Mutation-based Lifted Repair of Software Product Lines

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

Abstract

This paper presents a novel lifted repair algorithm for program families (Software Product Lines - SPLs) based on code mutations. The inputs of our algorithm are an erroneous SPL and a specification given in the form of assertions. We use variability encoding to transform the given SPL into a single program, called family simulator, which is translated into a set of SMT formulas whose conjunction is satisfiable iff the simulator (i.e., the input SPL) violates an assertion. We use a predefined set of mutations applied to feature and program expressions of the given SPL. The algorithm repeatedly mutates the erroneous family simulator and checks if it becomes (bounded) correct. Since mutating an expression corresponds to mutating a formula in the set of SMT formulas encoding the family simulator, the search for a correct mutant is reduced to searching an unsatisfiable set of SMT formulas. To efficiently explore the huge state space of mutants, we call SAT and SMT solvers in an incremental way. The outputs of our algorithm are all minimal repairs in the form of minimal number of (feature and program) expression replacements such that the repaired SPL is (bounded) correct with respect to a given set of assertions.

We have implemented our algorithm in a prototype tool and evaluated it on a set of `#ifdef`-based C programs (i.e., annotative SPLs). The experimental results show that our approach is able to successfully repair various interesting SPLs.

2012 ACM Subject Classification Software and its engineering → Software functional properties; Software and its engineering → Software creation and management; Theory of computation → Logic

Keywords and phrases Program repair, Software Product Lines, Code mutations, Variability encoding

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.36

1 Introduction

A *program family* (Software Product Line - SPL) represents a set of similar programs, known as *variants*, generated from a common code base [2]. SPL engineering has been successfully applied in industry to meet the need for custom-tailored software. For instance, different variants from an SPL can target different platforms or may serve customization requirements for different customers. The variants are specified in terms of *features* selected for that particular variant. The popular `#ifdef` directives from the C preprocessor CPP [43] represent the most common way to implement such (annotative) program families. An `#ifdef` directive specifies under which presence conditions (i.e., feature selections or feature expressions), parts of code should be included or excluded from a variant at compile-time. SPLs are often used in the development of the embedded and safety-critical systems (e.g., mobile devices, cars, medicine, avionics), where their behavioral correctness is of primary interest. In particular, the focus is on applying various verification and analysis techniques from the field of formal methods, which can give stronger guarantees on the correctness of software systems. In the last decade, much effort has been invested in designing specialized so-called *lifted* (family-based) formal verification and analysis algorithms [4, 6, 9, 43, 30, 14, 23, 15, 20, 22, 25, 55], which allow simultaneous verification of all variants of an SPL in a single run by exploiting the commonalities between the variants. They usually return an error trace, which shows how the given specification is violated. However, the users still need to process the obtained result, in order to isolate the cause of the error to a small part of the code and subsequently

© Aleksandar S. Dimovski;
licensed under Creative Commons License CC-BY 4.0

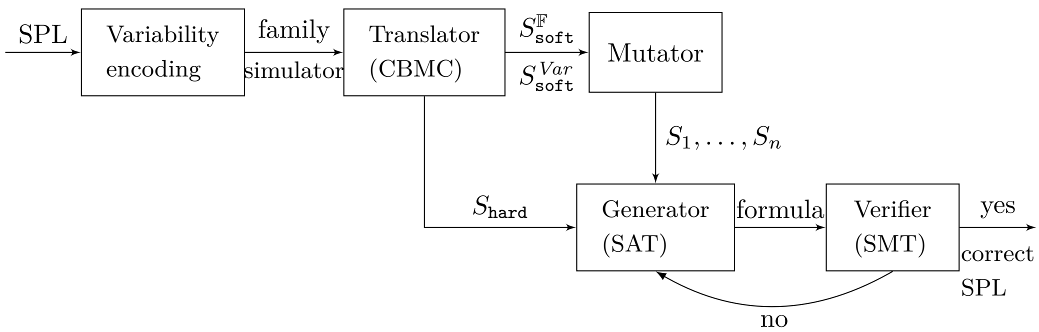

Figure 1 Diagram illustrating our lifted repair system.

to repair the given SPL. Here, we consider the problem of *SPL repair*, which is defined to be a code transformation such that the repaired SPL satisfies a given specification (e.g. assertion). Automatic SPL repair is an important problem since even if an error is identified in the verification phase, the manual error-repair is a nontrivial time-consuming task that requires close knowledge of the SPL. For instance, the error-repair of one variant may cause new errors to appear in other variants due to the feature interaction in the given SPL [3]. Recently, researchers have developed several successful single-program repair tools [28, 37, 40, 42, 45, 46, 48, 50, 51]. However, these tools cannot be directly applied to SPLs as they are only able to handle pre-processed single programs.

In this paper, we lift the mutation-based approach AllRepair [50, 51] for repairing single programs to program families (SPLs). Figure 1 illustrates our lifted repair system. More specifically, we use *variability encoding* [30, 56] to transform program families to single programs, called family simulators, by replacing compile-time variability with run-time variability (non-determinism). The (family) simulator, which contains the computations of all variants of a program family, is then translated into a set of SMT formulas using the CBMC bounded model checker [8]. The conjunction of the obtained SMT formulas is *satisfiable* iff there is an assertion violation in the given simulator iff there is an assertion violation in at least one variant of the original program family. On the other hand, the conjunction of the obtained SMT formulas is *unsatisfiable* iff all assertions are valid in the given simulator iff all assertions are valid in all variants of the original program family. We use a bounded notion of correctness, since we consider only bounded computations in which each loop and recursive call are inlined at most b times. Each statement in the simulator corresponds to a formula in the obtained set of SMT formulas, which can be partitioned into subset $S_{\texttt{hard}}$ encoding parts of the program that cannot be changed and subset $S_{\texttt{soft}} = S^{\mathbb{F}}_{\texttt{soft}} \cup S^{Var}_{\texttt{soft}}$ encoding parts of the program that can be changed. Therefore, mutating a feature or program expression found in a statement that can be changed corresponds to changing the respective SMT formula from $S^{\mathbb{F}}_{\texttt{soft}}$ or $S^{Var}_{\texttt{soft}}$, respectively. The *mutator* unit generates mutated family simulators (mutants) by using a predefined set of syntactic mutations/edits applied to feature and program expressions. Hence, in our repair model, we permit feature and program expressions to be changed but not statements. For example, we allow replacement of `#ifdef` guards (e.g., by applying $\neg$ to features, replacing $\wedge/\vee$ with $\vee/\wedge$) and right-hand sides of assignments (e.g., by increasing or decreasing a constant, replacing $+/-$ with $-/+$). Thus, the size of the space of mutants depends on the choice of permissable mutations/edits used for repair. The mutants are explored in increasing number of mutations applied to the original family simulator, so that only minimal sets of mutations are considered. Hence, the search in the space of mutants reduces to searching for an unsatisfiable set of SMT formulas. This search is performed using an iterative generate-and-verify process. The *generator* produces a minimally changed mutant using a SAT solver and the *verifier* checks the bounded-correctness of mutant using an SMT solver. This way, we find a solution with a minimal number of syntactic changes/edits to the original (incorrect) program family. Therefore, the type of errors that can be corrected is determined by the fixed set of syntactic mutations/edits, which can be applied to feature and program expressions. Hence, our approach can make repairs by replacing expressions in `#ifdef`-guards and right-hand sides of assignments with another expressions of the same form, but it cannot make repairs by replacing (adding/deleting) statements (e.g., replace assignment with if statement) or by replacing expressions with another expressions of different form (e.g., replace expression 5 with x+y). Both SAT and SMT solvers are used in an incremental way, which means that learned information is passed between successive calls. Since variants in a program family as well as mutated simulators are very similar, their encodings as sets of SMT formulas will have a lot in common. Hence, we can reuse the information that was gathered in checking previous mutated simulators to expedite the solution of the current one. The incremental solving was implemented via the mechanisms called assumptions and guard variables [26].

We have implemented our algorithm for repairing `#ifdef`-based C program families in a prototype tool, called SPLALLREPAIR, which is built on top of the ALLREPAIR tool [50, 51]. The tool uses the CBMC model checker [8] for translating single programs to SMT formulas, as well as the MINICARD [39] and Z3 [11] tools for SAT and SMT solving. We illustrate this approach for automatic repair on a number of C program families from the literature [10, 37, 46, 50, 51], and we report very encouraging results. We compare performances of two versions of our tool, with smaller and bigger sets of possible mutations, as well as with the `Brute-force` approach that repairs all variants from a program family one by one independently.

We summarize the contributions of this paper as follows:

- **Lifted Algorithm for SPL Repair:** We propose a novel lifted algorithm based on variability encoding and syntactic code mutations for repairing program families;
- **Synthesizing Minimally Repaired SPLs:** We automatically compute all minimal repaired program families (minimal in the number of code replacements) that are bounded correct by mutating feature and program expressions;
- **Implementation and Evaluation:** We build a prototype tool for automatically repairing `#ifdef`-based C program families, and present experimental results by evaluating it on a dozen of C benchmarks.

