Code-Level Functional Equivalence Checking of Annotative Software Product Lines

Shuolin Wang
shuolin.wang@mail.utoronto.ca
University of Toronto
Toronto, Ontario, Canada

Nick Feng
fengnick@cs.toronto.edu
University of Toronto
Toronto, Ontario, Canada

Marsha Chechik
chechik@cs.toronto.edu
University of Toronto
Toronto, Ontario, Canada

Abstract

Software functional equivalence checking is a technique for analyzing the impact of change of a portion of code on the rest of the system. The existing functional equivalence checking approaches are applicable only at the individual software product level. In this paper, we propose a *lifted* functional equivalence checking approach, CLEVER-V, that can efficiently handle annotative software product lines. Instead of checking functional equivalence of every product separately, CLEVER-V analyzes all products together to iteratively identify groups of non-equivalent products with common causes. We report on the implementation of the lifted functional equivalence checking approach and demonstrate its effectiveness and scalability on a suite of 288 realistic software updates from BusyBox.

ACM Reference Format:
Shuolin Wang, Nick Feng, and Marsha Chechik. 2023. Code-Level Functional Equivalence Checking of Annotative Software Product Lines. In *27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579027.3608978

1 Introduction

A *software product line (SPL)* [7, 8, 11] is a family of similar software systems (products) with shared assets. Maintaining multiple products via a product line facilitates reuse and thus is widely practised in many domains. Yet SPLs, like all software, are subject to frequent updates, e.g., new features, updates to underlying libraries, bug fixes. How do these updates affect behaviour of the products comprising the SPL? In principle, SPL updates may affect every product, making change impact analysis a time-consuming and potentially complex task. Several existing techniques, CC2 [14], CLEVER [25], Rêve [13] and SymDiff [23] can be used for validating functional equivalence between two versions of a program. Yet, these techniques are applicable only at the individual software product level, that is, they can be used to determine whether two products are functionally equivalent. *Variability-aware functional equivalence*, applied for SPLs, aims to determine the functional equivalence for every software product in the SPLs. A naive, or *brute-force* strategy [33], is to generate and analyze all products separately and then merge their results. However, such a strategy is unlikely to scale for commonly used SPLs such as Linux kernel, which contain over 15,000 features [24].

In this paper, we address the problem of change impact assessment for SPLs by proposing a *lifted* functional equivalence checking approach, CLEVER-V. To overcome the scalability challenge, CLEVER-V analyzes all products *together* to iteratively identify groups of non-equivalent products with a common cause. CLEVER-V exploits a key observation: non-equivalent products often share the same or similar counterexample to equivalence. This observation enables us to reuse the counterexample of one non-equivalent product to efficiently identify other non-equivalent products with the same counterexample as well as find similar counterexamples for other non-equivalent products.

Illustrative Example. Consider the pair of functions `F_1` and `F_2` in Fig. 1 with small differences on Lines 7–8. Both functions have three features, A, B, C and three inputs, x, y, z, and produce one output, r. Line 7 changes `defined` to `!defined`, and Line 8 changes `x` to `y`. To trigger the difference, Line 8 has to be enabled and Line 4 disabled so that $x \neq y$ before Line 6. The following is a sample analysis result of a variability-aware equivalence: (1) for feature configurations satisfying $(A \wedge B) \vee (\neg A \wedge C)$ (e.g., $A \wedge B \wedge C$, $A \wedge B \wedge \neg C$, $\neg A \wedge B \wedge C$, $\neg A \wedge \neg B \wedge C$), a counterexample to the equivalence of `F_1` and `F_2` is $x = 0 \wedge y = -1 \wedge z = 1$; (2) other feature configurations derive equivalent products.

CLEVER-V first finds that `F_1` and `F_2` in Fig. 1a and 1b produce different outputs for product $(A \wedge B \wedge C)$ under the input $(x = 1, y = -1, z = 1)$. Then, CLEVER-V tries to identify other non-equivalent products due to the same input. As the result, CLEVER-V discovered that `F_1` and `F_2` produce different outputs under any feature configuration satisfying $B \wedge A$ (i.e., $A \wedge B \wedge C$), when the inputs satisfy $z > 0 \wedge x > y \wedge z \leq 1 \wedge y < 0$. Then, CLEVER-V tries to find other feature configurations that derive non-equivalent products under inputs satisfying $z > 0 \wedge x > y \wedge z \leq 1 \wedge y < 0$ and finds that the only other set of configurations are $C \wedge \neg A$. Thus, it produces the following formula, which we call a *featured counterexample* (FCEX), that captures the feature configurations of non-equivalent products and their counterexamples: $((B \wedge A) \vee (C \wedge \neg A)) \wedge (z > 0 \wedge x > y \wedge z \leq 1)$. After repeating the previous step, CLEVER-V determines that there are no other FCEXs. Thus, all non-equivalence products and their counterexamples have been found. Using this strategy, our variability-aware equivalence checker can solve many similar instances efficiently.

Contributions. This paper makes the following contributions. (1) We formally define the problem of checking for functional equivalence between two software product lines (SPL); (2) We propose a lifted functional equivalence checking approach, CLEVER-V, that takes two annotative software product lines and produces a sound

SPLC '23, August 28-September 1, 2023, Tokyo, Japan

ACM ISBN 979-8-4007-0091-0/23/08…$15.00
https://doi.org/10.1145/3579027.3608978

```
1  int F_1(int x, int y, int z) {
2      int r = 0;
3  #if defined A && !defined B
4      x = y;
5  #endif
6      while (z > 0) {
7  #if defined A || defined C
8          r += x;
9  #else
10         r += y;
11 #endif
12         --z;
13     }
14     return r;
15 }
```

(a) F_1.

```
1  int F_2(int x, int y, int z) {
2      int r = 0;
3  #if defined A && !defined B
4      x = y;
5  #endif
6      while (z > 0) {
7  #if defined A || !defined C
8          r += y;
9  #else
10         r += y;
11 #endif
12         --z;
13     }
14     return r;
15 }
```

(b) F_2.

Figure 1: Two C functions with feature variabilities introduced by preprocessor macros. F_2 modifies F_1 via a preprocessor macro on Line 7 and right-hand side value on Line 8.

and complete *equivalence summary*. (3) We report on a prototype implementation of CLEVER-V and empirically evaluate it on a suite of 288 benchmarks inspired by real SPL updates from BusyBox.

Organization. The rest of the paper is structured as follows. Sec. 2 gives the necessary background. Sec. 3 formally defines the problem of checking functional equivalence of SPLs. Sec. 4 presents our approach at a high level, including the definition of what our analysis reports and the correctness proof of our algorithms. Sec. 5 discusses how these ideas have been implemented. Sec. 6 evaluates the performance of CLEVER-V compared to the product-based baseline (brute-force) approach that generates and checks all pairs of products. Sec. 7 examines related works. Finally, in Sec. 8, we conclude and outline some future research directions.

2 Preliminaries

In this section, we present the necessary background on program, program equivalence, and software product lines.

2.1 Program

For simplicity, we consider a simple many-sorted imperative programming language where all operations are assignments or assumptions. We further assume that every program is a function whose type and arity are statically known. Under the above assumptions, a program can be represented as a *Control-Flow Automaton* (CFA) [14]. Note that the assumptions on the considered programming language are made only for simplifying the presentation, but the language itself is Turing-complete and thus expressive enough to capture general programs.

Definition 2.1 (Control-Flow Automaton)**.** A CFA is a tuple $(L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E)$, where (1) L is a finite and non-empty set of *program locations*; (2) $l_0, l_f \in L$ are the *initial* and *final* locations; (3) X is a set of program *variables*; (4) $\vec{x}_{in}, \vec{x}_{out} \subseteq X$ are the *input* and *output* variables; (5) $E \subseteq L \times O \times L$ is a set of control flow edges, where O is one of the operations described below. (i) An *assignment* $x := a$, where $x \in X$ and a is a term of the same sort, or type, as x; (ii) an *assumption* $\mathbf{assume}(b)$, where b is a Boolean term; or (iii) a *sequential composition* $o_1; o_2$, where both $o_1, o_2 \in O$. A *term* is either a variable (x), a constant (1) or a function application ($x + 1$).

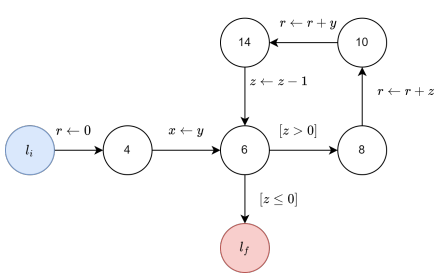
Semantics. Given a CFA $\Lambda = (L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E)$, its semantics is defined as a set of execution traces $\mathcal{L}(\Lambda)$. Each trace $\sigma \in \mathcal{L}(\Lambda)$ is a sequence of states $s_1, s_2, \ldots, s_n$, where each state s_i, $i \in \{1, \cdots, n\}$ is a tuple (l, v) that specifies its program location and variable assignment, respectively. Given a term a and an assignment v, let $v[a]$ denote the result of evaluating a under v. For variable assignments v and v', a variable x and a term a, $v' = v[x \leftarrow a]$ iff $v'[x] = a$ and $v'[y] = v[y]$ for variable $y \neq x$. We say that a trace σ is in $\mathcal{L}(\Lambda)$ iff (1) σ starts at location l_i, and (2) for every pair of consecutive states (l, v) and (l', v') in σ, there is an edge (l, o, l') such that v and v' satisfy the semantic relation induced by o:

$$R_o(v, v') = \begin{cases} v' = v[x \leftarrow a], & \text{if } o \text{ is } x := a \\ v[b] = \top, & \text{if } o \text{ is } \mathbf{assume}(b) \\ \exists v* . (R_{o_1}(v, v*) \wedge Ro_2(v*, v)) & \text{if } o \text{ is } o_1; o_2 \end{cases}$$

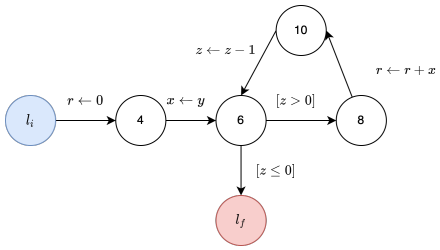
Definition 2.2 (Function Application)**.** Let a CFA Λ and two vectors $\vec{t}_{in}$ and $\vec{t}_{out}$ be given. We say that *the application of* Λ *on the input* $\vec{t}_{in}$ *produces the output* $\vec{t}_{out}$, denoted as $\Lambda(\vec{t}_{in}) = \vec{t}_{out}$ if and only if there exists a finite trace $\sigma = (l_0, v_0) \ldots (l_f, v_f)$ in $\mathcal{L}(\Lambda)$ s.t. $v_0[\vec{x}_{in}] = \vec{t}_{in}$ and $v_f[\vec{x}_{out}] = \vec{t}_{out}$.

