only mutating expressions: A, $n + 1$ at loc. ①, and A, z at loc. ③, whereas all other expressions in irrelevant statements will not be mutated at all. Note that our repair model permits mutation of `#if`'s presence conditions (feature expressions) as well as right-hand sides of assignments, `if`'s and `while`'s guards (program expressions). Note that due to the obtained slice family, the statement $z = n + 1$ at loc. ① before repair is replaced with the semantically equivalent one: `int #if (A)` $z = n + 1$; `#endif; #if (¬A)` $z = n + 1$; `#endif`. We use the following pre-defined mutations: a feature can be negated; a variable/parameter can be replaced by any other parameter in the program; an integer constant can be increased or decreased by one, or replaced by zero; an operator from the sets $\{+, -, *, \%, \div\}$, $\{<, \leq, >, \geq\}$, and $\{\wedge, \vee\}$ can be replaced by any other operator from the same set. This way, if we mutate all (feature and program) expressions in `intro.c` we would obtain 27000 mutants in the original `SPLAllRepair` tool. However, if we combine variability fault localization with the `SPLAllRepair`, we will mutate only expressions A, n+1, A, and z, thus reducing the space of mutants to 120. Note that `SPLAllRepair` does not *explore* the space of all possible mutants in general. When a possible solution is found, then all supersets of the found solutions are not further explored. The explored search space and the running time have also been reduced from 16206 mutants and 53.47 sec using `SPLAllRepair` to only 78 mutants and 0.585 sec using `SPL(FaultLoc+AllRepair)`. Both approaches generate the same repaired program by replacing statement at ① with `#if (A)` $z = n - 1$; `#endif;#if (¬A)` $z = n + 1$;`#endif` via replacement of the program expression $n + 1$ with $n - 1$.

The program family `cond.c` in Fig. 3 violates the assertion $(x \leq 0)$ when the feature A is enabled and the parameter n is greater than 41. The first iteration of backward-forward analyses is performed on the BDD abstract domain with one decision node: feature A, and infers imprecise results due to the over-approximation when analyzing the `if` statement. The second backward-forward iteration is performed on the refined BDD abstract domain with two decision nodes: feature A and predicate $P \equiv (y < 0)$. This way, we can analyze the `if` statement more precisely, and so obtain more precise error invariants, as shown in Fig. 3 using boxed code, indicating that the body of `if` statement (locs. ④ - ⑥) is irrelevant for the bug. Hence, the obtained slice family is:

① `#if (A)` $x = 1$; `#endif` ② `#if (A)` $y = n - 42$; `#endif`
③ `#if (A) if (`y<0`) then skip; #endif`

Therefore, only expressions: A, 1 in loc. ①; A, $n - 42$ at loc. ②; and A, y<0 at loc. ③ will be mutated. As a result, the explored/total mutant search space and running time will be reduced from 9006/19440 mutants and 40.67 sec for `SPLAllRepair` to 241/368 mutants and 1.28 sec for the improved `SPL(FaultLoc+AllRepair)`. Both approaches repair the program by replacing statement at loc. ① with

`#if (A)` $x = 0$; `#endif`; `#if` $(\neg$`A`$)$ $x = 0$; `#endif` via replacement of the program expression 1 with 0.

3 A Language for Program Families

Syntax. The *features* in a program family are given by a finite and totaly ordered set of Boolean variables, $\mathcal{F} = \{A_1, \ldots, A_n\}$. A *configuration* $k : \mathcal{F} \to \{true, false\}$, which specifies a *variant* of a program family, is a truth assignment assigning a truth value to each feature. We write $\mathcal{K}$ for the set of all *valid* configurations. We can also represent a configuration $k \in \mathcal{K}$ by a formula: $(k(A_1) \cdot A_1 \wedge \ldots \wedge k(A_n) \cdot A_n)$, where $true \cdot A = A$ and $false \cdot A = \neg A$. A *feature expression* θ is a propositional logic formula defined over $\mathcal{F}$ as:

$$\theta\ (\theta \in FeatExp(\mathcal{F})) ::= true \,|\, A \in \mathcal{F} \,|\, \neg\theta \,|\, \theta \wedge \theta \,|\, \theta \vee \theta$$

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

$s\ (s \in Stm) ::=$ `skip` $|$ `x`$=ae$ $|$ $s;s$ $|$ `if` (be) `then` s `else` s $|$
`while` (be) `do` s $|$ `#if` (θ) s `#endif` $|$ `assert`(be) $|$ `assume`(be)
$ae\ (ae \in AExp) ::= n \mid [n, n'] \mid$ `x` $\mid ae \oplus ae$,
$be\ (be \in BExp) ::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be$

where $n \in \mathbb{Z}$, $\mathtt{x} \in Var$, $\oplus \in \{+, -, *, \%, /\}$, $\bowtie \in \{<, \leq, ==, !=\}$, and $[n, n']$ is an integer interval. The compile-time conditional statement "`#if` (θ) s `#endif`" is used for encoding multiple variants of a program family, such that the statement s will be included in the variant corresponding to configuration $k \in \mathcal{K}$ only if the presence condition θ is satisfied by k ($k \models \theta$). Other compile-time conditional statements from the C preprocessor `CPP` [33] can be desugared and represented by `#if`, e.g., `#if` (θ) s_0 `#else` s_1 `#endif` is translated to `#if` (θ) s_0 `#endif`; `#if` $(\neg\theta)$ s_1 `#endif`.

Without loss of generality, we assume that a program family P is a sequence of statements followed by a single assertion: $l_{in} : s; l_{ass} :$ `assert`(be^{ass}).

Semantics. A program family is evaluated in two phases. In the first phase, the C preprocessor `CPP` [33], specified by the projection function π_k, takes a program family (statement) s and a configuration $k \in \mathcal{K}$ as inputs, and produces a variant (single program without `#if`-s) corresponding to k as the output. The projection function π_k is an identity for all basic statements and recursively pre-processes all substatements of compound statements, i.e. $\pi_k(\mathtt{x}{=}ae) := \mathtt{x}{=}ae$ and $\pi_k(s;s') := \pi_k(s);\pi_k(s')$. For "`#if` (θ) s `#endif`", the statement s is included in the variant k if $k \models \theta$, i.e. we have:

$$\pi_k(\texttt{\#if } (\theta)\ s\ \texttt{\#endif}) := \begin{cases} \pi_k(s) & \text{if } k \models \theta \\ \texttt{skip} & \text{if } k \not\models \theta \end{cases}.$$

In the second phase, the obtained variants, $\pi_k(s)$ for $k \in \mathcal{K}$, are evaluated using the standard single-program semantics [17]. More specifically, a *memory store* $\sigma \in \Sigma = Var \to \mathbb{Z}$ is a mapping from program variables to values. The semantics of arithmetic $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$ and Boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{true, false\})$ are the sets of possible integer and Boolean values for expressions ae and be in the store σ. The *necessary precondition* (backward) semantics $\overleftarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ 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, whereas the *invariance* (forward) semantics $\overrightarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ 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.

We say that a program family $P \equiv l_{in} : s; l_{ass} :$ `assert`(be^{ass}) is *buggy* if there exists a configuration $k \in \mathcal{K}$, such that the assertion in $\pi_k(P)$ is not valid, i.e. $\overrightarrow{[\![\pi_k(s)]\!]}\Sigma \not\subseteq \{\sigma \in \Sigma \mid [\![be^{ass}]\!]\sigma = \{true\}\}$ for some $k \in \mathcal{K}$. Unlike the single programs, the bugs in program families can be *variable* and only cause the assertion failures in certain variants. On the other hand, we say that a program family P is *correct* if for all configurations $k \in \mathcal{K}$, the assertion in $\pi_k(P)$ is valid, i.e. $\overrightarrow{[\![\pi_k(s)]\!]}\Sigma \subseteq \{\sigma \in \Sigma \mid [\![be^{ass}]\!]\sigma = \{true\}\}$ for all $k \in \mathcal{K}$.

4 Variability Fault Localization

In this section, we begin with the definition of the lifted BDD abstract domain, followed with the definition of two lifted abstract analyses for program families: the necessary precondition (backward) and the invariance (forward) analyses. Finally, we present our iterative algorithm of abstract analyses for calculating lifted error invariants as well as a slice of the buggy program family relevant for the bug.