2 Motivating Example

We now present an overview of our approach using a motivating example. Consider the `#ifdef`-based C program family `intro1`, shown in Fig. 2, which uses two Boolean features `A` and `B`. They induce a family of four variants defined by the set of configurations $\mathbb{K} = \{\mathtt{A} \land \mathtt{B}, \neg\mathtt{A} \land \mathtt{B}, \mathtt{A} \land \neg\mathtt{B}, \neg\mathtt{A} \land \neg\mathtt{B}\}$. For each configuration, a different variant (single program) can be generated by appropriately resolving `#if` directives. For example, the variant for configuration $(\mathtt{A} \land \mathtt{B})$ will have both features `A` and `B` enabled (set to true or 1), thus yielding the body of `main(): int x=0; x=x+2; assert(x`$\geq$`0); return x`. The variant for $(\neg\mathtt{A} \land \neg\mathtt{B})$ will have both features `A` and `B` disabled (set to false or 0), so it has the following body of `main(): int x=0; x=x-2; assert(x`$\geq$`0); return x`. In such program families, it may happen that errors (e.g., assertion violations) occur in some variants but not in others. In the `intro1` family, the assertion is valid for variants $(\mathtt{A} \land \mathtt{B})$, $(\mathtt{A} \land \neg\mathtt{B})$, $(\neg\mathtt{A} \land \mathtt{B})$ since the returned

```
int main(){
① int x := 0;
② #if (A) x := x+2; #endif
④ #if (¬A ∧ ¬B) x := x-2; #endif
⑥ assert (x ≥ 0);
⑦ return x;
}
```

Figure 2 intro1.

```
int A := [0,1];
int B := [0,1];
int main(){
  int x := 0;
  if (A) x := x+2;
  if (¬A∧¬B) x := x-2;
  assert (x ≥ 0);
  return x;
}
```

Figure 3 intro2.

```
A0 := [0,1];
B0 := [0,1];
int main(){
  x0 := 0;
  g0 := A0;
  x1 := x0+2;
  x2 := g0?x1 : x0;
  g1 := ¬A0 ∧ ¬B0;
  x3 := x2-2;
  x4 := g1?x3 : x2;
  assert (x4 ≥ 0);
  return x4; }
```

Figure 4 intro3.

```
S_intro = {
 A0=[0,1],
 B0=[0,1],
 x0=0,
 g0=A0,
 x1=x0+2,
 x2=ite(g0,x1,x0),
 g1=¬A0 ∧ ¬B0,
 x3=x2-2,
 x4=ite(g1,x3,x2),
 ¬(x4 ≥ 0)
}
```

Figure 5 $S_{\texttt{intro}}$

value `x` will be 2, 2, 0, respectively. However, the assertion fails for variant ($\neg$A $\wedge$ $\neg$B) since the returned value `x` will be -2 in this case. The goal is to automatically repair this program family, so that the assertion is valid for all its variants.

If we make mutations only to feature expressions, there are two possible repairs of `intro1` that remedy the feature interaction ($\neg$A $\wedge$ $\neg$B) responsible for the fault. First, the feature expression (A) at loc. ② can be replaced with ($\neg$A), thus making the assertion correct for all variants: the returned value `x` will be 0 for variants (A $\wedge$ B), (A $\wedge$ $\neg$B), ($\neg$A $\wedge$ $\neg$B); and 2 for ($\neg$A $\wedge$ B). Second, the feature expression ($\neg$A $\wedge$ $\neg$B) at loc. ④ can be replaced with (A $\wedge$ $\neg$B), thus making the assertion correct for all variants: the returned value `x` will be 0 for variants ($\neg$A $\wedge$ B), (A $\wedge$ $\neg$B), ($\neg$A $\wedge$ $\neg$B); and 2 for (A $\wedge$ B). If we make mutations only to program expressions, then one possible repair is the program expression (`x-2`) at loc. ⑤ to be replaced with (`x+2`). The above three repairs are all minimal patched mutations obtained by applying only one code mutation to the original program family. Note that the found repairs depend on the sets of mutations applied to feature and program expressions. For example, if we allow mutations of the arithmetic operator `-` to `*` and of the integer constant n to 0, we will also find additional minimal repairs that replace the expression (`x-2`) at loc. ⑤ with (`x*2`) or (`x-0`).

Our algorithm for repairing program families goes through four steps. We refer to the running example `intro1` in Fig. 2 to demonstrate the steps.

(1) We transform the program family to a single program, called family simulator, using variability encoding [30, 56], such that all features are first declared as global variables and non-deterministically initialized to 0 or 1, and then all `#if` directives are transformed into ordinary `if` statements with the same branch condition. For example, the single program `intro2` in Fig. 3 is a simulator for the program family `intro1` in Fig. 2. Features `A` and `B` are defined as non-deterministically initialized global variables and two `#if` directives are replaced with `if`-s.

(2) The simulator is simplified (e.g., branch conditions are replaced with fresh Boolean variables), unwinded by unrolling loops and recursive functions b times, and converted to static single assignment (SSA) form. In the SSA form, time-stamped versions of program variables are created: every time a variable is assigned, the time-stamp is incremented by one and then the variable is renamed; every time a variable is read, it is renamed using the current time-stamp. Thus, the single program `intro3` in Fig. 4 is obtained by simplifying and converting to SSA form the simulator `intro2` in Fig. 3. For example, the `if` condition $(\neg\mathtt{A0} \wedge \neg\mathtt{B0})$ is assigned to a fresh Boolean variable $g1$, the first assignment to `x` is replaced by an assignment to `x0`, the second by an assignment to `x1`, etc. We use Φ-assignments to determine which copy of `x` will be used after `if`-s. For example, the Φ-assignment `x2 := g0?x1 : x0` means that `x1` is used if `g0` is true, and `x0` is used if `g0` is false.

(3) The simplified program in SSA form is converted to a program formula. Hence, the program `intro3` in Fig. 4 is converted to a set of SMT formulas $S_{\mathtt{intro}}$ shown in Fig. 5, such that the corresponding program formula $\varphi_{\mathtt{intro}}$ is a conjunction of all SMT formulas in $S_{\mathtt{intro}}$. Note that the Φ-assignment `x2 := g0?x1 : x0` is converted to the formula `x2=ite(g0,x1,x0)`, which means $(\mathtt{g0} \wedge \mathtt{x2{=}x1}) \vee (\neg\mathtt{g0} \wedge \mathtt{x2{=}x0})$, while `assert(be)` is converted to $(\neg\mathtt{be})$. Therefore, a program is correct (i.e., all assertions in it are valid) iff the corresponding program formula is unsatisfiable.

(4) By making mutations in the set of SMT formulas, we aim to construct an unsatisfiable program formula and report the corresponding program as repaired. In the running example, if one of the following mutations: $(\mathtt{g0{=}}\neg\mathtt{A})$, $(\mathtt{g1{=}A} \wedge \neg\mathtt{B})$, or `(x3=x2+2)`, is applied to the set of SMT formulas $S_{\mathtt{intro}}$ in Fig. 5, we obtain an unsatisfiable program formula. This way, we generate a minimally mutated program family, which contains only one code mutation, that is correct.

3 Background

In this section, we introduce the background concepts used in later developments. We begin with the definition of syntax and semantics of program families. Then, we proceed to introducing the bounded program analysis for translating single programs to SMT formulas.

3.1 Program Families

Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite set of *Boolean features* available in a program family. A *configuration* $k : \mathbb{F} \to \{\text{true}, \text{false}\}$ is a truth assignment or a *valuation*, which gives a truth value to each feature. If $k(A) = \text{true}$, then feature A is enabled in configuration k, otherwise A is disabled. We assume that only a subset $\mathbb{K}$ of all possible configurations are *valid*. Each configuration $k \in \mathbb{K}$ can also be represented by a formula: $(k(A_1) \cdot A_1 \wedge \ldots \wedge k(A_n) \cdot A_n)$, where $\text{true} \cdot A = A$ and $\text{false} \cdot A = \neg A$. We write $\mathbb{K}$ for the set of all valid configurations. We define *feature expressions*, denoted $\mathit{FeatExp}(\mathbb{F})$, as the set of propositional logic formulas over $\mathbb{F}$:

$$\theta\,(\theta \in \mathit{FeatExp}(\mathbb{F})) ::= \text{true} \mid A \in \mathbb{F} \mid \neg\theta \mid \theta \wedge \theta \mid \theta \vee \theta$$

We consider a simple sequential non-deterministic programming language, in which the program variables $\mathit{Var} {=} \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. To define program families, a new compile-time conditional statement is introduced: "`#if` (θ) s `#endif`", such that the statement s will be included in the variant corresponding to configuration $k \in \mathbb{K}$ only if θ is satisfied by k, i.e. $k \models \theta$. The syntax is:

$$\begin{aligned}
&s\,(s \in \mathit{Stm}) ::= \texttt{skip} \mid \texttt{x:=}ae \mid s;s \mid \texttt{if}\,(be)\,\texttt{then}\,s\,\texttt{else}\,s \mid \texttt{while}\,(be)\,\texttt{do}\,s \mid \\
&\qquad\qquad \texttt{\#if}\,(\theta)\,s\,\texttt{\#endif} \mid \texttt{assert}(be) \mid \texttt{assume}(be) \\
&ae\,(ae \in \mathit{AExp}) ::= n \mid [n,n'] \mid \texttt{x} \mid ae \oplus ae, \\
&be\,(be \in \mathit{BExp}) ::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be
\end{aligned}$$

where $n \in \mathbb{Z}$, $\texttt{x} \in \mathit{Var}$, $\oplus \in \{+,-,*,\%,/\}$, $\bowtie \in \{<,\leq,\texttt{==},\texttt{!=}\}$, and integer interval $[n,n']$ denotes a random integer in the interval. Without loss of generality, we assume that a program family P is a sequence of statements followed by a single assertion, whereas a single program p is a sequence of statements without `#if`-s followed by an assertion.

▶ Remark 1. The C preprocessor `CPP` [32] also uses other compile-time conditional statements that can be desugared and represented only by the `#if` construct we use in this work, e.g. `#if` (θ) s_0 `#else` s_1 `#endif` is translated into `#if` $(\theta)\, s_0$ `#endif` ; `#if` $(\neg\theta)\, s_1$ `#endif`. Compile-time conditional constructs can also be defined at the level of expressions, e.g. `#if` (θ) ae_0 `#else` ae_1 `#endif`, and they can be translated into compile-time conditional statements by code duplication [32]. We use variability at the level of statements for pedagogical reasons in order to keep the presentation focussed.

A program family is evaluated in two phases. First, the C preprocessor `CPP` [32] takes a program family s and a configuration $k \in \mathbb{K}$ as inputs, and produces a variant (single program without `#if`-s) corresponding to k as output. Second, the obtained variant is evaluated using the standard single-program semantics [20]. The first phase is specified by the projection function π_k, which is an identity for all basic statements and recursively pre-processes all sub-statements of compound statements. Hence, $\pi_k(\texttt{skip}) = \texttt{skip}$ and $\pi_k(s;s') = \pi_k(s);\pi_k(s')$. The most interesting case is "`#if` (θ) s `#endif`", where the statement s is included in the variant k if $k \models \theta$; [1] otherwise s is excluded from the variant k. That is:

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

Given a program family P, the set of all variants derived from P is $\{\pi_k(P) \mid k \in \mathbb{K}\}$.