A trace σ *terminates* if σ is finite and the last state is at l_f. A CFA Λ is *deterministic* if each state has a fixed successor state in all traces of $\mathcal{L}(\Lambda)$. Λ is complete if every trace in $\mathcal{L}(\Lambda)$ either terminates or is an infinite sequence (i.e., the execution does not get stuck at a non-final location). In the rest of the paper, we assume that all CFAs are deterministic and complete. Note that, under the deterministic and complete assumption, for every input $\vec{t}_{in}$, if Λ terminates, then $\Lambda(\vec{t}_{in})$ has a unique return value ($\vec{t}_{out}$).

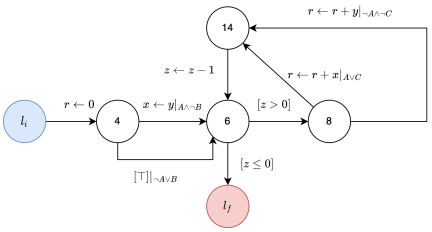
Example 2.1 (CFA). Consider the function F_1 in Fig. 1a, ignoring the presence conditions. The CFA of F_1 is shown in Fig. 2a. The trace $(l_i, \{r : 0,\ x : 0,\ y : 5,\ z : 0\})$, $(4, \{r : 0,\ x : 5,\ y : 5,\ z : 0\})$, $(l_f, \{r : 0\ x : 5,\ y : 5,\ z : 0\})$ is in the language $\mathcal{L}$(F_1). F_1 is deterministic since the only branching location is 4, and the branching conditions $[z > 0]$ and $[z \leq 0]$ are mutually exclusive.



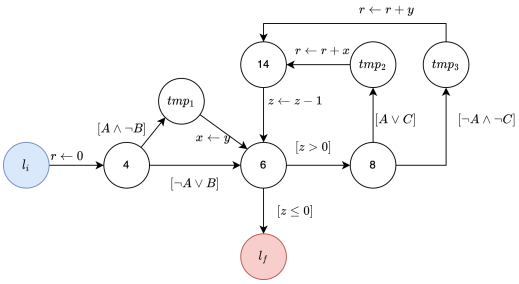
(a) The CFA of F_1 (without features).



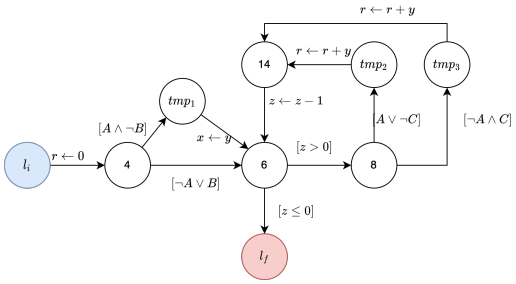
(b) The derived CFA of F_1 with configuration $(A, \neg B, \neg C)$.



(c) The variability-aware CFA of F_1.



(d) The metaproduct of F_1.



(e) The metaproduct of F_2.

Figure 2: (a) The CFA (without features); (b) a derived product (with configuration A); (c) a variability-aware CFA; (d) a metaproduct of the function F_1 (Fig. 1a); (e) a metaproduct of the function F_1 (Fig. 1b). Numbers in the nodes represent the program locations.

F_1 is complete because at least one transition is enabled for every location, including 4, because $z > 0 \vee z \leq 0 \Rightarrow \top$.

Definition 2.3 (Functional Equivalence)**.** Given two complete and deterministic CFAs Λ and Λ', Λ is *functionally equivalent* to Λ', denoted as $\Lambda \equiv \Lambda'$, iff on every input $\vec{t}_{in}$ where Λ and Λ' both terminate, $\Lambda(\vec{t}_{in}) = \Lambda'(\vec{t}_{in})$.

Note that the definition of functional equivalence excludes cases where Λ terminates and Λ' does not (or vice versa), and the study of termination is not within the scope of this paper.

2.2 Program Analysis

Hoare Triples and Counterexamples. Let a CFA $\Lambda = (L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E)$ be given. The assume-guarantee style analysis on Λ is represented as a Hoare triple [17]: $\{pre\}\Lambda\{post\}$ where pre and $post$ are expressions over $\vec{x}_{in}$ and $\vec{x}_{in} \cup \vec{x}_{out}$, respectively. The Hoare triple $\{pre\}\Lambda\{post\}$ is *valid* iff for all input values $\vec{t}_{in}$, $pre[x_{in} \leftarrow \vec{t}_{in}] = \top$ (denoted as $\vec{t}_{in} \models pre$) implies $post[x_{in} \leftarrow \vec{t}_{in}\ ; x_{out} \leftarrow \Lambda(\vec{t}_{in})] = \top$ (denoted as $\vec{t}_{in} \bigcup \Lambda(\vec{t}_{in}) \models post$). On the other hand, if there exists an input value $\vec{t}_{in}$ such that $t_{in} \models pre$ and $\vec{t}_{in} \bigcup \Lambda(\vec{t}_{in}) \not\models post$, then $\vec{t}_{in}$ is a *counterexample* (CEX) to the validity of the Hoare triple. Given a Hoare triple, a program verification tool (e.g., SeaHorn [16]) can be used to prove the validity of the Hoare triple or to generate a CEX.

Definition 2.4 (Generalized CEX)**.** Let a CFA $\Lambda = (L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E)$ and a Hoare triple $\{pre\}\Lambda\{post\}$ be given. A Boolean expression (over $\vec{x}_{in}$) g_{cex} is a *generalized counterexample (GCEX)* to the validity of $\{pre\}\Lambda\{post\}$ iff every input value $\vec{t}_{in}$ that satisfies g_{exp} ($\vec{t}_{in} \models g_{exp}$) is a CEX to the validity of $\{pre\}\Lambda\{post\}$.

We can compute a GCEX from a single CEX $\vec{t}_{in}$ by maximally identifying the set of (conjunction of atoms) $g_{cex} \leftarrow c_1 \wedge c_2 \ldots$ as long as $t_{in} \models g_{cex}$ and g_{cex} remains a correct GCEX. We will discuss the details of the GCEX computation in Sec. 5.

Example 2.2. Consider the function F_2 (see Fig. 1b) without presence conditions. The Hoare triple $\{y > 0 \wedge z > 0\}$ F_2 $\{r > 0\}$ is valid while $\{y > 0 \wedge z >= 0\}$ F_2 $\{r > 0\}$ has a counterexample $\vec{t}_{in} = (1, 1, 0)$. A GCEX generalized from $\vec{t}_{in}$ is $z \leq 0$.

Definition 2.5 (Self-Composition)**.** Let $\Lambda_1 = (L_1, l_0, l_f, X_1, \vec{x}_{in}, \vec{x1}_{out}, E_1)$ and $\Lambda_2 = (L_2, l_0, l_f, X_2, \vec{x}_{in}, \vec{x2}_{out}, E_2)$ be two CFAs with the same inputs $\vec{x}_{in}$. A *self-composition* of Λ and Λ' is a CFA $\Lambda_\times = (L_\times, l_0, l_f, X_\times, \vec{x}_{in}, \vec{x}_{out} \cup \vec{x}_{2out}, E_\times)$ with inputs $\vec{x}_{in}$ and outputs $\vec{x}_{out} \cup \vec{x}_{2out}$ such that for any input $\vec{t}_{in}$ where both Λ_1 and Λ_2 terminate, $\Lambda_1(\vec{t}_{in}) = \Lambda_\times(\vec{t}_{in})[\vec{x1}_{out}]$ and $\Lambda_2(\vec{t}_{in}) = \Lambda_\times(\vec{t}_{in})[\vec{x}_{2out}]$.

In contrast to the classical definition of self-composition [6], where multiple copies of the same CFAs are composed to verify k-safety properties such as determinism and noninterference, Def. 2.5 considers compositions between different (but similar) CFAs for equivalence checking. Existing self-composition approaches [32] are applicable as long as the declarative constraints in Def. 2.5 are satisfied. In Sec. 4, we demonstrate a concrete example of self-composition for equivalence checking (Example 4.4).

Theorem 2.3. *Let two CFAs Λ_1 and Λ_2 be given. If $\Lambda_\times$ is the self-composition of Λ_1 and Λ_2, then $\Lambda_1 \equiv \Lambda_2$ iff $\{\top\}\Lambda_\times\{\vec{x1}_{out} = \vec{x}_{2out}\}$.*

The proof follows directly from Def. 2.5. Thm. 2.3 allows us to reduce the problem of checking $\Lambda_1 \equiv \Lambda_2$ to checking the validity of a Hoare triple $\{\top\}\Lambda_\times\{\vec{x1}_{out} = \vec{x}_{2out}\}$.

2.3 Software Product Lines

A *software product line* (SPL) defines a collection of related software products, where each product can be enabled by a specific *feature configuration*. In *compositional* SPLs, features are implemented as separate units and are composed together by a desired feature configuration [1, 22]. *Delta-oriented programming* [29] uses a core module to define shared code for all products and specifies feature variability using delta modules. In *annotative* SPLs, features are explicitly annotated in the code base. In this work, we consider annotative SPLs at the level of programs represented as CFAs where variabilities are annotations on nodes and edges. Software product lines can thus be represented with *variability-aware CFAs*:

Definition 2.6 (Variability-Aware CFA and Derived CFA)**.** A *variability-aware CFA* $\mathscr{L}$ is a tuple $(F, \Phi, L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E, \phi)$, where L, l_0, l_f, X, $\vec{x}_{in}, \vec{x}_{out}, E$ is defined analogously to CFA. The additional attributes are: F, a finite set of *features*, Φ, a propositional formula on F called a *feature model*, and ϕ, a function that defines a *presence condition* $\phi(e)$ for the edges in E where $\phi(e)$ is a Boolean expression over F. A feature configuration ω is *valid* if $\Phi[F \leftarrow \omega]$ is evaluated to $\top$. Since F is a finite set of Boolean variables, we abuse the notation and refer to Φ as the set of all valid feature configurations. Given a valid feature configuration $\omega \in \Phi$, a derived product of ω, denoted as $\mathscr{L}|_\omega$, is a CFA $(L, l_0, l_f, \vec{x}_{in}, \vec{x}_{out}, E|_\omega)$, where $E|_\omega$ is the subset of E such that for every edge $e \in E|_\omega$, the presence condition $\phi(e)[F \leftarrow \omega]$ is evaluated to $\top$. The variability-aware CFA $\mathscr{L}$ defines the set of all the CFAs that can be derived from the feature models Φ and the presence condition ϕ. We assume that ϕ satisfies the following properties:

(**Determinism**). Let $g_1 = (l, o, l_1) \in G$ and $g_2 = (l, o, l_2) \in G$ be two edges that have the same source location l and operation o but a different destination location (i.e., $l_1 \neq l_2$). Then $\Phi \Rightarrow \neg(\phi(g_1) \wedge \phi(g_2))$.

