Faster Bug Detection for Software Product Lines with Incomplete Feature Models

Sabrina Souto[‡] Divya Gopinath[†] Marcelo d'Amorim[‡] Darko Marinov[*]
Sarfraz Khurshid[†] Don Batory[†]

[‡] Federal University of Pernambuco
Recife, PE, Brazil

[†] University of Texas
Texas, Austin, USA

[*] University of Illinois
Urbana, IL, USA

ABSTRACT

A *software product line (SPL)* is a family of programs that are differentiated by *features* — increments in functionality. Systematically testing an SPL is challenging because it requires running each test of a test suite against a combinatorial number of programs. *Feature models* capture dependencies among features and can (1) reduce the space of programs to test and (2) enable accurate categorization of failing tests as failures of programs or the tests themselves, not as failures due to illegal combinations of features. In practice, sadly, feature models are not always available.

We introduce `SPLif`, the first approach for testing SPLs that does not require the a priori availability of feature models. Our insight is to use a profile of passing and failing test runs to quickly identify failures that are indicative of real problems in test or code rather than specious failures due to illegal feature combinations.

Experimental results on five SPLs and one large configurable system (GCC) demonstrate the effectiveness of our approach. `SPLif` enabled the discovery of five news bugs in GCC, three of which have already been fixed.

CCS Concepts

•Software and its engineering → Software verification and validation;

Keywords

Software Testing, Feature Models, GCC.

1. INTRODUCTION

Software product lines (SPLs) improve the quality and reduce maintenance costs for a family of related programs [45, 52]. Each program of an SPL is identified by a unique combination of *features* – increments in functionality. SPLs can be constructed using *feature variables*, which guard the execution of feature-specific code (in the spirit of `ifdef` preprocessing directives) [11, 17].

Feature models (FMs) formally capture dependencies among features; they distinguish which combinations of features are *legal* from those that are not. Feature models play a key role in testing SPLs. They constrain the space of products to test and enable accurate categorization of failing tests as failures of programs or the tests themselves, not as failures due to illegal configurations. Consequently, testing SPLs ignoring such dependencies is senseless. Although *most* cited prior work on testing SPLs assumes the availability of complete formally-specified feature models [10, 12, 14, 26, 36, 37, 39, 51, 53], in practice, feature models are not always available, presenting a huge challenge to SPL testers.

We address the important problem of *finding tests that fail on legal configurations of SPLs regardless of how complete a feature model may be*. Intuitively, only tests that fail on legal configurations indicate real faults so finding such fault-revealing legal configurations soon is important for quick bug detection.

We introduce *`SPLif`*, a technique for testing SPLs that does not require a priori availability of complete feature models. `SPLif` guides the developer by classifying configurations related to the parts of the incomplete model that are relevant for failing test runs. Our insight is that by running each test against many configurations, we can utilize information from failing and passing runs to help developers prioritize their inspections.

It is important to note that this problem is distinct compared to reverse-engineering feature models from code [5, 7, 8, 18, 21, 32, 42, 50, 56]. Although it is possible to use testing to complement existing feature models the goal of this paper is to detect legal fault-revealing configurations quickly.

This paper makes the following contributions:

- **Technique:** We present `SPLif`, a technique that synergistically exploits tests and incomplete feature models (in the limit, starting with an empty feature model) to help users both (1) distinguish test failures caused by problems in test/code from those caused by illegal configurations and (2) build a more complete feature model, as a consequence.
- **Implementation:** `SPLif` builds on the `SPLat` tool [39] to execute tests on reachable configurations. `SPLif` works both for Java SPLs and configurable systems such as GCC.
- **Evaluation:** `SPLif` has been evaluated on five SPLs. The results demonstrate the utility of `SPLif` in testing SPLs with incomplete feature models. Further, we evaluated `SPLif` on one large extensively tested configurable system, GCC, where it helped to reveal 5 new bugs, 3 of which have been fixed after our bug reports.

