Proc. FASE 2015, (c) Springer

Facilitating Reuse in Multi-Goal Test-Suite Generation for Software Product Lines

Johannes Bürdek[1], Malte Lochau[1], Stefan Bauregger[1], Andreas Holzer[2],
Alexander von Rhein[3], Sven Apel[3], and Dirk Beyer[3]

[1] TU Darmstadt, Germany
[2] TU Wien, Austria
[3] University of Passau, Germany

Abstract. Software testing is still the most established and scalable quality-assurance technique in practice. However, generating effective test suites remains computationally expensive, consisting of repetitive reachability analyses for multiple test goals according to a coverage criterion. This situation is even worse when testing entire software product lines, i.e., families of similar program variants, requiring a sufficient coverage of all derivable program variants. Instead of considering every product variant one-by-one, family-based approaches are variability-aware analysis techniques in that they systematically explore similarities among the different variants. Based on this principle, we present a novel approach for automated product-line test-suite generation incorporating extensive reuse of reachability information among test cases derived for different test goals and/or program variants. We present a tool implementation on top of CPA/tiger which is based on CPAchecker, and provide evaluation results obtained from various experiments, revealing a considerable increase in efficiency compared to existing techniques.

Keywords: Software Product Lines, Automated Test Generation, Symbolic Model Checking, CPAchecker, CPA/tiger

1 Introduction

Software-product-line engineering [15] has become a key technology to cope with the variability of highly-configurable (software) systems, prevalent in various application domains today. In recent years, software product lines (SPL) have found their way into numerous industrial application domains, e.g., automotive, information and mobile systems [29]. SPL engineering aims at developing a family of similar, yet well-distinguished software products based on a common core platform, where commonality and variability among the family members (product variants) are explicitly specified in terms of *features*. Each feature corresponds to (1) user-visible product characteristics in the problem domain, relevant for product configuration, as well as to (2) composable implementation artifacts for (automated) assembling of implementation variants. This philosophy of extensive *reuse* of common feature artifacts among product variants facilitates a remarkable gain in efficiency compared to one-by-one variant development [15].

For SPLs to obtain full acceptance in practice, established quality-assurance techniques have to be lifted to become *variability-aware* as well, to also benefit from reuse principles [24, 26, 27]. Various promising attempts have been proposed, enhancing respective model-checking and software-testing techniques to efficiently verify entire families of software products instead of every single variant [4, 12, 14, 16]. In practice, systematic software testing remains the most established and elaborated quality-assurance technique, as it is directly applicable to real-world applications at any level of abstraction [28]. Furthermore, testing allows for a realistic and controllable trade-off between effectiveness and efficiency. In particular, white-box test generation aims at (automatically) deriving sample inputs for a given program under test to meet certain *test goals*. Thereupon, the derivation of an entire *test suite* is usually guided by test-selection measures by means of coverage criteria, e.g., *basic-block coverage* and *condition coverage* [8]. Coverage criteria impose multiple, arbitrarily complex test goals, thus requiring *sets* of test-input vectors to achieve a complete coverage [8]. Depending on the concrete application domain and the respective level of mission-/safety-criticality, it is imperative, or even enforced by industrial standards, to guarantee a certain degree of code coverage for every delivered product [11]. The computational problem underlying automated test generation consists of expensive reachability analyses of the program state space. Symbolic model checking has emerged in the past as a promising approach for fully automated white-box test generation using counterexamples as test inputs [7]. However, in case of large sets of complex test goals, scalability issues still hinder efficient test-case generation when being performed for every test goal anew. This is even worse when generating test inputs for covering entire product-line implementations. To avoid product-by-product (re-)generation of test cases with many redundant generation runs, a family-based test-generation approach must enhance test-suite-derivation techniques to be likewise applicable to product-line implementations [12, 24].

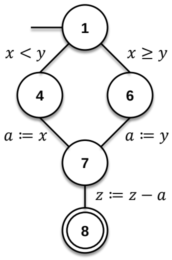
In this paper, we present a novel approach for efficient white-box test-suite generation that guarantees multi-goal test coverage of product-line implementations. The approach systematically exploits reuse potential among reachability-analysis results by means of similarity among test cases

- derived for different test goals [8], and/or
- derived for different product variants [12].

The interleaving of both techniques leads to an incremental, coverage-driven exploration of the state space of the product line under test implemented in C enriched with feature parameters [19]. We implemented an SPL test-suite generator for arbitrary coverage criteria on top of CPA/TIGER which is based on CPACHECKER [9], a symbolic model checker using the CEGAR approach [13]. In our evaluation, we consider sample product-line implementations of varying sizes to investigate the general applicability of the novel test-suite-derivation approach, as well as the gain in efficiency obtained from the reuse of reachability analyis results. The results reveal a considerable improvement in efficiency compared to test-suite-generation approaches without systematic reuse.

```
1 int func(int x, int y, int z) {
2   int a;
3   if (x < y)
4     a = x;
5   else
6     a = y;
7   z = z - a;
8   return z;
9 }
```

(a) Sample C-Code



(b) CFA

Fig. 1. C Code and CFA Representation of a Program Under Test

2 Background

In this section, we give a brief overview on the general principles and notions of white-box test-case derivation and its application to software product lines.