Lifted BDD abstract domain. The lifted binary decision diagram (BDD) abstract domain [15], denoted $\mathbb{D}(\mathcal{F} \cup \mathbb{P}, \mathbb{A})$, where $\mathcal{F}$ and $\mathbb{P}$ are sets of features and predicates (defined over program variables) and $\mathbb{A}$ is a base abstract domain (e.g., Polyhedra), is used for defining an efficient lifted abstract analysis. The abstract elements of the domain $\mathbb{D}(\mathcal{F} \cup \mathbb{P}, \mathbb{A})$ are disjunctions of leaf nodes that belong to the base $\mathbb{A}$, which are separated by the values of Boolean variables (features from $\mathcal{F}$ and predicates from $\mathbb{P}$) organized in decision nodes. Hence, the set of configurations $\mathcal{K}$ and the memory store space $\mathcal{P}(\Sigma)$ are partitioned with respect to the sets of features $\mathcal{F}$ and predicates $\mathbb{P}$ respectively, such that each top-down path of a BDD abstract element represents one or several configurations of $\mathcal{K}$ and partitionings of $\mathcal{P}(\Sigma)$, whereas the leaf node is the analysis property inferred for those configurations/variants and partitionings of $\mathcal{P}(\Sigma)$.

A *binary decision tree* (BDT) [7, 15] $t \in \mathbb{T}(\mathcal{F} \cup \mathbb{P}, \mathbb{A})$, defined over the set $\mathcal{F} \cup \mathbb{P}$ of features and predicates and the leaf abstract domain $\mathbb{A}$, is either a leaf node $\ll a \gg$ with $a \in \mathbb{A}$ and $\mathcal{F} = \mathbb{P} = \emptyset$, or $[\![B : tl, tr]\!]$, where B is *the smallest element* of $\mathcal{F} \cup \mathbb{P}$ 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 $tl, tr \in \mathbb{T}((\mathcal{F} \cup \mathbb{P}) \backslash \{B\}, \mathbb{A})$. Note that, $\mathcal{F} = \{A_1, \dots, A_n\}$ and $\mathbb{P} = \{P_1, \dots, P_m\}$ are totally ordered sets with ordering: $A_1 < \dots < A_n < P_1 < \dots < P_m$. The left and right subtrees are either both leafs or both rooted at decision nodes labeled with the same Boolean variable $B \in \mathcal{F} \cup \mathbb{P}$.

The BDTs contain redundancy, which can be eliminated by applying the following three optimizations [6]: (1) Removal of duplicate leaves; (2) Removal of redundant tests; (3) Removal of duplicate non-leaves. If reductions (**1**)-(**3**) are applied to a BDT $t \in \mathbb{T}(\mathcal{F} \cup \mathbb{P}, \mathbb{A})$ until no further reductions are possible, and moreover if the ordering on the Boolean variables from $\mathcal{F} \cup \mathbb{P}$ occurring on any path is fixed to the ordered list $[A_1, \dots, A_n, P_1, \dots, P_m]$, then a *reduced ordered binary decision diagram* (BDD) $b \in \mathbb{D}(\mathcal{F} \cup \mathbb{P}, \mathbb{A})$ is obtained [7, 15]. Note that the resulting BDDs have a canonical form, which means that any disjunctive property from the BDT domain can be represented in an unique way by a BDD. Sometimes we omit to write the parameters $\mathcal{F} \cup \mathbb{P}$ and $\mathbb{A}$ in $\mathbb{D}(\mathcal{F} \cup \mathbb{P}, \mathbb{A})$ when they are clear from the context.

Suppose that we have a base (single-program) abstract domain $(\mathbb{A}, \sqsubseteq_{\mathbb{A}})$, such that there exists a concretization-based abstraction $\langle \mathcal{P}(\Sigma), \subseteq \rangle \xleftarrow{\gamma_{\mathbb{A}}} \langle \mathbb{A}, \sqsubseteq_{\mathbb{A}} \rangle$ that works only with a concretization function $\gamma_{\mathbb{A}} : \mathbb{A} \to \mathcal{P}(\Sigma)$ expressing the meaning of abstract elements from $\mathbb{A}$ in terms of concrete elements from $\mathcal{P}(\Sigma)$. Concretization-based abstraction is a relaxation of the known Galois connection abstraction, which is more used in practice (e.g., Polyhedra domain). We also assume that the abstract domain $\mathbb{A}$ is equipped with sound operators for ordering $\sqsubseteq_{\mathbb{A}}$, least upper bound (join) $\sqcup_{\mathbb{A}}$, greatest lower bound (meet) $\sqcap_{\mathbb{A}}$, bottom $\bot_{\mathbb{A}}$, top $\top_{\mathbb{A}}$, widening $\nabla_{\mathbb{A}}$, and narrowing $\triangle_{\mathbb{A}}$, as well as sound transfer functions for forward assignments $\texttt{ASSIGN}_{\mathbb{A}} : \mathit{Stm} \times \mathbb{A} \to \mathbb{A}$, tests $\texttt{FILTER}_{\mathbb{A}} : \mathit{BExp} \times \mathbb{A} \to \mathbb{A}$, and backward assignments $\texttt{B-ASSIGN}_{\mathbb{A}} : \mathit{Stm} \times \mathbb{A} \to \mathbb{A}$. The abstract operations: $\gamma_{\mathbb{D}}, \sqsubseteq_{\mathbb{D}}, \sqcup_{\mathbb{D}}, \sqcap_{\mathbb{D}}, \bot_{\mathbb{D}}, \top_{\mathbb{D}}, \nabla_{\mathbb{D}}, \triangle_{\mathbb{D}}$, transfer functions: $\texttt{ASSIGN}_{\mathbb{D}}, \texttt{FILTER}_{\mathbb{D}}, \texttt{B-ASSIGN}_{\mathbb{D}}$, and the soundness of the domain $\mathbb{D}(\mathcal{F} \cup \mathbb{P}, \mathbb{A})$ are obtained by lifting the corresponding operations, transfer functions and the soundness of the leaf domain $\mathbb{A}$. The basic operation we use for this aim is $\texttt{apply}_2(\texttt{op}_{\mathbb{A}}, d_1, d_2)$ [6], which lifts any binary operation $\texttt{op}_{\mathbb{A}}$ from the domain $\mathbb{A}$ to BDDs, thus computing "$d_1 \texttt{op}_{\mathbb{D}}\, d_2$". For example, $d_1 \sqcup_{\mathbb{D}} d_2 = \texttt{apply}_2(\lambda(a_1, a_2).a_1 \sqcup_{\mathbb{A}} a_2, d_1, d_2)$. We refer to [7, 15] for more details. However, the assignment transfer functions, $\texttt{ASSIGN}_{\mathbb{D}}$ and $\texttt{B-ASSIGN}_{\mathbb{D}}$, need more care since their application on a leaf node in one partitioning of $\mathcal{P}(\Sigma)$ (i.e., one truth assignment to $\mathbb{P}$) 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* [7, 17]. We will also use the $\texttt{meet\_cond}(d, b)$ operation [15], which restricts the top-down paths (Boolean part) of the BDD d to those paths that satisfy the condition b.

$$
\begin{array}{|l|}
\hline
\overleftarrow{[\![\texttt{skip}]\!]}^{\sharp} d = d \\
\overleftarrow{[\![\texttt{x = }ae]\!]}^{\sharp} d = \texttt{B-ASSIGN}_{\mathbb{D}}(\texttt{x} = ae, d) \\
\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]}^{\sharp} d = \overleftarrow{[\![s_1]\!]}^{\sharp}(\overleftarrow{[\![s_2]\!]}^{\sharp} d) \\
\overleftarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}^{\sharp} d = \\
\quad \texttt{FILTER}_{\mathbb{D}}(be, \overleftarrow{[\![s_1]\!]}^{\sharp} d) \sqcup_{\mathbb{D}} \texttt{FILTER}_{\mathbb{D}}(\neg be, \overleftarrow{[\![s_2]\!]}^{\sharp} d) \\
\overleftarrow{[\![\texttt{\#if } \theta\ s \texttt{ \#endif}]\!]}^{\sharp} d = \\
\quad \texttt{meet\_cond}(\overleftarrow{[\![s]\!]}^{\sharp} d, \theta) \sqcup_{\mathbb{D}} \texttt{meet\_cond}(d, \neg\theta) \\
\overleftarrow{[\![\texttt{while } be \texttt{ do } s]\!]}^{\sharp} d = \texttt{lfp}^{\sharp}\, \overleftarrow{\phi}^{\sharp} \\
\overleftarrow{\phi}^{\sharp}(x) = \texttt{FILTER}_{\mathbb{D}}(\neg be, d) \sqcup_{\mathbb{D}} \texttt{FILTER}_{\mathbb{D}}(be, \overleftarrow{[\![s]\!]}^{\sharp} x) \\
\hline
\end{array}
$$

