On Synthesizing Presence Conditions in Numerical Software Product Lines

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

Abstract

This paper introduces a novel approach for synthesizing assertion-safe `#if`-guards (presence conditions) in numerical program families (Software Product Lines - SPLs). Given a partial SPL with missing `#if`-guards, our approach synthesizes concrete predicates defined over numerical features that are strong enough to prove the assertions in all variants of the given SPL. Moreover, the `#if`-guards synthesized by our approach are the simplest and the weakest among all possible that guarantee the assertion validity. To address this problem, we use abstract interpretation-based forward and backward lifted (SPL) analyses to generate constraints that are then solved by using the logical abduction technique.

We have implemented our approach in a prototype synthesis tool, called `SPLSynthesize`, for `#if`-annotated SPLs written in C. We show that our tool is able to successfully synthesize assertion-safe presence conditions defined over numerical features for some interesting C program families.

CCS Concepts

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

Keywords

`#if`-guard Synthesis, Abstract Interpretation, SPL Lifted Analysis, Logical Abduction

ACM Reference Format:
Aleksandar S. Dimovski. 2025. On Synthesizing Presence Conditions in Numerical Software Product Lines. In *29th ACM International Systems and Software Product Line Conference - Volume A (SPLC-A '25), September 01–05, 2025, A Coruña, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3744915.3748474

1 Introduction

Software Product Line (SPL) Engineering [6] has been successfully applied in industry to meet the need for custom-tailored software via building families of related software systems. End users can derive different single programs, called *variants*, from an SPL (program family) that are suitable to particular application scenarios. The variants (family members) are specified in terms of concrete values assigned to numerical *features* (statically configured options) [7] relevant for that particular variant at compile-time. This way, the features are used to control the presence and absence of software functionality. The popular `#if` directives from the C preprocessor CPP [32] represent the most common way to implement SPLs. An `#if` 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 today quite popular in the embedded and safety-critical systems.

Recently, various verification and static analysis techniques have been developed to ensure the semantic correctness of SPLs by using specialized so-called *lifted* (SPL) algorithms [1, 3–5, 15, 16, 25, 28, 29, 36–38], which allow simultaneous verification/analysis of all variants of an SPL in a single run by exploiting the commonalities between the variants. These tools can detect various types of bugs (errors). However, the identified bugs are difficult to understand, debug, and fix. This is especially so due to the variable nature of the bugs in SPLs, *variability bugs* [23, 35], which may only be exposed in certain variants under some selections of features but not in the other variants. This property is the source of substantial difficulties for analyzing/designing SPLs.

In this paper, we introduce a novel synthesis approach for the design of safe SPLs with respect to assertions. In particular, suppose that a programmer wishes to write a region of code S implementing a given functionality, but S can make assertions to fail in some variants. In our approach, the programmer embeds S within the scope of an unknown `#if`-guard ?? given in the following form: "`#if` (??) S `#endif`". Our approach uses a new guard synthesis algorithm to compute a predicate G over features (i.e., feature expression), such that when ?? is replaced by G, all assertions within the resulting complete SPL (i.e., within all its variants) become valid. Therefore, our approach goes beyond verification/static analysis: it not only guarantees the assertion validity, but also helps programmers write safe-by-construction code. That is, the programmers are only asked to mark which code snippets must be protected by an `#if`-guard, and our approach will statically infer a single feature expression that guarantees the assertion validity of all variants in the SPL.

Our synthesis algorithm consists of two phases: *constraint generation* based on lifted (SPL) abstract analyses and *constraint solving* based on logical abduction. More specifically, in *the first phase*, we use forward and backward lifted analyses of SPLs [15, 26, 28] based on abstract interpretation [8, 34]. The forward analysis works similarly to strongest postcondition inference by propagating the

SPLC-A '25, A Coruña, Spain

ACM ISBN 979-8-4007-2024-6/25/09
https://doi.org/10.1145/3744915.3748474

input fact towards the final locations, thus computing the lifted invariant postcondition before the hole. The backward analysis works similarly to weakest precondition inference by propagating the assertion fact towards the initial locations, thus computing the sufficient precondition that ensures assertion validity after the hole. The elements of the lifted analysis domain are *decision-trees*, in which decision nodes are labelled with linear constraints over numerical features, whereas leaf nodes are labelled with linear constraints over program variables. The decision nodes recursively partition the space of all variants, whereas the program properties at the leaf nodes provide analysis information corresponding to each partition. The efficiency of the decision-tree lifted domain comes from the opportunity to share equal subtrees, in case some program properties are independent from some features. Let us assume that the forward lifted analysis computes a lifted invariant (premise) P representing the facts known before the hole, while the backward lifted analysis computes a lifted sufficient precondition (conclusion) C that guarantees that the code after the hole satisfies all assertions. Subsequently, in *the second phase*, we use abduction [2, 10, 14, 24] to find missing hypothesis in a logical inference task. That is, the abduction infers the simplest and most general explanation E such that $P \wedge E \models C$ and $P \wedge E \not\models \text{false}$. The first condition states that the abduction solution E together with premise P should imply conclusion C, whereas the second condition states that the abduction solution E should not contradict premise P. Finally, we use the explanation E to synthesize a feature expression for the missing hole ??.

We have implemented our approach in a prototype tool, called `SPLSynthesize`. For the constraint generation phase, the tool uses the numerical abstract domains (e.g., Polyhedra [9]) from the APRON library [31] for generating linear constraints, and the decision-tree lifted domain from the SPLNum^{2}Analyzer tool [26, 28] for performing lifted (SPL) abstract analyses. For the constraint solving phase, the tool calls the EXPLAIN tool [10] for logical abduction and the MISTRAL tool [11] for SMT solving. We experimentally evaluate this approach on a selected set of benchmarks written in C, which are taken from the SV-COMP and the literature [28, 33]. The experiments demonstrate the practicality of our approach. This work brings the following contributions: (1) We explore the idea of synthesizing arbitrary feature expressions for missing `#if`-guards in SPLs; (2) We show how this `#if`-guard synthesis problem can be solved by interaction between lifted (SPL) abstract analyses and logical abduction; (3) We implement our approach, and we present the evaluation results for various interesting SPLs.

2 Motivating Examples

To illustrate our approach, we now present two motivating examples. Consider the SPL `intro.c` shown in Fig. 1, which contains an unknown `#if`-guard ??. There is only one available numerical feature `A` with domain $[0, 49] = \{0, 1, \ldots, 49\}$, which induces the following set of valid configurations $\mathcal{K} = \{(\texttt{A=0}), (\texttt{A=1}), \ldots, (\texttt{A=49})\}$. For each configuration from $\mathcal{K}$, we can derive one variant (single program) from `intro.c` by assigning a concrete value to the feature `A` and by appropriately resolving `#if` directives at compile-time. Hence, the variant for configuration (`A=0`) will have feature `A` set to `0` and so the statements at locs. ② and ③ become: `x = x+1`, whereas the variant for configuration (`A=10`) will have feature `A` set to `10` and so the statements at locs. ② and ③ become: `skip`. 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. For example, if the code at loc. ①, `x = x-1`, is included in variants satisfying $(\texttt{A} \leq 1)$ then the assertion will fail only for the variant (`A=2`) but it will be valid for all other variants. Our aim is to complete the hole ?? in loc. ④ so that the assertion is valid for all variants. Moreover, the synthesized guard for ?? should be the most general, so that the code in loc. ① is included in as many variants as possible.

Our approach proceeds in two phases, consisting of constraint generation and constraint solving. In *the first phase*, our approach performs forward and backward lifted abstract analyses that compute lifted numerical invariants and sufficient conditions. They are performed on a decision-tree (lifted) abstract domain parameterized by the Polyhedra abstract domain [9]. The leaves in the decision-tree lifted domain are polyhedra linear constraints over program variables (`x`), whereas decision nodes are polyhedra linear constraints over features (`A`). The edges are labelled with the truth value of the decision on the parent (decision) node: we use solid edges for true (i.e., the constraint in the parent node is satisfied) and dashed edges for false (i.e., the negation of the constraint in the parent node is satisfied). The (over-approximating) forward lifted abstract analysis of `intro.c` infers the lifted invariant represented by the decision-tree in Fig. 3a at loc. ④ before the hole. The (under-approximating) backward lifted abstract analysis of `intro.c` starts with the assertion fact $(\texttt{0} \leq \texttt{x} \leq \texttt{1})$ at loc. ⑤, and by propagating it backwards it infers the lifted precondition shown in Fig. 3b (resp., Fig. 3c) at loc. ① (resp., loc. ⑤) after the hole. Note that the lifted facts (invariants and preconditions) are disjunctive properties represented as decision-trees. For example, the decision-tree in Fig. 3a can be written as the following disjunctive formula (constraint) in first-order logic: $(\texttt{A} \leq \texttt{2} \wedge \texttt{x=2}) \vee (\texttt{3} \leq \texttt{A} \leq \texttt{5} \wedge \texttt{x=1}) \vee (\texttt{A} > \texttt{5} \wedge \texttt{x=0})$. Furthermore, note that in all decision-trees (disjunctive properties) the domains of features are implicitly assumed. For example, the decision-tree in Fig. 3b means that the fact $(\texttt{1} \leq \texttt{x} \leq \texttt{2})$ holds for all valid variants in $\mathcal{K}$, that is, it corresponds to the formula: $(\texttt{0} \leq \texttt{A} \leq \texttt{49} \wedge \texttt{1} \leq \texttt{x} \leq \texttt{2})$. We use these inferred facts to construct two abduction queries (constraints):