2.1 White-Box Test-Suite Derivation

Consider the C code snippet in Fig. 1(a) implementing a simple function `func` that takes three integer parameters, `x`, `y` and `z`, subtracts the smaller value of `x` and `y` from `z`, and returns the result. The program operates on a set $V = \{x, y, z, a\}$ of typed program variables, where the subset $\{x, y, z\} \subseteq V$ constitutes *input variables* of the program. The purpose of test-case derivation is to systematically derive sample vectors of concrete *input values*, one for each input variable, to stimulate and therefore investigate a particular behavior of the program under test. For example, executing the given sample program with the *input vector* $[x = 1, y = 2, z = 3]$ enforces a test run that traverses the `if` branch in Lines 3 – 4. In addition to the input vector, a test case often further consists of a *test oracle*, i.e., a specification of the intended outcome of running the test, e.g., by means of the expected return value of a function in case of unit testing. Hence, a *test case* consists of a pair of an input vector and a test oracle, e.g., $([x = 1, y = 2, z = 3], [z = 2])$. To properly test a program, typically a whole *set* of test cases is required. However, due to the large and even (theoretically) unbounded size of input-value domains (integer in this example), exhaustive testing is, in general, infeasible, or even impossible. To guide the selection of appropriate subsets of input vectors into a *test suite*, various adequacy criteria have been proposed [28]. Concerning white-box testing, in particular, *code-coverage measures* refer to syntactic program elements as multiple *test goals* to be covered by a test suite. For this purpose, an abstract representation of the program under test is considered, e.g., a control-flow automaton (CFA) [8], as shown in Fig. 1(b). Nodes of the CFA refer to program locations (program-counter valuations), and edges denote control transfers between locations according to the syntactic structure of the program. Edges referring to (blocks of) operations are labeled with respective value assignments on program variables, whereas edges referring control-flow branches are labeled with corresponding predicates (assumptions) over program-variable values. Covering a test goal on a CFA means to find an input vector that traverses a CFA path that matches this goal. For instance, applying *statement coverage* to the CFA of Fig. 1(b), the statements in Lines 4, 6, and 7 each constitute a test goal, where the aforementioned test case ($[x = 1, y = 2, z = 3], [z = 2]$) covers the goals 4 and 7. To obtain complete statement coverage, a further test case, e.g., ($[x = 2, y = 1, z = 3], [z = 2]$), is required to cover the `else` branch (Lines 5 – 6). Hence, both test-input vectors in combination constitute a *complete* test suite for statement coverage.

Statement coverage is one of the weakest code-coverage criteria, solely ensuring a small subset of the reachable program state space to be actually explored. In the context of safety-critical systems, coverage criteria such as MC/DC [11] comprise more complex test-goal specifications including sets of pairs (ℓ, φ) of program locations ℓ together with predicates φ over program variables in V. Coverage criteria not only define quality measures for existing test suites, but also serve as test-end criteria during test-suite generation. Therefore, a test-suite generator iterates over the set of test goals and derives a satisfying test case for each test goal. For example, statement coverage selects the program locations 4, 6, 7 as test goals and starts with location 7, which is trivially reached by any input vector, e.g., $[x = 1, y = 2, z = 3]$. In addition, location 4 is also already covered by this test case. Thus, results of test-case generation may be *reused* among test goals to reduce test-generation/execution efforts.

2.2 Test-Suite Derivation for Product-Line Implementations

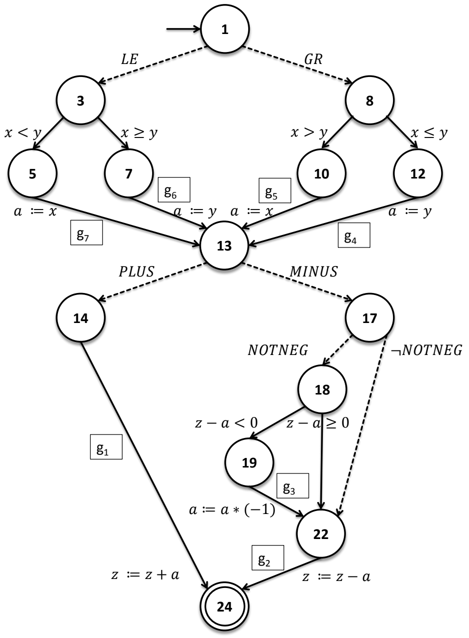
The program in Fig. 1 implements a fixed functionality without any behavioral variability, as apparent, e.g., in software product lines [15]. In a product line, features are related to dedicated artifacts within the solution space. These artifacts are composable into product variants derivable for a given configuration, thus facilitating systematic reuse of artifacts among product-family members at any level of abstraction [1]. At source-code level, features may occur as implementation parameters annotating dedicated pieces of code as feature-specific implementation variability, e.g., using C preprocessor [19].

Consider the product-line implementation `func-spl` in Fig. 2(a) (line breaks are added for better readability), which extends the example from Fig. 1(a) by variability. Therein, `#ifdef` conditions over (Boolean) feature parameters annotate those code pieces being conditionally compiled into a variant implementation depending on the feature selection. In the example of Fig. 2, feature parameters control that, from `x` and `y`, either the smaller (LE) or the greater value (GR) is either added (PLUS) or subtracted (MINUS) from `z`. In the case of subtraction, feature NOTNEG ensures the result not to be negative for positve input values.