3.2 Bounded Program Analysis

Unbounded loops with memory allocation are the reason for the undecidability of the assertion verification problem [24]. To avoid undecidability, we impose a bound on the loops by discarding all executions paths in which a loop is iterated more than a pre-determined number of times. That is, we analyze a new bounded program that under-approximates the original program. Using such bounded program, we can build a SMT formula that represents its semantics. We now briefly explain how a pre-processed program without `#if`-s is translated into a set of SMT formulas using the CBMC bounded model checker [8]. We present only the details that are important to understand our algorithm.

The given pre-processed (single) program undergoes three transformations: simplification, unwinding, and conversion to SSA form. Recall from Section 2 that the simplification ensures

---

[1] Since $k \in \mathbb{K}$ is a valuation function, either $k \models \theta$ holds or $k \not\models \theta$ holds for any θ.

that all branch conditions are replaced with fresh Boolean variables, whereas the SSA-form guarantees that each local variable has a single static point of definition. More specifically, in SSA-form each assignment to a variable $\mathtt{x}$ is changed into an unique assignment to a new variable $\mathtt{x}_i$. Hence, if variable $\mathtt{x}$ has n assignments to it throughout the program, then n new variables $\mathtt{x}_0$ to $\mathtt{x}_{n-1}$ are created to replace $\mathtt{x}$. All uses of $\mathtt{x}$ are replaced by a use of some $\mathtt{x}_i$. To decide which definition of a variable reaches a particular use after an `if`-statement with the guard g, we add the Φ-assignment $\mathtt{x}_k$ `:=` $g?\mathtt{x}_i : \mathtt{x}_j$ after the `if`. This means that if control reaches the Φ-assignment via the path on which g is true, Φ selects $\mathtt{x}_i$; otherwise Φ selects $\mathtt{x}_j$. This way, all uses of $\mathtt{x}$ after an Φ-assignment $\mathtt{x}_k$ `:=` $g?\mathtt{x}_i : \mathtt{x}_j$ become uses of Φ-assignment $\mathtt{x}_k$ until the next assignment of $\mathtt{x}$. The unwinding with bound b means that all `while` loops and recursive functions are unwound b times, so that we consider only so-called *b-bounded paths* that are going through them at most b times. For example, the statement "`while` (be) `do` s" after unwinding with $b = 2$ will be transformed to:

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

where we use `assume`$(\neg g)$ to block all paths longer than the bound b. After the above three transformations, in the obtained simplified program all original expressions are right-hand sides (RHSs) of assignments, loops are replaced with `if`-s, and each variable is assigned once. For example, the simplified program `intro3` is obtained from `intro2` by the above three transformations.

The generated simplified program is converted to a set of SMT formulas S as follows. An assignment `x:=`ae is converted to equation formula `x=`ae; a Φ-assignment `x := be?x1 : x2` is converted to formula `x=ite(be,x1,x2)`; an `assume`(be) is converted to formula be; and an `assert`(be) is converted to formula $\neg be$. A statement that is part of a `while` body may be encoded by several formulas $\phi_1, \dots, \phi_k$ in S due to the unwinding. In this case, we remove $\phi_1, \dots, \phi_k$ from S, and add instead one conjunctive formula $(\phi_1 \wedge \dots \wedge \phi_k)$ in S. In effect, we obtain that one formula in S encodes a single statement in the original program. For example, the set $S_{\mathtt{intro}}$ is obtained from `intro3` by the above conversion.

The obtained set of formulas S is partitioned into three subsets: $S^{Var}_{\mathtt{soft}}$ that contains all formulas corresponding to statements containing original program expressions, $S^{\mathbb{F}}_{\mathtt{soft}}$ that contains all formulas corresponding to statements containing original feature expressions, and $S_{\mathtt{hard}}$ that contains the other formulas corresponding to assertions, assumptions, Φ-assignments, and feature variable-assignments. Since all original program and feature expressions are RHSs of assignments after the simplification phase, all formulas in $S^{Var}_{\mathtt{soft}}$ and $S^{\mathbb{F}}_{\mathtt{soft}}$ are either single assignment formulas (`x=`ae) or multiple assignment formulas $\big((\mathtt{x}_1 = ae_1) \wedge \dots \wedge (\mathtt{x}_k = ae_k)\big)$. For example, the set $S_{\mathtt{intro}}$ in Fig. 5 is partitioned as follows:

$$\begin{aligned}
S^{Var}_{\mathtt{soft}} &= \{\mathtt{x0{=}0}, \mathtt{x1{=}x0{+}2}, \mathtt{x3{=}x2{-}2}\},\\
S^{\mathbb{F}}_{\mathtt{soft}} &= \{\mathtt{g0{=}A0}, \mathtt{g1{=}}\neg\mathtt{A0} \wedge \neg\mathtt{B0}\},\\
S_{\mathtt{hard}} &= \{\mathtt{A0{=}[0,1]}, \mathtt{B0{=}[0,1]}, \mathtt{x2{=}ite(g0,x1,x0)}, \mathtt{x4{=}ite(g1,x3,x2)}, \neg(\mathtt{x4} \geq 0)\}
\end{aligned}$$

Given a pre-processed (single) program p, the program formula φ^b_p is the conjunction of all formulas in S, where b denotes the unwinding bound used in the transformation phase of p. The formula φ^b_p encodes all possible b-bounded paths in the program p that go through each loop at most b times. We say that a program p is *b-correct* if all assertions in it are valid in all b-bounded paths of p.

▶ Proposition 2 ([8]). *A pre-processed (single) program p is b-correct iff φ^b_p is unsatisfiable.*

A satisfying assignment (model) of φ_p^b represents a b-bounded path of p that satisfies all assumptions but violates at least one assertion. In the following, we omit to write p and b in the program formula φ_p^b when they are clear from the context.

Our approach reasons about loops by unrolling them, so it is sensitive to the unrolling bound. We now present an example, where the unrolling bound has impact on the assertion validity.

▶ Example 3. Consider the program:

```
int i:=0, x:=0; while(i<3)do{i:=i+1; x:=x+1; }
```

Suppose that the assertion to be checked is `assert(x≥3)` at the final location. If we use the unrolling bound $b = 2$, we will find that the program is incorrect due to the spurious execution path that runs the `while`-body 2 times. Hence, we will needlessly try to repair this correct program. However, if we use the bound $b \geq 3$, then we will establish that the program is correct and so no repair is needed.

Suppose that the assertion to be checked is `assert(x<3)` at the final location. If we use the unrolling bound $b = 2$, we will find that the program is correct since the assertion is valid for all 2-bounded paths, so no repair will be performed. However, if we use the bound $b \geq 3$, then we will truly establish that the program is incorrect and so a repair is needed.

To enable incremental SMT solving, the program formula φ is instrumented with Boolean variables called *guard variables*. More specifically, a formula $\varphi = \phi_1 \wedge \ldots \wedge \phi_n$ is replaced with $\varphi' = (x_1 \implies \phi_1) \wedge \ldots \wedge (x_n \implies \phi_n)$, where $x_1, \ldots, x_n$ are fresh guard variables. In effect, the formula $(x_i \implies \phi_i)$ can be satisfied by setting x_i to false. Some guard variables called *assumptions* are conjuncted with φ' and passed to an incremental SMT solver. For example, $\varphi' \wedge x_1 \wedge x_2$ is satisfiable iff ϕ_1 and ϕ_2 are satisfiable, since the satisfying assignment will set $x_3, \ldots, x_n$ to false thus making $(x_3 \implies \phi_3), \ldots, (x_n \implies \phi_n)$ true. Thus, an incremental SMT-solver checking the satisfiability of $\varphi' \wedge x_1 \wedge x_2$ will only check satisfiability of ϕ_1 and ϕ_2, thus essentially disabling formulas $\phi_3, \ldots, \phi_n$.

We will use formulas of the form $\texttt{AtMost}(\{l_1, \ldots, l_n\}, k)$ (resp., $\texttt{AtLeast}(\{l_1, \ldots, l_n\}, k)$) to require that at most (resp., at least) k of the literals $l_1, \ldots, l_n$ are true. They are called *Boolean cardinality formulas* encoding that $\sum_{i=1}^{n} l_i \leq k$ (resp., $\sum_{i=1}^{n} l_i \geq k$), where l_i is a literal assigned the value 1 if true and the value 0 if false, and $k \in \mathbb{N}$. We will use the MiniCard SAT-solver [39] to check their satisfiability.

4 Lifted Repair Algorithm

In this section, we present our lifted repair algorithm, called SPLAllRepair, for repairing program families. We first give a high-level overview of the algorithm, and then describe its components more formally.

High-level Description.

The SPLAllRepair is given in Algorithm 1. It takes as input a program family P, an unwinding bound b, and a repair size r that limits the search space to only mutated programs with at most r mutations (changes to the original code) applied at once. The algorithm goes through an iterative generate-and-verify procedure, implemented using an interplay between an SAT solver and an SMT solver. In particular, we use an SAT solver in the generate phase to find a mutated program from the search space, whereas we use an SMT solver in the verify phase to check if the mutated program is correct.