2. FEATURE MODELS

Let ϕ be a set of boolean variables denoting SPL features. A *configuration* $c : \phi \rightharpoonup \{false, true\}$ is a partial function from variables to boolean values; c maps *some* (not necessarily all) feature variables to values *false* or *true*. A configuration can be encoded as a boolean formula $f_c = \bigwedge p_i$, where $p_i = (x_i | \neg x_i)$ for $x_i \in \phi$. We denote with $|f_c|$ the number of variables referenced in f_c. We say

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA
© 2015 ACM. ISBN 978-1-4503-3613-0/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2791060.2791093

that a configuration c is *complete* iff $|f_c|$=$|\phi|$; it is *incomplete* otherwise. An incomplete configuration c represents a set of $2^{|\phi|-|f_c|}$ complete configurations (we call them *extensions* of f_c), which map *all* variables to boolean values.

Example. Let $\phi = \{A,B,C,D,E\}$. Configuration f_{c_1}=$A \wedge B \wedge \neg C \wedge D \wedge E$ is complete. Configurations f_{c_2}=$A \wedge \neg B$ and f_{c_3}=$A \wedge B \wedge \neg C \wedge E$ are incomplete. c_2 can be also written as "10???", where the i_{th} position corresponds to the i_{th} feature variable according to a given total order of variables. We use the numbers 1 and 0 to indicate, respectively, the presence or absence of a feature and the symbol "?" to indicate "undefined".

A feature model defines which configurations are legal for a given SPL. We use the label "✓" (for *valid*) to indicate that a complete configuration does not violate model constraints, while the label "✗" (for *invalid*) indicates that some constraint has been violated.

Incomplete Feature Models An incomplete feature model is a partial function $M_ : 2^{\phi} \rightharpoonup \{✓, ✗\}$ that maps *complete configuration* $c \in 2^{\phi}$ to a configuration label. If the label of a configuration is not in $M_$, it is "*unknown*". In one limit, the feature model is *empty* if it assigns an "*unknown*" label to all $2^{|\phi|}$ configurations. In another, the feature model is *complete* if it assigns ✓or ✗ to every configuration.

Modeling Incomplete Feature Models Analogous to the labels ✓and ✗ used to indicate validity of complete configurations, we use the labels "L" (for *legal*) and "$notL$" (for *illegal*) to indicate legality of *incomplete configurations*. An incomplete configuration $c \in 2^{\phi}$ is illegal if it violates some constraint in the feature model; it is legal otherwise. We model an incomplete feature model as a total function $M : 2^{\phi} \to \{L, notL\}$.

Example. Formula f_M=$(x \to \neg z) \wedge (\neg y \to z)$ encodes an incomplete feature model M. The configuration 1?1?? is illegal as it violates $x \to \neg z$ whereas the configuration 11?11 is legal as it does not violate any constraint in M. Indeed, it is possible to obtain legal complete configuration (aka valid ✓) in M from 11?11 with the assignment of 0 to z, however the assignment of 1 to z produces an illegal complete configuration (aka invalid ✗).

The diagram from Figure 1 characterizes the key relationships across configurations that we use. The diagram unifies the notions of completeness and legality of configurations. An edge from one configuration kind to another indicates extension.

Note that similarity to legal configurations does not imply legality: when an incomplete configuration is legal (see top element of the hierarchy), extensions of that configuration *may or may not* be legal. In contrast, when an incomplete configuration is illegal, all its extensions *must be* illegal. Provided that one preserves this invariant, it is possible to augment f_M incrementally by reducing uncertainty associated with incomplete configurations.

3. EXAMPLE

The main problem for testing SPLs with incomplete feature models (in the limit empty models) is that if a test fails it is difficult to determine with absolute certainty if there is a bug in the code, a bug in the test, or the bug manifests because of an illegal combination of features. `SPLif` addresses this problem.

`Notepad`. To illustrate, we use `Notepad`, a simple visual text editor that has been previously used in related research [37–39]. `Notepad` has 2,074 lines of code, 17 features, and 62 tests. While `Notepad` has a complete feature model formally specified, for the sake of illustration, we assume here an empty model.