Figure 5. Lifted necessary precond. analysis $\overleftarrow{[\![s]\!]}^{\sharp} : \mathbb{D} \to \mathbb{D}$.

$$
\begin{array}{|l|}
\hline
\overrightarrow{[\![\texttt{skip}]\!]}^{\sharp} d = d \\
\overrightarrow{[\![\texttt{x = }ae]\!]}^{\sharp} d = \texttt{ASSIGN}_{\mathbb{D}}(\texttt{x} = ae, d) \\
\overrightarrow{[\![s_1 \texttt{ ; } s_2]\!]}^{\sharp} d = \overrightarrow{[\![s_2]\!]}^{\sharp}(\overrightarrow{[\![s_1]\!]}^{\sharp} d) \\
\overrightarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}^{\sharp} d = \\
\quad \overrightarrow{[\![s_1]\!]}^{\sharp}(\texttt{FILTER}_{\mathbb{D}}(be, d)) \sqcup_{\mathbb{D}} \overrightarrow{[\![s_2]\!]}^{\sharp}(\texttt{FILTER}_{\mathbb{D}}(\neg be, d)) \\
\overrightarrow{[\![\texttt{\#if } \theta\ s \texttt{ \#endif}]\!]}^{\sharp} d = \\
\quad \overrightarrow{[\![s]\!]}^{\sharp}(\texttt{meet\_cond}(d, \theta)) \sqcup_{\mathbb{D}} \texttt{meet\_cond}_{\mathbb{D}}(d, \neg\theta) \\
\overrightarrow{[\![\texttt{while } be \texttt{ do } s]\!]}^{\sharp} d = \texttt{FILTER}_{\mathbb{D}}(\neg be, \texttt{lfp}^{\sharp}\, \overrightarrow{\phi}^{\sharp}) \\
\overrightarrow{\phi}^{\sharp}(x) = d \sqcup_{\mathbb{D}} \overrightarrow{[\![s]\!]}^{\sharp}(\texttt{FILTER}_{\mathbb{D}}(be, x)) \\
\hline
\end{array}
$$

Figure 6. Lifted abstract invariance analysis $\overrightarrow{[\![s]\!]}^{\sharp} : \mathbb{D} \to \mathbb{D}$.

Example 4.1. Consider the BDT given in Fig. 4d. It has decision nodes labelled with features A and B, and leaves are Polyhedra constraints. In first order logic, it expresses the disjunctive formula:

$$(\mathsf{A} \wedge \mathsf{B} \wedge (y = n + 1)) \vee (\mathsf{A} \wedge \neg\mathsf{B} \wedge (y = n + 1)) \vee (\neg\mathsf{A} \wedge \mathsf{B} \wedge \bot) \vee (\neg\mathsf{A} \wedge \neg\mathsf{B} \wedge \bot)$$

After reducing this BDT, we obtain the BDD in Fig. 4c.

Lifted Abstract Analyses. We now define computable lifted abstract analyses that over-approximate the concrete necessary precondition $\overleftarrow{[\![s]\!]}$ and invariance $\overrightarrow{[\![s]\!]}$ semantics. They are designed to work directly on program families, rather than on single programs. The lifted abstract necessary precondition analysis $\overleftarrow{[\![s]\!]}^{\sharp} : \mathbb{D} \to \mathbb{D}$ and the lifted abstract invariance analysis $\overrightarrow{[\![s]\!]}^{\sharp} : \mathbb{D} \to \mathbb{D}$ are defined in Fig. 5 and Fig. 6, respectively. The lifted backward analysis $\overleftarrow{[\![s]\!]}^{\sharp}$ of statement s takes as input a BDD $d \in \mathbb{D}$ at the final label of s, and outputs a necessary precondition for the satisfaction of d at the initial label of s by using operators and backward transfer functions of $\mathbb{D}$. The lifted forward analysis $\overrightarrow{[\![s]\!]}^{\sharp}$ of a statement s takes as input a BDD $d \in \mathbb{D}$ at the initial label of s, and outputs an invariant representing the reachable states

Algorithm 1: SPLFaultLoc(P,$\mathcal{F}$,$\mathbb{A}$)

```
Input: Program family
       $P \equiv l_{in} : s; l_{ass} : \mathtt{assert}(be^{ass})$, features $\mathcal{F}$,
       domain $\mathbb{A}$
Output: Slice family $P'$
1  $\mathbb{P} := \mathbb{P}' := \emptyset$; $Dom := \mathbb{D}(\mathcal{F}, \mathbb{A})$; $d_{err} := \bot_{Dom}$; $d'_{err} :=$
     $\mathtt{FILTER}_{Dom}(\neg be^{ass}, \top_{Dom})$ ;
2  while $(\mathbb{P} \neq \mathbb{P}')$ or $(d_{err} \neq d'_{err})$ do
3      $\mathbb{P} := \mathbb{P}'$;  $d_{err} := d'_{err}$ ;
4      $\mathtt{Cond}^\sharp, \mathbb{P}' := \overleftarrow{[\![s]\!]}^\sharp\, d_{err}$ ;
5      if $(\mathtt{Cond}^\sharp(l_{in}) = \bot_{\mathbb{D}})$ then  return skip ;
6      $\mathtt{Inv}^\sharp := \overrightarrow{[\![s]\!]}^\sharp\ \mathtt{Cond}^\sharp(l_{in})$;
7      $\mathtt{ErrInv}^\sharp :=$
         $\textsc{MinSupport}(\mathtt{Inv}^\sharp \sqcap_{Dom} \mathtt{Cond}^\sharp, \mathtt{Cond}^\sharp)$ ;
8      $d'_{err} := \mathtt{ErrInv}^\sharp(l_{ass})$ ;
9      if $(d'_{err} = \bot_{Dom})$ then return skip ;
10     $Dom := \mathbb{D}(\mathcal{F} \cup \mathbb{P}', \mathbb{A})$
11 return $Slice(s, \mathtt{ErrInv}^\sharp, \vee_{k \in \mathcal{K}} k)$
```

at the final label of s by using operators and forward transfer functions of $\mathbb{D}$. For the transfer function of "while be do s", we use the abstract fixed-point functionals $\overleftarrow{\phi}^\sharp, \overrightarrow{\phi}^\sharp : \mathbb{D} \to \mathbb{D}$ that accumulate possible abstract stores after another while iteration from an abstract store d going in a backward and forward direction, respectively. Then, $\mathtt{lfp}^\sharp\, \overleftarrow{\phi}^\sharp$ and $\mathtt{lfp}^\sharp\, \overrightarrow{\phi}^\sharp$ are the limits of the following increasing chains defined by the widening [34]: $y_0 = \mathtt{FILTER}_{\mathbb{D}}(\neg be, d)$, $y_{n+1} = y_n \triangledown_{\mathbb{D}} \overleftarrow{\phi}^\sharp(y_n)$ for backward; and $y_0 = d$, $y_{n+1} = y_n \triangledown_{\mathbb{D}} \overrightarrow{\phi}^\sharp(y_n)$ for forward analysis. For the transfer function of "#if (θ) s #endif", we use the BDD operation $\mathtt{meet\_cond}(d, \theta)$ that restricts the top-down paths of the BDD d with θ.

The soundness of the lifted abstract analyses follows from the soundness of the lifted domain $\mathbb{D}$ [7, 15] and the leaf domain $\mathbb{A}$ [34]. Given $d \in \mathbb{D}$, let $\mathtt{proj}_k(d)$ denote the leaf of d reached along the top-down path that satisfies $k \in \mathcal{K}$.

Proposition 4.2. *$\overleftarrow{[\![\pi_k(s)]\!]}\gamma_{\mathbb{A}}(\mathtt{proj}_k(d)) \subseteq \gamma_{\mathbb{A}}(\pi_k(\overleftarrow{[\![s]\!]}^\sharp d))$; and $\overrightarrow{[\![\pi_k(s)]\!]}\gamma_{\mathbb{A}}(\mathtt{proj}_k(d)) \subseteq \gamma_{\mathbb{A}}(\pi_k(\overrightarrow{[\![s]\!]}^\sharp d))$.*

SPLFaultLoc Algorithm. We now present the iterative algorithm, called SPLFaultLoc, for inferring lifted error invariants and the slice family of variability-specific statements relevant for the faults.