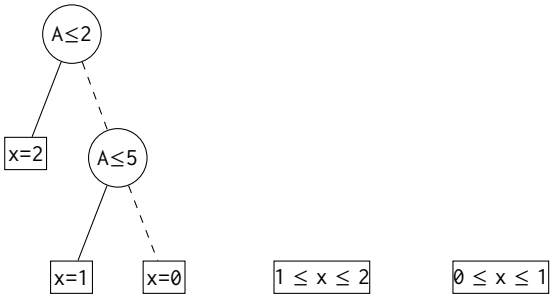
$$\big((\texttt{A} \leq \texttt{2} \wedge \texttt{x=2}) \vee (\texttt{3} \leq \texttt{A} \leq \texttt{5} \wedge \texttt{x=1}) \vee (\texttt{A} > \texttt{5} \wedge \texttt{x=0})\big) \wedge R_{tt}(\texttt{A}) \implies (\texttt{1} \leq \texttt{x} \leq \texttt{2})$$

$$\big((\texttt{A} \leq \texttt{2} \wedge \texttt{x=2}) \vee (\texttt{3} \leq \texttt{A} \leq \texttt{5} \wedge \texttt{x=1}) \vee (\texttt{A} > \texttt{5} \wedge \texttt{x=0})\big) \wedge R_{ff}(\texttt{A}) \implies (\texttt{0} \leq \texttt{x} \leq \texttt{1})$$

The left-hand side of the above implications encodes the generated constraints up to the hole ??, and the unknown predicates $R_{tt}(\texttt{A})$ and $R_{ff}(\texttt{A})$ encode the constraints over the feature `A`. The right-hand side of the above two implications encodes the postconditions ensuring that the assertion must be valid when the protected code of the `#if`-guard ?? (i.e., the statement at loc. ①) is taken and not taken into account, respectively. In *the second phase*, we use the abduction procedure [10] to solve the above queries (constraints) for $R_{tt}(\texttt{A})$ and $R_{ff}(\texttt{A})$. The logically weakest and simplest solutions we obtain are: $R_{tt}(\texttt{A}) \equiv (\texttt{A} \leq \texttt{5})$ and $R_{ff}(\texttt{A}) \equiv (\texttt{A} \geq \texttt{3})$. Then, we check whether $\neg R_{tt}(\texttt{A}) \implies R_{ff}(\texttt{A})$ using an SMT solver. The intuition

```
void main(){
① int x = 0;
② #if (A ≤ 2) x = x+1; #endif
③ #if (A ≤ 5) x = x+1; #endif
④ #if (??) ⓣ x = x-1; #endif
⑤ assert (0 ≤ x ≤ 1);
}
```

Figure 1: `intro.c`.



(a) Tree at loc. ④. (b) Tree at loc. ⓣ. (c) Tree at loc. ⑤.

Figure 3: Decision-trees in `intro.c` (solid edges = true, dashed edges = false).

behind this check is that if the hole ?? is replaced with $R_{tt}(\mathtt{A})$, then for all variants satisfying $\neg R_{tt}(\mathtt{A})$ the statement at loc. ⓣ will not be executed, so they should also satisfy $R_{ff}(\mathtt{A})$ describing the maximal set of variants for which the assertion remains valid when the statement at loc. ⓣ is not executed. Since the above implication is valid, i.e. $\neg(\mathtt{A} \leq 5) \implies (\mathtt{A} \geq 3)$, we fill the hole ?? in loc. ④ with $(\mathtt{A} \leq 5)$ as the most general condition for the assertion to hold. Observe that there are other guards that ensure safety, such as $(\mathtt{A} \leq 2)$, but they are not sufficiently general.

Consider `spl.c` shown in Fig. 2, which contains two features $\mathcal{F} = \{\mathtt{A}, \mathtt{B}\}$ where A is a numerical feature with the domain $[0, 49]$ and B is a Boolean feature. Thus, the set of valid configurations is $\mathcal{K} = \{(\mathtt{A{=}0}) \wedge \mathtt{B}, (\mathtt{A{=}1}) \wedge \mathtt{B}, \ldots, (\mathtt{A{=}49}) \wedge \mathtt{B}, (\mathtt{A{=}0}) \wedge \neg\mathtt{B}, (\mathtt{A{=}1}) \wedge \neg\mathtt{B}, \ldots (\mathtt{A{=}49}) \wedge \neg\mathtt{B}, \}$. The forward lifted abstract analysis infers the invariant $(\mathtt{A} \leq 5 \wedge \mathtt{y{=}n{+}1} \wedge \mathtt{z{=}n{+}1}) \vee (\mathtt{A} > 5 \wedge \mathtt{y{=}n} \wedge \mathtt{z{=}n{+}1})$ at loc. ③. The under-approximating backward lifted abstract analysis starts by propagating the assertion fact $(\mathtt{z} > \mathtt{y})$ backwards. After the missing `#if`-guard, it infers that the lifted precondition at loc. ⓣ is $(\mathtt{B} \wedge \mathtt{z} > \mathtt{y}) \vee (\neg\mathtt{B} \wedge \mathtt{z} > \mathtt{y{+}1})$, while the lifted precondition at loc. ④ is $(\mathtt{B} \wedge \mathtt{z{+}2} > \mathtt{y}) \vee (\neg\mathtt{B} \wedge \mathtt{z{+}1} > \mathtt{y})$. [1] Then, we construct two abduction queries:

$$((\mathtt{A} \leq 5 \wedge \mathtt{y{=}n{+}1} \wedge \mathtt{z{=}n{+}1}) \vee (\mathtt{A} > 5 \wedge \mathtt{y{=}n} \wedge \mathtt{z{=}n{+}1})) \wedge R_{tt}(\mathtt{A},\mathtt{B}) \implies (\mathtt{B} \wedge \mathtt{z} > \mathtt{y}) \vee (\neg\mathtt{B} \wedge \mathtt{z} > \mathtt{y{+}1})$$

$$((\mathtt{A} \leq 5 \wedge \mathtt{y{=}n{+}1} \wedge \mathtt{z{=}n{+}1}) \vee (\mathtt{A} > 5 \wedge \mathtt{y{=}n} \wedge \mathtt{z{=}n{+}1})) \wedge R_{ff}(\mathtt{A},\mathtt{B}) \implies (\mathtt{B} \wedge \mathtt{z{+}2} > \mathtt{y}) \vee (\neg\mathtt{B} \wedge \mathtt{z{+}1} > \mathtt{y})$$

[1] The inferred preconditions guarantee that the assertion is valid for any non-deterministic choice of the integer interval $[1, 2]$. Note that if the *over-approximating* backward lifted analysis is used instead of the under-approximating one, we will infer the necessary preconditions that may lead to the assertion validity for some non-deterministic choices of $[1, 2]$.

```
void main(int n){
① int z = n+1;
② #if (A ≤ 5) y = z; #else y = n; #endif
③ #if (??) ⓣ y = y+2; #endif
④ z = z+[1, 2];
⑤ #if (B) z = z+1; #endif
⑥ assert (z > y); }
```

Figure 2: `spl.c`.

The found weakest solutions are: $R_{tt}(\mathtt{A},\mathtt{B}) \equiv (\mathtt{A} > 5 \wedge \mathtt{B})$ and $R_{ff}(\mathtt{A},\mathtt{B}) \equiv \text{true}$. Since $\neg R_{tt}(\mathtt{A},\mathtt{B}) \implies R_{ff}(\mathtt{A},\mathtt{B})$ is valid, we fill the hole ?? at loc. ③ with $(\mathtt{A} > 5 \wedge \mathtt{B})$.

3 Background

In this section, we introduce a language for writing program families as well as its concrete and abstract semantics. Furthermore, we also define the logical abduction problem.

3.1 Syntax

Let $\mathcal{F} = \{A_1, \ldots, A_n\}$ be a finite and totally ordered set of *numerical features* available in a program family. The domain of a numerical feature $A \in \mathcal{F}$, denoted $\text{dom}(A) \subseteq \mathbb{Z}$, represents the set of possible values that can be assigned to it. For example, the domain of a Boolean feature $B \in \mathcal{F}$ is $\text{dom}(B) = \{0, 1\}$. A *configuration* is a *valuation function* $k \in \mathcal{K} = \mathcal{F} \to \mathbb{Z}$, which represents a mapping that assigns a value from $\text{dom}(A)$ to each feature A. We assume that only a subset $\mathcal{K}$ of all possible configurations are *valid*. Any valid configuration $k \in \mathcal{K}$ specifies one *variant* of a program family. Alternatively, we can represent a configuration by a propositional formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_n = k(A_n))$. We define *complete feature expressions* as the set of propositional logic formulas over constraints of $\mathcal{F}$ generated by:

$$\theta' ::= \text{true} \mid e_{\mathcal{F}} \bowtie e_{\mathcal{F}} \mid \neg\theta \mid \theta \wedge \theta, \quad e_{\mathcal{F}} ::= n{\in}\mathbb{Z} \mid A{\in}\mathcal{F} \mid e_{\mathcal{F}} \oplus e_{\mathcal{F}}$$

where $\bowtie \in \{<, \leq, =, \neq\}$ and $\oplus \in \{+, -, *, /\}$.

We use a simple C-like imperative language to illustrate our work. The program variables *Var* are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. To encode multiple variants, the preprocessor conditional statement is introduced "`#if` (θ) s `#endif`" [32], such that only if the presence condition $\theta \in \mathit{FeatExp}(\mathcal{F})$ is satisfied by the configuration $k \in \mathcal{K}$ the statement s will be included in the variant corresponding to k. To encode unknown `#if`-guards, we use the hole constructs $??_i$ representing placeholders that the synthesizer must replace with suitable complete feature expressions θ', such that the resulting completed program family will satisfy all assertions. The syntax of the language is:

$$\begin{aligned}
&s\ (s \in \mathit{Stm}) ::= \mathtt{skip} \mid \mathtt{x{=}}ae \mid s; s \mid \mathtt{if}\ (be)\ s\ \mathtt{else}\ s \mid \mathtt{while}\ (be)\ \mathtt{do}\ s \mid \\
&\qquad\qquad \mathtt{\#if}\ (\theta)\ s\ \mathtt{\#endif} \mid s\ \mathtt{or}\ s \mid \mathtt{assert}(be) \\
&ae\ (ae \in \mathit{AExp}) ::= n \in \mathbb{Z} \mid [n, n'] \mid \mathtt{x} \in \mathit{Var} \mid ae \oplus ae, \\
&be\ (be \in \mathit{BExp}) ::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be, \\
&\theta\ (\theta \in \mathit{FeatExp}(\mathcal{F})) ::= \theta' \mid ??_i
\end{aligned}$$

where "or" non-deterministically chooses to execute either of its arguments, and $[n, n']$ ranges over integer intervals denoting a random choice of an integer in the given interval.