The set of features, together with constraints limiting their possible combinations, are usually captured in a *variability model*. Here, we limit our attention to a representation in terms of propositional formulae over Boolean feature vari-

```
1  int func-spl(int x, int y,
           int z) {
2      int a;
3      #ifdef LE
4      if (x < y)
5        a = x;
6      else
7        a = y;
8      #elsif GR
9      if (x > y)
10       a = x;
11     else
12       a = y;
13     #endif
14     #ifdef PLUS
15     z = z + a;
16     #elsif MINUS
17     #ifdef NOTNEG {
18     if ((z - a) < 0)
19       a = a * (-1);
20     }
21     #endif
22     z = z - a;
23     #endif
24     return z;
25 }
```

(a) Sample Product-Line C-Code



(b) CFA

Fig. 2. Parameterized C-Code and CFA Representation of a Product-Line Under Test

ables [5]. For our example in Fig. 2(a), the variability model states that

$$(\mathsf{LE} \veebar \mathsf{GR}) \wedge (\mathsf{PLUS} \veebar \mathsf{MINUS}) \wedge (\mathsf{NOTNEG} \rightarrow \mathsf{MINUS})$$

holds, where $\veebar$ denotes exclusive or. The features LE and GR, as well as PLUS and MINUS denote alternative choices, whereas NOTNEG is an optional feature, only selectable in combination with MINUS. Thus, there are 6 valid configurations.

The inherent variability of product-line implementations has to be taken into account also during test-case derivation [12]. The enhanced CFA representation for the example of Fig. 2 is depicted in Fig. 2(b), where each *variation point* is represented as an additional (dashed) branch edge labeled over respective feature constraints. Hence, actual test runs, i.e., the code parts traversed and the final results computed for a test case applied to a product-line implementation, depend on the configuration of the product variant under test. For instance, the input vector $[x = 1, y = 2, z = 3]$ traverses the `if` branch (Lines 4 – 5) only if feature LE is selected, and the `else` branch (Lines 11 – 12) if feature GR is selected. Next, either Line 15 is executed for feature PLUS, or Line 22 for feature MINUS. In the latter case, the execution of Line 19 for feature NOTNEG further depends on whether LE or GR is selected, whereas the `if` branch (Lines 18 – 19) is not covered and Line 19 is skipped in both cases. Finally, the expected value returned in Line 24, again, depends on the selected feature combination, i.e., $[z = 4]$ for LE and PLUS, $[z = 2]$ for LE and MINUS, $[z = 5]$ for GR and PLUS, and $[z = 1]$ for GR and MINUS, whereas feature NOTNEG does not affect the outcome of the test run.

To cope with behavioral variability, the derivation of test cases must be *variability-aware*. Therefore, a test-case specification is extended to a triple, e.g., $([x = 1, y = 2, z = 3], [z = 2], \phi)$, where ϕ denotes the *presence condition*, i.e., a propositional formula over feature parameters constraining the set of configurations for which this test case is *valid*. In the example, $\phi = \mathsf{LE} \wedge \mathsf{MINUS}$ holds, i.e., this test case is (re-)usable for configurations with and without feature NOTNEG. In the same way, the notion of code coverage must be adapted to product-line implementations. For instance, concerning location 19, the input vector $[x = 1, y = 3, z = 2]$ covers this location on configurations with GR, MINUS, and NOTNEG being selected, whereas in program variants with feature LE instead of GR, location 19 remains uncovered. In contrast, input vector $[x = 3, y = 2, z = 1]$ covers location 19 on both program variants, and, together with input vector $[x = 2, y = 3, z = 1]$, the resulting test suite achieves complete statement coverage for the entire product line.

To summarize, a systematic approach for variability-aware, yet efficient test-suite derivation for complete product-line coverage has to take both dimensions of reuse into account: among test goals as well as among variants.

3 Test-Suite Generation for Product Lines

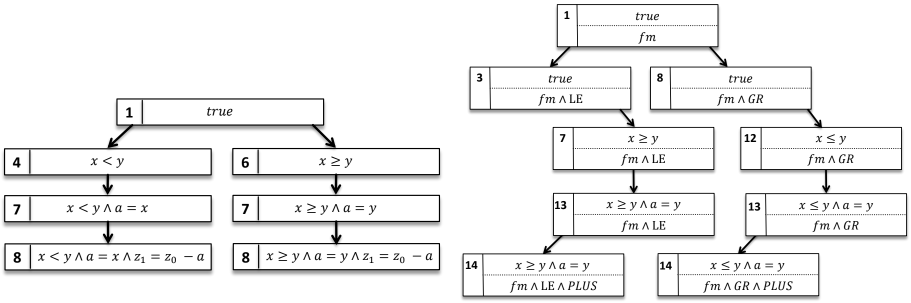
In this section, we introduce an approach for white-box test-suite generation for covering entire product-line implementations based on symbolic model checking. The approach incorporates systematic reuse of reachability-analysis results among different test goals as well as different product variants.

3.1 Test-Case Generation based on Symbolic Model Checking

The derivation of test inputs for covering a particular test goal requires a *reachability analysis* of the state space of the program under test. Model checking has been extensively investigated as a viable technique for automating the evaluation of respective *reachability queries*. The basic idea of those approaches is to pass the (negated) test goal to the model checker to derive a counterexample serving as test input for covering that goal [7].

Considering white-box testing in particular, symbolic model checking has recently shown promising performance improvements when applied to derive test inputs for a given program under test [8]. Symbolic test-input derivation for a test goal consisting of a pair (ℓ, φ) is performed in two interleaved steps.

1. Find a path through the CFA of the program under test to location ℓ.
2. Derive input values satisfying the *path condition* for ℓ and state predicate φ.