The SPLFaultLoc$(P, \mathcal{F}, \mathbb{A})$ procedure is shown in Algorithm 1. It takes as input: a program family $P \equiv l_{in} : s; l_{ass} : \mathtt{assert}(be^{ass})$, a set of features $\mathcal{F}$, and a base abstract domain $\mathbb{A}$. After initializing the local variables (Line 1), the algorithm goes through an iterative while loop performing lifted backward and forward abstract analyses (Lines 2–10), in which the refinement process makes use both of the predicates $\mathbb{P}$ inferred at the if-s where analysis precision loss occurs as well as of the reduced abstract error store d_{err}. This way, more precise (refined) analyses are generated that can also express disjunctive program properties.

First, the algorithm calls the lifted backward abstract analysis $\overleftarrow{[\![s]\!]}^\sharp\, d_{err}$ (Line 4), where the abstract domain Dom is $\mathbb{D}(\mathcal{F}, \mathbb{A})$ and the abstract error store d_{err} is the negated assertion be^{ass}. Note that the input BDD d_{err} at the assertion label l_{ass} has only one leaf node $\mathtt{FILTER}_{\mathbb{A}}(\neg be^{ass}, \top_{\mathbb{A}})$ and decision nodes define the set $\mathcal{K}$. The analysis $\overleftarrow{[\![s]\!]}^\sharp\, d_{err}$ proceeds by structural induction on the program syntax, thus computing the necessary preconditions $\mathtt{Cond}^\sharp$ of statement s in all labels $l \in \mathbb{L}$. 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 statement $l : \mathtt{if}\ (be)\ \mathtt{then}\ l_{tt} : s_{tt}\ \mathtt{else}\ l_{ff} : s_{ff}$, the precondition in l is obtained by joining the preconditions found in the then (label l_{tt}) and else (label l_{ff}) branches. If those preconditions are different, $\mathtt{Cond}^\sharp(l_{tt}) \neq \mathtt{Cond}^\sharp(l_{ff})$, then we collect the branch condition be in $\mathbb{P}'$.

If the inferred precondition at the initial label $\mathtt{Cond}^\sharp(l_{in})$ is $\bot_{Dom}$ (Line 5), the assertion is valid and the algorithm terminates by returning the empty slice family: skip (i.e., no statement is relevant for the bug). Otherwise, the lifted forward analysis $\overrightarrow{[\![s]\!]}^\sharp\ \mathtt{Cond}^\sharp(l_{in})$ is performed to compute the invariants $\mathtt{Inv}^\sharp$ in all labels $l \in \mathbb{L}$ (Line 6). The lifted error invariants $\mathtt{ErrInv}^\sharp(l)$ are then generated via (Line 7):

$$\mathtt{apply}_2(\lambda(a_1, a_2).\mathtt{MinSupport}(a_1, a_2), \mathtt{Inv}^\sharp(l) \sqcap_{Dom} \mathtt{Cond}^\sharp(l), \mathtt{Cond}^\sharp(l))$$

where $l \in \mathbb{L}$ and MinSupport is the minimal support set [28]. Given $P = \{p_1, \dots, p_n\}$ and $p_1 \wedge \dots \wedge p_n \models p'_1 \wedge \dots \wedge p'_k$, the $\mathtt{MinSupport}(P, p'_1 \wedge \dots \wedge p'_k)$ denotes the set of minimal supporting conjuncts in P (i.e., a subset of P) that imply $p'_1 \wedge \dots \wedge p'_k$. An implementation of MinSupport through unsatisfiability cores is available in existing SMT solvers (e.g., Z3 [14]) for many first-order logic theories, such as linear arithmetic. We assume that the abstract elements of $\mathbb{A}$ are finite conjunctions of linear constraints over program variables. This way, for any pair of leafs from $\mathtt{Inv}^\sharp(l) \sqcap_{Dom} \mathtt{Cond}^\sharp(l)$ and $\mathtt{Cond}^\sharp(l)$ that can be reached along the same top-down path, the application of MinSupport removes the redundant conjuncts from the leaf of $\mathtt{Inv}^\sharp_{d_{in}}(l) \sqcap_{\mathbb{D}} \mathtt{Cond}^\sharp_{d_{err}}(l)$, which do not support the inference of the leaf of $\mathtt{Cond}^\sharp(l)$, thus producing the corresponding leaf for $\mathtt{ErrInv}^\sharp(l)$. If the refined abstract error store $d'_{err} = \mathtt{ErrInv}^\sharp(l_{ass})$ is $\bot_{Dom}$ (Line 9), the assertion is proved *true* and the algorithm terminates by returning the empty slice family. Otherwise, we perform another iteration of while loop with the refined parameters d'_{err} and $\mathbb{D}(\mathcal{F} \cup \mathbb{P}', \mathbb{A})$. The procedure terminates when either the refinement is no longer enabled, $\mathbb{P} = \mathbb{P}'$ and $d_{err} = d'_{err}$ (Line 2), or the assertion is found valid (Lines 5, 9).

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

$$\texttt{Slice}(\texttt{x} = ae, \texttt{ErrInv}^{\sharp}, \theta) = \begin{cases} \texttt{skip}, \text{ if } \texttt{ErrInv}^{\sharp}(l) = \texttt{ErrInv}^{\sharp}(l') \\ \texttt{\#if } (\theta \wedge \theta')\ \texttt{x} = ae\ \texttt{\#endif}, \ \text{if} \ \begin{array}{l}\texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l), \neg\theta') = \\ \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l'), \neg\theta')\end{array} \\ \texttt{x} = ae, \text{ otherwise} \end{cases}$$

$$\texttt{Slice}(s_1 \ ; \ s_2, \texttt{ErrInv}^{\sharp}, \theta) = \begin{cases} \texttt{skip}, \quad \text{if } \texttt{ErrInv}^{\sharp}(l) = \texttt{ErrInv}^{\sharp}(l') \\ \texttt{\#if } (\theta \wedge \theta')\ \texttt{Slice}(s_1, \texttt{ErrInv}^{\sharp}, \theta \wedge \theta') \texttt{\#endif} \ ; \ \texttt{\#if } (\theta \wedge \theta') \texttt{Slice}(s_2, \texttt{ErrInv}^{\sharp}, \theta \wedge \theta')\ \texttt{\#endif}, \\ \qquad \text{if } \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l), \neg\theta') = \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l'), \neg\theta') \\ \texttt{Slice}(s_1, \texttt{ErrInv}^{\sharp}, \theta); \texttt{Slice}(s_2, \texttt{ErrInv}^{\sharp}, \theta), \text{otherwise} \end{cases}$$

$$\texttt{Slice}(\texttt{\#if } (\theta_1)\ s\ \texttt{\#endif}, \texttt{ErrInv}^{\sharp}, \theta) = \begin{cases} \texttt{skip}, \quad \text{if } \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l), \theta_1) = \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l'), \theta_1) \\ \texttt{\#if } (\theta_1 \wedge \theta \wedge \theta')\ \texttt{Slice}(s, \texttt{ErrInv}^{\sharp}, \theta_1 \wedge \theta \wedge \theta')\ \texttt{\#endif}, \\ \qquad \text{if } \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l), \theta_1 \wedge \neg\theta') = \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l'), \theta_1 \wedge \neg\theta') \\ \texttt{\#if } (\theta_1)\ \texttt{Slice}(s, \texttt{ErrInv}^{\sharp}, \theta)\ \texttt{\#endif}, \text{ otherwise} \end{cases}$$

Figure 7. Selected definitions of $\texttt{Slice}(s, \texttt{ErrInv}^{\sharp}, \theta)$, where "$l : s; l'$" is a statement with lifted error invariants $\texttt{ErrInv}^{\sharp}$ considered in the context of variants satisfying θ.