Any other preprocessor conditional constructs can be desugared and represented by `#if`. E.g., `#if` (θ_0) s_0 `#elif` (θ_1) s_1 `#endif` is translated into `#if` (θ_0) s_0 `#endif;` `#if` $(\neg\theta_0 \wedge \theta_1)$ s_1 `#endif`. We assume that statements are tagged with unique syntactic labels $l \in \mathbb{L}$. Without loss of generality, we assume that a program family P is a sequence of statements followed by a single assertion "$l_i : s; l_f : \texttt{assert}(be^f)$". We say that P is *complete* program family if it contains no hole constructs. Otherwise, we say that P is *partial* program family.

Let H be a set of uniquely labelled holes $??_i$ in partial program family P. A *control function* $\phi : H \to \mathit{FeatExp}(\mathcal{F})$ is a mapping from holes to complete feature expressions. We say that ϕ is *complete* if $dom(\phi) = H$, i.e. ϕ is defined for all holes in the program family. Otherwise, we say that ϕ is a *partial* control function. We write $P[\phi]$ to denote the program family obtained by substituting each $??_i$ in P with $\phi(??_i)$, if $\phi(??_i)$ is defined.

Definition 3.1. A complete control function ϕ is a *solution* to the synthesis problem defined by a partial program family P if $P[\phi]$ is a complete family that satisfies its assertion.

A solution ϕ is the *weakest* if no $\phi(??_i)$ can be weakened. That is, for any θ' that is logically weaker than $\phi(??_i)$, i.e. $\phi(??_i) \implies \theta'$, the map $\phi[??_i \mapsto \theta']$ is not a solution.

3.2 Concrete Semantics

The semantics of a program family is defined in two phases. First, the preprocessor `CPP` [32] takes a program family s and a configuration $k \in \mathcal{K}$ as inputs, and produces a variant (single program without `#if`-s) as the output. Second, the standard single-program semantics [19, 34] is used to evaluate the obtained variants.

The first phase is specified by the projection function $\mathtt{P}_k$ (where $k \in \mathcal{K}$), which is an identity for all basic statements, e.g. $\mathtt{P}_k(\mathtt{x}{=}ae) = \mathtt{x}{=}ae$, and recursively pre-processes all sub-statements of compound statements, e.g. $\mathtt{P}_k(s;s') = \mathtt{P}_k(s);\mathtt{P}_k(s')$. For "`#if` (θ') s `#endif`", where θ' is a complete feature expression, the statement s is included in the variant for $k \in \mathcal{K}$ if $k \models \theta'$. Note that for any complete feature expression θ', either $k \models \theta'$ holds or $k \not\models \theta'$ holds since $k \in \mathcal{K}$ is a valuation function. That is, we have: $\mathtt{P}_k(\texttt{\#if}\ (\theta')\ s\ \texttt{\#endif}) = \begin{cases} \mathtt{P}_k(s) & \text{if } k \models \theta' \\ \texttt{skip} & \text{if } k \not\models \theta' \end{cases}$. On the other hand, when the presence condition is a hole construct $??_i$ we consider both possibilities of including and excluding s from the resulting variants. That is, we have: $\mathtt{P}_k(\texttt{\#if}\ (??_i)\ s\ \texttt{\#endif}) = \mathtt{P}_k(s)\ \texttt{or}\ \texttt{skip}$.

The second phase, which is used for evaluating variants, is specified by the memory *store*, denoted $\sigma \in \Sigma = \mathit{Var} \to \mathbb{Z}$, which is a mapping from program variables to values. The concrete domain is the powerset complete lattice $\langle \mathcal{P}(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma \rangle$. We define the semantics of arithmetic expressions $[\![ae]\!] : \Sigma \to \mathcal{P}(\mathbb{Z})$ and boolean expressions $[\![be]\!] : \Sigma \to \mathcal{P}(\{\text{true}, \text{false}\})$ as the sets of possible (numerical and boolean) values for expressions ae and be in a given store $\sigma \in \Sigma$. For example, $[\![n]\!]\sigma = \{n\}$, $[\![[n, n']]\!]\sigma = \{n, \ldots, n'\}$, and $[\![\mathtt{x}]\!]\sigma = \{\sigma(\mathtt{x})\}$ for any $\sigma \in \Sigma$. We consider two concrete semantics/analyses of statements: an *invariance* (forward) semantics $\overrightarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ [8, 19] that infers a set of reachable stores (invariants) from a given set of initial stores; and a *sufficient condition* (backward) semantics $\overleftarrow{[\![s]\!]} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ [19, 34] that infers a set of stores (sufficient condition) from which only stores satisfying a given postcondition are reached. For example, we now define the concrete semantics of assignment and "`or`" statements. Given a set of states $S \in \mathcal{P}(\Sigma)$, we have that $\overrightarrow{[\![\mathtt{x} = ae]\!]}S = \{\sigma[\mathtt{x} \mapsto n] \mid \sigma \in S, n \in [\![ae]\!]\sigma\}$ is the set of final states reachable after executing $\mathtt{x} = ae$, whereas $\overleftarrow{[\![\mathtt{x} = ae]\!]}S = \{\sigma \mid \forall n \in [\![ae]\!]\sigma, \sigma[\mathtt{x} \mapsto n] \in S\}$ is the set of initial states such that any execution of $\mathtt{x} = ae$ from them leads to a state in S. Moreover, $\overrightarrow{[\![s_1\ \texttt{or}\ s_2]\!]}S = \overrightarrow{[\![s_1]\!]}S \cup \overrightarrow{[\![s_2]\!]}S$ and $\overleftarrow{[\![s_1\ \texttt{or}\ s_2]\!]}S = \overleftarrow{[\![s_1]\!]}S \cap \overleftarrow{[\![s_2]\!]}S$. The full list of definitions of $\overrightarrow{[\![s]\!]}$ and $\overleftarrow{[\![s]\!]}$ can be found in [19]. The semantics of a `while`-loop is given in a fixed-point formulation [8, 34] using the least and greatest fix-point operators `lfp` and `gfp`.

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

3.3 Abstract Lifted Analyses

The concrete semantics and analyses are uncomputable since our language is Turing complete. Therefore, we now define *computable* lifted abstract analyses that are approximations of the concrete semantics and analyses, which rely on the *decision-tree lifted domain* [26, 28].

Abstract domains for leaf nodes. We assume that numerical abstract domain $\mathbb{D}$ defined over a set of variables V is equipped with sound operators for concretization $\gamma_{\mathbb{D}}$, ordering $\sqsubseteq_{\mathbb{D}}$, join $\sqcup_{\mathbb{D}}$, meet $\sqcap_{\mathbb{D}}$, the least element (bottom) $\bot_{\mathbb{D}}$, the greatest element (top) $\top_{\mathbb{D}}$, widening $\nabla_{\mathbb{D}}$, narrowing $\triangle_{\mathbb{D}}$, and lower widening $\underline{\nabla}_{\mathbb{D}}$. There are sound over-approximating forward transfer functions for expression-based tests $\text{FILTER}_{\mathbb{D}} : \mathit{BExp} \times \mathbb{D} \to \mathbb{D}$ and forward assignments $\text{ASSIGN}_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$, as well as sound under-approximating backward transfer functions for expression-based tests $\text{B-FILTER}^u_{\mathbb{D}} : \mathit{BExp} \times \mathbb{D} \to \mathbb{D}$ and backward assignments $\text{B-ASSIGN}^u_{\mathbb{D}} : \mathit{Stm} \times \mathbb{D} \to \mathbb{D}$. In particular, $\text{FILTER}_{\mathbb{D}}(be, d)$ returns an abstract element from $\mathbb{D}$ obtained by restricting d to satisfy the test be; $\text{ASSIGN}_{\mathbb{D}}(\mathtt{x}{=}ae, d)$ returns an updated version of d obtained by abstractly evaluating $\mathtt{x}{=}ae$ in d; $\text{B-FILTER}^u_{\mathbb{D}}(be, d)$ returns an abstract element from $\mathbb{D}$ such that by restricting it to satisfy test be produces d; and $\text{B-ASSIGN}^u_{\mathbb{D}}(\mathtt{x}{=}ae, d)$ returns an abstract element from $\mathbb{D}$ such that by abstractly evaluating $\mathtt{x}{=}ae$ in it produces d. We let $\mathtt{lfp}^{\#}$ (resp., $\mathtt{gfp}^{\#}$) denote an abstract fix-point operator derived using widening and narrowing (resp., lower widening) that over-approximates (resp, under-approximates) the corresponding concrete operator `lfp` (resp., `gfp`). In practice, the domain $\mathbb{D}$ will be instantiated with some known numerical domains, such as the Polyhedra domain $\langle P, \sqsubseteq_P \rangle$ [9] whose elements are conjunctions of polyhedra linear constraints of the form $\alpha_1 x_1 + \ldots + \alpha_k x_k + \alpha_{k+1} \geq 0$, where $x_1, \ldots x_k \in V, \alpha_1, \ldots, \alpha_k, \alpha_{k+1} \in \mathbb{Z}$. We will sometimes write $\mathbb{D}_V$ to explicitly denote the set of variables V over which the domain $\mathbb{D}$ is defined.