A `Notepad` test (failure). Figure 2 shows a test for `Notepad` – *testEditToolBar*. This test is implemented using the GUI testing framework FEST [25] and uses buttons available from the toolbars

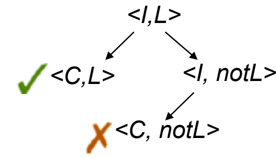

Figure 1: Hierarchy of configuration types. The first element of the pair indicates completeness of a configuration: "*I*" is Incomplete and "*C*" is Complete. The second element of the pair indicates legality of a configuration: "*notL*" is for not Legal and "*L*" is for Legal. An edge indicates that it is possible to extend a configuration of one type to another. Complete configuration types appear at the leaves of the tree. The check mark ✓ labels a complete and legal configuration (aka valid) whereas the cross mark ✗ labels a complete and illegal configuration (aka invalid).

```
1  public void testEditToolBar() {
2   JTextComponentFixture textArea = window.textBox();
3   textArea.enterText("Hi");
4   JButtonFixture cutButton =
               getButtonByTooltip("Cut", window);
5   cutButton.requireVisible();
6   cutButton.requireEnabled();
7   textArea.selectAll();
8   cutButton.click();
9   assertThat(textArea.text()).contains("");
10  JButtonFixture pasteButton =
            getButtonByTooltip("Paste", window);
11  pasteButton.click();
12  pasteButton.click();
13  assertThat(textArea.text().contains("HiHi");  } }
14
15 private JButtonFixture getButtonByTooltip(
          String toolTipText, ContainerFixture  frame){
16  GenericTypeMatcher<JButton> buttonMatcher =
17    new GenericTypeMatcher<JButton>(JButton.class){
18      @Override
19      protected boolean isMatching(JButton button){
20       return button.getToolTipText().equals(msg);
21      }
22    }; return frame.button(buttonMatcher); }
```

Figure 2: Test of `Notepad` that checks the cut and paste functionalities. This test fills the text area with a string, selects the text area, and presses the "cut" button. It then asserts that the text area is indeed empty, presses the "paste" button twice, and checks for the presence of the repeated string.

to cut and paste a string written to the text area. To find these buttons, this test uses the auxiliary function *getButtonByToolTip* (line 15), which iterates over all buttons reachable from the parent window of the text area until finding one that matches the given tooltip text, passed as parameter. This test fails when the feature *UNDOREDOTOOLBAR* is enabled. That feature adds to the `Notepad` window a new toolbar containing the buttons "undo" and "redo"; they become reachable in the search for buttons. The failure manifests because this feature initializes the *toolTipText* field of these buttons with **null**. Consequently, execution raises *NullPointerException* at line 20 when trying to derreference this field from the "undo" button.

`SPLat`. `SPLif` builds on `SPLat` [39], a technique that we previously developed, to execute each test several times, once for each configuration that a test encounters during execution. *Failing tests* are tests whose executions failed on at least one configuration. We call such executions *failing runs*, and call configurations on which tests fail *failing configurations*. With `Notepad` and an empty model, `SPLat` explores all given 62 tests, a total of 5,094 runs (excluding time-outs) on distinct configurations. Out of these 5,094 runs, 300 were failing runs that were attributed to 3 tests on different configurations. It is daunting to inspect all 300 failures to find real problems. `SPLif` reduces the number of inspections to 10.

`SPLif` in a nutshell. `SPLif` takes as input an SPL with its test suite and incomplete feature model. It explores each test with various configurations, and produces a ranked list of failing tests and configurations for the user to inspect. Inspired by Tarantula [35], `SPLif` *ranks tests* based on the *ratio* of failing and passing configurations.

SPLif also *ranks configurations* using various strategies to prioritize the order in which they should be inspected. An inspection can result in a repair of the test, a repair of the feature model, and/or a repair of the code under test. If a test fails on a legal configuration, a real problem is revealed in the target source code or in the test itself.

If a test fails on an illegal configuration, the incomplete feature model can be updated to incorporate the violated constraint. Tests that pass for all the unknown or known legal configurations do not appear in the ranking that SPLif reports. The user continues inspecting failing tests and configurations until a budget limit is reached; the goal is to quickly find fault-revealing legal configurations that cover (all) failing tests.