Algorithm 1 $\texttt{SPLAllRepair}(P, b, r)$

Input: Program family P, unwinding bound b, repair size r
Output: Set of solutions Sol

1 $p_{sim} := \texttt{VarEncode}(P)$;
2 $(S_{\texttt{hard}}, S^{Var}_{\texttt{soft}}, S^{\mathbb{F}}_{\texttt{soft}}) := \texttt{CBMC}(p_{sim}, b)$;
3 $(S_1, \ldots, S_n) := \texttt{Mutate}(S^{Var}_{\texttt{soft}}, S^{\mathbb{F}}_{\texttt{soft}})$;
4 $(S'_1, \ldots, S'_n, V_1, \ldots, V_n, V_{\texttt{orig}}) := \texttt{InstGuardVars}(S_1, \ldots, S_n)$;
5 $\varphi^b_{sim} := (\wedge_{s \in S_{\texttt{hard}}} s) \wedge (\wedge_{s \in S'_1 \cup \ldots \cup S'_n} s)$;
6 $\varphi := (\wedge^n_{i=1} \texttt{AtMost}(V_i, 1)) \wedge (\wedge^n_{i=1} \texttt{AtLeast}(V_i, 1))$;
7 $k := 1$; $Sol := \emptyset$;
8 **while** $(k \leq n) \wedge (k \leq r)$ **do**
9     $\varphi_k := \varphi \wedge \texttt{AtLeast}(V_{\texttt{orig}}, n - k)$;
10     $satres, V := \texttt{SAT}(\varphi_k)$;
11   **if** $(satres)$ **then**
12         $smtres := \texttt{IncrementalSMT}(\varphi^b_{sim} \wedge \wedge_{v \in V} v)$;
13   **if** $(\neg smtres)$ **then**
14             $Sol := Sol \cup V$;
15             $\varphi_k := \varphi_k \wedge (\vee_{v \in V \setminus (V_{\texttt{orig}})} \neg v)$;
16   **else**
17             $\varphi_k := \varphi_k \wedge (\vee_{v \in V} \neg v)$;
18   **else**
19         $k := k + 1$;
20   **if** $(Timeout)$ **then return Sol** ;
21 **return** Sol**;**

The SPLAllRepair starts by generating the family simulator p_{sim} using the preprocessor `VarEncode` procedure (line 1). Then, the `CBMC` translation procedure calls the CBMC model checker to generate the triple $(S_{\texttt{hard}}, S^{Var}_{\texttt{soft}}, S^{\mathbb{F}}_{\texttt{soft}})$ of sets of formulas corresponding to p_{sim} as explained in Section 3.2 (line 2). By calling the `Mutate` procedure, we generate all possible mutations $S_1, \ldots, S_n$ of formulas in $S^{Var}_{\texttt{soft}}$ and $S^{\mathbb{F}}_{\texttt{soft}}$ (line 3). Here S_i is a set of formulas obtained by mutating some $\phi_i \in S^{Var}_{\texttt{soft}} \cup S^{\mathbb{F}}_{\texttt{soft}}$. Thus, $S_1, \ldots, S_n$ correspond to n program locations where an error may occur. Next, we use the `InstGuardVars` procedure to instrument all formulas in $S_1, \ldots, S_n$ by fresh guard variables, so that the results are sets of instrumented formulas $S'_1, \ldots, S'_n$ and sets of fresh guard variables $V_1, \ldots, V_n$ used to guard formulas in $S'_1, \ldots, S'_n$ (line 4). Here $S'_i = \{(x \implies \phi) \mid \phi \in S_i, x \text{ is a fresh guard variable}\}$. The set $V_{\texttt{orig}}$ contains guard variables corresponding to original formulas in $S^{Var}_{\texttt{soft}}$ and $S^{\mathbb{F}}_{\texttt{soft}}$. The program formula φ^b_{sim} is then initialized to be the conjunction of all formulas from $S_{\texttt{hard}}$ and all instrumented formulas from $S'_1 \cup \ldots \cup S'_n$ (line 5). Subsequently, we search the space of all mutated formulas in increasing size order using the variable k, which is initialized to 1 and increased after each iteration (lines 8–20). In particular, we generate the boolean formula φ_k [13] (line 9) expressing that k guard variables are not original, that is $n - k$ are original (by using $\texttt{AtLeast}(V_{\texttt{orig}}, n - k)$), and there is exactly one guard variable selected for each statement in the program (by using $\varphi \equiv \wedge^n_{i=1} \texttt{AtMost}(V_i, 1) \wedge \wedge^n_{i=1} \texttt{AtLeast}(V_i, 1)$, line 6). This means that every satisfying assignment of φ_k represents one mutated program formula of size at most k (i.e. with k changes to the original code). The boolean formula φ_k is fed to an SAT solver, which can handle Boolean cardinality formulas, to check its satisfiability. If

φ_k is unsatisfiable, this means that there are no unexplored mutated program formulas of size k so we increase k by one (line 19) and generate a new formula φ_k. Otherwise, if φ_k is satisfiable, we store in a set V all guard variables assigned true in the given satisfying assignment of φ_k (line 10). To check the correctness of the mutated program corresponding to the satisfying assignment V of φ_k, we call an incremental SMT solver to check φ^b_{sim} with all guards in V passed as assumptions (i.e., $\varphi^b_{sim} \wedge \wedge_{v \in V} v$) (line 12). This is the same to checking the conjunction of all formulas in $S_{\texttt{hard}}$ and all soft formulas guarded by variables in V, since all other soft formulas will get satisfied by setting their guard variables to false. Notice that SMT formulas solved consecutively in the iteration are very similar, thus sharing majority of their assumptions and all hard formulas. This means that most of what was learnt in solving the previous formula can be reused to solve the current one. If the result of incremental SMT solving is true, the mutated program is not correct so we block V from further exploration (line 17). Otherwise, we report V as a possible solution (i.e., a repaired program family) and block all supersets of V for further exploration (lines 14,15). The algorithm terminates when either the whole search space of mutated programs is inspected, i.e. all possible combinations of guard variables in n locations are explored as assumptions ($k > n$, line 8), or the subspace of mutated programs with at most r mutations is inspected ($k > r$, line 8), or a time limit is reached (line 20).

▶ Example 4. Let p be a simulator with 4 statements that can be mutated. Let p_1 be a repaired mutant of p consisting of mutating statement 1 with mutation M^1_1 (guard variable v^1_1) and statement 3 with mutation M^2_3 (guard variable v^2_3). Then blocking any superset of this mutation is done by adding the blocking clause $(\neg v^1_1 \vee \neg v^2_3)$ to the Boolean formula φ_k representing the search space of all mutants. This means do not apply either M^1_1 to statement 1 or do not apply M^2_3 to statement 3.

On the other hand, let p_2 be a buggy mutant of p consisting of mutating statement 1 with mutation M^2_1 (guard variable v^2_1) and statement 4 with mutation M^2_4 (guard variable v^2_4). The guards for original statements 2 and 3 are v^{orig}_2 and v^{orig}_3. Then the blocking clause $(\neg v^2_1 \vee \neg v^{orig}_2 \vee \neg v^{orig}_3 \vee \neg v^2_4)$ will be added to prune from the search space exactly the mutant p_2. Note that smaller blocking clause (with smaller number of literals) will result in a larger set of pruned mutants.

Pre-Processor: `VarEncode`.

The aim of the pre-processor `VarEncode` procedure is to transform an input program family P with sets of features $\mathbb{F}$ and configurations $\mathbb{K}$ into an output pre-processed (single) program without `#if`-s, called family simulator. The set of configurations $\mathbb{K}$ includes all possible combinations of feature values. In the pre-transformation phase, we convert each feature $A \in \mathbb{F}$ into the global variable A non-deterministically initialized to 0 or 1. Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be the set of available *features* in the program family P. We generate the following pre-transformed program:

$$\texttt{pre-t}(P) \equiv \texttt{int}\, A_1 \texttt{ := } [0,1], \ldots, A_n \texttt{ := } [0,1]; P$$

We now define a rewrite rule for eliminating `#if`-s from $\texttt{pre-t}(P)$. Let $\mathbb{K}$ be the set of configurations in the family P that can be equated to a propositional formula $\kappa = \vee_{k \in \mathbb{K}} k$. Note that if $\mathbb{K}$ contains all possible combinations of feature values, then $\kappa \equiv$ true. The rewrite rule replaces `#if`-s with ordinary `if`-s whose guards are strengthened with the feature model κ.

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

If the current program family being transformed matches the abstract syntax tree node of the shape `#if` $(\theta)\, s$ `#endif`, then replace it with the RHS of rule (R-1). We write $\texttt{VarEncode}(P)$ to be the final transformed single program obtained by repeatedly applying rule (R-1) on $\texttt{pre-t}(P)$ and on its transformed versions until we reach a point at which this rule can no longer be applied.

A memory *state* $\sigma : \Sigma = \mathit{Var} \to \mathbb{Z}$ is a function mapping each program variable to a value. Given a single program p and a memory state σ, we write $[\![p]\!]\sigma$ for the set of final states that can be derived by executing all terminating paths (computations) of p starting in the input state σ. Note that the result is a set of states since our language is non-deterministic. We define $[\![p]\!] = \cup_{\sigma \in \mathcal{P}(\Sigma)} [\![p]\!]\sigma$ to be the set of final states that can be reached by p from any possible input state $\sigma \in \mathcal{P}(\Sigma)$ (where $\mathcal{P}(\Sigma)$ is the powerset of Σ). The following result shows that the set of final states from terminating computations of $\texttt{VarEncode}(P)$ coincides with the union of final states from terminating computations of all variants derived from the program family P.

▶ Proposition 5 ([30]). *For a program family P, $[\![\texttt{VarEncode}(P)]\!] = \cup_{k\in\mathbb{K}} [\![\pi_k(P)]\!]$.*

▶ Example 6. Consider the program family `intro1` in Fig. 2 and its family simulator `intro2`≡`VarEncode(intro1)` in Fig. 3. The states σ contain only one program variable `x`. Hence, the semantics of all variants of `intro1` is:

$$\begin{array}{ll} [\![\pi_{\mathtt{A}\wedge\mathtt{B}}(\mathtt{intro1})]\!] = [\mathtt{x} \mapsto 2], & [\![\pi_{\mathtt{A}\wedge\neg\mathtt{B}}(\mathtt{intro1})]\!] = [\mathtt{x} \mapsto 2] \\ [\![\pi_{\neg\mathtt{A}\wedge\mathtt{B}}(\mathtt{intro1})]\!] = [\mathtt{x} \mapsto 0], & [\![\pi_{\neg\mathtt{A}\wedge\neg\mathtt{B}}(\mathtt{intro1})]\!] = [\mathtt{x} \mapsto -2] \end{array}$$

On the other hand, the semantics of `intro2`≡`VarEncode(intro1)` is:

$$[\![\mathtt{VarEncode(intro1)}]\!] = \{[\mathtt{x} \mapsto -2], [\mathtt{x} \mapsto 0], [\mathtt{x} \mapsto 2]\}$$

`Mutate`.