The procedure $\texttt{Slice}(s, \texttt{ErrInv}^{\sharp}, \vee_{k \in \mathcal{K}} k)$ (Line 11) returns a slice family of the input $P \equiv l_{in} : s; l_{ass} : \texttt{assert}(be^{ass})$ containing only the variability-specific statements *relevant* for the assertion failure considered in the context of valid configurations $\mathcal{K}$. Given a statement "$l : s; l'$", we define $\texttt{Slice}(s, \texttt{ErrInv}^{\sharp}, \theta)$ as follows. (1) `Slice` replaces the statement s with `skip` when $\texttt{ErrInv}^{\sharp}(l) = \texttt{ErrInv}^{\sharp}(l')$. In this case, the statement with an encompassing error invariant is not needed to reproduce the bug and so it is irrelevant for the bug. (2) `Slice` replaces the statement s with the statement $\texttt{\#if } (\theta \wedge \theta')\ \texttt{Slice}(s, \texttt{ErrInv}^{\sharp}, \theta \wedge \theta')\ \texttt{\#endif}$ when $\texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l), \neg\theta') = \texttt{meet\_cond}(\texttt{ErrInv}^{\sharp}(l'), \neg\theta')$. In this case, statement s for variants satisfying $\neg\theta'$ is irrelevant for the bug, but s for variants satisfying θ' is relevant for the bug. Otherwise, (3) `Slice` recursively pre-process all sub-statements of compound statements or returns basic statements. The definitions of $\texttt{Slice}(s, \texttt{ErrInv}^{\sharp}, \theta)$ for some selected statements is given in Fig. 7. Note that for "$\texttt{\#if } (\theta_1)\ s\ \texttt{\#endif}$", all checks are done with respect to the presence condition θ_1 (see Fig. 7).

5 SPL Repair

In this section, we combine our variability fault localization algorithm and the mutation-based SPL repair [21]. First, we transform the input SPL to a single program, called family simulator, by replacing compile-time with run-time variability. Then, the obtained family simulator is translated into a SMT formula, which is satisfiable iff the simulator (i.e., the given SPL) violates the assertion. Finally, we describe an efficient SPL repair algorithm that repeatedly mutates expressions identified by `SPLFaultLoc` as relevant for the variability bug until a correct SPL is found.

Syntactic Transformations. The variability encoding procedure [29], `VarEncode`, transforms an input program family P with sets of features $\mathcal{F}$ and configurations $\mathcal{K}$ into a single program without `#if`-s, called *family simulator*.

First, if a statement s in P is relevant for the bug only in the context of some variants satisfying $\theta \in FeatExp(\mathcal{F})$, then we replace s with the following semantically equivalent statement: $\texttt{\#if } (\theta)\ s\ \texttt{\#endif}; \texttt{\#if } (\neg\theta)\ s\ \texttt{\#endif}$. Let us denote the resulting program family after this transformation as $\texttt{transf1}(P)$.

Then, we convert each feature $A \in \mathcal{F}$ into the global variable A non-deterministically initialized to 0 (*false*) or 1 (*true*), such that $\vee_{k \in \mathcal{K}} k$ is satisfied. Given $\mathcal{F} = \{A_1, \ldots, A_n\}$ and $\kappa = \vee_{k \in \mathcal{K}} k$, the transformation $\texttt{transf2}(P)$ is:

$$\texttt{int } A_1 = [0,1], \ldots, A_n = [0,1]; \texttt{assume}(\kappa); \texttt{transf1}(P)$$

Next, we define a rewrite rule for replacing `#if`-s from $\texttt{transf2}(P)$ with ordinary `if`-s:

$$\texttt{\#if } (\theta)\ s\ \texttt{\#endif} \rightsquigarrow \texttt{if } (\theta)\ \texttt{then } s\ \texttt{else skip} \qquad \text{(R-1)}$$

We write $\texttt{VarEncode}(P)$ to be the final transformed single program generated by repeatedly applying rule (R-1) on $\texttt{transf2}(P)$ until this rule can no longer be applied. For example, the single program in Fig. 2 is a family simulator for `intro.c` in Fig. 1.

The following result, which states that the set of final states from executions of $\texttt{VarEncode}(P)$ coincides with the union of final states from executions of all variants derived from the program family P, is proved in [29].

Proposition 5.1 ([29]). $\overrightarrow{[\![\texttt{VarEncode}(P)]\!]}\Sigma = \cup_{k \in \mathcal{K}} \overrightarrow{[\![\pi_k(P)]\!]}\Sigma$.

Translating programs to SMT formulas. The obtained family simulator is translated into a SMT formula using the

CBMC bounded model checker [9]. Three transformations are applied to the input family simulator. *Simplification* ensures that all `if` and `while` guards are replaced with fresh Boolean variables. *Unwinding* with bound b ensures that all `while`-s are unrolled b-times, so that we only consider b-bounded paths that are going through them at most b-times. For example, "`while` (be) `do` s" after unwinding with $b = 2$ is transformed to:

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

where `assume`($\neg g$) is used to block all paths longer than the bound b. Finally, *conversion to the Static Single Assignment (SSA)-form* [13] ensures that any assignment to a variable x is changed into an unique assignment to a new variable $\mathtt{x}_i$ ($i \geq 0$). To decide which definition of a given variable reaches a particular use after an `if` with the guard g, we insert the Φ-assignment $\mathtt{x}_k = g?\mathtt{x}_i : \mathtt{x}_j$ after the `if`. This means that Φ selects $\mathtt{x}_i$ if control reaches the Φ-assignment via the path on which g is *true*; otherwise Φ selects $\mathtt{x}_j$. In effect, in the obtained simplified program all original expressions are right-hand sides of assignments, loops are replaced with `if`-s, and each variable is assigned once.

The generated simplified program p is translated to a set of SMT formulas S_p as follows. We translate: assignment x=e to equation formula x=e; Φ-assignment `x2 = be?x0 : x1` to formula `x2=ite(be,x0,x1)` (meaning $(be \wedge \mathtt{x2{=}x0}) \vee (\neg be \wedge \mathtt{x2{=}x1})$); `assume`($be$) to formula be; and `assert`(be) to formula $\neg be$. The obtained set of formulas S_p is partitioned into three subsets: $S_p^{\mathtt{soft,rel}}$ that includes formulas corresponding to statements (assignments) containing original expressions that are classified by `SPLFaultLoc` as relevant for the bug, $S_p^{\mathtt{soft,irrel}}$ that includes formulas corresponding to statements (assignments) containing original expressions that are classified by `SPLFaultLoc` as irrelevant for the bug, and $S_p^{\mathtt{hard}}$ that includes the other formulas corresponding to assertions, assumptions, assignments to features, and Φ-assignments. For example, the family simulator of `intro.c` in Fig. 2 is converted to a set of SMT formulas $S_{\mathtt{intro}}$, which is then partitioned into subsets:

$S_{\mathtt{intro}}^{\mathtt{soft,rel}} = \{g0 = \mathtt{A0}, z0 = n0+1, g2 = \mathtt{A0}, y1 = z3\}$,
$S_{\mathtt{intro}}^{\mathtt{soft,irrel}} = \{g1 = \neg\mathtt{A0}, z1 = n0+1, y0 = n0-1, g3 = \neg\mathtt{B0} \wedge \mathtt{C0}, z2 = z1+1\}$,
$S_{\mathtt{intro}}^{\mathtt{hard}} = \{\mathtt{A0} = [0,1], \mathtt{B0} = [0,1], \mathtt{C0} = [0,1], z1 = \mathtt{ite}(g0, z0, 0),$
$z3 = \mathtt{ite}(g1, z2, z1), y2 = \mathtt{ite}(g2, y1, y0), z5 = \mathtt{ite}(g3, z4, z3), \neg(y2 < n0)\}$

The formula φ_p^b for (single) 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 5.2 ([9]). *A single-program p is b-correct iff φ_p^b is unsatisfiable.*

By Proposition 5.1 and 5.2, for program families P we have:

Corollary 5.3. **(i)** *$\varphi^b_{\mathtt{VarEncode}(P)}$ is unsatisfiable iff $\forall k \in \mathcal{K}.\pi_k(P)$ is b-correct iff P is b-correct.*

(ii) *$\varphi^b_{\mathtt{VarEncode}(P)}$ is satisfiable iff $\exists k \in \mathcal{K}.\pi_k(P)$ is not b-correct iff P is not b-correct.*