SPLif on Notepad Tests. With an empty feature model, SPLat explores all given 62 tests of Notepad and a total of 5,094 runs. Out of these, 300 were failing and were attributed to 3 tests on 265 distinct configurations.

Most failing runs can be attributed to illegal configurations. We want to find failing tests for *legal* configurations. If a user randomly inspected the 133 failed runs for *testEditToolBar* (t_i=31 for Notepad table under Figure 7), (s)he would examine 37 configurations on average before arriving at the first legal configuration. SPLif does better than this: by following its ranking, the first legal configuration is found in the first 10 inspections. By prioritizing tests and configurations to inspect, SPLif makes users more productive.

Proceeding with this process until all failing tests have been inspected, a random ranking would require a total of 90 inspections, and a ranking generated by SPLif would require no additional inspection, because the same legal configurations that SPLif reports for the first test inspected is reachable by the other 2 failing tests as well. So, SPLif detects that all 3 tests expose a bug in the test code or in the source code by inspecting 10 configurations. This result represents a reduction of 88% compared to random inspection, and a reduction of 96% compared to inspect all 300 failing configurations (the worst case).

4. TECHNIQUE

SPLif takes as input a test suite and an incomplete feature model for an SPL and reports as output a ranked list of failing tests sorted by their likelihood of containing legal failing configurations.

We represent an incomplete feature model as two sets of configurations, one encoding legal configurations ($\mathcal{M}_L$) and one encoding illegal configurations ($\mathcal{M}_{notL}$). As shown in Section 2 we can view each set as a formula, a disjunction of the (conjunctive) formulas that represent each configuration in the set. A well-formed feature model should have disjoint legal and illegal sets of configurations, i.e., $\mathcal{M}_L \wedge \mathcal{M}_{notL}$ should be unsatisfiable. A feature model is empty if both $\mathcal{M}_L$ and $\mathcal{M}_{notL}$ are empty, and a feature model is complete if $\mathcal{M}_L \vee \mathcal{M}_{notL}$ is equivalent to $true$.

For such test from the input test suite, SPLif reports a ranked list of configurations for inspection by their likelihood of being legal. These rankings expose real problems in code or tests more quickly.

SPLif has three fully automated parts: Test Exploration (Section 4.1), Test Ranking (Section 4.2), and Configuration Ranking (Section 4.3). The overall approach consists of the following steps:

1. Use SPLif to run a given test suite with an incomplete or empty feature model (Section 4.1).
2. SPLif ranks tests (Section 4.2).
3. User chooses the highest ranked test.
4. SPLif ranks failing configurations for that test (Section 4.3).
5. User inspects the highest ranked configuration.
6. If the configuration is illegal, user provides that information to SPLif to make the feature model more complete.
7. If the configuration is legal, user repairs the test or the code under test.
8. Repeat steps 1-7 until running out of time budget or finishing the inspection of all failing tests.

Note that we could have chosen to inspect a configuration that fails for multiple tests and then sort tests to inspect. We prioritized tests and then configurations because we found it more intuitive to focus on one test at a time and to reason about multiple configurations for that one test.

Further, SPLif exploits illegal configurations discovered during the inspection process. When that happens SPLif can reduce incompleteness of feature models, which can help to better categorize remaining tests and configurations.

4.1 Test Exploration

SPLif uses SPLat [39] to obtain, for each test, all (not known to be illegal) configurations that have a unique trace during the test execution. We described SPLat in detail elsewhere [39], so we summarize it only briefly here, and then describe the changes we made for SPLif.

Given an SPL code base, a test and an *complete* feature model (we discuss incomplete models in the next paragraph), SPLat executes the test on one configuration, observes the values of feature variables that have been accessed during the execution, and uses these values to determine what other configurations should be considered in subsequent test executions. For example, if a test execution accessed only one feature variable, f, with value $false$, then SPLat re-executes the test with f set to $true$. If that second execution accesses no other feature variables, the search stops. Otherwise, it continues to explore the combinations of values of the other accessed variables. SPLat repeats this process until it explores all dynamically reachable configurations or until it reaches a specified bound on the number of configurations.