We replace the computation in the concrete domain $\mathcal{P}(\Sigma)$ with a computation in the abstract domain $\mathbb{D}_{Var}$ that is defined over program variables *Var*, so that there is a concretization based abstraction $\langle \mathcal{P}(\Sigma), \subseteq \rangle \xleftarrow{\gamma_{\mathbb{D}},} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$. [2] We will use domain $\mathbb{D}_{Var}$ to implement *leaf nodes* of decision-trees.

Abstract domains for decision nodes. Given the abstract domain $\mathbb{D}_{\mathcal{F}}$, we will construct the finite set $\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}$ of linear constraints over features $\mathcal{F}$ by Galois connection $(\alpha_{\mathbb{C}_{\mathbb{D}}}, \gamma_{\mathbb{C}_{\mathbb{D}}})$ between $\langle \mathcal{P}(\mathbb{C}_{\mathbb{D}}), \sqsubseteq_{\mathbb{D}} \rangle$ and $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$. The concretization function $\gamma_{\mathbb{C}_{\mathbb{D}}} : \mathbb{D} \to \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ maps a conjunction of constraints from $\mathbb{D}$ to a finite set of constraints in $\mathcal{P}(\mathbb{C}_{\mathbb{D}})$. We will use $\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}$ to implement *decision nodes* of decision-trees.

We assume the set of features $\mathcal{F} = \{A_1, \ldots, A_n\}$ to be totally ordered, such that the ordering is $A_1 > \ldots > A_n$. We impose a total order $<_{\mathbb{C}_{\mathbb{D}}}$ on $\mathbb{C}_{\mathbb{D}}$ to be the lexicographic order on $\alpha_1, \ldots, \alpha_n$ and α_{n+1} of the linear constraints, such that:

$$(\alpha_1 \cdot A_1 + \ldots + \alpha_n \cdot A_n + \alpha_{n+1} \geq 0) <_{\mathbb{C}_{\mathbb{D}}} (\alpha'_1 \cdot A_1 + \ldots + \alpha'_n \cdot A_n + \alpha'_{n+1} \geq 0)$$
$$\iff \exists j > 0. \forall i < j. (\alpha_i = \alpha'_i) \wedge (\alpha_j < \alpha'_j)$$

The negation of linear constraints is formed as: $\neg(\alpha_1 A_1 + \ldots \alpha_n A_n + \alpha_{n+1} \geq 0) = -\alpha_1 A_1 - \ldots - \alpha_n A_n - \alpha_{n+1} - 1 \geq 0$. For example, the negation of $A - 3 \geq 0$ is $-A + 2 \geq 0$. To ensure canonical representation of decision-trees, we only keep the largest constraint with respect to $<_{C_{\mathbb{D}}}$ between c and $\neg c$.

Abstract domain for decision-trees. A *decision-tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{D}_{Var})$ over the sets $\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}$ of linear constraints defined over features $\mathcal{F}$ and the leaf abstract domain $\mathbb{D}_{Var}$ defined over program variables *Var* is: either a leaf node $\ll d \gg$ with $d \in \mathbb{D}_{Var}$, or $[\![c : tl, tr]\!]$, where $c \in \mathbb{C}_{\mathbb{D}_{\mathcal{F}}}$ (denoted by $t.c$) is *the smallest constraint* with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ appearing in the tree t, tl (denoted by $t.l$) is the left subtree of t representing its *true branch*, and tr (denoted by $t.r$) is the right subtree of t representing its *false branch*. The path along a decision tree establishes the set of configurations (those that satisfy the encountered constraints), and the leaf nodes represent the analysis properties for the corresponding configurations. We often omit to write $\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}$ and $\mathbb{D}_{Var}$ in the domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{D}_{Var})$ when they are clear from the context.

By lifting the operators and transfer functions for the base abstract domain $\mathbb{D}$, we can define the operators and transfer functions for the lifted decision-tree domain $\mathbb{T}$ [15, 26, 28]. Thus, there are sound operators for concretization $\gamma_{\mathbb{T}}$, ordering $\sqsubseteq_{\mathbb{T}}$, join $\sqcup_{\mathbb{T}}$, meet $\sqcap_{\mathbb{T}}$, bottom $\bot_{\mathbb{T}}$, top $\top_{\mathbb{T}}$, widening $\nabla_{\mathbb{T}}$, narrowing $\triangle_{\mathbb{T}}$, and lower widening $\underline{\nabla}_{\mathbb{T}}$. There are also sound over-approximating forward transfer functions for expression-based tests $\text{FILTER}_{\mathbb{T}} : \mathit{BExp} \times \mathbb{T} \to \mathbb{T}$, forward assignments $\text{ASSIGN}_{\mathbb{T}} : \mathit{Stm} \times \mathbb{T} \to \mathbb{T}$, and feature-based tests $\text{FEATFILTER}_{\mathbb{T}} : \mathit{FeatExp}(\mathcal{F}) \times \mathbb{T} \to \mathbb{T}$, as well as sound under-approximating backward transfer function for expression-based tests $\text{B-FILTER}^u_{\mathbb{T}} : \mathit{BExp} \times \mathbb{T} \to \mathbb{T}$, backward assignments $\text{B-ASSIGN}^u_{\mathbb{T}} : \mathit{Stm} \times \mathbb{T} \to \mathbb{T}$, and feature-based tests $\text{B-FEATFILTER}^u_{\mathbb{T}} : \mathit{FeatExp}(\mathcal{F}) \times \mathbb{T} \to \mathbb{T}$. The transfer function $\text{FEATFILTER}_{\mathbb{T}}(\theta, t)$ returns a decision-tree obtained by adding the constraint θ to the tree t, thus discarding all paths of t that do not satisfy θ, whereas $\text{B-FEATFILTER}^u(\theta, t)$ represents the corresponding backward transfer function. We refer to [15, 26, 28] for their exact definitions.

[2] Concretization-based abstraction uses only a concretization function $\gamma_{\mathbb{D}} : \mathbb{D} \to \mathcal{P}(\Sigma)$ and represents a relaxation of the known Galois connection abstraction, which uses two functions: abstraction and concretization. However, some abstract domains (e.g., Polyhedra) do not enjoy Galois connection, but only the relaxed concretization-based abstraction.

$$\overrightarrow{[\![\texttt{skip}]\!]}^\sharp t = t$$
$$\overrightarrow{[\![s_1 \texttt{ or } s_2]\!]}^\sharp t = \overrightarrow{[\![s_1]\!]}^\sharp t \sqcup_{\mathbb{T}} \overrightarrow{[\![s_2]\!]}^\sharp t$$
$$\overrightarrow{[\![\texttt{x = } ae]\!]}^\sharp t = \text{ASSIGN}_{\mathbb{T}}(\texttt{x = } ae, t)$$
$$\overrightarrow{[\![s_1 \texttt{ ; } s_2]\!]}^\sharp t = \overrightarrow{[\![s_2]\!]}^\sharp (\overrightarrow{[\![s_1]\!]}^\sharp t)$$
$$\overrightarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}^\sharp t =$$
$$\overrightarrow{[\![s_1]\!]}^\sharp (\text{FILTER}_{\mathbb{T}}(be, t)) \sqcup_{\mathbb{T}} \overrightarrow{[\![s_2]\!]}^\sharp (\text{FILTER}_{\mathbb{T}}(\neg be, t))$$
$$\overrightarrow{[\![\texttt{\#if } (\theta)\ s\ \texttt{\#endif}]\!]}^\sharp t =$$
$$\overrightarrow{[\![s_1]\!]}^\sharp (\text{FEATFILTER}_{\mathbb{T}}(\theta, t)) \sqcup_{\mathbb{T}} \overrightarrow{[\![s_2]\!]}^\sharp (\text{FEATFILTER}_{\mathbb{T}}(\neg\theta, t))$$
$$\overrightarrow{[\![\texttt{while } be \texttt{ do } s]\!]}^\sharp t = \texttt{lfp}^\sharp\ \overrightarrow{\phi^\sharp}(\neg be, t)$$
$$\overrightarrow{\phi^\sharp}(x) = t \sqcup_{\mathbb{T}} \overrightarrow{[\![s]\!]}^\sharp (\text{FILTER}_{\mathbb{T}}(be, x))$$

Figure 4: Lifted invariance analysis $\overrightarrow{[\![s]\!]}^\sharp : \mathbb{T} \to \mathbb{T}$.

Lifted analyses. The operations and transfer functions of the lifted decision-tree domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{D}_{Var})$ can be used to define lifted invariance (resp., sufficient condition) abstract analysis that is over- (resp., under-) approximation of the corresponding concrete analysis. The lifted invariance abstract semantics $\overrightarrow{[\![s]\!]}^\sharp$ is defined in Fig. 4, whereas the lifted sufficient condition abstract semantics $\overleftarrow{[\![s]\!]}^\sharp$ is defined in Fig. 5. For a `while` loop, $\texttt{lfp}^\sharp\ \overrightarrow{\phi^\sharp}$ and $\texttt{gfp}^\sharp\ \overleftarrow{\phi^\sharp}$ are the limits of the following increasing and decreasing chains: $y_0 = t,\ y_{n+1} = y_n \nabla_{\mathbb{T}} \overrightarrow{\phi^\sharp}(y_n)$ for forward analysis; and $y_0 = \text{B-FILTER}^u_{\mathbb{T}}(\neg be, t),\ y_{n+1} = y_n \underline{\nabla}_{\mathbb{T}} \overleftarrow{\phi^\sharp}(y_n)$ for backward analysis. Since we have $\text{FEATFILTER}_{\mathbb{T}}(\theta, t) \sqsubseteq_{\mathbb{T}} t$ and $\text{B-FEATFILTER}^u_{\mathbb{T}}(\theta, t) \sqsupseteq_{\mathbb{T}} t$, we will use the following definitions: $\text{FEATFILTER}_{\mathbb{T}}(??_i, t) = t$, $\text{B-FEATFILTER}^u_{\mathbb{T}}(??_i, t) = t$ to handle holes $??_i$ as missing `#if`-guards.