1 $true$
4 $x < y$
6 $x \geq y$
7 $x < y \wedge a = x$
7 $x \geq y \wedge a = y$
8 $x < y \wedge a = x \wedge z_1 = z_0 - a$
8 $x \geq y \wedge a = y \wedge z_1 = z_0 - a$
1 $true$ fm
3 $true$ $fm \wedge \mathrm{LE}$
8 $true$ $fm \wedge GR$
7 $x \geq y$ $fm \wedge \mathrm{LE}$
12 $x \leq y$ $fm \wedge GR$
13 $x \geq y \wedge a = y$ $fm \wedge \mathrm{LE}$
13 $x \leq y \wedge a = y$ $fm \wedge GR$
14 $x \geq y \wedge a = y$ $fm \wedge \mathrm{LE} \wedge PLUS$
14 $x \leq y \wedge a = y$ $fm \wedge GR \wedge PLUS$

(a) ARG for func

(b) Partial ARG for func-spl

Fig. 3. ARG for C Program and Product-Line Implementation

For Step 2, the model checker uses an abstract reachability graph (ARG) as central data structure to symbolically represent the reachable program state space [7]. Based on the approach of lazy abstraction [6], the ARG is iteratively constructed and refined for any possible CFA path reaching ℓ, until either a satisfying variable assignment has been found, or the path conditions of all possible paths leading to (ℓ, φ) turn out to be unsatisfiable.

The fully-explored ARG for the program in Fig. 1(a) is depicted in Fig. 3(a). Each ARG path refers to some path of the CFA in Fig. 1(b), where the accumulated state predicates over program variables denote the path condition to be satisfied by input values to reach that location in a concrete execution. Fresh temporal variables, e.g., z_0, z_1, are introduced using SSA encoding. For instance, for the ARG path $1, 4, 7, 8$, derived for test goal $(\ell = 8, \varphi = \langle x < y \rangle)$, there is an initial variable assignment satisfying the respective path condition as well as φ, whereas, for path $1, 6, 7, 8$, no such assignment exists.

Formally, the set V of program variables is typed over a compound domain $\mathcal{D}$, where we limit our considerations to $\mathcal{D} = \mathbb{B} \cup \mathbb{Z}$, i.e,. Boolean and Integer variables. Thereupon, the labeling alphabet of a CFA is given as $\mathcal{L}(V) = \mathcal{L}_{assume}(V) \cup \mathcal{L}_{assign}(V)$, where $\mathcal{L}_{assume}(V) = (V \to \mathcal{D}) \to \mathbb{B}$ comprises the sub-alphabet of *predicates* over V, and $\mathcal{L}_{assign}(V) = (V \to \mathcal{D}) \to (V \to \mathcal{D})$ denotes the sub-alphabet of (blocks) of *value assignments* on V. We refer to elements of the labeling alphabet as $a \in \mathcal{L}(V)$.

Definition 1 (Control-Flow Automaton). *A* CFA *is a triple* (L, ℓ_0, E)*, where* L *is a finite set of* control locations*,* $\ell_0 \in L$ *is the* initial control location*, and* $E \subseteq (L \times \mathcal{L}(V) \times L)$ *is a* labeled transition relation*.*

For a CFA to be well-structured, we require each node except termination nodes to have either exactly one outgoing edge labeled over $\mathcal{L}_{assign}(V)$, or all outgoing edges labeled with mutually excluding predicates from $\mathcal{L}_{assume}(V)$. A *concrete execution* of a program traverses some path of the corresponding CFA, denoted

$$\ell_0 \xrightarrow{a_0} \ell_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} \ell_k,$$

where $(\ell_i, a_i, \ell_{i+1}) \in E$, $0 \leq i < k$, holds. Semantically, a concrete execution corresponds to a sequence

$$\pi = (\ell_0, c_0) \xrightarrow{a_0} (\ell_1, c_1) \xrightarrow{a_1} \cdots \xrightarrow{a_{k-1}} (\ell_k, c_k)$$

of *program states*, i.e., pairs (ℓ_i, c_i), $0 \leq i < k$, of locations $\ell_i \in L$ and current values of program variables given by a mapping $c_i : V \to \mathcal{D}$, such that either

- $c_{i+1} = a_i(c_i)$ if $a_i \in \mathcal{L}_{assign}(V)$, or
- $c_{i+1} = c_i$ and $c_i \models a_i$, if $a_i \in \mathcal{L}_{assume}(V)$

holds. Based on this notion of program-execution semantics, input-variable assignments of an initial program state c_0 constitute a (concrete) test-case-satisfying test goal (ℓ, φ) iff there is a concrete program execution reaching some state (ℓ, c_k) with $c_k \models \varphi$. Stepping from concrete executions to symbolic-execution semantics is done by predicate representation of (sets of) concrete data values in program states: a set of possible program-variable assignments, called a *region*, in a program location is symbolically characterized by a *state predicate* $r \in \mathcal{L}_{assume}(V)$, where $R(V)$ denotes the set of all regions over V. For constructing symbolic executions, the *strongest postcondition* operator $sp_V : (R(V) \times \mathcal{L}(V)) \to R(V)$ is defined such that, for $r' = sp_V(r, a_i)$ and for each $c_i \models r$, either

- $c_{i+1} = a_i(c_i)$ and $c_{i+1} \models r'$, if $a_i \in \mathcal{L}_{assign}(V)$, or
- $c_i \wedge a_i \models r'$, if $a_i \in \mathcal{L}_{assume}(V)$

holds. The symbolic representation of regions as predicates supports *abstraction*, i.e., omitting information about the state space that is not necessary for the analysis: the *abstract states* might *over-approximate* sets of concrete states. Based on these notions, an ARG is defined as follows.