As explained in Section 3.2, the SMT formulas in $S^{\mathit{Var}}_{\mathtt{soft}}$ and $S^{\mathbb{F}}_{\mathtt{soft}}$ correspond to statements containing program and feature expressions, so our goal is to repair the given erroneous program family by applying mutations to those formulas. A *mutation* is a replacement of a program/feature expression with another expression of the same type. For example, feature expressions A and $A \wedge B$ can be replaced with $\neg A$ and $(A \vee \neg B)$, while program expressions x and $x+2$ can be replaced with 0 and $x-2$. We maintain a fixed list of syntactic mutations for each type of program and feature expressions. Let us assume that mutations $M_1, \ldots, M_j$ can be applied to a formula $\phi \in S^{\mathit{Var}}_{\mathtt{soft}} \cup S^{\mathbb{F}}_{\mathtt{soft}}$. Then, $\texttt{Mutate}(\phi) = \{\phi, M_1(\phi), \ldots, M_j(\phi)\}$. Finally, we have $\texttt{Mutate}(S^{\mathit{Var}}_{\mathtt{soft}}, S^{\mathbb{F}}_{\mathtt{soft}}) = \Pi_{\phi \in S^{\mathit{Var}}_{\mathtt{soft}} \cup S^{\mathbb{F}}_{\mathtt{soft}}} \texttt{Mutate}(\phi)$.

We now present the variability-specific mutations applied to feature expressions: $A \to \neg A$ (read: feature A is replaced by $\neg A$) and $\neg A \to A$ for features $A \in \mathbb{F}$, as well as $\{\wedge, \vee\}$ (read: logical operator $\wedge$ can be replaced with $\vee$, and vice versa).

▶ Example 7. Recall that $S^{\mathbb{F}}_{\mathtt{soft}} = \{\mathtt{g0{=}A0}, \mathtt{g1{=}}\neg\mathtt{A0} \wedge \neg\mathtt{B0}\}$ for our running example `intro1`. If we use the variability-specific mutations $A \to \neg A$, $\neg A \to A$ for $A \in \mathbb{F}$ and $\{\wedge, \vee\}$, we obtain:

$$\begin{array}{l} \texttt{Mutate}(S^{\mathbb{F}}_{\mathtt{soft}}) = \{\mathtt{g0{=}A0},\ \mathtt{g0{=}}\neg\mathtt{A0},\ \mathtt{g1{=}}\neg\mathtt{A0} \wedge \neg\mathtt{B0},\ \mathtt{g1{=}A0} \wedge \neg\mathtt{B0},\ \mathtt{g1{=}}\neg\mathtt{A0} \wedge \mathtt{B0},\ \mathtt{g1{=}A0} \wedge \mathtt{B0}, \\ \qquad \mathtt{g1{=}}\neg\mathtt{A0} \vee \neg\mathtt{B0},\ \mathtt{g1{=}A0} \vee \neg\mathtt{B0},\ \mathtt{g1{=}}\neg\mathtt{A0} \vee \mathtt{B0},\ \mathtt{g1{=}A0} \vee \mathtt{B0}\} \end{array}$$

Post-Processor: Interpreting results.

The solutions obtained by calling the AllRepair tool to repair $\texttt{VarEncode}(P)$ are interpreted back on the original program family P. Any possible repair for $\texttt{VarEncode}(P)$, which consists of replacing some feature and program expressions, represents a valid repair for P as well. This is due to the fact that our transformed program `VarEncode`(P) contains all possible paths that may occur in any variant $\pi_k(P)$ for $k \in \mathbb{K}$. A single program (variant) is b-correct if it has no b-bounded path that leads to an assertion failure, while a program family is b-correct if all its variants are b-correct. Therefore, the b-correctness and possible repair of `VarEncode`(P) and P are isomorphic (identical).

More formally, by using Propositions 2 and 5, we can prove the following result.

▶ Corollary 8. *Let P and b be a program family and an unwinding bound.*
(i) $\varphi^b_{\mathtt{VarEncode}(P)}$ *is unsatisfiable iff $\forall k \in \mathbb{K}.\pi_k(P)$ is b-correct iff P is b-correct.*
(ii) $\varphi^b_{\mathtt{VarEncode}(P)}$ *is satisfiable iff $\exists k \in \mathbb{K}.\pi_k(P)$ is not b-correct iff P is not b-correct.*

`Correctness.`

We first use Corollary 8 to show the b-correctness of the SPLAllRepair algorithm (where b is the unwinding bound). That is, every solution returned by SPLAllRepair is minimal repaired program family (b-soundness), and every minimal repaired program family with respect to mutations we apply is eventually returned by SPLAllRepair (b-relative completeness). Our algorithm explores all mutated programs in increasing size order starting with size 1. Every returned solution is minimally repaired due to the fact that it would have been blocked by another smaller solution in a previous iteration. Therefore, the b-correctness (b-soundness and b-relative completeness) of SPLAllRepair follows from the b-correctness of AllRepair shown in [50] and Corollary 8 (i.e., the fact that `VarEncode`(P) and P are isomorphic with respect to b-correctness).

The SPLAllRepair always terminates, as there are only finitely many mutations that can be applied to any type of (feature and program) expressions so the algorithm enumerates all possible mutated programs (simulators) until it finds the minimal repaired ones if any. This way, we have proved the following result.

▶ Theorem 9. *The algorithm* ***SPLAllRepair***(P, b, r) *is b-bounded correct and terminates.*

5 Evaluation

We now evaluate our approach for mutation-based lifted repair of SPLs. We show that our approach can efficiently repair various interesting `#ifdef`-based C program families, and we compare the runtime performances and precision of two versions of our algorithm, with smaller and bigger sets of mutations, as well as with the `Brute-force` approach that repairs all variants of a program family one by one independently.

Implementation.

We have implemented our lifted repair algorithm SPLAllRepair in a prototype tool, which is built on top of the AllRepair tool [50, 51] for repairing single programs. The pre-processor `VarEncode` procedure is implemented in Java, while the translation and mutation procedures (`CBMC` and `Mutate` in Algorihtm 1) are implemented by modifying the CBMC model checker [8] written in C++, where variability-specific mutations are defined. Moreover, we have experimented by defining various mutations to other types of program expressions (see below). The repair phase is implemented by calling the AllRepair tool [50] written in Python. We also call the MiniCard SAT solver [39] and the Z3 SMT solver [11]. The altered CBMC (plus ∼1K LOC) takes as input a family simulator, and generates a `gsmt2` file containing SMT formulas for all possible mutations of the corresponding statements in the input program. The AllRepair ($\sim$2K LOC) takes as input a `gsmt2` file, generates formulas for SAT and SMT solving, and handles all calls to them.

The tool accepts programs written in C with `#ifdef`/`#if` directives. It uses three main parameters: *mutation level* that defines the kind of mutations that will be applied to feature and program expressions; *unwinding bound b* that shows how many times loops and recursive functions will be inlined; and *repair size r* that specifies how many mutations will be applied at most to buggy programs. We use two mutation levels: *level 1* contains simpler mutations that are often sufficient for repairment, while *level 2* contains all possible mutations we apply. For each type of feature and program expression, the list of syntactic mutations/edits in level 1 and level 2 is given below:

type of exp.	level 1	level 2
`arithmetic op.`	$\{+,-\},\{*,\%,\div\}$	$\{+,-,*,\%,\div\}$
`relational op.`	$\{<,\leq\},\{>,\geq\},\{==,!=\}$	$\{<,\leq,>,\geq,==,!=\}$
`logical op.`	$\{\&\&,\|\|\}$	$\{\&\&,\|\|\}$
`bit-wise op.`	$\{>>,<<\},\{\&,\|,\hat{}\ \}$	$\{>>,<<,\&,\|,\hat{}\ \}$
`program vars`		$x\to 0, x\to -x$
`integer constants`		$n\to n+1, n\to n-1, n\to -n, n\to 0$
`feature vars`	$A\to\neg A, \neg A\to A$	$A\to\neg A, \neg A\to A$

For example, 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. On the other hand, in mutation level 2 we have one set $\{+,-,*,\%,\div\}$, which means that any arithmetic operator from the set can be replaced with any other. Mutations on feature variables $A\in\mathbb{F}$ in both levels include negations of feature variables ($A\to\neg A, \neg A\to A$), whereas for program variables $x\in \mathit{Var}$ in level 2 we have mutations for replacing them with 0 ($x\to 0$) and changing the sign ($x\to -x$). Integer constants $n\in\mathbb{Z}$ in mutation level 2 can be increased by one, decreased by one, minused, or replaced with 0.

Experimental setup and Benchmarks.

Experiments are run on 64-bit Intel®Core^{TM} i7-1165G7 CPU@2.80GHz, VM Ubuntu 22.04.3 LTS, with 8 GB memory. We use a timeout value of 400 sec. The implementation, benchmarks, and all obtained results are available from: `https://zenodo.org/records/11179373`. For the aim of evaluation, we ran: (1) our tool with mutation level 1, denoted SPLAllRepair$_1$; (2) our tool with mutation level 2, denoted SPLAllRepair$_2$; and (3) the `Brute-force` approach that uses a preprocessor to generate all variants of a program family and then applies the single-program repair tool AllRepair to each individual variant independently.

The evaluation is performed on a dozen of C programs: two warming-up examples (`intro1` in Fig. 2 and `feat-inter` in Fig. 6); four commonly known algorithms (`feat_power` in Fig. 7, `factorial` in Fig. 8, `sum` in Fig. 9 and `sum_mton` in Fig. 10); `Codeflaws` [53], `TCAS` [29], and `Qlose` [10] benchmarks that are widely used for evaluating program repair tools [10, 37, 46, 50, 51]; as well as `MinePump` system [38] from the `product-lines` category of `SV-COMP 2024` (`https://sv-comp.sosy-lab.org/2024`) that is often used to assess product-line verification in the SPL community [4, 9, 56, 55]. `Codeflaws` consists of programs taken from buggy user submissions to the programming contest site Codeforces (`http://codeforces.com`). For each program, there is a correct reference version and several buggy versions. Traffic Alert and Collision Avoidance System (`TCAS`) represents an aircraft collision

```
void main(){
  int x := 0;
  #if (A) x := x+2; #endif
  #if (B ∧ C) x := x-2; #endif
  assert (x ≥ 0 && x < 4);
}
```

Figure 6 feat-inter.

```
int feat_power(int n){
  assume(n ≥ 1);
  int res := 0;
  #if (¬A) int i := 1;
    #else int i := 0; #endif
  while (i < 3) {
    res=res*n;
    i++; }
  #if (A) assert (sum==n*n*n);
    #else assert (sum==n*n*n*n); #endif
  return res; }
```