Algorithm 2: `SPL(FaultLoc+AllRepair)`$(P, \mathcal{F}, \mathbb{A}, b)$

```
   Input: Program family $P$, abstract domain $\mathbb{A}$,
          unwinding bound $b$
   Output: Set of solutions $Sol$
 1 $p := \mathtt{VarEncode}(P)$ ;
 2 $(S_p^{\mathtt{hard}}, S_p^{\mathtt{soft,rel}}, S_p^{\mathtt{soft,irrel}}) :=$
       $\mathtt{CBMC}(p, \mathtt{SPLFaultLoc}(p, \mathcal{F}, \mathbb{A}), b)$ ;
 3 $(S_1, \dots, S_n) := \mathtt{Mutate}(S_p^{\mathtt{soft,rel}})$ ;
 4 $(S'_1, \dots, S'_n, V_1, \dots, V_n, V_{\mathtt{orig}}) := \mathtt{InstGuardVar}(S_1, \dots, S_n)$ ;
 5 $\varphi_p^b := (\wedge_{s \in S_p^{\mathtt{hard}} \cup S_p^{\mathtt{soft,irrel}}} s) \wedge (\wedge_{s \in S'_1 \cup \dots \cup S'_n} s)$ ;
 6 $k := 1$; $Sol := \emptyset$; $\varphi := \wedge_{i=1}^n \mathtt{Exact}(V_i, 1)$ ;
 7 while $(k \leq n)$ do
 8     $\varphi_k := \varphi \wedge \mathtt{AtLeast}(V_{\mathtt{orig}}, n-k)$ ;
 9     $satres, V := \mathtt{SAT}(\varphi_k)$ ;
10     if $(satres)$ then
11         $smtres := \mathtt{IncrementalSMT}(\varphi_p^b \wedge \wedge_{v \in V} v)$ ;
12         if $(\neg smtres)$ then
13             $Sol := Sol \cup V$; $\varphi := \varphi \wedge (\vee_{v \in V \setminus (V_{\mathtt{orig}})} \neg v)$;
14         else
15             $\varphi := \varphi \wedge (\vee_{v \in V} \neg v)$ ;
16     else
17         $k := k + 1$ ;
18 return $Sol$;
```

The formula $\varphi_p^b = \phi_1 \wedge \dots \wedge \phi_n$ for a program p is instrumented with Boolean variables called *guard variables*, thus obtaining $\mathtt{inst}(\varphi_p^b) = (x_1 \Longrightarrow \phi_1) \wedge \dots \wedge (x_n \Longrightarrow \phi_n)$, 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 $\mathtt{inst}(\varphi_p^b)$ 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 also use Boolean cardinality formulas $\mathtt{AtLeast}(\{l_1, \dots, l_n\}, k)$ and $\mathtt{Exact}(\{l_1, \dots, l_n\}, k)$ to require that at least k and exactly k, respectively, of the literals $l_1, \dots, l_n$ are *true*.

`SPL(FaultLoc+ALLRepair)`. We now integrate the procedure `SPLAllRepair` [21] with the `SPLFaultLoc` in Algorithm 1, in order to reduce the mutant's search space by not mutating statements classified by `SPLFaultLoc` as irrelevant for the bug. The novel procedure, `SPL(FaultLoc+AllRepair)`, is shown in Algorithm 2. It starts by generating the family simulator p of the input program family P using the

VarEncode procedure (Line 1). The simulator p is translated to formulas ($S_p^{\mathtt{hard}}, S_p^{\mathtt{soft,rel}}, S_p^{\mathtt{soft,irrel}}$) by using the CBMC model checker, where the information of the SPLFaultLoc procedure is used for constructing subsets $S_p^{\mathtt{soft,rel}}$ and $S_p^{\mathtt{soft,irrel}}$ (Line 2). Next, we use the Mutate procedure to generate all possible mutations $S_1, \dots, S_n$ of formulas in $S_p^{\mathtt{soft,rel}}$, where S_i is a set of formulas obtained by mutating some $\phi_i \in S_p^{\mathtt{soft,rel}}$ (Line 3). Hence, $S_1, \dots, S_n$ correspond to n program locations where statements relevant for the error may occur. A *mutation* is a replacement of an (feature or program) expression of the RHS of an assignment with another expression. We have a fixed set of mutations for each type of expressions. For example, any feature A can be substituted by ¬A and vice versa. The InstGuardVars procedure instruments all formulas in $S_1, \dots, S_n$ by new guard variables recorded in sets $V_1, \dots, V_n$, thus producing instrumented formulas $S'_1, \dots, S'_n$ (Line 4). That is, $S'_i = \{(x_i \implies \phi_i) \mid \phi_i \in S_i, x_i \in V_i\}$. The set $V_{\mathtt{orig}}$ contains guard variables corresponding to original formulas in $S_p^{\mathtt{soft,rel}}$. The formula φ_p^b is initialized to be the conjunction of formulas from $S^{\mathtt{hard}}$ and $S_p^{\mathtt{soft,irrel}}$, and all instrumented formulas from $S'_1 \cup \ldots \cup S'_n$ (Line 5). Then, we search the space of all mutated formulas using an iterative generate-and-verify procedure (Lines 7-17). 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 $\mathtt{AtLeast}(V_{\mathtt{orig}}, n-k)$) and there is exactly one guard variable selected for any of n relevant statements (Line 8). A SAT solver checks the satisfiability of φ_k, such that any satisfying assignment V of φ_k corresponds to one mutated program (Line 9). 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 11). If the formula is found unsatisfiable, we report the mutant 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).

6 Evaluation

Implementation and Experimental setup. We have implemented a prototype tool for variability fault localization via inferring lifted error invariants and for its application to program repair. It uses the APRON library [31], which includes the Polyhedra abstract domain, and the BDDAPRON library [30]. It also calls the Z3 SMT solver [14] 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, with no support for pointers, struct and union types. We have also implemented our SPL repair algorithm that combines our tool for variability fault localization and the SPLAllRepair tool [21] for repairing SPLs written in C. The implementation is available online [22].

For the aim of evaluating our variability fault localization approach, we ran: (1) our approach, denoted SPLFaultLoc; (2) an approach that uses single backward abstract analysis $\mathtt{Cond}^\sharp$ to compute abstract error invariants, denoted SPLFaultLoc_Single; and (3) the Brute-force approach that uses a preprocessor to generate all variants of a program family and then applies the single-program fault localization tool FaultLoc [17, 20] to each individual variant one by one. Moreover, we also evaluate the benefits that our variability fault localization approach brings to the SPL repair. For this purpose, we compare: (1) the tool for SPL repair with no fault localization, called SPLAllRepair [21]; and (2) the approach that combines our SPLFaultLoc and SPLAllRepair, called SPL(FaultLoc+AllRepair). We use two mutation levels: *level 1* contains simpler mutations that are often sufficient for repairment, while *level 2* contains a richer set mutations. For arithmetic operators in mutation level 1, we have two sets $\{+,-\}$ and $\{*,\%,\div\}$, which means that + is replaced with − and vice versa, and $*, \%, \div$ can be replaced with each other. In mutation level 2, we have one set $\{+,-,*,\%,\div\}$, which means that any arithmetic operator from the set can be replaced with any other. Similarly, for relational operators and integer constants n, we use sets $\{<,\leq\}$, $\{>,\geq\}$, $\{==,!=\}$ and $\{n, n+1, n-1\}$ in mutation level 1 as well as the sets $\{<,\leq,>,\geq,==,!=\}$ and $\{n, n+1, n-1, 0\}$ in mutation level 2. We use sets $\{\&\&, ||\}$ and $\{\mathtt{A}, \neg\mathtt{A}\}$ for logical operators and features in both levels. We use $\{x, y \in \mathit{Param}(p)\}$ for program variables $x \in \mathit{Var}$, where $\mathit{Param}(p)$ are the parameters used in program p, in mutation level 2, whereas they are not mutated in mutation level 1. We use a set of numerical benchmarks taken from the literature [17, 21] as well as different folders of SV-COMP (https://sv-comp.sosy-lab.org/).

Experiments are run on 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM LUbuntu 20.10, with 8 GB memory, and we use a timeout value of 800 seconds. All times are reported as average over five independent executions. We report total running times, measured via real values of the time command, needed for the tasks to be performed. In the case of SV-COMP benchmarks, we have first selected some programs with integer variables, and then we have manually added variability (features and #if-s). We have inserted presence conditions with different complexities in different locations of the code. Table 1 presents characteristics of the benchmarks: file name (Benchmark), number of configurations $|\mathcal{K}|$ ($= 2^{|\mathcal{F}|}$), and lines of code (LOC).