SPLat assumes that, in addition to the test, a *complete* feature model is provided as input. This allows SPLat to substantially reduce the space of possible configurations to explore. In contrast, SPLif assumes an *incomplete* feature model, so we had to change SPLat to treat every unknown configuration as valid (✓), if the model is complete, or as legal, if the model is incomplete. When SPLat runs now, it prunes test execution paths that correspond to definitely illegal configurations and explores all other paths. Additionally, we changed SPLat to accept a time limit for each test execution, because executing illegal configurations can take a long time or even lead to infinite loops.

4.2 Test Ranking

SPLif ranks tests for all executions with unknown configurations. For illegal configurations, SPLif/SPLat does not even execute a test. For legal configurations, the situation is quickly resolved: if the test passes, the execution is ignored; if the test fails, the execution immediately shows a real problem in the test or the code.

For each test t, we define the following terms:

- $P_t \stackrel{\text{def.}}{=}$ number of passing unknown configurations of t,
- $F_t \stackrel{\text{def.}}{=}$ number of failing unknown configurations of t,
- $FL_t \stackrel{\text{def.}}{=}$ number of legal configurations for which t fails,
- $S_t \stackrel{\text{def.}}{=}$ *suspiciousness rating* or fraction of failed unknown configurations. (*i.e.*, $S_t = \frac{F_t}{(F_t+P_t)}$).

Ranking is obtained by lexicographically sorting the triples $\langle FL_t,\ S_t,\ F_t\rangle$ associated with each test t. Spectrum-based fault

localization algorithms (*e.g.*, Tarantula [35] and Barinel [2]) use a similar criterion to classify suspicious statements.

Tests that appear higher in this ranking are considered more likely to contain a legal configuration $FL_t > 0$. Note that if a test fails for a legal configuration, we know it has a bug (in code or test) and that test needs inspection irrespective of the number of legal configurations it succeeds or fails.

The metrics S_t and F_t differentiate tests in the middle of the ranking. In case of ties on metric FL, test t_1 will be ranked higher than t_2 if $S_{t1} > S_{t2}$. The rationale for using S_t is that there is a higher chance of finding a failing legal configuration when `SPLat` reaches a relatively high number of failing configurations. Finally, if the first two metrics do not differentiate t_1 and t_2, `SPLif` uses the third metric as a tie-breaker ($F_{t1} > F_{t2}$).

`SPLif` can rank tests once at the beginning of its execution (option `STATIC`) or rerank tests after every test inspection (option `DYNAMIC`). Intuitively, reranking could help `SPLif` to better categorize the tests remaining for inspection: after the user labels the configurations from one test, the feature model becomes more complete. So if inspecting one test determines that some configuration is illegal, then the same configuration can be ignored for all other tests. Likewise, if inspecting one test determines that some configuration is legal, and another test fails for that configuration, then that other test immediately shows a real problem and goes to the top of the ranking.

4.3 Configuration Ranking

The previous section explained how `SPLif` ranks tests. Now, we explain how `SPLif` ranks configurations for a given test. `SPLif` has two independently selectable options to rank configurations, namely `UPDATEFM` and `WEIGHTED`. If no option is selected, `SPLif` randomly orders configurations for inspection.

If `UPDATEFM` is set, `SPLif` uses previously labeled configurations to update the incomplete feature model such that configurations whose labels can be inferred from the model are not inspected again. Note that `UPDATEFM` does not define an explicit order of configurations. If `WEIGHTED` is set, `SPLif` sorts the configurations according to three different criteria. For each configuration c, we define the following terms:

- $?_c \stackrel{\text{def.}}{=}$ number of undefined features variables in c,
- $F_c \stackrel{\text{def.}}{=}$ number of failing tests that execute c, and
- $SL_c \stackrel{\text{def.}}{=}$ boolean that indicates if c is *similar* to some previously seen legal configuration.

If the option `WEIGHTED` is set, the ranking of configurations is obtained by lexicographically sorting the triples $\langle ?_c,\ F_c,\ SL_c \rangle$ associated with each configuration c. Configurations that appear higher in the ranking are considered more helpful to improve overall performance of `SPLif`.