Figure 7 feat_power.

```
void main(int n){
  assume(n ≥ 0);
  int res1 := fact(n);
  int res2 := fact_correct(n);
  assert (res1 == res2);
}
int fact_correct(int x){
  int res=1;
  for (int i=2; i ≤ x; i++)
    res *= i;
  return res;
}
```

```
int fact(int x){
  int res=1, i=2;
  while (#if (A) (i<x) #else (i ≤ x) #endif){
    res = mult(res,i);
    i++; }
  return res; }
int mult(int x, int y){
  int res=0;
  for (int i=1; i ≤ y; i++)
    #if (B) res-=x; #else res+=x; #endif
  return res;
}
```

Figure 8 factorial

detection system used by all US commercial aircrafts. The TCAS benchmark suite consists of a reference (correct) implementation and 41 faulty versions. In our experiments, we use 10 faulty versions that can be repaired using the mutations we apply in our approach. The Qlose benchmarks are used for evaluating the Qlose program repair tool [10], which consist of a reference (correct) implementation and several faulty versions for each programming task. In the case of Codeflaws, TCAS, and Qlose, we have selected several faulty versions of each benchmark and we have created a buggy program family out of them. For example, we use tcas_v3 and tcas_v12 (resp., tcas_v16 and tcas_v17) to create the tcas_spl1 (resp., tcas_spl2) program family. Then, we use assertions to check the equivalence of the results returned by the program family and the reference (correct) version (for example, see main() of factorial in Fig. 8). Note that the correct version is marked so that it will not be mutated. The MinePump SPL system contains 730 LOC and six independent optional features: start, stop, methaneAlarm, methaneQuery, lowWaterSensor, highWaterSensor. When activated, the controller should switch on the pump when the water level is high, but only if there is no methane in the mine. We consider two specifications of the MinePump system encoded as assertions in SV-COMP 2024: MinePump_spec1 checks whether the pump is not running if the level of methane is critical; and MinePump_spec3 checks whether the pump is running if the level of water is high. Table 1 presents characteristics of the benchmarks, such as: the file name (Benchmark), the number of features $|\mathbb{F}|$ (note that $|\mathbb{K}| = 2^{|\mathbb{F}|}$), and the lines of code (LOC).

```
int sum(int n){
  assume(n ≥ 1);
  int sum := 0, i := 0;
  #if (A) i := 1; #endif
  while (i < n) {
    #if (B) sum+=i;
      #else sum-=i; #endif
    i++; }
  assert (sum==n*(n+1)/2);
  return sum;
}
```

Figure 9 sum.

```
int sum_mton(int n, int m){
  assume(n ≥ 1&&m ≥ 1);
  #if (A) assume(n ≥ m);
    #else assume(m ≥ n); #endif
  int sum := 0;
  #if (A) int i := n;
    #else int i := m; #endif
  while (#if (A) (i ≤ n) #else (i ≤ m) #endif)
  {  sum:=sum-i;
     i++; }
  #if (A) assert(sum==(n*(n+1)-m*(m-1))/2);
  #else assert(sum==(m*(m+1)-n*(n-1))/2);
  #endif
  return sum;}
```

Figure 10 sum_mton.

Examples.

We now present several of our examples in detail. Consider the program family `feat-inter` in Fig. 6. The error occurs due to the feature interaction ($\neg$A $\wedge$ B $\wedge$ C). In particular, the variant ($\neg$A $\wedge$ B $\wedge$ C) is: `int x=0; x=x-2; assert(x≥0 && x<4)`. So the assertion fails since `x` has value -2 at the assertion location. The simplest fix from mutation level 1, which replaces `x:=x-2` with `x:=x+2`, does not work as it introduces a new error in other variants. In this case, the feature interaction (A $\wedge$ B $\wedge$ C) causes the assertion failure since the value of `x` will be 4 at the assertion location for variant (A $\wedge$ B $\wedge$ C). Therefore, SPLALLREPAIR$_1$ reports that no repair is found by searching the space of 7 mutants in 0.254 sec. However, if we consider mutations of level 2 then SPLALLREPAIR$_2$ successfully finds a repair, which replaces `x:=x-2` with `x:=x-0`, by searching the space of 25 mutants in 0.315 sec. On the other hand, the `Brute-force` approach applies mutations to all faulty variants independently. As the only faulty variant is ($\neg$A $\wedge$ B $\wedge$ C), it will report the repair that replaces `x:=x-2` with `x:=x+2`. This is a correct repair for the variant ($\neg$A $\wedge$ B $\wedge$ C), but not for the entire family. This example shows that sometimes the `Brute-force` approach may not report correct results due to the feature interaction.

The program family `feat_power` in Fig. 7 should find the third power of `n` when feature A is enabled and the fourth power of `n` when A is disabled. SPLALLREPAIR$_1$ suggests fixes in 0.722 sec that replace the feature expression ($\neg$ A) with (A) when initializing variable `i` and replace `while`-guard (`i` $<$ 3) with (`i` $\leq$ 3). The `Brute-force` finds that variant (A) is correct, but variant ($\neg$ A) is not correct and no fix is suggested as integer constants cannot be mutated in level 1. Some possible repairs of variant ($\neg$ A) in level 2 will make variant (A) incorrect. For example, changing the `while`-guard to (`i` $\leq$ 3) will make variant (A) incorrect since it is initialized to `0` so it will return the fourth power of `n` instead of the third.

The program `factorial` in Fig. 8 contains two implementations of the factorial function: a correct one, called `fact_correct`, and a buggy one, called `fact`, that represents a program family with four variants. The assertion requires that the results returned from each variant of `fact` are equivalent with the result returned from `fact_correct`. We do not apply mutations to `fact_correct`, but only to the program family `fact`. All three approaches suggest fixes that replace the `while`-guard (`i` $<$ `x`) with (`i` $\leq$ `x`) and the assignment `res-=x` with `res+=x`.

Consider the program family `sum` in Fig. 9, which computes the sum of all integers from 0

Table 1 Performance results of SPLALLREPAIR$_1$ vs. SPLALLREPAIR$_2$ vs. `Brute-force`. All times in sec.

Benchmarks	$\vert\mathbb{F}\vert$	LOC	SPLALLREPAIR$_1$			SPLALLREPAIR$_2$			Brute-force		
			Fix	Space	Time	Fix	Space	Time	Fix	Space	Time
`intro1`	2	20	✓	7	0.252	✓	25	0.304	✓	5	0.981
`feat-inter`	3	20	×	7	0.254	✓	25	0.315	×	9	2.110
`feat_power`	1	20	✓	16	0.722	✓	403	7.79	×	8	0.882
`factorial`	2	50	✓	86	2.540	✓	1603	107.3	✓	81	4.196
`sum`	2	30	✓	17	0.376	✓	266	2.656	✓	18	1.147
`sum_mton`	1	20	✓	32	0.770	✓	681	15.22	×	10	0.556
`4-A-Codeflaws`	2	95	×	52	0.426	✓	1390	2.578	×	36	1.180
`651-A-Codeflaws`	2	85	✓	180	3.394	✓	2829	38.53	✓	237	5.78
`tcas_spl1`	1	305	×	37	0.99	✓	158	6.10	×	37	1.41
`tcas_spl2`	1	305	×	38	1.19	✓	164	8.94	×	38	1.47
`Qlose_multiA`	3	32	×	122	0.711	✓	5415	69.21	×	65	5.781
`Qlose_iterPower`	2	30	×	9	0.973	✓	38	2.921	×	16	1.391
`MinePump_spec1`	6	730	✓	38	300.0	$\underline{\checkmark}$	-	timeout	$\underline{\checkmark}$	-	timeout
`MinePump_spec3`	6	730	✓	39	291.0	$\underline{\checkmark}$	-	timeout	$\underline{\checkmark}$	-	timeout

to a given input integer `n`. The specification indicates that given a positive input `n` ($\texttt{n} \geq 1$), the output represented by the variable `sum` is `n*(n+1)/2`. The body of `sum` is implemented in an iterative fashion. There are two features `A` and `B` that enable different initializations of `i` and different updates of `sum`. Let us consider mutations of level 1. If the repair size is 1 (i.e., only one original expression can be mutated), our tool cannot find a repair by searching the space of 7 mutants in 0.321 sec. However, if the repair size is 2, then SPLALLREPAIR$_1$ suggests a fix that replaces the `while`-guard ($\texttt{i} < \texttt{n}$) with ($\texttt{i} \leq \texttt{n}$) and the assignment `sum-=i` with `sum+=i`. The search space contains 17 mutants and the tool explores it in 0.376 sec. The `Brute-force` approach reports a correct repair in 1.147 sec.

The program family `sum_mton` in Fig. 10 computes the sum $\texttt{m} + (\texttt{m}+1) + \ldots \texttt{n}$ when feature `A` is enabled and ($\texttt{n} \geq \texttt{m}$), and the sum $\texttt{n} + (\texttt{n}+1) + \ldots \texttt{m}$ when feature `A` is disabled and ($\texttt{m} \geq \texttt{n}$). The corresponding specifications assert that the returned value `sum` is equal to $\big(\texttt{n} * (\texttt{n}+1) - \texttt{m} * (\texttt{m}-1)\big)/2$ when `A` is on and $\big(\texttt{m} * (\texttt{m}+1) - \texttt{n} * (\texttt{n}-1)\big)/2$ when `A` is off. The programmer has made two mistakes: when initializing variable `i` and when updating variable `sum` in the `while`-body. SPLALLREPAIR$_1$ suggests fixes in 0.770 sec that replace the feature expression (`A`) with ($\neg$ `A`) when initializing variable `i` and replace `sum:=sum-i` with `sum:=sum+i` when updating `sum`. However, the `Brute-force` cannot fix any of the two variants since mutating variable `n` (resp., `m`) to other variable `m` (resp., `n`) is not allowed.

Performance.

Table 1 shows performance results of running SPLALLREPAIR$_1$, SPLALLREPAIR$_2$, and the `Brute-force` approach on the given benchmarks. We use mutation level 1 for `Brute-force`. Note that the `Brute-force` approach calls translation, mutation, and repair procedures for each variant separately, whereas SPLALLREPAIR$_1$ and SPLALLREPAIR$_2$ call these procedures only once per program family. Moreover, the `Brute-force` approach can only find repairs by mutating program expressions. The default values for unwinding bound is

Table 2 Performance results of SPLAllRepair$_1$ for different values of the unwinding bound $b = 2, 5, 8$. All times in sec.