(**Completeness**). Suppose $\gamma(l)$ is the set of all edges that start from location l. Then for any location $l \in L \backslash \{l_f\}, \Phi \Rightarrow \bigvee_{g \in \gamma(l)} \phi(g)$.

Example 2.4. Consider the function F_1 from Fig. 1a. The variability-aware CFA (denoted as $\mathscr{L}_{F_1}$) of F_1 is shown in Fig. 2c with feature variables $F = \{A, B, C\}$ and the feature model $\Phi = \top$. The two outgoing edges of location 4 have the presence conditions $A \wedge \neg B$ and $\neg A \vee B$, respectively, and the two outgoing edges of location 8 have the presence conditions $A \vee C$ and $\neg A \wedge \neg C$, respectively. Since the presence conditions of edges from the same source node are always mutually exclusive and complementary, the presence conditions are deterministic and complete. Under the feature model Φ, $\mathscr{L}_{F_1}$ has eight different feature configurations, and the product derived from the configuration $(A, \neg B, \neg C)$ is shown in Fig. 2b, where the edges with presence conditions $A \wedge \neg B$ and $A \vee C$ are enabled, while the edges with presence conditions $\neg A \vee B$ and $\neg A \wedge \neg C$ are disabled. The derived products are also deterministic and complete.

3 Variability-Aware Functional Equivalence

In this section, we formally define the problem of checking the functional equivalence of software product lines (called the *VEQ problem*). Then we describe the expected outcome of our approach that would solve the VEQ problem.

Definition 3.1 (VEQ Problem)**.** Let $\mathscr{L} = (F, \Phi, L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E, \phi)$ and $\mathscr{L}' = (F, \Phi, L', l_0, l_f, X', \vec{x}_{in}, \vec{x}_{out}, E', \phi')$ be two variability-aware CFAs that have the same feature variables F, feature models Φ, inputs $\vec{x}_{in}$ and outputs $\vec{x}_{out}$. The problem of verifying the functional equivalence of $\mathscr{L}$ and $\mathscr{L}'$ is to determine the functional equivalence of every derived product $\mathscr{L}|_\omega \equiv \mathscr{L}'|_\omega$ for a feature configuration $\omega \in \Phi$. Formally, the VEQ of $\mathscr{L}$ and $\mathscr{L}'$ computes a function $VEQ_{\mathscr{L} \equiv \mathscr{L}'} : \{\top, \bot\}^{|F|} \rightarrow \{\top, \bot\}$ such that for every $\omega \in \Phi$, $VEQ_{\mathscr{L} \equiv \mathscr{L}'}(\omega)$ iff $\mathscr{L}|_\omega \equiv \mathscr{L}'|_\omega$.

Naively, one could solve the VEQ problem by checking the functional equivalence for each derived product separately and then report the result for each feature configuration. However, the naive approach is unlikely to scale due to the number of feature configurations, as discussed in Sec. 1. Furthermore, the analysis results are produced for each individual product, which is not succinct ($O(2^{|F|})$). Instead of considering one configuration at a time, we propose to analyze a set of feature configurations and generate a *featured counterexample* (FCEX) as a summary of the equivalence status for the set. Intuitively, an FCEX reports the set of feature configurations whose derived products are nonequivalent due to the same set of CEXs.

Definition 3.2 (Featured Counterexample)**.** Given two variability-aware CFAs $\mathscr{L} = (F, \Phi, L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E, \phi)$ and $\mathscr{L}' = (F, \Phi, L', l_0, l_f, X', \vec{x}_{in}, \vec{x}_{out}, E', \phi')$, a *featured counterexample* (FCEX) to $\mathscr{L} \equiv \mathscr{L}'$ is a tuple (p, q), where p is a propositional formula over F and q is a quantifier-free formula over $\vec{x}_{in}$. An FCEX is *sound* iff for every feature configuration $\omega \in \Phi$ that satisfies p and for every input value $\vec{t}_{in}$ that satisfies q, we have $\mathscr{L}|_\omega(\vec{t}_{in}) \neq \mathscr{L}'|_\omega(\vec{t}_{in})$. We refer to p and q as the *head* and *body* of the FCEX, respectively.

Example 3.1. Consider the functions F_1 and F_2 in Fig. 1a and Fig. 1b, respectively. Let $\mathscr{L}_{F\_1}$ and $\mathscr{L}_{F\_2}$ be the variability CFA of F_1 and F_2, respectively. Then the tuple $(\neg A \wedge C, x = y > 0 \wedge z > 0)$ is a FCEX for $\mathscr{L}_{F\_1} \equiv \mathscr{L}_{F\_2}$. From the FCEX, we can show that $\mathscr{L}_{F\_1}|_{\{\neg A, B, C\}}(1, 1, 1) \neq \mathscr{L}_{F\_2}|_{\{\neg A, B, C\}}(1, 1, 1)$ since the feature configuration $\{\neg A, B, C\}$ satisfies the head $\neg A \wedge C$ and the input $(x = 1 \wedge y = 1 \wedge z = 1)$ satisfies the body $x = y > 0 \wedge z > 0$.

Definition 3.3 (NEQ-Summary)**.** Given two variability-aware CFAs $\mathscr{L} = (F, \Phi, L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E, \phi)$ and $\mathscr{L}' = (F, \Phi, L', l_0, l_f, X', \vec{x}_{in}, \vec{x}_{out}, E', \phi')$, an *NEQ-summary* $\text{Cex}^{\uparrow}(\mathscr{L}, \mathscr{L}')$ is a set of FCEXs to $\mathscr{L} \equiv \mathscr{L}'$. The summary is sound if all FCEXs in the summary are sound. The summary is complete if for every feature configuration $\omega \in \Phi$ such that $\mathscr{L}|_\omega \not\equiv \mathscr{L}'|_\omega$ (the derived products are not functionally equivalent), there exists an FCEX = (p, q) such that ω satisfies p. The summary is *minimal* if, for every pair of FCEXs (p, q) and (p', q') in the summary, $q \wedge q'$ is UNSAT.

Theorem 3.2 (Validity of NEQ-Summary). *Let two variability-aware CFAs $\mathscr{L}, \mathscr{L}'$ with feature model Φ be given. If a NEQ-summary $\text{Cex}^{\uparrow}(\mathscr{L}, \mathscr{L}')$ is sound and complete, then $\text{Cex}^{\uparrow}(\mathscr{L}, \mathscr{L}')$ answers the VEQ problem. More specifically, for every feature configuration $\omega \in \Phi$, $\mathscr{L}|_\omega \not\equiv \mathscr{L}'|_\omega$ iff there exists some FCEX (p, q) in $\text{Cex}^{\uparrow}(\mathscr{L}, \mathscr{L}')$ such that $\omega \models p$.*

Proof. For every feature configuration $\omega \in \Phi$, if there is an FCEX (p, q) in $\text{Cex}^{\uparrow}(\mathscr{L}, \mathscr{L}')$ such that ω satisfies p, then $\mathscr{L}|_\omega \not\equiv \mathscr{L}'_\omega$ due to the soundness of the NEQ-summary. On the other hand,

if there is no such FCEX (p, q), then $\mathscr{L}|_\omega \equiv \mathscr{L}'_\omega$ due to the completeness of the NEQ-summary. □

Given two variability-aware CFAs $\mathscr{L}$ and $\mathscr{L}'$, we want to compute a complete and sound NEQ-summary while minimizing the FCEXs. Finding the summary with the smallest number of FCEXs is expensive, since it reduces to the high-dimensional geometric set cover problem, which is NP-complete [15]. Instead, we wish to compute a minimal NEQ-summary and merge FCEXs as much as possible (while preserving soundness) based on Thm. 3.3.

Theorem 3.3. *Let (p, q) and $(p', q') \in \mathrm{Cex}^{\uparrow}(\mathscr{L}, \mathscr{L}')$ be two FCEXs in a sound and complete NEQ-summary. If $q \wedge q'$ is satisfiable, replacing (p, q) and (p', q') with $(p \vee p', q \wedge q')$ in the summary preserves soundness and completeness.*

Proof. The replacement satisfies the completeness property because $\vee$ is associative and commutative. The replacement is sound because $q \wedge q'$ is satisfiable and is stronger than q and q'. □

Example 3.4. Consider functions F_1 and F_2 shown in Fig. 1a and Fig. 1b, respectively. Suppose that $\mathscr{L}_{F\_1}$ and $\mathscr{L}_{F\_2}$ are the variability CFA of F_1 and F_2, respectively. The NEQ-summary (for $\mathscr{L}_{F\_1} \equiv \mathscr{L}_{F\_2}$),

$$\{(\neg A \wedge C,\ x \neq y \wedge z > 0),\ (B \wedge A,\ x \neq y \wedge z > 0)\}$$

is sound and complete. From the first FCEX of the summary, we can show that $\vec{x}_{in} = \{x = 1,\ y = 2,\ z = 3\}$ is a CEX under the feature configuration $\omega = \{\neg A, B, C\}$ because $\omega \models \neg A \wedge \neg C$ and $\vec{x}_{in} \models x \neq y \wedge z > 0$. Moreover, we can show that $\mathscr{L}_{F\_1}|_\omega \equiv \mathscr{L}_{F\_2}|_\omega$ for any feature configuration $\omega \not\models (\neg A \wedge C) \vee (B \wedge A)$.