By using the soundness of the abstract domains $\mathbb{D}$ [8, 34] and $\mathbb{T}$ [15, 26, 28], we can establish the soundness of lifted abstract analyses with respect to concrete semantics. Given $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}_{\mathcal{F}}}, \mathbb{D}_{Var})$, we denote by $\pi_k(t) \in \mathbb{D}_{Var}$ the leaf node of t that corresponds to the variant $k \in \mathcal{K}$.

Proposition 3.2 ([15, 26, 28]). *For all $t \in \mathbb{T}$ and all $k \in \mathcal{K}$, $\overrightarrow{[\![\texttt{P}_k(s)]\!]}\gamma_{\mathbb{D}}(\pi_k(t)) \subseteq \pi_k(\overrightarrow{[\![s]\!]}^\sharp t)$; $\overleftarrow{[\![\texttt{P}_k(s)]\!]}\gamma_{\mathbb{D}}(\pi_k(t)) \supseteq \pi_k(\overleftarrow{[\![s]\!]}^\sharp t)$.*

3.4 Abduction

The standard abduction allows the inference of a single unknown predicate $R(\mathbf{x})$ defined over a vector of variables $\mathbf{x}$, known as *abducible*, from a formula $R(\mathbf{x}) \wedge \chi \implies C$. That is, the standard abduction finds a formula ϕ over variables $\mathbf{x}$, such that (1) $\phi \wedge \chi \not\models \text{false}$; and (2) $\phi \wedge \chi \models C$. A solution ϕ to the standard abduction problem is an interpretation of $R(\mathbf{x})$ that strengthens the left-hand side of the implication in order to make the implication logically

$$\overleftarrow{[\![\texttt{skip}]\!]}^\sharp t = t$$
$$\overleftarrow{[\![s_1 \texttt{ or } s_2]\!]}^\sharp t = \overleftarrow{[\![s_1]\!]}^\sharp t \sqcap_{\mathbb{T}} \overleftarrow{[\![s_2]\!]}^\sharp t$$
$$\overleftarrow{[\![\texttt{x} = ae]\!]}^\sharp t = \texttt{B-ASSIGN}^u_{\mathbb{T}}(\texttt{x} = ae, t)$$
$$\overleftarrow{[\![s_1 \texttt{ ; } s_2]\!]}^\sharp t = \overleftarrow{[\![s_1]\!]}^\sharp(\overleftarrow{[\![s_2]\!]}^\sharp t)$$
$$\overleftarrow{[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]}^\sharp t = \texttt{B-FILTER}^u_{\mathbb{T}}(be, \overleftarrow{[\![s_1]\!]}^\sharp t) \sqcap_{\mathbb{T}} \texttt{B-FILTER}^u_{\mathbb{T}}(\neg be, \overleftarrow{[\![s_2]\!]}^\sharp t)$$
$$\overleftarrow{[\![\texttt{\#if } (\theta)\ s \texttt{ \#endif}]\!]}^\sharp t = \overleftarrow{[\![s_1]\!]}^\sharp(\texttt{B-FEATFILTER}^u_{\mathbb{T}}(\theta, t)) \sqcap_{\mathbb{T}} \overleftarrow{[\![s_2]\!]}^\sharp(\texttt{B-FEATFILTER}^u_{\mathbb{T}}(\neg\theta, t))$$
$$\overleftarrow{[\![\texttt{while } be \texttt{ do } s]\!]}^\sharp t = \texttt{gfp}^\sharp\, \overleftarrow{\phi}^\sharp$$
$$\overleftarrow{\phi}^\sharp(x) = \texttt{B-FILTER}^u_{\mathbb{T}}(\neg be, t) \sqcap_{\mathbb{T}} \texttt{B-FILTER}^u_{\mathbb{T}}(be, \overleftarrow{[\![s]\!]}^\sharp x)$$

Figure 5: Lifted sufficient condition analysis $\overleftarrow{[\![s]\!]}^\sharp : \mathbb{T} \to \mathbb{T}$.

valid. Every solvable abduction problem has an unique logically weakest solution. The procedure $\texttt{Abduce}(\chi, C, \mathbf{x})$ that solves the problem $R(\mathbf{x}) \wedge \chi \implies C$ is implemented in the EXPLAIN tool [10]. It computes the logically weakest (i.e., the most general) solution containing the fewest number of variables.

Proposition 3.3 ([10]). *If the abduction problem is solvable, then* $\texttt{Abduce}(\chi, C, \boldsymbol{x})$ *terminates with an unique weakest solution containing a fewest number of variables.*

4 Synthesis Algorithm

In this section, we describe our synthesis algorithm for automatically synthesizing complete feature expressions for the missing #if-guards in a partial program family. In particular, we employ the lifted abstract analyses, $\overrightarrow{[\![s]\!]}^\sharp$ and $\overleftarrow{[\![s]\!]}^\sharp$, for constraint generation as well as the logical abduction procedure, Abduce, for constraint solving.

High-level description. The $\texttt{SPLSynthesize}(P, H)$ procedure is shown in Algorithm 1. It takes as input two parameters: a partial program family $P \equiv l_i : s; l_f : \texttt{assert}(be^f)$ and a set of holes H in P, and returns as output the weakest solution or $\emptyset$ if no solution exists. For each hole $??_i$ in H, we first generate an initial solution, a complete feature expression θ'_i, where all other holes are treated as non-deterministic choices over Booleans, $\mathcal{P}(\{\text{true}, \text{false}\})$ (lines 2–7). First, we identify the statement "l: #if $(??_i)$ $l_t : s_t$ #endif l':" in which $??_i$ occurs by using $\texttt{Extract}(P, ??_i)$ procedure (line 3). Second, we construct a partial control function ϕ_i^{ndet}, such that $\phi_i^{ndet}(??_i) = \bot$ (undefined) and $\phi_i^{ndet}(??_j) = \mathcal{P}(\{\text{true}, \text{false}\})$ for all other $??_j \in H$ (line 4), and we construct a program family $P[\phi_i^{ndet}]$ in which the hole $??_i$ is the only one. This has the effect of treating any hole other than $??_i$ in $P[\phi_i^{ndet}]$ as a non-deterministic choice. Finally, we call the one-hole synthesis procedure, called $\texttt{OneHoleSynth}(P[\phi_i^{ndet}], l, l_t, l')$ to generate the solution θ'_i corresponding to hole $??_i$ that is located after label l and before labels l_t and l' (line 5). This way, we generate an *initial* complete control function $\phi : [??_i \mapsto \theta'_i]$ by using the above initial solutions for all holes $??_i$ in H.

Algorithm 1: $\texttt{SPLSynthesize}(P, H)$

Input: Program family P, and a set of holes H
Output: Complete control function ϕ or an empty set

1 $\phi := \phi' := \emptyset$;
2 **for** $(??_i \in H)$ **do**
3 $\quad l$: #if $(??_i)$ $l_t : s_t$ #endif $l' ::= \texttt{Extract}(P, ??_i)$;
4 $\quad \phi_i^{ndet} := [??_i \mapsto \bot] \uplus \{[??_j \mapsto \mathcal{P}(\{\text{true}, \text{false}\})] \mid j \neq i\}$;
5 $\quad \theta'_i := \texttt{OneHoleSynth}(P[\phi_i^{ndet}], l, l_t, l')$;
6 $\quad$ **if** $(\theta'_i = \emptyset)$ **then return** $\emptyset$;
7 $\quad \phi := \phi \uplus [??_i \mapsto \theta'_i]$
8 **while** $(\phi \neq \phi')$ **do**
9 $\quad$ **if** $(\phi' \neq \emptyset)$ **then** $\phi = \phi'; \phi' = \emptyset$;
10 $\quad$ **for** $(??_i \in H)$ **do**
11 $\quad\quad l$: #if $(??_i)$ $l_t : s_t$ #endif $l' ::= \texttt{Extract}(P, ??_i)$;
12 $\quad\quad \phi_i := \phi[??_i \mapsto \bot]$;
13 $\quad\quad \theta''_i := \texttt{OneHoleSynth}(P[\phi_i], l, l_t, l')$;
14 $\quad\quad$ **if** $(\theta''_i = \emptyset)$ **then return** $\emptyset$;
15 $\quad\quad \phi' := \phi' \uplus [??_i \mapsto \theta''_i]$
16 **return** ϕ

Algorithm 2: $\texttt{OneHoleSynth}(P, l, l_t, l')$

Input: Program family $P \equiv l_i : s; l_f : \texttt{assert}(be^f)$, and labels l, l_t, l'
Output: Complete feature expression θ' or empty set

1 $\texttt{Inv}^\sharp := \overrightarrow{[\![s]\!]}^\sharp\, \top_{\mathbb{T}}$;
2 $\texttt{Cond}^\sharp := \overleftarrow{[\![s]\!]}^\sharp\, \texttt{FILTER}_{\mathbb{T}}(be^f, \top_{\mathbb{T}})$;
3 $R_{tt}(\mathcal{F}) := \texttt{Abduce}(\texttt{Inv}^\sharp(l), \texttt{Cond}^\sharp(l_t), \mathcal{F})$;
4 $R_{ff}(\mathcal{F}) := \texttt{Abduce}(\texttt{Inv}^\sharp(l), \texttt{Cond}^\sharp(l'), \mathcal{F})$;
5 **if** $(\texttt{unsat}(R_{tt}(\mathcal{F})) \vee \texttt{unsat}(R_{ff}(\mathcal{F})))$ **then return** $\emptyset$;
6 **return** $\texttt{SMTCheck}(R_{tt}(\mathcal{F}), R_{ff}(\mathcal{F}))$

Subsequently, Algorithm 1 goes through an iterative while-loop (lines 8-15) performing the weakening of the current solution ϕ, thus generating more general solutions ϕ' until we obtain the weakest solution. The algorithm terminates when the refinement is no longer enabled, i.e. $\phi = \phi'$ (line 8). In particular, in each while-iteration we weaken the current solution ϕ by iteratively weakening ϕ-solutions for all holes (lines 10–15). To weaken the ϕ-solution for each $??_i$, we fix the ϕ-solutions θ'_j for all other $??_j$ in a partial control function ϕ_i, such that $\phi_i(??_i) = \bot$ and $\phi_i(??_j) = \phi(??_j)$ for all other $??_j \in H$ (line 12). Then, we construct a program family $P[\phi_i]$ in which the hole $??_i$ is the only one. Finally, we call $\texttt{OneHoleSynth}(P[\phi_i], l, l_t, l')$ to find a weaker feature expression θ''_i for $??_i$ (line 13), which is located at statement l: #if $(??_i)$ $l_t : s_t$ #endif l':. The generated θ''_i will be included in the updated solution ϕ' (line 15). That is, we use the existing solution given by the current control function for all other holes and infer the weakest expression for $??_i$ that implies the assertion validity.