Benchmarks	$b = 2$		$b = 5$		$b = 8$	
	`Fix`	`Time`	`Fix`	`Time`	`Fix`	`Time`
`feat_power`	×	0.254	✓	0.722	✓	0.978
`factorial`	×	1.231	✓	3.540	✓	6.524
`sum`	×	0.304	✓	0.376	✓	0.456
`sum_mton`	×	0.589	✓	0.770	✓	0.922
`651-A-Codeflaws`	✓	1.814	✓	3.394	✓	6.828

$b = 5$ and for repair size is $r = 1$. However, for some benchmarks whose repaired versions contain more than one code mutation, we use the minimal value of repair size r that allows one approach to find a correct solution. For example, we use repair size $r = 2$ for `sum`. For each approach, there are three columns: "`Fix`" that specifies with ✓ (resp., ×) whether the given approach finds (resp., does not find) a correct repair for a given benchmark; "`Space`" that specifies how many mutants have been inspected; and "`Time`" that specifies the total time (in seconds) needed for the given tasks to be performed.

From Table 1, we can see that SPLAllRepair$_1$ and SPLAllRepair$_2$ combined outperform the `Brute-force` approach with respect to both repairability and runtime. In particular, SPLAllRepair$_2$ fully repairs 12 benchmarks, which is better than 8 full correct repairs reported by SPLAllRepair$_1$ and 4 full correct repairs reported by the `Brute-force` approach that use the same mutations of level 1 (see also Discussion below). Note that SPLAllRepair$_2$ and the `Brute-force` timeout after 400 sec for the `MinePump` system. Hence, they report only a partial list of possible repairs, denoted by $\underline{\checkmark}$. On the other hand, SPLAllRepair$_1$ achieves time speed-ups compared to `Brute-force` when report the same results, that range from 1.2 to 4 times. If we compare SPLAllRepair$_1$ and SPLAllRepair$_2$, we can see that there is a trade-off between repairability and runtime. That is, SPLAllRepair$_2$ is more precise (12 vs. 8 fixes) but slower (from 1.2 to 42 times slow-down when report the same results) compared to SPLAllRepair$_1$.

Table 2 shows performance results of running SPLAllRepair$_1$ on a selected set of benchmarks for different unwinding bounds b. Recall that our approach reasons about loops by unrolling (unwinding) them, so it is sensitive to the chosen unwinding bound. By choosing larger bounds b, we will obtain more precise results (more genuine repairs), but we will also obtain longer SMT formulas and slower speeds of the repairing tasks. We can see that the running times of all repairing tasks grow with the number of bound b. This is due to the fact that longer SMT formulas are generated, which need more time to be verified. Of course, we will also obtain more precise results for bigger values of b, and less precise results (i.e., some genuine repairs will not be reported) for smaller values of b. Hence, there is a preision/speed tradeoff when choosing the unwinding bound b. We obtain similar results for SPLAllRepair$_2$ and the `Brute-force`.

Discussion

In summary, our experiments demonstrate that our tool outperforms the `Brute-force` approach, and moreover it can be used for repairing various SPLs with different sizes of LOC, configuration space, and mutation space. Although SPLAllRepair$_1$ and `Brute-force` have similar precision (8 vs. 4 fixes) due to the use of same sets of mutations, there is still a difference in the quality of the reported results. As we argued before, SPLAllRepair$_1$ and SPLAllRepair$_2$ report repaired program families obtained by fixing both feature and program expressions, whereas `Brute-force` only reports the repaired variants obtained by fixing program expressions. Hence, the results from `Brute-force` have to be analyzed by the user to produce information comparable to that returned by SPLAllRepair$_1$ and SPLAllRepair$_2$ in the form of repaired program families. Moreover, the fixes of individual variants may cause errors in other variants as evidenced by `feat-inter` and `feat_power`.

The main bottleneck for real-world SPLs, such as `MinePump` with 730 LOC and 6 features, is the huge space of mutants. The problem is that the search space of mutants grows very rapidly as the number of changeable expressions (statements) included in $S_{\texttt{soft}}$ grows. For example, the space of mutants for `MinePump` is $\sim 10^{12}$ for mutation level 1 and $\sim 10^{34}$ for mutation level 2. Hence, to explore even the sub-space of mutants with only 1 edit ($r = 1$) we need around 300 sec for SPLAllRepair$_1$ and >400 sec (timeout) for SPLAllRepair$_2$. One way to address this problem is to use variability fault localization [5, 47], which will first identify feature and program expressions relevant for a variability bug, so that the SPL repair algorithm will apply mutations only to those expressions. This way, we will significantly reduce the space of all mutants without dropping any potentially correct mutant, and so we will improve the performance of the SPLAllRepair algorithm.

The runtime performance results confirm that our lifted (family-based) repair algorithm is indeed effective and especially so for large values of $|\mathbb{F}|$ and $|\mathbb{K}| = 2^{|\mathbb{F}|}$. That is, the focus of lifted repair algorithm is to combat the configuration space explosion of SPLs, not their LOC or mutation space sizes. As an experiment, we took `feat-inter`, and we have gradually added optional features into it by conjoining them to the presence conditions of `#if`-s. For $|\mathbb{F}| = 3$, SPLAllRepair$_1$ achieves speed-up of 8.3 times compared to `Brute-force`, whereas for $|\mathbb{F}| = 4$ and $|\mathbb{F}| = 5$ we observe speed-ups of 14.7 and 26.7 times, respectively. The key for those speed-ups is the linear growth of the running times of SPLAllRepair$_1$ with the number of features $|\mathbb{F}|$ compared to the exponential growth of the running times of `Brute-force` with $|\mathbb{F}|$.

Finally, the evaluation shows that for bigger values of the unwinding bound b, we obtain repairing tasks with slower runtime speed, but reporting more precise results.

6 Related Work

We divide our discussion of related work into two categories: lifted SPL analysis and program repair.

Lifted SPL analysis

Formal analysis and verification of program families have been a topic of considerable research in recent times. The challenge is to develop efficient techniques that work directly on program families, rather than on single programs. Various lifted techniques have been introduced that lift existing single-program analysis techniques to work on the level of program families. Some examples are lifted syntax checking [27, 34], lifted type checking [7, 33], lifted static analysis [6, 30, 15, 20, 55], lifted model checking [9, 16, 25], etc. There are two main lifted techniques: to develop dedicated lifted (family-based) algorithms and tools (e.g. [9, 7, 6, 20]); or to use specific simulators and variability encodings which transform program families into single programs that can be analyzed by the standard single-program analysis tools. The two approaches have different strengths and weaknesses. The advantage of the dedicated lifted algorithms is that precise (conclusive) results are reported for every variant, but the disadvantage is that their implementation and maintenance can be labor intensive and expensive. For example, CBMC [8] is prominent (single-system) software model checker that contains many optimization algorithms, which are result of more than two decades research in advanced formal verification. Adapting and implementing all these algorithms in the context of lifted software model checking would require an enormous amount of work. Moreover, the performance of dedicated lifted algorithms still heavily depends on the size and complexity of the configuration space of the analyzed SPL.

On the other hand, the approaches based on variability encoding [30, 56] generate a family simulator that simulates the behaviour of all variants in an SPL. They re-use existing tools from single-program world, but some precision may be lost when interpreting the obtained results. The work [56] defines variability encoding on the top of TypeChef parser [34] for C and Java SPLs, while the work [30] defines variability encoding on the top of SuperC parser [27] for C SPLs. The results of variability encoding have been applied to testing [35], software model checking [4], formal verification [30], and theorem proving [54] of SPLs. In this work, we pursue this line of research by presenting a lifted repair algorithm that is based on variability encoding of program families and an existing single-program mutation-based repair algorithm AllRepair [50, 51].

Program repair

Automated program repair has been extensively examined in software engineering as a way to efficiently maintain software systems [28, 37, 40, 42, 45, 46, 48, 50, 51]. These works aim to repair the buggy program, so that the transformed program does not exhibit any faults. Most of them use test suits as the only specification, so the correctness of a candidate is checked by running all tests in the test suite against it. They iteratively generate a candidate from the repair search space and check its validity by testing. Some examples are GenProg [28], RsRepair [48], SPR [40]. The main problem of all testing-based approaches is the generation of overfitting repairs that pass all the test cases, but they break some untested required functionality of correct programs. This happens when the test suites do not cover all the functionality of a program.

In contrast to testing-based approaches, our work belongs to the category of repair tools that use formal techniques to guide the repair process. Several techniques, such as SemFix [45] and Angelix [42], use symbolic execution to find a repair constraint and then generate a correct fix based on it. Similarly to our work, Könighofer et al. [37] also use assertions as formal specifications, but instead of mutations they use on-the-fly concolic execution (a variant of symbolic execution that uses both symbolic and concrete input values) and templates (linear expressions of program variables with unknown coefficients) as repairs. The solutions for unknown coefficients are found by SMT solving, thus discovering the repaired program. The Maple tool [46] utilizes a formal verification system to locate buggy expressions, which are again replaced with templates in which the unknown coefficients are determined using constraint solving. The work [36] uses a deductive synthesis framework for repairing recursive functional programs with respect to specifications expressed in the form of pre- and post-conditions.

Finally, our approach is inspired by Rothenberg and Grumberg [50, 51] that have developed the AllRepair tool for automatic program repair based on code mutations. In this paper, we pursue this line of work by applying it in a new context of SPL repair, which is done by taking into account all specific characteristics of SPLs. This way, we broaden the space of programs that can be repaired.

The Qlose tool [10] introduces a quantitative program repair algorithm that finds the "optimal" solutions by taking into account multiple quantitative objectives, such as the number of syntactic edits and semantic changes in program behaviours/executions. The work [41] proposes a semantic program repair technique that performs counterexample-guided inductive repair loop via symbolic execution. In this work, we currently find a solution with minimal number of syntactic changes to the original program family. The semantics of the program family is encoded as an SMT formula that is mutated and checked for correctness by an SMT solver. In the future, we plan to investigate some semantics-based learning techniques that will use the counterexamples returned by the SMT solver to guide the algorithm towards finding faster solutions.