The summary is not minimal and can be further minimized by merging FCEXs since their bodies (see Def. 3.2) have an non-empty intersection. More specifically, all three FCEXs have the same body $x \neq y \wedge z > 0$. Therefore, FCEXs can be merged by taking the union of their heads. After merging FCEXs, we obtain the minimized NEQ-summary: $\{((\neg A \wedge C) \vee (A \wedge B), x \neq y \wedge z > 0)\}$ (the head is simplified). For every feature configuration $\omega \models (\neg A \wedge C) \vee (A \wedge B)$, the derived products $\mathscr{L}_{F\_1}|_\omega$ and $\mathscr{L}_{F\_2}|_\omega$ are equivalent.

4 Computing NEQ-Summary

In this section, we present our approach for computing the NEQ-summary for the VEQ problem. The approach consists of three steps (see Fig. 3): (1) transforming the variability-aware CFA into a *metaproduct* CFA where the feature variables are replaced with program variables; (2) reducing the problem of CFA equivalence checking into verification of a safety property, expressed as a Hoare triple, over the self-composed CFA; (3) iteratively producing generalized counterexamples of the Hoare triple and adding them into NEQ-summary until the summary is complete. We describe Step (1) in Sec. 4.1 and Steps (2)–(3) in Sec. 4.2. The proofs of soundness, complexity, and optimality of the approach are in Sec. 4.3.

4.1 Metaproduct

In this section, we adopted a semantic-preserving transformation (a.k.a. *variability encoding*), proposed by Apel et al. [19], that encodes variability-aware CFAs into CFAs, denoted as *metaproducts*. The transformation allows us to reduce the VEQ problem (Def. 3.1)

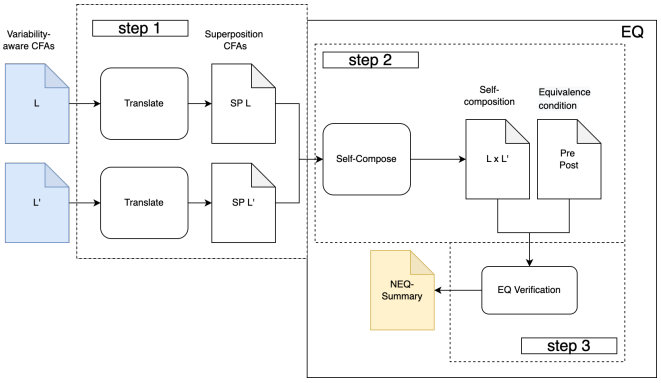


Figure 3: The workflow for computing the NEQ-summary to the VEQ problem. Inputs are in blue, outputs in yellow.

on the variability-aware CFAs to the equivalence checking (Def. 2.3) problem on their metaproducts.

Definition 4.1 (Metaproduct)**.** Let a variability-aware CFA $\mathscr{L} = (F, \Phi, L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E, \phi)$ be given. A CFA $\mathscr{L}\uparrow = (L \cup l'_0, l'_0, l_f, F \cup X, F \cup \vec{x}_{in}, \vec{x}_{out}, E')$ is a *metaproduct* of $\mathscr{L}$ iff the edges in E' satisfy the following rules: (1) the edge connecting the initial location $(l'_0,\ \mathbf{assume}(\Phi),\ l_0)$ is an edge in E' and (2) for every edge $e = (l_i, o, l_f) \in E$, there exists an edge $e' = (l_i, o', l_f)$ where o' is a sequential composition of $\mathbf{assume}(\phi(e))$ and o.

Intuitively, the metaproduct of a variability-aware CFA converts the feature variable, F, to the program variables, $X \cup F$, and pushes the enabling conditions, ϕ, as assumptions on the edges (rule (2) of E'). In addition, a new initial location l'_0 and its connecting edge $(l'_0,\ \mathbf{assume}(\Phi),\ l_0)$ are introduced to restrict the space of feature configurations by the feature model Φ.

Example 4.1. Consider the variability-aware CFA $\mathscr{L}_{F\_1}$ shown in Fig. 2c for function F_1 (in Fig. 1a). The metaproduct of $\mathscr{L}_{F\_1}$ is shown in Fig. 2d, where the feature variables A, B and C are converted into program inputs. In the metaproduct, edges connecting to a temporary location (e.g., tmp_1, tmp_2 and tmp_3) are the assumption edges for capturing presence conditions. For example, the sequential composition $(4, \mathbf{assume}(A \wedge \neg B), tmp_1); (tmp_1, x \leftarrow y, 6)$ in the metaproduct corresponds to the edge $(4, x \leftarrow y|_{A \wedge \neg B}, 6)$ in $\mathscr{L}_{F\_1}$ with the presence condition $A \wedge \neg B$. The metaproduct is deterministic and complete.

The metaproduct preserves all traces in any CFAs derived from the variability-aware CFA.

Theorem 4.2. *Given a variability-aware CFA $\mathscr{L} = (F, \Phi, L, l_0, l_f, X, \vec{x}_{in}, \vec{x}_{out}, E, \phi)$ and its metaproduct $\mathscr{L}\uparrow$, the traces of the metaproduct $\mathcal{L}(\mathscr{L}\uparrow)$ are equivalent to the union of all traces from the derived products $\bigcup_{\omega \in \Phi} \mathcal{L}(\mathscr{L}|_\omega)$*

Proof. We prove trace equivalence in two directions: (1) $\mathcal{L}(\mathscr{L}\uparrow) \supseteq \bigcup_{\omega \in \Phi} \mathcal{L}(\mathscr{L}|_\omega)$ and (2) $\mathcal{L}(\mathscr{L}\uparrow) \subseteq \bigcup_{\omega \in \Phi} \mathcal{L}(\mathscr{L}|_\omega)$.

We prove (1) by contradiction: Suppose there exists a trace $\sigma \in \mathcal{L}(\mathscr{L}|_\omega)$ but not in $\mathcal{L}(\mathscr{L}^{\uparrow})$. Then either (1a) there is an initial state s_0 allowed in $\mathscr{L}|_\omega$ but not in $\mathscr{L}^{\uparrow}$, or (1b) there is a state transition between s_i, s_{i+1} that is permitted in $\mathscr{L}|_\omega$ but not in $\mathscr{L}^{\uparrow}$. Case (1a) is not possible, since the program variables X are the same for

Algorithm 1 EQ checks the equivalence of two metaproduct CFAs and produces a NEQ-summary.

Input Metaproduct CFAs $\mathscr{L}^{\uparrow}$ and $\mathscr{L}^{\uparrow\prime}$
Output A NEQ-summary

1: **procedure** EQ($\mathscr{L}^{\uparrow}, \mathscr{L}^{\uparrow\prime}$)
2: $\quad\mathscr{L}^{\uparrow}{}_{\times} \leftarrow$ Self-compose($\mathscr{L}^{\uparrow}, \mathscr{L}^{\uparrow\prime}$)
3: $\quad$*NEQ-summary* $\leftarrow \{\}$, $pre \leftarrow \top$
4: $\quad\vec{t}_{in} \leftarrow$ Verify&CEX$\{pre\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$
5: $\quad$**while** $\vec{t}_{in} \neq \varnothing$ **do**
6: $\quad\quad c \leftarrow$ Generalize($\vec{t}_{in}$)
7: $\quad\quad p, q \leftarrow$ Split(c)
8: $\quad\quad p*, q* \leftarrow$ GeneralizeFCEXs($pre, p, q, \mathscr{L}^{\uparrow}{}_{\times}$)
9: $\quad\quad$*NEQ-summary* $\leftarrow$ *NEQ-summary* $\cup \{(p*, q*)\}$
10: $\quad\quad pre \leftarrow pre \wedge \neg p*$
11: $\quad\quad\vec{t}_{in} \leftarrow$ Verify&CEX$\{pre\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$
12: $\quad$**end while**
13: $\quad$**return** *NEQ-summary*
14: **end procedure**

Algorithm 2 GeneralizeFCEXs generalizes an FCEX (p, q) into $(p*, q*)$ where $q*$ is stronger than q, and $p*$ represents the set of all feature configurations that are non-equivalent due to the input $q*$.

Input A FCEX (p, q), a precondition *pre* and a self-composition $\mathscr{L}^{\uparrow}{}_{\times}$
Output A NEQ-summary

1: **procedure** GeneralizeFCEXs(pre, p, q, $\mathscr{L}^{\uparrow}{}_{\times}$)
2: $\quad p* \leftarrow p$ and $q* \leftarrow q$
3: $\quad\vec{t}_{in} \leftarrow$ Verify&CEX$\{pre \wedge \neg p* \wedge q*\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$
4: $\quad$**while** $\vec{t}_{in} \neq \varnothing$ **do**
5: $\quad\quad c \leftarrow$ Generalize($\vec{t}_{in}$)
6: $\quad\quad p', q' \leftarrow$ Split(c)
7: $\quad\quad p* \leftarrow p* \vee p'$ and $q* \leftarrow q* \wedge q'$
8: $\quad\quad\vec{t}_{in} \leftarrow$ Verify&CEX$\{pre \wedge \neg p* \wedge q*\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$
9: $\quad$**end while**
10: $\quad$**return** $p*, q*$
11: **end procedure**

the two CFAs, and the assignment over the feature variables F is fixed by ω in $\mathscr{L}|_{\omega}$ but free in $\mathscr{L}^{\uparrow}$. For case (1b), suppose that the transition from s_i to s_{i+1} is due to an edge $e = (l, o, l')$ of $\mathscr{L}|_{\omega}$. By construction of E' (the edges of $\mathscr{L}^{\uparrow}$), there exists an edge $e' = (l, o', l')$ in E' where $o' = \textbf{assume}(\phi(e))\ ; o$. Since e is enabled by ϕ under assignment ω, assigning F by ω will pass the assumption in e'. Therefore, the transition from s_i to s_{i+1} is also available in $\mathscr{L}^{\uparrow}$, which is a contradiction.