One-hole Synthesis. The $\texttt{OneHoleSynth}(P, l, l_t, l')$ procedure, shown in Algorithm 2, takes four parameters: a program family $P \equiv l_i : s; l_f : \texttt{assert}(be^f)$ with one hole in it and three labels: l, l_t, and l', where the hole ?? we want to handle is located (l is location before ??, and l_t, l' are locations after ??). First, we call the lifted forward abstract analysis $\overrightarrow{[\![s]\!]}^\sharp \top_{\mathbb{T}}$ to compute the lifted invariants $\texttt{Inv}^\sharp$ in all locations of s (line 1). Then, we call the lifted backward abstract analysis $\overleftarrow{[\![s]\!]}^\sharp t_f$, where $t_f = \texttt{FILTER}_{\mathbb{T}}(be^f, \top_{\mathbb{T}})$ represents the assertion fact, to compute the lifted sufficient conditions $\texttt{Cond}^\sharp$ in all locations of s (line 2). Assume that $\texttt{Inv}^\sharp(l)$ is the inferred lifted invariant at loc. l before the hole, while $\texttt{Cond}^\sharp(l_t)$ and $\texttt{Cond}^\sharp(l')$ are the inferred lifted sufficient conditions at locs. l_t and l', respectively, after the hole. They ensure assertion-safety of the code that come after the hole ??, such that $\texttt{Cond}^\sharp(l_t)$ includes the effects of statement s_t, whereas $\texttt{Cond}^\sharp(l')$ does not include the effects of statement s_t. We construct two abduction queries in which the premise is $\texttt{Inv}^\sharp(l)$ and the unknown predicate is defined over all features $\mathcal{F}$. The conclusions are $\texttt{Cond}^\sharp(l_t)$ and $\texttt{Cond}^\sharp(l')$ in the first and second abduction query, respectively (lines 3, 4). They can be solved using the standard abduction procedure Abduce [10], which returns the unique weakest solutions. If one of the abduction queries is not solvable, then an empty set is returned (line 5). Otherwise, we call $\texttt{SMTCheck}(R_{tt}(\mathcal{F}), R_{ff}(\mathcal{F}))$ to check if $\neg R_{tt}(\mathcal{F}) \implies R_{ff}(\mathcal{F})$ by using an SMT solver (line 6), where $R_{tt}(\mathcal{F})$ and $R_{ff}(\mathcal{F})$ are solutions of the first and second abduction query. This check is done to ensure that if the hole ?? is replaced with $R_{tt}(\mathcal{F})$ then the assertion will remain valid for all variants that will not execute the protected code s_t by ??, that is, if $k \models \neg R_{tt}(\mathcal{F})$ then $k \models R_{ff}(\mathcal{F})$ for all $k \in \mathcal{K}$. If the above implication is valid, then $R_{tt}(\mathcal{F})$ is reported as solution. Otherwise, $R_{tt}(\mathcal{F})$ is strengthen until $\neg R_{tt}(\mathcal{F}) \implies R_{ff}(\mathcal{F})$ holds, in which case the strengthen $R_{tt}(\mathcal{F})$ is solution.

Correctness. We now state the correctness result.

Theorem 4.1. $\texttt{SPLSynthesize}(P, H)$ *is correct and terminates. Furthermore, it returns the weakest solution.*

Proof Sketch. The procedure $\texttt{SPLSynthesize}(P, H)$ terminates since all steps in it are terminating. The correctness of the procedure $\texttt{SPLSynthesize}(P, H)$ follows from the soundness of the lifted abstract analyses $\overrightarrow{[\![s]\!]}^\sharp$ and $\overleftarrow{[\![s]\!]}^\sharp$ (see Proposition 3.2) and the correctness of Abduce (see Proposition 3.3). The correctness proof is by structural induction on statements s in program families $P \equiv l_i : s; l_f : \texttt{assert}(be^f)$. □

Example 4.2. Let us consider the following program family `intro-b.c`, which contains two holes $??_1$ and $??_2$:

```
① int x=0;                          ④ #if (??1) ℓ1 x=x+1;#endif
② #if (A ≤ 2) x=x+1;#endif          ⑤ #if (??2) ℓ2 x=x-2;#endif
③ #if (A ≤ 5) x=x+1;#endif          ⑥ assert (0 ≤ x ≤ 2);
```

We first find initial solution for $??_1$ (resp., $??_2$) under assumption that $??_2$ (resp., $??_1$) is a non-deterministic choice. For $??_1$, we construct two abduction queries:

$$\big((\texttt{A} \le 2 \wedge \texttt{x=2}) \vee (3 \le \texttt{A} \le 5 \wedge \texttt{x=1}) \vee (\texttt{A} > 5 \wedge \texttt{x=0})\big) \wedge R^1_{tt}(\texttt{A}) \implies (-1 \le \texttt{x} \le 1 \wedge 1 \le \texttt{x} \le 3)$$

$$\big((\texttt{A} \le 2 \wedge \texttt{x=2}) \vee (3 \le \texttt{A} \le 5 \wedge \texttt{x=1}) \vee (\texttt{A} > 5 \wedge \texttt{x=0})\big) \wedge R^1_{ff}(\texttt{A}) \implies (0 \le \texttt{x} \le 2 \wedge 2 \le \texttt{x} \le 4)$$

Note that the right-hand sides (RHSs) of the above implications, which represent sufficient conditions at locs. ℓ_1 and ⑤, are the under-approximations of the effects of executing and not executing statement at loc. ℓ_2 (`x = x-2 or skip`). We obtain $R^1_{tt}(\texttt{A}) \equiv (3 \le \texttt{A} \le 5)$ and $R^1_{ff}(\texttt{A}) \equiv (\texttt{A} \le 2)$. Since $\neg R^1_{tt}(\texttt{A}) \implies R^1_{ff}(\texttt{A})$ is not valid, the reported solution for $??_1$ is (false). For $??_2$, we construct abduction queries:

$$\big((\texttt{A} \le 2 \wedge 2 \le \texttt{x} \le 3) \vee (3 \le \texttt{A} \le 5 \wedge 1 \le \texttt{x} \le 2) \vee (\texttt{A} > 5 \wedge 0 \le \texttt{x} \le 1)\big) \wedge R^2_{tt}(\texttt{A}) \implies (2 \le \texttt{x} \le 4)$$

$$\big((\texttt{A} \le 2 \wedge 2 \le \texttt{x} \le 3) \vee (3 \le \texttt{A} \le 5 \wedge 1 \le \texttt{x} \le 2) \vee (\texttt{A} > 5 \wedge 0 \le \texttt{x} \le 1)\big) \wedge R^2_{ff}(\texttt{A}) \implies (0 \le \texttt{x} \le 2)$$

Note that the left-hand side of the above implications, which represents the invariant at loc. ⑤, is the over-approximation of the effects of executing and not executing statement at loc. ℓ_1 (`x = x+1 or skip`). We obtain $R^2_{tt}(\texttt{A}) \equiv (\texttt{A} \le 2)$ and $R^1_{ff}(\texttt{A}) \equiv (\texttt{A} \ge 3)$. Since $\neg R^2_{tt}(\texttt{A}) \implies R^2_{ff}(\texttt{A})$ is valid, the reported solution for $??_2$ is $(\texttt{A} \le 2)$.

In the second iteration, we weaken the initial solutions for $??_1$ and $??_2$. For $??_1$, we assume that $??_2$ is $(\texttt{A} \le 2)$, thus we obtain the abduction queries by updating the RHSs of the above queries for $??_1$ with: $(\texttt{A} > 2 \wedge -1 \le \texttt{x} \le 1) \vee (\texttt{A} \le 2 \wedge 1 \le \texttt{x} \le 3)$ for the first query and $(\texttt{A} > 2 \wedge 0 \le \texttt{x} \le 2) \vee (\texttt{A} \le 2 \wedge 2 \le \texttt{x} \le 4)$ for the second query. Thus, we generate the solution (true) for $??_1$. In the next iteration, we assume that $??_1$ is (true), and we generate the solution $(\texttt{A} \le 5)$ for $??_2$. Then, we report solutions (true) for $??_1$ and $(\texttt{A} \le 5)$ for $??_2$.

5 Evaluation

We now evaluate our synthesis approach based on decision-tree lifted analysis. We compare its performances against approaches that use tuple-based lifted analysis and variability encoding combined with single-program analysis.