Automated program repair has often been combined with fault localization. Fault localization [31, 17] is a technique for automatically generating concise error explanations in the form of locations/statements relevant for a given error that describe why the error has occurred. The works [12, 49, 51] use fault localization to narrow down the repair search space, followed by applying program repair. Firstly, fault localization suggests locations in the erroneous program that might be the cause of the error. Subsequently, the program repair attempts to change only those locations detected by the fault localization in order to eliminate the error. This way, the original program repair procedure is speeded up without incurring any precision loss. Recently, variability fault localization in buggy SPL systems has also been a subject of research [5, 44, 47]. They use spectrum-based fault localization (SBFL) metric [1] to calculate the suspiciousness scores for localizing variability bugs at the level of features [5] and statements [44, 47] based on the test information (program spectra). We can combine the variability fault localization and our variability-aware repair method to additionally prune the search space of mutants, thus improving the performances of our approach.

Program repair is also related to program sketching [52], where a program with missing parts (holes) has to be completed in such a way that a given specification is satisfied. One of the earliest and widely-known approach to solve the sketching problem is the Sketch tool [52], which uses SAT-based counterexample-guided inductive synthesis. It iteratively performs SAT queries to find integer constants for the holes so that the resulting program is correct on all possible inputs. The works [19, 21] introduce the FamilySketcher tool that solves the sketching problem by using a lifted static analysis based on abstract interpretation. The key idea is that all possible sketch realizations represent a program family, and so the sketch search space is explored via an efficient lifted analysis of program families, which uses a specifically designed decision tree abstract domain. The FamilySketcher also generates an optimal solution to the sketching problem with respect to the number of execution steps to termination. Furthermore, the approach [18] uses abstract static analysis and logical abduction to solve the generalized program sketching problem where the missing holes can be replaced with arbitrary expressions, not only with integer constants as in the case of Sketch and FamilySketcher tools.

7 Conclusion

In this paper, we have introduced an automated SPL repair framework using variability encoding, bounded model checking and cooperation between SAT and SMT solvers. More specifically, we utilize the CBMC bounded model checker to translate the family simulator of a program family to a program formula. By checking the satisfiability of the program formula using an SMT solver, we verify the correctness of the given program family. Then, each formula corresponding to a buggy (feature or program) expression is replaced by a mutated patch, to create a new SMT formula that is again checked for satisfiability. To ensure that only minimally mutated programs are considered, we call a SAT solver. By experiments we have shown that our prototype tool can discover interesting patches for various buggy SPLs.

The huge space of mutants can be a bottleneck when dealing with real-world SPLs that have high sizes of LOCs and features. To overcome this problem, we can consider different techniques for pruning the search space of all possible mutations in the future. One possibility is to use variability fault localization [5, 47], which will find statements relevant for the variability bug. The formulas corresponding to all other statements will be included in $S_{\texttt{hard}}$ and so no mutations will be applied to them. By mutating only statements relevant for the bug, we will significantly reduce the space of all mutants, thus speeding up the SPL repair method without any precision loss.

References

1 Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 88–99. IEEE Computer Society, 2009. doi:10.1109/ASE.2009.25.

2 Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

3 Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, pages 372–375, 2011. URL: http://dx.doi.org/10.1109/ASE.2011.6100075, doi:10.1109/ASE.2011.6100075.

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

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

6 Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl^{lift}: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.

7 Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40, 2012. URL: http://doi.acm.org/10.1145/2364527.2364535, doi:10.1145/2364527.2364535.

8 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. doi:10.1007/978-3-540-24730-2_15.

9 Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86, doi:10.1109/TSE.2012.86.

10 Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II*, volume 9780 of *LNCS*, pages 383–401. Springer, 2016. doi:10.1007/978-3-319-41540-6_21.

11 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. `doi:10.1007/978-3-540-78800-3\_24`.

12 Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010*, pages 65–74. IEEE Computer Society, 2010. `doi:10.1109/ICST.2010.66`.

13 Aleksandar Dimovski and Danilo 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*, volume 2 of *IEEE*, pages 604–607, 2003. `doi:10.1109/TELSKS.2003.1246297`.

14 Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.

15 Aleksandar S. Dimovski. Lifted static analysis using a binary decision diagram abstract domain. In *Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019*, pages 102–114. ACM, 2019. `doi:10.1145/3357765.3359518`.

16 Aleksandar S. Dimovski. Ctl* family-based model checking using variability abstractions and modal transition systems. *Int. J. Softw. Tools Technol. Transf.*, 22(1):35–55, 2020. `doi:10.1007/s10009-019-00528-0`.

17 Aleksandar S. Dimovski. Error invariants for fault localization via abstract interpretation. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings*, volume 14284 of *LNCS*, pages 190–211. Springer, 2023. `doi:10.1007/978-3-031-44245-2\_10`.

18 Aleksandar S. Dimovski. Generalized program sketching by abstract interpretation and logical abduction. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings*, volume 14284 of *LNCS*, pages 212–230. Springer, 2023. `doi:10.1007/978-3-031-44245-2\_11`.

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

20 Aleksandar S. Dimovski and Sven Apel. Lifted static analysis of dynamic program families by abstract interpretation. In *35th European Conference on Object-Oriented Programming, ECOOP 2021*, volume 194 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.14`.

21 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. Program sketching using lifted analysis for numerical program families. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*, volume 12673 of *LNCS*, pages 95–112. Springer, 2021. `doi:10.1007/978-3-030-76384-8\_7`.

22 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. Several lifted abstract domains for static analysis of numerical program families. *Sci. Comput. Program.*, 213:102725, 2022. URL: `https://doi.org/10.1016/j.scico.2021.102725`, `doi:10.1016/J.SCICO.2021.102725`.

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

24 Aleksandar S. Dimovski and Ranko Lazic. Compositional software verification based on game semantics and process algebra. *Int. J. Softw. Tools Technol. Transf.*, 9(1):37–51, 2007. URL: `https://doi.org/10.1007/s10009-006-0005-y`, `doi:10.1007/S10009-006-0005-Y`.

25 Aleksandar S. Dimovski and Andrzej Wasowski. From transition systems to variability models and from lifted model checking back to UPPAAL. In *Models, Algorithms, Logics and Tools*, volume 10460 of *LNCS*, pages 249–268. Springer, 2017. `doi:10.1007/978-3-319-63121-9_13`.

26 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003. `doi:10.1007/978-3-540-24605-3\_37`.

27 Paul Gazzillo and Robert Grimm. Superc: parsing all of C by taming the preprocessor. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 323–334. ACM, 2012. URL: `http://doi.acm.org/10.1145/2254064.2254103`, `doi:10.1145/2254064.2254103`.

28 Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012. `doi:10.1109/TSE.2011.104`.

29 Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam A. Porter, and Gregg Rothermel. An empirical study of regression test selection techiques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001. `doi:10.1145/367008.367020`.

30 Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 597–607, 2015. URL: `http://dx.doi.org/10.1109/ASE.2015.84`, `doi:10.1109/ASE.2015.84`.

31 Manu Jose and Rupak 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*, pages 437–446. ACM, 2011. `doi:10.1145/1993498.1993550`.

32 Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* PhD thesis, University of Magdeburg, Germany, May 2010.

33 Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.

34 Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA'11*, pages 805–824. ACM, 2011.

35 Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *FOSD '12*, pages 1–8, 2012.

36 Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II*, volume 9207 of *LNCS*, pages 217–233. Springer, 2015. `doi:10.1007/978-3-319-21668-3\_13`.

37 Robert Könighofer and Roderick Bloem. Repair with on-the-fly program analysis. In *8th International Haifa Verification Conference, HVC 2012*, volume 7857 of *LNCS*, pages 56–71. Springer, 2012. `doi:10.1007/978-3-642-39611-3\_11`.

38 Jeff Kramer, Jeff Magee, Morris Sloman, and A. Lister. Conic: An integrated approach to distributed computer control systems. *IEE Proc.*, 130(1):1–10, 1983.

39 Mark H. Liffiton and Jordyn 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*, volume 7317 of *LNCS*, pages 485–486. Springer, 2012. `doi:10.1007/978-3-642-31612-8\_47`.

40 Fan Long and Martin C. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on ESEC/FSE 2015*, pages 166–178. ACM, 2015. `doi:10.1145/2786805.2786811`.

41 Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*, pages 129–139. ACM, 2018. `doi:10.1145/3180155.3180247`.

42 Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 691–701. ACM, 2016. `doi:10.1145/2884781.2884807`.

43 Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015. URL: `http://dx.doi.org/10.1016/j.scico.2015.04.005`, `doi:10.1016/j.scico.2015.04.005`.

44 Kien-Tuan Ngo, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Variability fault localization: a benchmark. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, pages 120–125. ACM, 2021. `doi:10.1145/3461001.3473058`.

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

46 Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. Automatic program repair using formal verification and expression templates. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Proceedings*, volume 11388 of *LNCS*, pages 70–91. Springer, 2019. `doi:10.1007/978-3-030-11245-5\_4`.

47 Thu-Trang Nguyen, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. A variability fault localization approach for software product lines. *IEEE Trans. Software Eng.*, 48(10):4100–4118, 2022. `doi:10.1109/TSE.2021.3113859`.

48 Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE '14*, pages 254–265. ACM, 2014. `doi:10.1145/2568225.2568254`.

49 Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Jaan Raik, Raimund Ubar, Giuseppe Di Guglielmo, Graziano Pravadelli, and Franco Fummi. Combining dynamic slicing and mutation operators for ESL correction. In *17th IEEE European Test Symposium, ETS 2012*, pages 1–6. IEEE Computer Society, 2012. `doi:10.1109/ETS.2012.6233020`.

50 Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 593–611, 2016. `doi:10.1007/978-3-319-48989-6\_36`.

51 Bat-Chen Rothenberg and Orna Grumberg. Must fault localization for program repair. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II*, volume 12225 of *LNCS*, pages 658–680. Springer, 2020. `doi:10.1007/978-3-030-53291-8\_33`.

52 Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. `doi:10.1007/s10009-012-0249-7`.

53 Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017 - Companion Volume*, pages 180–182. IEEE Computer Society, 2017. `doi:10.1109/ICSE-C.2017.76`.

54 Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. Family-based deductive verification of software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 11–20. ACM, 2012. URL: `http://doi.acm.org/10.1145/2371401.2371404`, `doi:10.1145/2371401.2371404`.

55 Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 27(4):18:1–18:33, 2018. `doi:10.1145/3280986`.

56 Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *J. Log. Algebraic Methods Program.*, 85(1):125–145, 2016. `doi:10.1016/j.jlamp.2015.06.007`.