We prove (2) by realizing that the assignment over F in $\mathscr{L}^{\uparrow}$ is static (it is determined in the initial state s_0 and does not change over states). For every trace $\sigma \in \mathcal{L}(\mathscr{L}^{\uparrow})$, we denote the static assignment in F by ω', and then σ is also a trace in $\mathscr{L}|_{\omega'}$. Moreover, the initial transition of the metaproduct $(l'_0, \textbf{assume}(\Phi), l_0)$ restricts the valid space of ω' by Φ. Therefore, $\mathscr{L}|_{\omega'}$ is a derived product of $\mathscr{L}$. □

Theorem 4.3. *Let $\mathscr{L}$ and $\mathscr{L}'$ be two variability-aware CFA with metaproducts $\mathscr{L}^{\uparrow}$ and $\mathscr{L}^{\uparrow\prime}$, respectively. For any configuration ω, if $\vec{x}_{in}$ is a CEX of equivalence in the derived product such that $\mathscr{L}|_{\omega}(\vec{x}_{in}) \neq \mathscr{L}'|_{\vec{x}_{in}}$, then $\vec{x}_{in} \cup \omega$ is a CEX of equivalence for their meta-products (i.e., $\mathscr{L}^{\uparrow}(\vec{x}_{in} \cup \omega) \neq \mathscr{L}^{\uparrow\prime}(\vec{x}_{in} \cup \omega)$).*

Thm. 4.3 is a direct consequence of Thm. 4.2. It enables performing equivalence analysis on the metaproduct instead of the derived products.

4.2 An Algorithm to Compute NEQ-Summary

In this section, we describe EQ, the equivalence analysis algorithm (see Alg. 1). We study its soundness, run-time complexity, and optimality in Sec. 4.3.

EQ takes two metaproduct CFAs, $\mathscr{L}^{\uparrow}$ and $\mathscr{L}^{\uparrow\prime}$ (transformed from variability-aware CFAs $\mathscr{L}$ and $\mathscr{L}'$, respectively), as input and produces a complete and sound NEQ-summary. EQ first self-composes $\mathscr{L}^{\uparrow}$ and $\mathscr{L}^{\uparrow\prime}$ to obtain $\mathscr{L}^{\uparrow}{}_{\times}$ as the self-composition (Line 2), and then, following Thm. 2.3, checks the equivalence of $\mathscr{L}^{\uparrow}$ and $\mathscr{L}^{\uparrow\prime}$ by verifying the validity of the Hoare triple $\{\top\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$(Line 4). If the Hoare triple is valid, an empty NEQ-summary is returned (Line 13). Otherwise, EQ obtains a CEX $\vec{t}_{in}$, and invokes Generalize to obtain a GCEX (see Def. 2.4) c (Line 6). EQ then splits c into subformulae p and q over feature variables F and program variables X, respectively (Line 7). EQ then adds (p, q) as a FCEX (see Def. 3.2) to the NEQ-summary (Line 9). Since p represents a set of non-equivalent feature configurations, EQ does not need to consider them again for equivalence analysis and hence adds $\neg p$ to the precondition of the Hoare triple (Line 10). After strengthening the precondition, EQ verifies the Hoare triple again (Line 11). If a CEX exists, EQ adds more FCEXs to the NEQ-summary and strengthens the precondition (Lines 5–12). EQ terminates when the Hoare triple becomes valid, returning the NEQ-summary (Line 13).

We observed in the example in Fig. 1 that if an input is an CEX for a feature configuration, then it is likely a CEX for other feature configurations as well. Based on this observation, we developed GeneralizeFCEXs — an optimization for EQ to further generalize FCEXs before adding them to NEQ-summary (Line 8 in Alg. 1). Computation of GeneralizeFCEXs is given in Alg. 2. The algorithm takes a featured counterexample (p, q), the current precondition pre, and the self-composition $\mathscr{L}^{\uparrow}{}_{\times}$ as inputs, and generalizes (p, q) into a new FCEX $(p*, q*)$, where $p \rightarrow p*$ and $q* \rightarrow q$. Intuitively, the input FCEX (p, q) is generalized by expanding the feature configurations covered in the head $(p*)$ while ensuring the resulting FCEX is sound and has a non-empty body $(q*)$. Generalization is achieved by repeatedly identifying GCEXs $p' \wedge q'$ to the validity of the Hoare triple $\{pre \wedge q* \wedge \neg p*\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$ (Line 6 of Alg. 2) and adding p' and q' to $p*$ and $q*$, respectively (Line 7). The generalization ends when the Hoare triple becomes valid.

Example 4.4. Consider the functions F_1 and F_2 in Fig. 1a and Fig. 1b, respectively, and let $\mathscr{L}_{F\_1}$ and $\mathscr{L}_{F\_2}$ be the variability-aware CFAs for F_1 and F_2. We illustrate how EQ (Alg. 1) computes NEQ-summary for $\mathscr{L}_{F\_1} \equiv \mathscr{L}_{F\_2}$.

Before calling EQ, we first pre-process variability-aware CFAs $\mathscr{L}_{F\_1}$ and $\mathscr{L}_{F\_2}$ into meta-products $\mathscr{L}^{\uparrow}{}_{F\_1}$ and $\mathscr{L}^{\uparrow}{}_{F\_2}$ shown in Fig. 2d and Fig. 2e, respectively. Then Alg. 1 self-composes (Line 2) $\mathscr{L}^{\uparrow}{}_{F\_1}$ and $\mathscr{L}^{\uparrow}{}_{F\_2}$ into $\mathscr{L}^{\uparrow}{}_{\times}$ (shown in Fig. 4). The variables r_1 and r_2 in $\mathscr{L}^{\uparrow}{}_{\times}$ store the values of the *same* variable r from $\mathscr{L}^{\uparrow}{}_{F\_1}$ and $\mathscr{L}^{\uparrow}{}_{F\_2}$, respectively. The other program variables in $\mathscr{L}^{\uparrow}{}_{F\_1}$ and $\mathscr{L}^{\uparrow}{}_{F\_2}$ always have the same value and thus are preserved in

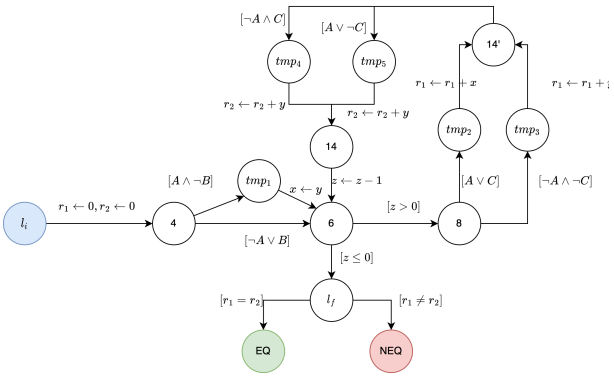


Figure 4: A self-composition of the F_1 and F_2 metaproduct.

$\mathscr{L}^{\uparrow}{}_{\times}$. When an execution reaches the final location l_f in $\mathscr{L}^{\uparrow}{}_{\times}$, the values of r_1 and r_2 are then returned as the output of $\mathscr{L}^{\uparrow}{}_{\times}$.

Then EQ invokes Verify&CEX (Line 4) to identify a CEX to the Hoare tripe $\{pre\}\mathscr{L}^{\uparrow}{}_{\times}(r_1 = r_2)$ where pre is initially $\top$. Suppose the CEX $\vec{t}_{in} = (A,\ B,\ C,\ x = 1,\ y = 0,\ z = 1)$ is returned. Then EQ generalizes $\vec{t}_{in}$ (Line 6) into a generalized counterexample (GCEX) $c \leftarrow (A \wedge B \wedge z = 1 \wedge x \neq y)$. Then EQ splits the GCEX c (Line 7) into a FCEX (p, q) where p is $(A \wedge B)$ and q is $(z = 1 \wedge x \neq y)$. Then EQ calls GeneralizeFCEXs to generalize (Line 8) the FCEX (p, q).

GeneralizeFCEXs tries generalizing the FCEX (p, q) by first checking validity of the Hoare triple $\{pre \wedge \neg p* \wedge q*\}\mathscr{L}^{\uparrow}{}_{\times}(r_1 = r_2)$ where $pre \leftarrow \top$, $p* \leftarrow p$ and $q* \leftarrow q$ (Line 3). Suppose a CEX $\vec{t}_{in} = (\neg A,\ B,\ C,\ x = 1,\ y = 0,\ z = 1)$ is returned. The CEX is then generalized into a FCEX (p', q') where $p' = \neg A \wedge C$ and $q' = x \neq y \wedge z = 1$ (same as q). Then pre is updated as $p* \leftarrow p* \vee p'$ and $q*$ is updated as $q* \leftarrow q* \wedge q'$ (Lines 5–7). Then the updated Hoare triple's validity is checked again (Line 8) to obtain new CEXs. Eventually, $p*$ becomes $(A \wedge B) \vee (\neg A \wedge C)$ and $q*$ becomes $(z = 1 \wedge x \neq y)$. At this point, the Hoare triple $\{pre \wedge \neg p* \wedge q*\}\mathscr{L}^{\uparrow}{}_{\times}(r_1 = r_2)$ becomes valid, and GeneralizeFCEXs returns the generalized FCEX $(p*, q*)$.

After obtaining the generalized FCEX $(p*, q*)$ from GeneralizeFCEXs, EQ updates $pre \leftarrow pre \wedge \neg p*$ (Line 10), and then checks the validity of the Hoare triple $\{pre\}\mathscr{L}^{\uparrow}{}_{\times}(r_1 = r_2)$ again (Line 11). The Hoare triple is valid and EQ returns the single entry NEQ-summary $\{((A \wedge B) \vee (\neg A \wedge C), z = 1 \wedge x \neq y)\}$. The NEQ-summary is sound, complete and minimal.

4.3 Soundness, Complexity and Summary Minimality

In this section, we first prove the soundness of EQ (Alg. 1), and then study its runtime complexity. Finally, we prove that the NEQ-summary returned by EQ is minimal.

Theorem 4.5 (Soundness). *Let two variability-aware CFAs $\mathscr{L}$, $\mathscr{L}'$ and their respective meta-products $\mathscr{L}^{\uparrow}$, $\mathscr{L}^{\uparrow'}$ be given. The NEQ-summary produced by EQ($\mathscr{L}^{\uparrow}, \mathscr{L}^{\uparrow'}$) is complete and sound.*

Proof. First, we prove that the NEQ-summary returned by EQ is always sound. An FCEX $(p*, q*)$ is added to the NEQ-summary (Line 9 of Alg. 1) iff $(p*, q*)$ is an FCEX returned by GeneralizeFCEXs (Line 8 of Alg. 1). The formulae $\neg p*$ and $q*$ are incrementally strengthened from $\neg p$ and q by intersecting with $\neg p'$ and q', respectively, where (p', q') is a GCEX for the Hoare triple $\{pre \wedge \neg p* \wedge q*\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$ (Line 7 of Alg. 2). Since $\neg p* \rightarrow \neg p'$ and $q* \rightarrow q'$, $(p*, q*)$ is also a GCEX for the Hoare triple $\{pre \wedge q\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$. Since $pre \wedge q \rightarrow \top$, $(p*, q*)$ is also a GCEX for the Hoare triple $\{\top\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$ By Def. 2.4, every input $\vec{v'}_{in}$ that satisfies $p* \wedge q*$ is a CEX of the Hoare triple $\{\top\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$. By Thm. 2.3 and 4.3, these inputs are also CEXs of equivalence for the derived CFAs. Therefore, every FCEX added to NEQ-summary is sound, and NEQ-summary is sound as well.