Tuple-based lifted analysis. The lifted tuple domain for representing lifted invariants and sufficient conditions is $\langle \prod_{k \in \mathcal{K}} \mathbb{D}_{Var}, \dot{\sqsubseteq} \rangle$. That is, there is one separate copy of $\mathbb{D}$ for each configuration $k \in \mathcal{K}$. Hence, the elements of the lifted domain are tuples $\bar{t}$ that maintain one property element from $\mathbb{D}$ per configuration. All lifted operations and transfer functions are defined by lifting the corresponding operations and transfer functions of the domain $\mathbb{D}$ configuration-wise. We refer to [28] for precise definitions of all elements of the lifted tuple domain. The tuple-based lifted analysis represents an improvement over the "brute force" approach, which generates and analyzes all individual variants one by one, as evidenced in the literature [5, 28].

Variability encoding. The variability encoding approach [30, 39] replaces compile-time variability with run-time variability (non-determinism). In particular, a program family is transformed into a

single program by encoding features with ordinary program variables that are non-deterministically initialized to any value from their domain and by replacing `#if`-directives with ordinary conditional `if` statements. The resulting single programs, called *variability simulators*, can be analyzed using single-program analyzers. Unlike the lifted analyses, this approach does not consider disjunctive properties arising from features, which leads to imprecisions and rough approximations in the inferred results.

Implementation. We have implemented our synthesis algorithm in a prototype tool, called `SPLSynthesize`. The abstract operations and transfer functions of the numerical abstract domains (e.g. Polyhedra [9]) are provided by the APRON library [31], while the decision-tree and tuple-based lifted domains are provided by the SPLNum^{2}Analyzer tool [28]. The abduction and SMT queries are solved by the EXPLAIN [10] and the MISTRAL [11] tools. Our proof-of-concept tool is written in OCaml and consists of around 7K lines of code. The tool accepts programs written in a subset of C with `#if`-directives. It currently provides only a limited support for arrays, pointers, `struct` and `union` types.

Experiment setup and Benchmarks. All experiments are executed on a 64-bit Intel Core^{TM} i5 CPU, VM Lubuntu 18.04, with 4 GB memory, and we use a timeout value of 60 sec. All times are reported as average over five independent executions. The tool artifact is available from: https://zenodo.org/records/15839141. We compare three approaches for `#if`-guards synthesis: (1) an approach based on decision-tree lifted analyses; (2) an approach based on tuple lifted analyses; and (3) variability encoding approach. We report Time which is the total time to resolve a given problem, AbdTime which is the time to solve the abduction queries in the given synthesis task, and Prec which shows the precision of the reported solution. We use: ✓ when the weakest solution is found; and ≈ when a stronger than the weakest solution is found.

The evaluation is performed on several C numerical SPLs collected from SV-COMP and the literature [28, 33]. In particular, we use the following benchmarks: `intro.c` (Fig. 1), `spl.c` (Fig. 2), `minmax.c` (Fig. 6), `while.c` (Fig. 7), `mine.c` (Fig. 8), `Mysore.c` (Fig. 9). In all benchmarks, we assume that `A` is a numerical feature with domain [0,49] and `B` is a Boolean feature. In the case of some benchmarks from SV-COMP (e.g., `minmax.c`, `while.c`, `Mysore.c`), we have manually added variability (features and `#if`-s) via presence conditions with different complexities, from atomic to more complex.

Performance Results (`Decison-tree` approach). Table 1, in column `Decison-tree`, shows the performance results of our approach, `SPLSynthesize`, on the given benchmarks.

Let us construct a new version of the example `spl.c` in Fig. 2, denoted `spl-b.c`, by adding one additional `#if` with missing guard at loc. ④: `#if (??`$_2$`)` ⑫ `y = y-1; #endif`. In the first iteration, we obtain solutions $R^1_{tt}(\mathtt{A},\mathtt{B}) \equiv (\mathtt{B} \wedge \mathtt{A} > 5)$ for $??_1$ and $R^2_{tt}(\mathtt{A},\mathtt{B}) \equiv$ (false) for $??_2$. For example, the abduction query for $R^1_{tt}(\mathtt{A},\mathtt{B})$ is: $((\mathtt{A} \leq 5 \wedge \mathtt{y}=\mathtt{n}+1 \wedge \mathtt{z}=\mathtt{n}+1) \vee (\mathtt{A} > 5 \wedge \mathtt{y}=\mathtt{n} \wedge \mathtt{z}=\mathtt{n}+1)) \wedge R_{tt}(\mathtt{A},\mathtt{B}) \implies (\mathtt{B} \wedge \mathtt{z} > \mathtt{y} \wedge \mathtt{z} > \mathtt{y}-1) \vee (\neg\mathtt{B} \wedge \mathtt{z} > \mathtt{y}+1 \wedge \mathtt{z} > \mathtt{y})$. In the second iteration, by using $(\mathtt{B} \wedge \mathtt{A} > 5)$ for $??_1$, we obtain weaker solution $R^2_{tt}(\mathtt{A},\mathtt{B}) \equiv$ (true) for $??_2$. Thus, `SPLSynthesize` reports the final solutions $(\mathtt{B} \wedge \mathtt{A} > 5)$ for $??_1$ and (true) for $??_2$.

The program family `minmax.c` in Fig. 6 has a specification stating that the returned value `res` is the minimum of two parameters `n` and `m` when the feature `A` is greater than 5 and the maximum of `n` and `m` otherwise. The lifted abstract analyses find the constraints: $(\mathtt{res}=\mathtt{n} \wedge \mathtt{m} \geq \mathtt{n})$ before the hole, as well as $(\mathtt{A} \leq 5 \wedge \mathtt{m} \geq \mathtt{n}) \vee (\mathtt{A} > 5 \wedge \mathtt{n} \geq \mathtt{m})$ and $(\mathtt{A} \leq 5 \wedge \mathtt{res} \geq \mathtt{n} \wedge \mathtt{res} \geq \mathtt{m}) \vee (\mathtt{A} > 5 \wedge \mathtt{n} \geq \mathtt{res} \wedge \mathtt{m} \geq \mathtt{res})$ after the hole at locs. ⓣ and ⓕ, respectively. The solutions of two abduction queries are: $R_{tt}(\mathtt{A}) \equiv (\mathtt{A} \leq 5)$ and $R_{ff}(\mathtt{A}) \equiv (\mathtt{A} > 5)$. Finally, we obtain the solution $(\mathtt{A} \leq 5)$ for `??` in 0.75 sec.

Consider the program family `while.c` in Fig. 7. The inferred lifted invariant before the hole `??` is: $(\mathtt{x}=0 \wedge \mathtt{y}=0)$, while the lifted sufficient conditions after the hole are: $(\mathtt{A} \leq 1 \wedge \mathtt{x} \geq -10 \wedge \mathtt{y} \geq 0 \wedge \mathtt{x}+\mathtt{y} \leq 0) \vee (2 \leq \mathtt{A} \wedge \mathtt{x} \leq 10 \wedge \mathtt{y} \geq 0 \wedge 2\mathtt{x}+\mathtt{y} \leq -10)$ and $(\mathtt{A} \leq 1 \wedge \mathtt{x} \geq 0 \wedge \mathtt{y} \geq 0 \wedge \mathtt{x}+\mathtt{y} \leq 10) \vee (2 \leq \mathtt{A} \wedge \mathtt{x} \geq 0 \wedge \mathtt{y} \geq 0 \wedge 2\mathtt{x}+\mathtt{y} \leq 10)$. The solutions of abduction queries are: $R_{tt}(\mathtt{A}) \equiv (\mathtt{A} < 2)$ and $R_{ff}(\mathtt{A}) \equiv$ (true). Thus, we obtain the final solution $(\mathtt{A} < 2)$ for `??` in 0.78 sec.

Consider the program family `mine.c` in Fig. 8, which is taken from [19, 33]. The lifted invariant before the hole is: $(\mathtt{i}=0 \wedge 0 \leq \mathtt{j} \leq 5)$, whereas the lifted sufficient condition after the hole at loc. ⓣ is: $(\mathtt{A} \leq 5 \wedge \mathtt{i} \geq 0 \wedge -1 \leq \mathtt{j} \leq 104 \wedge \mathtt{i} \geq \mathtt{j}-4) \vee (6 \leq \mathtt{A} \leq 10 \wedge \mathtt{i} \geq 0 \wedge -1 \leq \mathtt{j} \leq 204 \wedge \mathtt{i} \geq \mathtt{j}-5) \vee (\mathtt{A} > 10 \wedge \mathtt{i} \geq 0 \wedge -1 \leq \mathtt{j} \leq 204 \wedge 2\mathtt{i} \geq \mathtt{j}-4)$. Note that the lifted backward analysis infers conditions at all locations so that all executions branching from those locations will satisfy the given assertion. Thus, even if the non-deterministic choice [0,1] always evaluates to 1 in the `while`-body, the final assertion will hold if the above sufficient conditions at loc. ⓣ are met. By solving the corresponding abduction query, we obtain the answer: $(5 < \mathtt{A} \leq 10)$ for `??` in 0.77 sec.

By performing lifted abstract analyses of the program family `Mysore.c` in Fig. 9, we obtain the lifted invariant before the hole: $(\mathtt{x}+2\mathtt{c} \geq 2 \wedge \mathtt{x}+\mathtt{c} \leq -1)$, whereas the lifted sufficient conditions after the hole are: $(\mathtt{A} < 3 \wedge \mathtt{x}+\mathtt{c} \geq 2) \vee (\mathtt{A} \geq 3 \wedge \mathtt{x}+\mathtt{c} \leq -1)$ at loc. ⓣ, and $(\mathtt{A} < 3 \wedge \mathtt{x}+2\mathtt{c} \geq 2) \vee (\mathtt{A} \geq 3 \wedge \mathtt{x}+2\mathtt{c} \leq -1)$ at loc. ⓕ. After solving the abduction queries, we obtain the solution $(\mathtt{A} > 2)$ for `??` in 0.78 sec.