Performances. Table 1 shows the result of running our SPLFaultLoc, SPLFaultLoc_Single, and Brute-force approach on the given benchmarks. The column "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

```
void main (int n){
① int #if (A) x = 15;
②     #else x = 5; #endif
③ int y = n;
④ while (x > n) do
⑤   x = x − 1;
⑥   #if (B ∨ C) y = y+1; #endif
⑦ od;
⑧ assert (x ≤ 10);
```

Figure 8. `while.c`

```
void main (int n) {
① int x, y, z;
② #if (A) x=6; #else x=0; #endif
③ y = 0;
④ #if (B) z=y; #else z=x; #endif
⑤ while (x>0) do
⑥   if (x<3) then y = y+1;
⑦   else z = z−1;
⑧   x = x − 1;
⑨ od;
Ⓐ assert (y ≤ 0);
```

Figure 9. `easy2-1.c`

```
void main (int x) {
① int #if (A) c = 0;
②     #else c = 5; #endif
③ while (x + c ≥ 0) do
④   #if (A) x = x−c;
②    #else x = x+c; #endif
⑤   c = c+1;
⑥ od;
⑦ assert (c > 0);
```

Figure 10. `Mysore-1.c`

error given by the concrete semantics is considered correct. Table 2 shows the performance of the repair approaches: SPL(FaultLoc+AllRepair)$_1$ and SPLAllRepair$_1$ with mutation level 1 as well as SPL(FaultLoc+AllRepair)$_2$ and SPLAllRepair$_2$ with mutation level 2. The column "Space" is given as n/m where n is the explored and m is the total size of mutant search space, and Time is the run-time in seconds.

Consider a variant of program intro.c in Fig. 1, denoted intro-b, where the assertion at loc. ⑤ is changed to $(y \geq n)$. The SPLFaultLoc reports that the only bug-relevant statement is $y = n - 1$ at loc. ② for variants satisfying ¬A, whereas the single backwards analysis infers imprecise results reporting the statements at locs. ① and ③ as relevant as well. Moreover, SPL(FaultLoc+AllRepair)$_1$ (resp., SPL(FaultLoc+AllRepair)$_2$) successfully repairs intro-b, e.g. by replacing the expression $n - 1$ with $n + 1$ at loc. ②, in 0.464 sec (resp., 0.517 sec) by exploring 8/12 (resp., 22/30) mutants, whereas SPLAllRepair$_1$ (resp., SPLAllRepair$_2$) needs 0.575 (resp., 50.61) sec by exploring 196/576 (resp., 12608/27000) mutants. Similarly, cond-b is obtained from cond.c in Fig. 3 by changing the assertion at loc. ⑧ to $(x > 0)$. The SPLFaultLoc is precise by identifying the assignment $x = 1$ at loc. ① as irrelevant for the bug, while the statement at loc. ② is wrongly reported as irrelevant for the bug by SPLFaultLoc_Single. The correct fix is found by replacing $x = 0$ at locs. ① and ④ with $x = 1$, which is done by SPLAllRepair$_1$ (resp., SPLAllRepair$_2$) in 2.620 (resp., 71.02) sec and by SPL(FaultLoc+AllRepair)$_1$ (resp., SPL(FaultLoc+AllRepair)$_2$) in 1.205 (resp., 8.348) sec.

The bug in while.c in Fig. 8 occurs when the main function is called with a value greater than 10 for the parameter n and the initial value of variable x is also greater than 10. In this case, the values n and x will be equal at the end of while. The SPLFaultLoc infers that statements at locs. ②, ③, and ⑥ as well as variable y are irrelevant for the bug, while SPLFaultLoc_Single wrongly identifies ② as relevant for the bug. In effect, only SPL(FaultLoc+AllRepair)$_2$ can fix the bug by replacing $x = 15$ at ① with $x = 0$ in 10.60 sec, while SPLAllRepair$_2$ timeouts after 800 sec. The SPLFaultLoc needs two iterations to analyze easy2-1.c in Fig. 9 by using the BDD domain with features A, B and predicate $(x < 3)$ as decision nodes, thus inferring that statements at locs. ④, ⑦, as well as $x=0$ at loc ② are irrelevant for the error. SPLFaultLoc_Single is very imprecise wrongly inferring that statements at locs. ③, ⑥, ⑧ are irrelevant for the bug. SPL(FaultLoc+AllRepair)$_1$ successfully finds several repairs (e.g., $y = -1$ at ③ and $x = x - 2$ at ⑧)in 20.76 sec by exploring 3093 mutants, while SPLAllRepair$_1$ and SPLAllRepair$_2$ timeout after 800 sec. Similarly, we analyze Mysore-1 in Fig. 10. The SPLFaultLoc finds that statements in the body of while as well as at loc. ② are redundant and so are eliminated from the slice family, whereas SPLFaultLoc_Single does not find any statement as irrelevant for the bug. As a result, SPL(FaultLoc+AllRepair)$_1$ and SPL(FaultLoc+AllRepair)$_2$ have to search through the space of only 6 mutants, while SPLAllRepair$_1$ (resp., SPLAllRepair$_2$) has to see 1152 (resp., 138240) mutants.

From Table 1 we can see that our technique SPLFaultLoc gives more precise information about potential error statements than simply performing a backward analysis. On average by using SPLFaultLoc, the number of locations to check for potential error (RLOC) is reduced to 66% of the total code (LOC). In fact, SPLFaultLoc pin-pointed the correct error locations for all examples, thus achieving the precision of 100%. On the other hand, the precision of SPLFaultLoc_Single is 68% on average. However, SPLFaultLoc is slower than SPLFaultLoc_Single with slow-downs that range from 2.8 to 42 times. However, this is an acceptable precision/cost tradeoff, since the more precise results of SPLFaultLoc can be applied in program repair (see Table 2). SPLFaultLoc and Brute-force have the same precision, but SPLFaultLoc outperforms Brute-force achieving time speed-ups ranging from 1.1 to 1.8 times. Moreover, Brute-force reports only statements (defined over program variables) relevant for the bug so its results have to be analyzed manually by the user to produce information comparable to that of SPLFaultLoc.

From Table 2 we can see that SPL(FaultLoc+AllRepair) significantly outperforms SPLAllRepair. For feasible tasks, SPL(FaultLoc+AllRepair)$_1$ achieves speed-ups that range from 1.2 to 10.3 times compared to SPLAllRepair$_1$.

Table 1. Performance results of `SPLFaultLoc` vs. `SPLFaultLoc_Single` vs. `Brute-force`. All times in sec.

Bench.	LOC	$\|\mathcal{K}\|$	SPLFaultLoc			SPLFaultLoc_Single			Brute-force	
			Time	RLOC	Prec	Time	RLOC	Prec	Time	Prec
intro	13	8	0.260	8	100%	0.031	9	90%	0.357	100%
intro-b	13	8	0.245	4	100%	0.028	9	50%	0.317	100%
cond	13	2	0.231	11	100%	0.030	13	80%	0.191	100%
cond-b	13	2	0.516	12	100%	0.028	7	50%	0.546	100%
while	17	8	1.528	11	100%	0.033	12	92%	2.298	100%
easy2-1	23	4	5.617	16	100%	0.031	6	50%	9.64	100%
Mysore-1	17	4	0.144	10	100%	0.051	17	50%	0.183	100%

Table 2. Performance results of `SPL(FaultLoc+AllRepair)` vs. `SPLAllRepair`. All times in sec.

Bench.	SPL(FaultLoc+AllRepair)$_1$		SPLAllRepair$_1$		SPL(FaultLoc+AllRepair)$_2$		SPLAllRepair$_2$	
	Space	Time	Space	Time	Space	Time	Space	Time
intro	18/24	0.420	386/576	0.841	78/120	0.585	16206/27000	53.47
intro-b	8/12	0.464	196/576	0.575	22/30	0.517	12608/27000	50.61
cond	97/144	0.624	1729/2592	2.004	241/360	1.285	9006/19440	40.67
cond-b	335/432	1.205	1985/2592	2.620	2222/3240	8.348	13052/19440	71.02
while	96/96	2.470	2304/2304	11.42	1083/1920	10.60	?/230400	timeout
easy2-1	3093/73728	20.76	?/3538944	timeout	34766/12441600	764.8	?/8.9 $\times 10^9$	timeout
Mysore-1	5/6	0.409	769/1152	4.217	5/6	0.446	?/138240	timeout

7 Related Work and Conclusion

Lifted formal analysis and repair. In the last decade, much effort has been invested in designing specialized lifted (variability-aware) formal analysis and verification algorithms [3, 5, 15, 23, 24, 26, 41] that work directly on program families, rather than on single programs. The work [15] has introduced the lifted static analysis based on abstract interpretation by using lifted BDD domain for program families with only Boolean features. Subsequently, the lifted decision tree domain has been proposed to analyze program families with numerical features [16, 24] as well as dynamic program families [23]. Another lifted technique is based on variability encoding [29, 42], which generates a simulator that simulates the computations of all variants in a program family. They re-use existing single-program analysis tools, but some precision may be lost when interpreting the obtained results.