Second, we prove that the NEQ-summary returned by EQ is always complete. EQ terminates and returns the NEQ-summary iff the Hoare triple $\{pre\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$ (Line 11 of Alg. 1) has no CEX (Line 5 of Alg. 1). At every iteration, the current NEQ-summary is $\{(p_1, q_1) \ldots, (p_n, q_n)\}$ iff the precondition pre is $\neg p_1 \wedge \ldots \neg p_n$. By Thm. 2.3 and 4.3, the absence of CEX for the Hoare triple implies that no feature configuration outside of $p_1 \vee \ldots p_n$ is non-equivalent. Therefore, the returned NEQ-summary is complete. □

EQ calls the procedure Verify&CEX to soundly verify the Hoare triples. Since program verification is undecidable in general, EQ might not terminate as Verify&CEX might not always terminate. Therefore, instead of analyzing the algorithm's worst-case runtime complexity, we analyze the number of times Verify&CEX is invoked in a single run, since Verify&CEX is a performance bottleneck.

Theorem 4.6 (Complexity). *Let two variability-aware CFAs $\mathscr{L}$, $\mathscr{L}'$ and their respective meta-products $\mathscr{L}^{\uparrow}$, $\mathscr{L}^{\uparrow'}$ be given. Suppose F is the set of feature variables for $\mathscr{L}$ and $\mathscr{L}'$. Then the verification procedure Verify&CEX is invoked at most $O(2^{|F|})$ times during a single run of EQ.*

Proof. Since F is a finite set of boolean variables, the number of feature configurations is bounded by $O(2^{|F|})$. Let $\overline{P}$ be the set of non-equivalent feature configurations $\overline{P} = \{\omega \mid (\mathscr{L}|_{\omega} \not\equiv \mathscr{L}'|_{\omega})\}$. We show that the satisfying solution of the formula $\overline{P} \wedge pre$ is monotonically decreasing by at least x at every iteration of EQ (Line 5 of Alg. 1), where x is the number of times Verify&CEX is invoked during the iteration. The decrease in the number of solutions can be proved by the fact that pre is strengthened by adding $\neg p$ at each iteration (Line 10 of Alg. 1), and $\neg p$ is a conjunction of feature expressions $\neg p_1 \wedge \ldots \neg p_x$, where each feature expression $\neg p_i \in \neg p$ contains at least one distinct feature configuration in $\overline{P} \wedge pre$. This is because $p_i \wedge q$ is a GCEX from the CEX $\vec{t}_i$ of the Hoare triple $\{pre \wedge \neg p_1 \ldots \neg p_{i-1}\}\mathscr{L}^{\uparrow}{}_{\times}\{\vec{x}_{1out} = \vec{x}_{2out}\}$ (Line 6 of Alg. 1). Since $\vec{t}_i$ is a CEX, $\vec{t}_{in} \in \overline{P}$ (valid), $\vec{t}_{in} \models pre$ (not blocked), and $t_i \models \neg p_1 \wedge \neg \ldots p_{i-1}$ (distinct). By Def. 2.4, $\vec{t}_i \models p_i$. Therefore, given $\neg p = \neg p_1 \wedge \ldots \neg p_x$, the number of solutions of $pre \wedge \neg p$ is at least x less than the number of solutions in pre. Note that EQ produces one $\neg p_i$ for every call to Verify&CEX (Line 11 of Alg. 1 and Line 8 of Alg. 2), and hence the number of invocations of Verify&CEX is x. Given that the algorithm ends when $pre \wedge \overline{P} = \varnothing$, EQ calls Verify&CEX at most $O(2^{|F|})$ times. □

Theorem 4.7 (Minimality). *Let two variability-aware CFAs $\mathscr{L}$, $\mathscr{L}'$ and their respective meta-products $\mathscr{L}^{\uparrow}, \mathscr{L}^{\uparrow\prime}$ be given. If EQ($\mathscr{L}^{\uparrow}, \mathscr{L}^{\uparrow\prime}$) terminates and returns a NEQ-summary S, then S is minimal.*

Proof. By Thm. 3.3, a summary S is minimal if for any pair of FCEXs (p, q) and (p', q') in S, $q \wedge q'$ is UNSAT. Every FCEX $(p*, q*)$ in the summary S is generalized by FCEX returned by GeneralizeFCEXs (Line 8 of Alg. 1). GeneralizeFCEXs terminates and returns $(p*, q)$ iff the Hoare triple $\{pre \wedge \neg p* \wedge q*\} \mathscr{L}^{\uparrow} \times \{\vec{x}_{1out} = \vec{x}_{2out}\}$ becomes valid (Line 8 of Alg. 2). This means that there are no feature configurations in the space of $pre \wedge \neg p*$ that can derive non-equivalent CFAs with CEXs in the space of $q*$. Since pre represents the space of feature configurations that are not already in the summary S, for any FCEX (p_δ, q_δ) added to S after $(p*, q*)$, $q_\delta \wedge q*$ is UNSAT. Moreover, for any FCEX (p_δ, q_δ) added before $(p*, q*)$, $q_\delta \wedge q*$ is also UNSAT (since $(p*, q*)$ is added to S after (p_δ, q_δ), and the constraint on $(p*, q*)$ is also applied to (p_δ, q_δ)). Therefore, S is a minimal summary. □

5 Implementation

We have implemented a prototype of the lifted functional equivalence checker, CLEVER-V, using 3000 lines of Scala code. CLEVER-V includes the front-end `C_to_VCFAs` for compiling C source code to variability-aware CFAs) and the implementation of Steps (1)–(3) in Fig. 3. CLEVER-V uses TypeChef [20] as its front end to parse annotated software product lines expressed in C language and convert them to variability-aware CFAs. For Step (1), CLEVER-V implements the technique proposed by Apel et al. [19] to generate metaproducts. For Step (2), CLEVER-V uses the algorithm implemented by Feng et al. [14] to self-compose metaproducts. For Step (3), CLEVER-V implements the algorithm EQ (Alg. 1) and the optimization GeneralizeFCEXs (Alg. 2). The implementation of Step (3) uses SeaHorn [16] as the back-end verifier (Verify&CEX on Lines 4 and 11 of Alg. 1 and Lines 3 and 8 of Alg. 1). It also uses the counter-example generalization technique proposed by Hui et al. [18] on Line 6 of Alg. 1 and Line 5 of Alg. 2 to obtain GCEXs.

Limitations. Our Variability-aware CFA model (Def. 2.6) assumes that the input functions are deterministic and complete. The model only captures feature variabilities in annotative SPLs but not compositional or delta-oriented ones. The definition of functional equivalence (Def. 3.1) assumes that two input functions have the same signatures (i.e., the same inputs and outputs). In addition, components that our implementation uses impose some technical constraints: source-level self-composition [14] requires that the functions be non-recursive, well-structured and exiting from their last statement. The back-end verifier [16] limits data types to chars, integers, Booleans and non-parametric arrays. Both of these technical limitations can be removed as better underlying components become available.

6 Evaluation

In this section, we evaluate our lifted equivalence analyzer, CLEVER-V, to answer the following research questions: **RQ1**: How effective is our approach at solving VEQ problems? **RQ2**: How does our approach scale compared to the brute-force approach as the number of features increases? To answer RQ1 (effectiveness), we prepared a case-study for analyzing the change impact of a real software update in BusyBox. To answer RQ2 (scalability), we synthesized a set of "hard" benchmark instances by increasing the variability of the case study and proposing a wide range of updates.

6.1 Case-study Preparation

We selected the case-study from historical commits of the BusyBox software product line [1] (a software suite that includes several Unix utilities). For each commit C, let S and S' be the set of C-language functions before and after C. C can only be selected if there exist functions $F \in S$ and $F' \in S'$ that satisfy the following properties: (i) F and F' both have return values (i.e., their type is not `void`), (ii) the signatures of F and F' are the same in terms of function names and the return type, as well as parameter names and types; (iii) the code structures of the function bodies of F and F' are different (i.e., the new code is not a reformat of the old code); and (iv) there is at least one feature and one feature-specific software artifact annotated using a feature expression (i.e., the two functions we are analyzing are SPLs with `#ifdef` variability and are not single products).

For each selected commit C and a pair of functions F and F' that satisfy the selection criteria, we aimed to prepare a VEQ problem instance for checking the functional equivalence of F and F' ($F \equiv F'$). We manually made a few modifications to eliminate programming constructs unsupported by our front-end (TypeChef), middle-end (self-composition), or back-end (SeaHorn). For example, we rewrote `enum`'s into constants and bitwise operations into equivalent C expressions. We also decomposed members in C `struct`'s into individual variables (e.g., `struct { int x; int y; } s;` became `int s_x; int s_y;`) to accommodate our middle-end. In addition, we turned C standard library functions for strings (e.g., `strcmp`, `strcoll`) into uninterpreted functions since our back-end did not support reasoning about them. We identified the first VEQ instance ($F \equiv F'$) where the use of uninterrupted functions in F and F' does not affect their equivalence, and used it for our case-study, referring to this instance as b-Orig.

6.2 RQ1: Effectiveness

To answer RQ1, we use the case-study instance b-Orig constructed from the commit `6b01b71e` (see Fig. 5). The function affected by the update (i.e., `sortcmp`) is part of the BusyBox's implementation of the Unix `ls` command, which is used to display file information in sorted order. It compares the statistics of two file entries and returns an integer, whose sign tells `ls` which entry should come first in the list (i.e., if it returns a positive integer, then the first entry should appear before the second and vice versa).