Definition 2 (Abstract Reachability Graph). *An* ARG *of a CFA* (L, ℓ_0, E) *is a triple* (S, s_0, T)*, where* $S \subseteq (L \times R(V))$ *is a set of* abstract program states*,* $s_0 = (l_0, true) \in S$ *is the* initial abstract program state*, and* $T \subseteq (S \times E \times S)$ *is a* labeled transition relation*, where* $(\ell, r) \xrightarrow{e} (\ell', r') \in T$ *if* $e = (\ell, a, \ell') \in E$ *and* $r' \models sp_V(r, a)$*.*

An ARG is *complete*, if it contains, for every concrete execution π of a program, a corresponding abstract path

$$(\ell_0, r_0) \xrightarrow{e_0} (\ell_1, r_1) \xrightarrow{e_1} \cdots \xrightarrow{e_{k-1}} (\ell_k, r_k),$$

with $c_i \models r_i$, $0 \leq i \leq k$. For deriving a test case that covers goal (ℓ, φ), it is often sufficient to only partially explore the ARG until an abstract state (ℓ, r) with $r \models \varphi$ is found, for which an initial program state c_0 with $c_0 \models r_0$ exists. Although being much more efficient than explicit state-space exploration in many cases, this process has to be repeated until every reachable test goal is covered by, at least, one input vector.

Symbolic test-case generation is applicable to derive test cases from product-line implementations as shown in Fig. 2. Formally, a product line enhances a single program implementation by adding a set of distinguished Boolean (read-only) input variables $V_F \to \mathbb{B}$ to the set of program variables V_P, thus leading to $V = V_P \cup V_F$. Based on the concept of *variability encoding* [25], preprocessor-based, compile-time variability can be transformed into run-time variability using plain `if-then-else` statements over feature variables in a fully-automated way [19]. Similarly, the feature model can be represented as propositional formula $FM \in \mathcal{L}_{assume}(V_F)$ over V_F [5]. Correspondingly, (partial and complete) product configurations are denoted as $\gamma \in \mathcal{L}_{assume}(V_F)$, where $\gamma \models FM$. Based on this encoding, the derivation of a test case satisfying a test goal (ℓ, φ) can be performed as described before. Starting with feature model FM as initial path condition, ARG regions $r = (pc \wedge \phi) \in R(V_P \cup V_F)$ now is a conjunction of two separated parts: (1) a *path condition* $pc \in \mathcal{L}_{assume}(V_P)$ over program variables V_P restricting the inputs and (2) a *presence condition* $\phi \in \mathcal{L}_{assume}(V_F)$ over feature variables V_F restricting the feature selections for which the test case is valid.

The partially explored ARG for test goal $(\ell = 14, true)$ in the CFA of Fig. 2(b) is shown in Fig. 3(b). The path condition and the presence condition are separated by a dashed line, for better readability. Path $1, 3, 7, 13, 14$ suffices to cover the goal on complete configurations with feature LE, whereas for configurations with feature GR, the additional path $1, 8, 12, 13, 14$ is required. As location 14 is only reachable in variants with feature PLUS, this ARG is sufficient to derive inputs for covering the test goal on every variant in which that goal is reachable.

Definition 3 (Complete Product-Line Test Suite). *Let G be a set of test goals on a product-line implementation. A set C_{TS} of initial program states is a* complete product-line test suite*, if for each $g = (\ell, \varphi) \in G$ and for each $\gamma \in \mathcal{L}_{assume}(V_F)$ with $\gamma \models FM$ it holds that, if $(\ell_0, \gamma) \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} (\ell, r_k)$ with $r_k \models \varphi$ exists, then there exists $c_0 \in C_{TS}$ with $c_0 \models \gamma$ and c_0 covering g.*

To achieve complete coverage of product-line implementations, test-case derivation has to be repeated for each test goal (ℓ, φ) until it is covered by, at least, one valid test case on each program variant in which the test goal is reachable.

3.2 Reuse of Reachability-Analysis Results

We now describe an efficient approach for deriving a test suite for multi-goal coverage of product lines. The approach incorporates systematic reuse of reachability information that is already obtained from a (partially explored) ARG during test-case derivation for other test goals and program variants. The overall approach consists of two phases: (1) incremental ARG exploration until sufficient reachability information is obtained to cover each test goal on every program variant in which it is reachable; (2) derivation of concrete test-input data from ARG path conditions for reachability counterexamples. In the first phase, ARG exploration is guided by repetitively traversing paths of the CFA until every test

Algorithm 1 Abstract Product-Line Test-Suite Derivation

```
Input: CFA $(L, \ell_0, E)$, Feature Model $FM$, Test Goals $G = \{g_1, g_2, \ldots, g_n\}$
Output: Abstract Test Suite $TS$
 1: $TS := \{\}$; for all $g \in G : CS[\,g\,] := FM$
 2: for all $g = (\ell, \varphi) \in G$ do
 3:     $r_0 := CS[\,g\,]$
 4:     if $(\ell_0, r_0) \xrightarrow{e_0} \cdots \xrightarrow{e_{k-1}} (\ell_k, r_k)$ with $\ell = \ell_k, r_k = (pc_k, \phi_k)$, and $pc_k \models \varphi$ then
 5:         for all $g' = (\ell_i, \varphi') \in G$ with $0 \leq i \leq k$, $r_i = (pc_i, \phi_i)$, and $pc_i \models \varphi'$ do
 6:             if $\nexists \phi_j \in \mathcal{L}_{assume}(V_F) : TS[\,g', \phi_j\,] \neq undef \wedge \phi_i \models \phi_j$ then
 7:                 $TS[\,g', \phi_i\,] := pc_i$; $CS[\,g'\,] := CS[\,g'\,] \wedge \neg\phi_i$
 8:             end if
 9:         end for
10:     else
11:         $G := G \setminus \{g\}$
12:     end if
13: end for
```

goal $g' = (\ell_i, \varphi') \in G$ from a set G of test goals is covered. The corresponding region r_i of the ARG state (ℓ_i, r_i) that is reached via the CFA path then contains the path condition pc_i, as well as the presence condition ϕ_i for concrete test input data.