The SPLAllRepair [21] implements a lifted repair algorithm that is based on variability encoding of program families and the single-program mutation-based repair tool AllRepair [39]. Automated program repair has also been combined with fault localization [40]. However, the above works use syntactic information in the form of graphs for static and dynamic dependency relations to localize the bugs. In contrast, here we use a pure semantics-based technique for this aim. Program repair is also related to program sketching, where a partial program with missing parts (holes) has to be completed in such a way that a given specification is satisfied [18]. Recently, lifted static analysis has been successfully employed for solving the program sketching [19, 25].

Lifted fault localization. Various fault localization techniques have been proposed to identify the locations of faults in single programs [1, 8, 27, 32, 40]. Among them, spectrum-based fault localization (SBFL) [1] is the most well-studied technique, which has been recently applied for fault localization of SPLs at the level of statements [36] and features [4]. SBFL uses the results of test cases and their corresponding code coverage to compute suspiciousness scores of each program component (e.g. statements, features) being faulty. To be effective in practice, SBFL requires more test cases, and so the test suite takes longer to run. In contrast to test-based approaches, our work belongs to formal verification-based techniques that exclusively relies on the semantics of the given SPL and does not require any test cases or user guidance. This way, our technique gives stronger guarantees on the correctness of the obtained solutions rather than a probabilistic estimation as in the SBFL techniques.

To conclude, we have presented an algorithm for variability fault localization of SPLs to obtain compact slices of buggy SPLs relevant for the bug. Moreover, we have shown how to combine our variability fault localization with a mutation-based SPL repair algorithm.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 88–99. https://doi.org/10.1109/ASE.2009.25

[2] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 372–375. https://doi.org/10.1109/ASE.2011.6100075

[3] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for product-line verification: case studies and experiments. In *35th Inter. Conference on Software Engineering, ICSE '13*. 482–491.

[4] Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2018. Spectrum-based fault localization in software product lines. *Inf. Softw. Technol.* 100 (2018), 18–31. https://doi.org/10.1016/J.INFSOF.2018.03.008

[5] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL$^{\text{LIFT}}$: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*. 355–364.

[6] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[7] Junjie Chen and Patrick Cousot. 2015. A Binary Decision Tree Abstract Domain Functor. In *Static Analysis - 22nd International Symposium, SAS 2015, Proceedings (LNCS, Vol. 9291)*. Springer, 36–53. https://doi.org/10.1007/978-3-662-48288-9_3

[8] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. 2013. Flow-Sensitive Fault Localization. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013. Proceedings (LNCS, Vol. 7737)*. Springer, 189–208. https://doi.org/10.1007/978-3-642-35873-9_13

[9] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings (LNCS, Vol. 2988)*. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[10] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[11] Patrick Cousot and Radhia Cousot. 1977. 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, 238–252. https://doi.org/10.1145/512950.512973

[12] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings (LNCS, Vol. 3444)*. Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. https://doi.org/10.1145/115372.115320

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[15] Aleksandar S. Dimovski. 2021. A binary decision diagram lifted domain for analyzing program families. *J. Comput. Lang.* 63 (2021), 101032. https://doi.org/10.1016/j.cola.2021.101032

[16] Aleksandar S. Dimovski. 2021. Lifted termination analysis by abstract interpretation and its applications. In *GPCE '21: Concepts and Experiences, 2021*. ACM, 96–109. https://doi.org/10.1145/3486609.3487202

[17] Aleksandar S. Dimovski. 2023. Error Invariants for Fault Localization via Abstract Interpretation. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings (LNCS, Vol. 14284)*. Springer, 190–211. https://doi.org/10.1007/978-3-031-44245-2_10

[18] Aleksandar S. Dimovski. 2023. Generalized Program Sketching by Abstract Interpretation and Logical Abduction. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings (LNCS, Vol. 14284)*. Springer, 212–230. https://doi.org/10.1007/978-3-031-44245-2_11

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

[20] Aleksandar S. Dimovski. 2024. Fault localization by abstract interpretation and its applications. *J. Comput. Lang.* 80 (2024), 101288. https://doi.org/10.1016/J.COLA.2024.101288

[21] Aleksandar S. Dimovski. 2024. Mutation-based Lifted Repair of Software Product Lines. In *38th European Conference on Object-Oriented Programming, ECOOP 2024 (LIPIcs, Vol. 313)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 36:1–36:24. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2024.36

[22] Aleksandar S. Dimovski. 2025. Tool Artifact for "Variability Fault Localization by Abstract Interpretation and its Application to SPL Repair". *Zenodo* (2025). https://doi.org/10.5281/zenodo.15243366

[23] Aleksandar S. Dimovski and Sven Apel. 2021. Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation. In *35th European Conference on Object-Oriented Programming, ECOOP 2021 (LIPIcs, Vol. 194)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2021.14

[24] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2021. A Decision Tree Lifted Domain for Analyzing Program Families with Numerical Features. In *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings (LNCS, Vol. 12649)*. Springer, 67–86. https://arxiv.org/abs/2012.05863

[25] Aleksandar S. Dimovski, Sven Apel, and Axel Legay. 2021. Program Sketching using Lifted Analysis for Numerical Program Families. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings (LNCS, Vol. 12673)*. Springer, 95–112. https://doi.org/10.1007/978-3-030-76384-8_7

[26] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2019. Finding suitable variability abstractions for lifted analysis. *Formal Aspects Comput.* 31, 2 (2019), 231–259. https://doi.org/10.1007/s00165-019-00479-y

[27] Evren Ermis, Martin Schäf, and Thomas Wies. 2012. Error Invariants. In *FM 2012: Formal Methods - 18th International Symposium, 2012. Proceedings (LNCS, Vol. 7436)*. Springer, 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[28] William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. 2010. 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, 71–82. https://doi.org/10.1145/1706299.1706309

[29] Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. 2015. Experiences from Designing and Validating a Software Modernization Transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*. 597–607. https://doi.org/10.1109/ASE.2015.84

[30] Bertrand Jeannet. 2009. Relational Interprocedural Verification of Concurrent Programs. In *Seventh IEEE Inter. Conf. on Software Engineering and Formal Methods, SEFM'09*. IEEE Computer Society, 83–92. https://doi.org/10.1109/SEFM.2009.29

[31] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st Inter. Conference, CAV 2009. Proceedings (LNCS, Vol. 5643)*. Springer, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

[32] Manu Jose and Rupak Majumdar. 2011. 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, 437–446. https://doi.org/10.1145/1993498.1993550

[33] Christian Kästner. 2010. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* Ph.D. Dissertation. University of Magdeburg, Germany.

[34] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372. https://doi.org/10.1561/2500000034

[35] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13*. IEEE Computer Society, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[36] Thu-Trang Nguyen, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. 2022. A Variability Fault Localization Approach for Software Product Lines. *IEEE Trans. Software Eng.* 48, 10 (2022), 4100–4118. https://doi.org/10.1109/TSE.2021.3113859

[37] Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. 2019. Automatic Program Repair Using Formal Verification and Expression Templates. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Proceedings (LNCS, Vol. 11388)*. Springer, 70–91. https://doi.org/10.1007/978-3-030-11245-5_4

[38] David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Trans. Software Eng.* 2, 1 (1976), 1–9. https://doi.org/10.1109/TSE.1976.233797

[39] Bat-Chen Rothenberg and Orna Grumberg. 2016. Sound and Complete Mutation-Based Program Repair. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings (LNCS, Vol. 9995)*. 593–611. https://doi.org/10.1007/978-3-319-48989-6_36

[40] Bat-Chen Rothenberg and Orna Grumberg. 2020. Must Fault Localization for Program Repair. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II (LNCS, Vol. 12225)*. Springer, 658–680. https://doi.org/10.1007/978-3-030-53291-8_33

[41] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 18:1–18:33. https://doi.org/10.1145/3280986

[42] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *J. Log. Algebr. Meth. Program.* 85, 1 (2016), 125–145. https://doi.org/10.1016/j.jlamp.2015.06.007