The commit removes the while loop inside the else branch and changes the number of bits to shift in the if branch from `sizeof(int)*8` to enum `BITS_TO_SHIFT`, or equivalently, `8 * (sizeof(dif) - sizeof(int))`. The code versions before and after the commit both aim to right shift the variable `dif` until it is small enough to fit into an `int`. This update should not change the behaviour of the program if `sizeof(dif) == sizeof(int) * 2` evaluates to `true`. `sortcmp` has two features: `CONFIG_LOCALE_SUPPORT` and `CONFIG_LFS`, both are not visible from the figure.

[1] https://busybox.net/

```
510 510     /* Make dif fit into an int */
511 511         if (sizeof(dif) > sizeof(int)) {
512     -       if (sizeof(dif) == sizeof(int)*2) {
513     -           /* typical on many arches */
514     -           if (dif != 0) {
515     -               dif = 1 | (int)((uoff_t)dif >> (sizeof(int)*8));
516     -           }
517     -       } else {
518     -           while ((dif & ~(off_t)INT_MAX) != 0) {
519     -               dif >>= (sizeof(int)*8 / 2);
520     -           }
    512 +       enum { BITS_TO_SHIFT = 8 * (sizeof(dif) - sizeof(int)) };
    513 +       /* shift leaving only "int" worth of bits */
    514 +       if (dif != 0) {
    515 +           dif = 1 | (int)((uoff_t)dif >> BITS_TO_SHIFT);
521 516         }
522 517     }
```

Figure 5: Changes to the file `coreutils/ls.c` in commit 6b01b71e. Lines starting with - (resp. +) represent the code deleted (resp. inserted) by the commit.

We ran our prototype on b-Orig, and the result is an empty NEQ-Summary, which means commit `6b01b71e` preserves functional equivalence under all feature configurations. This result is in line with our expectations.

To illustrate our approach for handling non-equivalent updates, we proposed a change. Suppose that commit `6b01b71e` also changed line 515 as followings:

```
-   dif = 1 | (int)((uoff_t))dif >> (sizeof(int)*8));
+   #ifdef CONFIG_LFS
+   dif = 1 | (int)((uoff_t))dif << (sizeof(int)*8));
+   #else
+   dif = 1 | (int)((uoff_t))dif >> (sizeof(int)*8));
+   #endif
```

That is, we changed the right shift (>>) to a left shift (<<) when `CONFIG_LFS` is enabled. The program should behave differently due to this change. Running lifted equivalence checker on the modified benchmark (which we call b-Modified) gives the following NEQ-Summary.

Feature Configurations:

$$(\texttt{CONFIG\_LOCALE\_SUPPORT} \wedge \texttt{CONFIG\_LFS}) \vee (\texttt{CONFIG\_LFS} \wedge \neg\texttt{CONFIG\_LOCALE\_SUPPORT})$$

Counterexamples:

$$(\texttt{sort\_dir} \wedge \neg\texttt{sort\_mtime} \wedge \neg\texttt{sort\_atime} \wedge \neg\texttt{sort\_ctime} \wedge (\texttt{strcoll12} < 0) \wedge \neg\texttt{sort\_size} \wedge \neg\texttt{sort\_reverse}) \wedge (\texttt{sort\_dir} \wedge (\texttt{strcoll12} < 0) \wedge \neg\texttt{sort\_reverse} \wedge (\texttt{strcmp12} < 0) \wedge \neg\texttt{sort\_ctime} \wedge \neg\texttt{sort\_mtime} \wedge \neg\texttt{sort\_atime} \wedge \neg\texttt{sort\_size})$$

The NEQ-summary contains a single entry representing the set of non-equivalent products that are characterized by the feature expressions. Any satisfying input to the counterexample expression would trigger functional differences on these products. As an example, CLEVER-V solved the VEQ problem ($F \equiv F'$) in the case study b-Orig, and produced a NEQ-summary. Using the NEQ-summary, we identified a counterexample to equivalence (shown in Fig. 6). The counterexample triggers the difference between F and F' for products derived by the feature configurations: `CONFIG_LOCALE_SUPPORT` ∧`CONFIG_LFS`.

Answer to RQ1. CLEVER-V is effective at solving VEQ problems. The NEQ-summary for b-Orig is sound and contains all feature configurations of non-equivalent products.

6.3 RQ2: Scalability

To answer RQ2: scalability of our approach compared to the brute-force one as the number of features increases, we systematically generated 288 benchmark instances by first increasing the variability of b-Orig and then using *program mutation* [12] on the modified b-Orig. We referred to the resulting set as B-Hard. B-Hard was generated based on two hypotheses on real-world updates to SPLs: (1) presence conditions are in the form of a conjunction of enabled and disabled features (e.g., `#if !defined FEATURE_A && defined FEATURE_B`), and (2) code changes are small (e.g., replacing + with -, or << with >>).

We first generated 72 pairs of complex SPLs based on hypothesis (1). Each pair of SPLs were generated by adding i additional features $f_1, \cdots, f_i$ to b-Orig (for each $i = 1, \cdots, 12$, we generated six benchmark cases) and injecting randomly-generated, conjunctive feature expressions as presence conditions. Specifically, the randomly generated presence condition $\phi(s)$ of statement s can be written as

$$\phi(s) = \bigwedge_{f_i \in F_e} f_i \wedge \bigwedge_{f_j \in F_d} \neg f_j$$

where $F_e, F_d \subseteq \{f_1, \cdots, f_i\}$ are sets of enabled and disabled features. Each feature f has an equal probability of $p_e = 0.1$ (resp. $p_d = 0.05$) to be selected into F_e (resp. F_d), and $1 - p_e - p_d$ to be unselected. Statements to which we inject presence conditions must be unaffected by commit `6b01b71e`, and the pair of SPLs from the same benchmark case should inject the same presence conditions to the same statements.

For each of the 72 pairs of complex SPLs, we created additional benchmarks by making minor changes to the complex SPLs according to hypothesis (2). Each additional benchmark was created by applying at least one of the two mutation operators we selected. The first mutation operator, $M_{op}(p_{op})$, randomly mutates C-language operations. Each operation (e.g., -, <, ||, >>) has a probability of p_{op} to be replaced by its counterpart (e.g., +, >, ||, <<). The second mutation operator, $M_{pc}(p_{pc})$, randomly mutates presence conditions. Specifically, for any presence condition $\phi(s) = \bigwedge_{f_i \in F_e} f_i \wedge \bigwedge_{f_j \in F_d} \neg f_j$, each f_i in F_e has a probability of p_{pc} to be moved to F_d, and vice versa.

The result of applying one or more mutation operators to the original program is called a *mutant* [12]. For each of the 72 pairs of complex SPLs, we apply $M_{op}(0.1)$, $M_{pc}(0.05)$ or both independently (i.e., with different random seed) on the pair of SPLs to obtain three pairs of mutants. This gives a total of 288 benchmarks including the 72 unmutated benchmarks.

We ran experiments on Ubuntu 22.04 with an Intel® Core™ i7-6700 CPU processor and 16 GiB of RAM. Each case was run with timeout set to 10 minutes, and memory limit set to 16 GB.

In addition to comparing the performance of CLEVER-V against the brute-force approach, we also aim to study the impact of the optimization `GeneralizeFCEXs` (Alg. 2) on CLEVER-V. Therefore, we evaluate performance on the following three configurations of B-Hard : (1) the *brute-force* approach that derives products by all feature configurations in the feature model and then analyzes them separately, (2) the *unoptimized CLEVER-V* that uses CLEVER-V without the optimization `GeneralizeFCEXs`, (3) *CLEVER-V* that uses CLEVER-V with `GeneralizeFCEXs`.

```
st_size1 = 0                st_size2 = 3
st_atim_tv_sec1 = 0         st_atim_tv_sec2 = 3   st_ctim_tv_sec1 = 0
st_ctim_tv_sec2 = 0         st_mtim_tv_sec1 = 0   st_mtim_tv_sec2 = 3
st_mode1 = 0                st_mode2 = 0
strcoll12 = -1              strcmp12 = -1         all_fmt = 0
sort_size = 0               sort_atime = 0        sort_ctime = 1
sort_mtime = 0              sort_dir = 0          sort_reverse = 1
CONFIG_LOCALE_SUPPORT = 0   CONFIG_LFS = 1
```

Figure 6: A concrete counterexample to equivalence for the case-study `b-orig`. `CONFIG_LOCALE_SUPPORT` and `CONFIG_LFS` are feature variables and the others are program variables.

Results. The plot in Fig. 7 shows the running times vs. the number of additional features. The times are displayed in ms. on a logarithmic scale. We plot the *Penalized Quantile Runtime (PQR)* [21] with $p = 0.5$ and $f = 10$ as the performance indicator. PQR is calculated by penalizing timeouts with a factor of f or taking the p-quantile of the successful runs depending on the number of successful runs. This penalizes unsuccessful runs and is resistant to outliers.

Comparing CLEVER-V against the *brute-force* approach, we observed that CLEVER-V significantly outperformed the brute-force approach in all categories. The PQRs (in ms) for CLEVER-V vs. *brute-force* are 581.5 vs. 60666.5 for unmutated; 1087.0 vs. 53916.5 for $M_{pc}(0.5)$; 4513.0 vs. 53785.5 for $M_{op}(0.1)$; 5164.0 vs. 56870.0 for $M_{pc}(0.5) + M_{op}(0.1)$; and 2262.0 vs. 54538.0 over all instances.

Comparing CLEVER-V against *unoptimized CLEVER-V*, we observed that unoptimized CLEVER-V remained competitive on the unmutated benchmarks and the benchmarks mutated by both mutation operators ($M_{pc}(0.5) + M_{op}(0.1)$). The PQRs (in ms) of unoptimized CLEVER-V and CLEVER-V were 601.0 and 581.5, respectively, for unmutated benchmarks, and 9206.0 and 5164.0, for $M_{pc}(0.5) + M_{op}(0.1)$. In other categories, CLEVER-V outperformed unoptimized CLEVER-V as the number of additional features increased. For benchmarks mutated by $M_{pc}(0.05)$, the PQRs (in ms) of unoptimized CLEVER-V and CLEVER-V were 997.5 and 922.5, respectively, with 6 additional features, and 40945.0 and 9142.5, respectively, with 12 features. Similarly, for benchmarks mutated by $M_{pc}(0.1)$, the PQRs of unoptimized CLEVER-V and CLEVER-V were 3208.0 and 2684.0, respectively, with 6 additional features, and 159960.5 and 7458.0, respectively, with 12 additional features. We observed that enabling the optimization `GeneralizeFCEXs` (see Alg. 2) reduces the number of iterations (Lines 5–12 of Alg. 1) required for converging to a complete NEQ-summary. Even though `GeneralizeFCEXs` added an overhead in each iteration, the benefit of faster convergence becomes important given a large number of feature configurations, which significantly improved CLEVER-V's performance.