The general procedure for the first phase is outlined in Alg. 1. The algorithm incorporates two reuse strategies for reachability-analysis results during ARG exploration: (1) reuse of reachability information for multiple product variants and (2) reuse of reachability information for multiple test goals. The algorithm operates on the CFA of the product-line implementation, the corresponding feature model *FM*, and a finite set G of test goals of the form $g = (\ell, \varphi)$. The data structure *TS* (test suite), holding the result of the algorithm, maps test goals g together with presence conditions ϕ onto path conditions *pc*, such that each concrete input vector satisfying *pc* covers g on every program variant whose configuration satisfies ϕ. In addition, the map *CS* (cover set) assigns to every test goal g a predicate over feature variables, denoting the subset of product configurations on which g is *not yet* covered. The cover set of every test goal $g \in G$ is initialized with *FM* (Line 1), thus requiring a reachability analysis of g for every valid product configuration. The algorithm iterates over the set of not yet completely processed test goals from G (Lines 2 – 13), and incrementally refines the ARG, correspondingly. To consider test goal $g \in G$, reachability analysis is initialized with region $r_0 = CS[\,g\,]$ (Line 3), thus restricting the search to the subset of product configurations on which g is not yet covered by a previous ARG refinement. If a further ARG refinement succeeds to obtain a feasible ARG path reaching g on configurations from the remaining product subset (Line 4), then *TS* is updated (Line 5 – 9) and otherwise, g is removed from the working set G (Line 11). In the first case, the ARG path potentially not only covers g, but may be also (re-)used to (partially) cover further test goals in G. In particular, every test goal $g' \in G$ (including g) whose location ℓ_i lies on the ARG path reaching g

TG	**TC**	**CFA Path**	**Presence Condition**
g_1	tc_1	1,3,5,13,14,24	LE ∧ PLUS
g_7	tc_1'	1,3,5,13	LE
g_1	tc_2	1,8,10,13,14,24	GR ∧ PLUS
g_5	tc_2'	1,8,10,13	GR
g_2	tc_3	1,8,12,13,17,18,19,22,24	GR ∧ MINUS ∧ NOTNEG
g_3	tc_3'	1,8,12,13,17,18,19,22	GR ∧ MINUS ∧ NOTNEG
g_4	tc_3''	1,8,12,13	GR
g_2	tc_4	1,3,7,13,17,18,19,22,24	LE ∧ MINUS ∧ NOTNEG
g_3	tc_4'	1,3,7,13,17,18,19,22	LE ∧ MINUS ∧ NOTNEG
g_6	tc_4''	1,3,7,13	LE
g_2	tc_5	1,3,7,13,17,22,24	LE ∧ MINUS
g_2	tc_6	1,8,10,13,17,22,24	GR ∧ MINUS

(a) Test-Case Derivation for func-spl

	$\mathbf{g}_1$	$\mathbf{g}_2$	$\mathbf{g}_3$	$\mathbf{g}_4$	$\mathbf{g}_5$	$\mathbf{g}_6$	$\mathbf{g}_7$
P$_1$	tc_1	–	–	–	–	tc_4''	tc_1'
P$_2$	–	tc_5	–	–	–	tc_4''	tc_1'
P$_3$	tc_2	–	–	tc_3''	tc_2'	–	–
P$_4$	–	tc_6	–	tc_3''	tc_2'	–	–
P$_5$	–	tc_4	tc_4''	–	–	tc_4''	tc_1'
P$_6$	–	tc_3	tc_3'	tc_3''	tc_2'	–	–

P_1 : LE, PLUS
P_2 : LE, MINUS
P_3 : GR, PLUS
P_4 : GR, MINUS
P_5 : LE, MINUS, NOTNEG
P_6 : GR, MINUS, NOTNEG

(b) Test-Goal Coverage for func-spl

Fig. 4. Sample SPL Test-Suite Derivation

and whose predicate φ' is satisfied by the path condition pc_i of the respective ARG state may be added to TS (Line 5). If g' is already covered by a previously obtained ARG path with presence condition ϕ_j, it is added to TS only if ϕ_i is not subsumed by any those existing ϕ_j in TS, i.e., $\phi_i \not\models \phi_j$ (Line 6). Thus, the corresponding path condition pc_i is added for g' and ϕ_i to TS and the cover set of g' is further restricted, accordingly (Line 7).

After termination, TS contains an *abstract*, *symbolic* product-line test suite in the sense that it provides enough reachability information to derive in the second phase a *concrete* test suite that completely covers the entire product-line implementation. In particular, for each pair (g, ϕ) with $TS[g, \phi] = pc$, a concrete test suite C_{TS} contains an initial program state c_0 with $c_0 \models pc$ as test input being applicable to all program variants whose configuration complies ϕ. The following result essentially relies on the soundness of ARG refinement (Line 4).