Performance Results (other approaches). Table 1, in columns `Tuple` and `Variability encoding`, shows the performance results when tuple-based lifted analysis and variability encoding combined with single program analysis are used for constraint generation. The obtained results confirm that sharing enabled by decision-trees is indeed effective and especially so for larger values of $|\mathcal{K}|$. On our benchmarks, this translates on speed-ups `Decision-tree` vs. `Tuple` of 2 times in average when $|\mathcal{K}| = 50$ and of 2.8 times when $|\mathcal{K}| = 100$. Both `Decision-tree`- and `Tuple`-based approaches achieve the same precision by reporting the same weakest solutions for all benchmarks. On the other hand, the `Variability encoding` approach reports less precise results, since features are considered as ordinary variables here. That is, it infers analysis properties that represent linear constraints over both program and feature variables, thus inducing flat conjunctive constraints in the abduction queries. This way, we obtain the precise answer only for `intro.c`, while in the case of `intro-b.c` and `spl-b.c` we obtain $(\mathtt{A} \leq 2)$ and true for the second holes. In other cases, we obtain the strongest solution: false.

Discussion. We observe that the constraint solving phase in all three approaches (see AbdTime column) takes more than 90% of the total synthesis time for all benchmarks. By building more powerful

```
void main(int n, int m){
 int res = n;
 if (m > n){
     ⓣ #if (??) res = m; #endif ⓕ }
 #if (A > 5) assert (res ≤ m ∧ res ≤ n);
   #else assert (res ≥ m ∧ res ≥ n); #endif
 return res;
 }
```

Figure 6: `minmax.c`.

```
void main() {
 int x = 0; int y = 0;
 #if (??) x = x+10; #endif
 while (x > 0) {
   x = x-1;
   #if (A ≥ 2) y = y+2;
     #else y = y+1; #endif }
 assert (y ≤ 10); }
```

Figure 7: `while.c`.

```
void main() {
 int j = [0,5];
 int i = 0;
 #if (??) ⓣ j = j+1; #endif
 while (i<100) {
   i = i+1;
   #if (A ≥ 10) j = j+[0,1]; #endif
   j = j+[0,1]; }
 #if (A≤ 5) assert (j≤105);
   #else assert (j≤205); #endif }
```

Figure 8: `mine.c`.

```
void main () {
 int x = 10; int c = 0;
 #if (B) x = x+10;
 while (x+c ≥ 0) {
   x = x-c;
   c = c+1; }
 #if (??) ⓣ x = x+c;
   #else ⓕ x = x+2c; #endif
 #if (A<3) assert (x≥2);
   #else assert (x<0); #endif }
```

Figure 9: `Mysore.c`.

Table 1: Performance results of `Decison-tree` vs. `Tuple` vs. `Variability encoding`-based approaches. All times in sec.

Bench.	$\lvert\mathcal{K}\rvert$	Decision-tree			Tuple			Variability encoding		
		Time	AbdTime	Prec	Time	AbdTime	Prec	Time	AbdTime	Prec
intro.c	50	0.752	0.740	✓	1.112	1.005	✓	6.766	6.751	✓
intro-b.c	50	3.076	2.996	✓	4.441	4.231	✓	3.018	3.001	≈
spl.c	100	0.772	0.755	✓	2.103	2.011	✓	0.767	0.758	≈
spl-b.c	100	3.146	3.007	✓	9.201	8.120	✓	2.983	2.950	≈
minmax.c	50	0.751	0.738	✓	1.468	1.337	✓	0.730	0.723	≈
while.c	50	0.780	0.766	✓	1.486	1.243	✓	0.741	0.726	≈
mine.c	50	0.771	0.742	✓	2.033	1.821	✓	0.766	0.749	≈
Mysore.c	100	0.781	0.751	✓	3.158	2.234	✓	0.762	0.735	≈

abduction solving tools, we believe that our approach will benefit and will be able to handle larger and more complex benchmarks.

6 Related Work

Lifted analysis. In the past decade, many static analysis techniques [3, 4, 15, 16, 25, 28, 38] have been successfully lifted to work at the level of program families, rather than on single programs. The lifted static analysis phrased in the abstract interpretation framework [8] employs a lifted BDD domain [15] to analyze program families with only Boolean features. Next, the lifted decision-tree domain [16, 28] has been introduced to analyze numerical and termination properties of program families with both Boolean and numerical features. Moreover, this domain has been extended to analyze dynamic program families [25].

Lifted abstract analysis based on the decision-tree domain has also been applied for variability fault localization of SPLs [23] by using lifted error invariants [18, 20] to compute compact slices of buggy SPLs relevant for the given bug. Moreover, the variability fault localization has been combined with a mutation-based SPL repair algorithm [21] in order to significantly speed-up the SPL repair.

Abduction and synthesis. An algorithm for solving abduction queries in first-order SMT theories is described in [10], which generates maximally general solutions. The SMT-based abduction has been applied for loop invariant generation [13], for finding the most permissive models of undefined functions to ensure safety of the given programs [2, 24], for error explanation and diagnosis of error reports generated by verification tools [12], etc. The abduction has been used to infer missing `if`-guards from low-level C code such that all buffer accesses are memory safe [14]. However, this approach uses Hoare logic-style inference mechanisms as well as loop invariants provided by external tools in the constraint generation phase. Abstract static analysis and logical abduction are combined in [19] to solve the generalized program sketching problem by inferring arbitrary arithmetic and boolean expressions for the missing holes in the given (single-program) sketch [17, 27]. The combination of abstract static analysis and SMT-based mutations

has been used for automatic synthesis of imperative programs from logical specifications [22].

7 Conclusion

In this work, we present an algorithm for automatically synthesizing arbitrary feature expressions for unknown `#if`-guards in program families. By experiments we show the effectiveness of our approach on a variety of C benchmarks.

References

[1] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective product-line testing using similarity-based product prioritization. *Softw. Syst. Model.* 18, 1 (2019), 499–521. https://doi.org/10.1007/S10270-016-0569-2

[2] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016*. ACM, USA, 789–801. https://doi.org/10.1145/2837614.2837628

[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*. IEEE Computer Society, USA, 482–491.

[4] 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*. ACM, USA, 355–364.

[5] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. 2013. Intraprocedural Dataflow Analysis for Software Product Lines. *T. Aspect-Oriented Software Development* 10 (2013), 73–108.

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

[7] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. 2013. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In *35th International Conference on Software Engineering, ICSE '13*. IEEE Computer Society, USA, 472–481. https://doi.org/10.1109/ICSE.2013.6606593

[8] 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, USA, 238–252. https://doi.org/10.1145/512950.512973

[9] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on POPL'78*. ACM Press, USA, 84–96. https://doi.org/10.1145/512760.512770

[10] Isil Dillig and Thomas Dillig. 2013. Explain: A Tool for Performing Abductive Inference. In *Computer Aided Verification - 25th International Conference, CAV 2013. Proceedings (LNCS, Vol. 8044)*. Springer, Russia, 684–689. https://doi.org/10.1007/978-3-642-39799-8_46

[11] Isil Dillig, Thomas Dillig, and Alex Aiken. 2009. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In *Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings (LNCS, Vol. 5643)*. Springer, France, 233–247. https://doi.org/10.1007/978-3-642-02658-4_20

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*. ACM, 181–192. https://doi.org/10.1145/2254064.2254087

[13] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*. ACM, USA, 443–456. https://doi.org/10.1145/2509136.2509511

[14] Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. 2014. Optimal Guard Synthesis for Memory Safety. In *Computer Aided Verification - 26th International Conference, CAV 2014. Proceedings (LNCS, Vol. 8559)*. Springer, Austria, 491–507. https://doi.org/10.1007/978-3-319-08867-9_32

[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, USA, 96–109. https://doi.org/10.1145/3486609.3487202

[17] Aleksandar S. Dimovski. 2022. Quantitative Program Sketching using Lifted Static Analysis. In *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Proceedings (LNCS, Vol. 13241)*. Springer, Germany, 102–122. https://doi.org/10.1007/978-3-030-99429-7_6

[18] 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, Portugal, 190–211. https://doi.org/10.1007/978-3-031-44245-2_10

[19] 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, Portugal, 212–230. https://doi.org/10.1007/978-3-031-44245-2_11

[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 für Informatik, Austria, 12:1–12:24. https://doi.org/10.4230/LIPICS.ECOOP.2024.12

[22] Aleksandar S. Dimovski. 2025. Imperative Program Synthesis by Abstract Static Analysis and SMT Mutations. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2025*. ACM, Norway, 27–40. https://doi.org/10.1145/3742876.3742884

[23] Aleksandar S. Dimovski. 2025. Variability Fault Localization by Abstract Interpretation and its Application to SPL Repair. In *Proceedings of the 18th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2025*. ACM, Germany, 175–188. https://doi.org/10.1145/3732771.3742722

[24] Aleksandar S. Dimovski. 2025. Weakest Safe Context Synthesis by Symbolic Game Semantics and Logical Abduction. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025*. ACM, Italy, 1990–1997. https://doi.org/10.1145/3672608.3707849

[25] 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, Denmark, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2021.14

[26] 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, Luxembourg, 67–86. https://arxiv.org/abs/2012.05863

[27] 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, USA, 95–112. https://doi.org/10.1007/978-3-030-76384-8_7

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

[29] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2016. Finding Suitable Variability Abstractions for Family-Based Analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings (LNCS, Vol. 9995)*. Springer, Cyprus, 217–234. https://doi.org/10.1007/978-3-319-48989-6_14

[30] 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*. IEEE Computer Society, USA, 597–607. https://doi.org/10.1109/ASE.2015.84

[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, France, 661–667. https://doi.org/10.1007/978-3-642-02658-4_52

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

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

[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] 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

[36] Ramy Shahin and Marsha Chechik. 2020. Automatic and efficient variability-aware lifting of functional programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 157:1–157:27. https://doi.org/10.1145/3428225

[37] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. 2022. Efficient static analysis and verification of featured transition systems. *Empir. Softw. Eng.* 27, 1 (2022), 10. https://doi.org/10.1007/S10664-020-09930-8

[38] 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

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