Answer to RQ2. CLEVER-V's performance scales better than the brute-force approach w.r.t. running times as the number of additional features increases. CLEVER-V scales better than unoptimized CLEVER-V when the change is either in the code or in the presence conditions, while remaining competitive in solving other instances.

6.4 Threats to Validity

Internal Threats. Our benchmark generation threatens internal validity. The choice of parameters p_e, p_d, p_{op}, and p_{pc} (see Sec. 6.3) may bias the effectiveness of FCEX generalization. Since we fixed the choice of parameters when generating all B-Hard benchmarks, the results could be skewed toward this generation pattern. However, as we are more concerned with the scalability as the number of features increases, we presume we can tolerate these biases.

External Threats. The choice of BusyBox as the subject system threatens external validity. Further work should consider expanding subject systems to other SPLs, such as the Linux kernel. Furthermore, all benchmark instances (b-Orig and b-Hard) used in the evaluation satisfy the limitations of CLEVER-V mentioned in Sec. 5, i.e., they do not contain global variables, jumps, or recursive function calls. Therefore, our evaluation needs to be repeated on a broader range of input programs. This is left for future work.

7 Related Work

Functional Equivalence Checking. Zaks et al. [35] reduces equivalence checking to analysis of a self-composition of the two input programs. This work is followed by many others that use self-composition to verify program equivalence. For example, Feng et al. [14] implement an equivalence checker based on the use of conditional model checking to verify self-compositions of extractable sub-CFGs (a.k.a. *impact boundaries*) containing calls to different versions of the library. During self-composition construction, *aligning* is typically used to help in equivalence checking by avoiding summarizing loops separately. Barthe et al. [5] align programs by pairing iterations of loops, which makes an inductive proof easier. Churchill et al. [10] describe an approach to construct product programs driven by semantics as opposed to syntax. Another equivalence checking technique is *differential symbolic execution* (DSE) [26], which is an extension of symbolic execution. In particular, *Shadow Symbolic Execution* (SSE) [9], *Directed incremental Symbolic Execution* (DiSE) [27], and ModDiff [34] are inspired by DSE. Rêve [13] converts the equivalence checking problem into Horn constraints that can be solved by an SMT solver. Finally, ARDiff [4] proposed a counterexample-guided abstraction refinement approach to identify a "simple" slice of the program to reduce the complexity of equivalence analysis.

We lift Feng at al.'s implementation of the self-composition algorithm. However, unlike all previously mentioned tools and techniques, our work is, to the best of our knowledge, the first equivalence checker that supports SPL analysis. Existing equivalence checkers must enumerate all product pairs, the number of which may grow exponentially in the number of features.

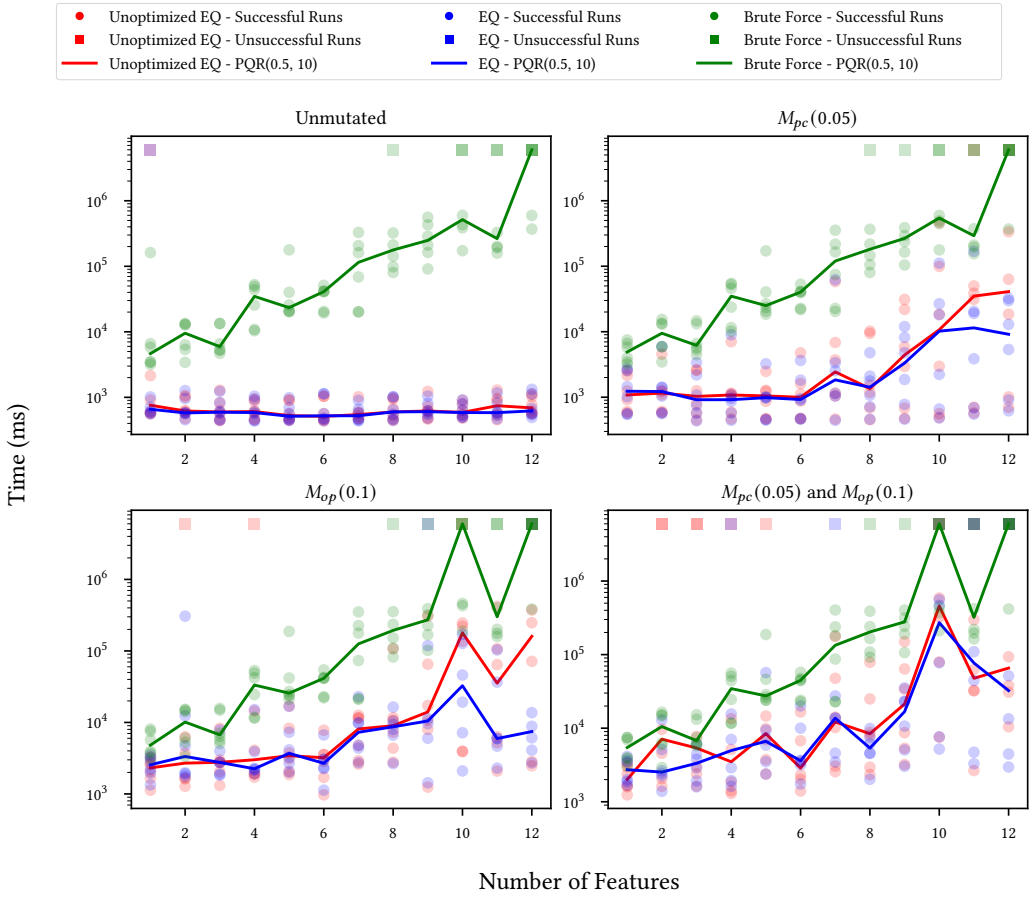


Figure 7: The running times of all 288 benchmarks classified by the mutation operators used.

Product-Line Software Model Checking. The second phase of EQ (see Alg. 1 in Sec. 4.2) can be viewed as product-line software model checking. Several authors propose approaches for software model checking of SPLs written in Java [3, 19, 30] or C [2, 3, 28]. The closest work to ours is [28] as it performs software model checking on annotative SPLs. As in our approach, the authors use variability encoding to convert all products into a metaproduct, then prove it using a verification back-end (i.e., CBMC). The main difference between our verification phase (i.e., Lines 3–13 of Alg. 1) and [28] is that we output an NEQ-summary that is sound and complete — it includes exactly the feature configurations of products with counterexamples, while [28] is only sound — it produces at most one product with counterexamples (if any exist). [28] is much faster because it produces a simpler output and is useful when counterexamples mean "bugs"; developers use counterexamples to fix bugs until the product-line software model checker finds no more counterexamples. Our approach is compliant with the definition of variability-aware lifting by Shahin and Chechik [31], and is more useful when certain products in the SPL are expected to have different behaviour in terms of inputs and outputs due to changes in specifications, and stakeholders are more interested in finding all such products.

8 Conclusion

In this paper, we tackled VEQ— the problem of functional equivalence checking for SPLs. We proposed a lifted equivalence checking algorithm EQ and implemented it in the tool CLEVER-V. We empirically demonstrated CLEVER-V's effectiveness and scalability for change impact analysis on realistic SPL updates.

As a future work, we aim to integrate CLEVER-V with existing equivalence checking techniques such as impact boundary search [14] and iterative abstraction and refinement [4] to improve scalability. We also intend to improve CLEVER-V's applicability by supporting equivalence checking between programs with recursive functions as well as stateful functions. Finally, we aim to expand CLEVER-V to support compositional and delta-oriented SPLs.

References

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

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

[3] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491, 2013.

[4] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 13–24. ACM, 2020.

[5] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *World Congress on Formal Methods*, pages 200–214, 06 2011.

[6] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[8] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004. Software Variability Management.

[9] Cristian Cadar and Hristina Palikareva. Shadow symbolic execution for better testing of evolving software. *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014.

[10] Berkeley R. Churchill, Oded Padon, Rahul Sharma, and Alexander Aiken. Semantic program alignment for equivalence checking. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[11] Krzysztof Czarnecki. Generative programming: Methods, techniques, and applications tutorial abstract. In Cristina Gacek, editor, *Software Reuse: Methods, Techniques, and Tools*, pages 351–352, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[12] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[13] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 349–360, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Nick Feng, Federico Mora, Vincent Hui, and Marsha Chechik. Scaling client-specific equivalence checking via impact boundary search. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 734–745, 2020.

[15] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are np-complete. *Information Processing Letters*, 12(3):133–137, 1981.

[16] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 343–361, Cham, 2015. Springer International Publishing.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[18] Vincent Hui and Nick Feng. Human-guided precondition synthesis. https://github.com/NickF0211/precondition_sys, 2021.

[19] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, page 1–8, New York, NY, USA, 2012. Association for Computing Machinery.

[20] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking #ifdef variability in c. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, page 25–32, New York, NY, USA, 2010. Association for Computing Machinery.

[21] Pascal Kerschke, Jakob Bossek, and Heike Trautmann. Parameterization of state-of-the-art performance indicators: A robustness study based on inexact tsp solvers. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, page 1737–1744, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[23] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 712–717. Springer, 2012.

[24] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. Transfer learning across variants and versions: The case of linux kernel size. *IEEE Transactions on Software Engineering*, 48(11):4274–4290, 2022.

[25] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. Client-specific equivalence checking. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 441–451. ACM, 2018.

[26] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pundefined-sundefinedreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 226–237, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2011.

[28] Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 347–350, 2008.

[29] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.

[30] Ina Schaefer, Dilian Gurov, and Siavash Soleimanifard. Compositional algorithmic verification of software product lines. In *Formal Methods for Components and Objects*, 2010.

[31] Ramy Shahin and Marsha Chechik. Automatic and efficient variability-aware lifting of functional programs. *Proceedings of the ACM on Programming Languages*, 4:1 – 27, 2020.

[32] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Property directed self composition. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 161–179. Springer, 2019.

[33] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1), jun 2014.

[34] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. Modular demand-driven analysis of semantic difference for program versions. In *SAS*, 2017.

[35] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *World Congress on Formal Methods*, 2008.