Theorem 1. *A concrete test suite C_{TS} obtained from the abstract test suite TS generated by Algorithm 1 is a* complete product-line test suite *(cf. Def. 3).*

A sample application of Alg. 1 to the example from Fig. 2 is illustrated in Fig. 4, considering statement coverage. The resulting set of test goals $G = \{g_1, \ldots, g_7\}$ are attached (in boxes) to the CFA edges in Fig. 2(b). Fig. 4(a) lists the iterations over G, where each horizontal line marks the next iteration of the outer loop. Starting with test goal g_1, the corresponding ARG path obtained for test case tc_1 traverses the CFA such that the resulting path condition requires the features LE and PLUS. In addition, test case tc_1 is reusable as test case tc_1' covering g_7. The presence condition of tc_1' is, therefore, weakened to LE, as the predicate requiring PLUS does not occur in the respective sub-path. A further ARG exploration is required to finally cover g_1 also on those configurations with GR instead of LE selected. After termination, six abstract test cases have been generated explicitly to cover all products $P_1 - P_6$, as summarized in the table in Fig. 4(b). In contrast, considering every single test goal on every program variant anew without any information reuse would have required at least 20 reachability-analysis steps.

4 Evaluation

To investigate the effects of the two proposed reuse strategies, we compare both of them to a product-by-product (PbP) product-line test-suite generation approach. For quantifying the effects of reuse among product variants, we apply a family-based (FB), i.e., variability-aware, generation approach, whereas for the reuse among test goals (TG), we apply a PbP approach. Finally, we combine both reuse strategies (FBTG) to measure potential synergies. The two product-by-product approaches are regarded for comparison purposes. These considerations led us to the following research questions.

- **RQ1:** Do the presented reuse strategies improve the *efficiency* of product-line test-suite generation in terms of CPU time and model-checker calls?
- **RQ2:** Does a family-based approach decrease testing *effectiveness* by leaving more test goals uncovered than a product-by-product approach due to the presumably more complex control-flow of product-line implementations after variability encoding?

To answer the research questions, we implemented a product-line test-suite generator based on CPA/TIGER[4], a test-case generator on top of CPACHECKER[5], which is a software model checker for C programs. Internally, CPACHECKER uses CFA and ARG representations and applies the CEGAR approach, as described in Sect. 3. Furthermore, we use FQL[6] (FShell Query Language), which is part of CPA/TIGER, to specify complex coverage criteria. For a variability-aware generation of product-line test suites, we integrated a BDD-based feature-parameter analysis into CPA/TIGER and added further code to determine which test goals of which variants are covered by a given test case. For our experiments, we selected two subject systems that are well-known from several benchmarks, e.g., in context of product-line verification and feature interaction detection [3].

- *Mine-Pump (MP)* implements a water pump system based on the CONIC project [20]. The product-line implementation has 279 LoC, 7 features, e.g., a methane sensor, and 64 configurations.
- *E-Mail (EM)* is based on a model of an e-mail system developed by Hall [18]. The product-line implementation we used has 233 LoC, 4 features, e.g., encryption and automatic forwarding, and 8 configurations.

As coverage criterion, we applied *Basic-Block Coverage*, i.e., each basic statement block constitutes a test goal. Our measurements, e.g., the CPU time and the number of model-checker calls, were obtained by hooking into the test-case generator. We performed our evaluation on a E5-2650 (2 GHz) machine with 30 GB of RAM. CPA/TIGER ran with value-based analysis, an overall timeout of 24 h, and a timeout per test goal of 900 s, i.e., if a test goal had not been reached after 900 s, the next test goal was processed. Table 1 summarizes the re-

[4] http://forsyte.at/software/cpatiger/
[5] http://cpachecker.sosy-lab.org/
[6] http://forsyte.at/software/fshell/fql/

		#TG	**processed**	**feasible**	**infeasible**	**timeout**	**#TC**	**#CPA**	**t (in h)**
MP	PbP	3792	3792	3298	490	4	3298	3720	13.9
	TG	3792	3792	3300	490	2	427	847	3.7
	FB	93	93	42	1	50	52	145	24
	FBTG	93	81	42	1	38	46	127	24
EM	PbP	1084	1084	804	280	0	800	1874	32.9
	TG	1084	1084	804	280	0	169	533	10.4
	FB	198	131	78	16	37	154	285	24
	FBTG	198	160	91	30	39	103	224	24

Table 1. Overview of Evaluation Results (#TG: Test Goals, #TC: Test Cases, #CPA: Model-checker Calls, t: CPU Time)

sults of our experiments. Compared to PbP, we observe the following effects: (1) the number of model-checker calls and generated test cases have been reduced by both reuse strategies, (2) TG led to high time savings, and (3) FB led to more timeouts. For test goals not reached by FB within 24 h, we extrapolated the results.

Considering **RQ1**, both reuse strategies are able to drastically reduce the number of test cases to be generated, as well as the generation time in terms of model-checker calls. However, a decrease of CPU time was only observable for TG, whereas FB led to increased CPU time compared to PbP due to the additional control-flow for variability encoding. The number of model-checker calls and generated test cases has been further reduced when using both reuse strategies in combination (FBTG), compared to applying them independently. Furthermore, we obtained no valuable CPU time results for the FB and the FBTG approaches due to timeouts after 24 h. Finally, concerning **RQ2**, we observed that more test goals stayed uncovered using FB/FBTG than using PbP/TG since more CPU time is required, which leads to more timeouts. This might be avoided by increasing the overall timeout, as well as the test-goal timeout.

5 Related Work

In previous work, the problem of generating test suites for product-line coverage has been investigated in the context of model-based (black-box) testing [12]. There, the explicit-state model checker SPIN has been used for reachability analysis, with a posteriori reasoning about test-case reuse among product variants, but without any reuse of reachability results among test goals. Applying CPAchecker for product-line verification has been proposed [3], incorporating BDD analysis for reuse of verification results [2]. Reuse of reachability analysis results for different test goals [7, 8] has been presented and implemented as CPA/tiger on top of CPAchecker and corresponding reuse concepts have been applied to intermediate verification results [10]. Both approaches are limited to single systems without variability. Recent approaches for variability-aware product-line analysis can be roughly categorized into four strategies: sample-based, family-based, feature-based, and incremental techniques [27]. Most of these approaches consider the adoption of formal methods for product-line verification [26]. Similar to the problem of test generation, these approaches are, in general, also concerned with exploring the state space of entire product lines incorporating inherent variability. Family-based SPL analysis approaches focus on lifting static analysis and model-checking techniques to entire product lines. Some of those approaches also use symbolic model-checking techniques, handling features as special inputs, as in our approach [14,16]. However, those approaches consider a family-based evaluation of one particular model-checking query without systematic reuse of analysis results. Feature-based [21], as well as incremental SPL model-checking techniques [22], perform a step-wise state-space exploration similar to the CEGAR approach, but, again, considering a single query instead of multiple analysis/test goals. Finally, incremental approaches focus on step-wise test-suite refinement and retest selection inspired by regression testing, where the test-case generation approach is out of scope [17,23].

6 Conclusion

We presented a test-suite-generation approach for efficiently achieving complete multi-goal test-coverage of product-line implementations. Our approach exploits similarity information to facilitate systematic reuse of reachabiltity information among product-line variants. To this end, we extended the test-suite generator CPA/TIGER and presented evaluation results showing a considerable gain in efficiency compared to approaches without reuse. In addition, we plan to further improve reuse strategies, e.g, by fully adopting the concepts from previous work [8] and considering optimization criteria for the ordering of test goals.

Acknowledgments. This work has been supported by the German Research Foundation (DFG) in the Priority Programme SPP 1593: Design For Future – Managed Software Evolution (SCHU 1309/6-1, AP 206/5), by the DFG grants AP 206/4 and AP 206/6, as well as by the Austrian National Research Network S11403 and S11405 (RiSE) of the Austrian Science Fund (FWF), and by the Vienna Science and Technology Fund (WWTF) through grant PROSEED.

References

1. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)
2. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., von Rhein, A.: Domain Types: Abstract-Domain Selection Based on Variable Usage. In: HVC, pp. 262–278. Springer (2013)
3. Apel, S., Rhein, A.v., Wendler, P., Grösslinger, A., Beyer, D.: Strategies for Product-Line Verification: Case Studies and Experiments. In: ICSE. pp. 482–491. IEEE Press (2013)
4. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Model-Checking Tool for Families of Services. In: FMOODS/FORTE'11. pp. 44–58. Springer (2011)
5. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: SPLC. pp. 7–20. Springer (2005)

6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer 9(5-6), 505–525 (2007)
7. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE. pp. 326–335 (2004)
8. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information Reuse for Multi-goal Reachability Analyses. In: ESOP, pp. 472–491. Springer (2013)
9. Beyer, D., Keremoglu, M.: CPAchecker: A Tool for Configurable Software Verification. In: CAV, pp. 184–190. Springer (2011)
10. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision Reuse for Efficient Regression Verification. In: ESEC/FSE 2013. pp. 389–399. ACM (2013)
11. Chilenski, J.J., Miller, S.P.: Applicability of Modified Condition/Decision Coverage to Software Testing. Software Engineering Journal 9, 193–200(7) (1994)
12. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-Based Coverage-Driven Test-Suite Generation for Software Product Lines. In: MoDELS. pp. 425–439. Springer (2011)
13. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: CAV, pp. 154–169. Springer (2000)
14. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic Model Checking of Software Product Lines. In: ICSE. pp. 321–330 (2011)
15. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
16. Cordy, M., Classen, A., Schobbens, P.Y., Heymans, P., Legay, A.: Simulation-Based Abstractions for Software Product-Line Model Checking. In: ICSE. pp. 672–682. IEEE (2012)
17. Engström, E.: Exploring Regression Testing and Software Product Line Testing - Research and State of Practice. Lic dissertation, Lund University (May 2010)
18. Hall, R.J.: Fundamental Nonmodularity in Electronic Mail. ASE 12, 41–79 (2005)
19. Kästner, C., Giarrusso, P., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In: OOPSLA. pp. 805–824. ACM (2011)
20. Kramer, J., Magee, J., Sloman, M., Lister, A.: CONIC: An Integrated Approach to Distributed Computer Control Systems. Computers and Digital Techniques 130, 1–20 (1983)
21. Li, H.C., Krishnamurthi, S., Fisler, K.: Interfaces for Modular Feature Verification. In: ASE'02. pp. 195–204. IEEE (2002)
22. Lochau, M., Mennicke, S., Baller, H., Ribbeck, L.: DeltaCCS: A Core Calculus for Behavioral Change. In: FMSPLE 2014, pp. 320–335. Springer (2014)
23. Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In: TAP. pp. 67–82. Springer (2012)
24. McGregor, J.D.: Testing a Software Product Line. Tech. Rep. CMU/SEI-2001-TR-022, Software Engineering Inst. (2001)
25. Post, H., Sinz, C.: Configuration Lifting: Verification Meets Software Configuration. In: ASE. pp. 347–350. IEEE (2008)
26. Schaefer, I., Hähnle, R.: Formal Methods in Software Product Line Engineering. Computer 44(2), 82–85 (2011)
27. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A Classification and Survey of Analysis Strategies for Software Product Lines. ACM Comput. Surv. 47(1), 6:1–6:45 (2014)
28. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
29. Weiss, D.M.: The Product Line Hall of Fame. In: SPLC. p. 395. IEEE (2008)