The first element of the triple helps to rank higher those configurations that could be obtained with instantiations of undefined variables, and the second element helps to rank higher those configurations that occur in yet-to-be-inspected tests. The rationale for these metrics is to optimize future labelings (L or $notL$) of configurations. For example, if a configuration is illegal, all complete extensions of it must be illegal; hence it is beneficial to label such configurations quickly as they label more complete configurations. The third element ranks higher those configurations that look similar to legal configurations. Similarity is determined by checking if $\mathcal{M}_L \wedge f_c$ is satisfiable, which suggests that the configuration is in accordance with the current feature model learned from the previously labeled legal configurations. Note, however, that similarity to legal configuration does not imply legality. In case of ties in the ranking `SPLif` uses random ordering.

```
1   /* summary of SPLat execution for a test */
2   class TestInfo { Test, Set<Conf> pass, fail; }
3
4   /* models and test suite */
5   INPUTS: T, M_L, M_notL
6   SPLif()
7
8    /* collect test profiles */
9    Set<TestInfo> Π = ∅
10   Π = {SPLat(M_notL, t_i) | t_i ∈ T} /*test info*/
11
12   R = list(Π)
13   /* rank tests by their execution profiles */
14   if (STATIC) R = rankTests(R)
15
16   attest:
17   while R ≠ [] do
18
19    /* dynamically (re-)rank tests */
20    if (DYNAMIC) R = rankTests(R)
21    t = head(R); R = tail(R)
22
23    /* inspect test t if a configuration has been
          previously inspected and labeled as legal */
24    if (t.fail ∩ M_L ≠ ∅)
25      break  /* inspect test! */
26
27    /* f_k is the logical encoding of configuration k */
28    Set<Conf> Δ = {k ∈ t.fail | isSAT(f_k ∧ ¬M_notL)}
29    if (Δ = ∅) continue /* ignore test! */
30
31    /* ranking confs. with unknown labels */
32    while Δ ≠ ∅
33      c = head(rankConfs(Δ))
34      Δ = Δ - {c}
35      switch ulabel(c) /* user labels c */
36        case L:
37          if (UPDATEFM)
38            M_L = M_L ∨ f_c
39          break attest  /* inspect test! */
40        case notL:
41          if (UPDATEFM)
42            M_notL = M_notL ∨ f_c
43            /* update set of unknown configurations */
44            Δ = {k ∈ Δ | isSAT(f_k ∧ ¬M_notL)}
45          break
46      update(c,T); /* update the counts of passing and
              failing configurations */
47
48  /* ranking tests*/
49  List<TestInfo> rankTests(List<TestInfo> R)
50   return sortLexicographically(R, λt:Test.⟨FL_t,S_t,F_t⟩)
51
52  /* ranking configurations*/
53  List<Conf> rankConfs(Set<Conf> Δ)
54   if (WEIGHTED)
55    return sortLexicographically(Δ, λc:Conf.⟨?_c,F_c,SL_c⟩)
56   else return randomOrder(Δ)
```

Figure 3: The `SPLif` Algorithm.

4.4 Algorithm

Figure 3 shows the pseudo-code of `SPLif`. It takes as input a test suite T for a software product line and its incomplete feature model.

At line 10 the algorithm calls `SPLat`, which returns results for each test in the test suite. For example, it outputs the results (pass or fail) for each configurations `SPLat` explores on a given test. The algorithm proceeds by iteratively choosing the top ranked test and then focusing on the failing configurations for each selected test.

There are two strategies for resorting the ranking of tests. The basic mode (option `STATIC`) sorts all the tests at the beginning of execution whereas the adaptive mode (option `DYNAMIC`) resorts the remaining tests after each test is inspected.

If any failing configuration is already in the legal set, the test shows a real problem that should be repaired (lines 24–25). If that does not hold but all configurations are in the illegal set, then the test should be ignored (line 29). Otherwise, the algorithm iterates through the set of still unknown configurations (lines 17–46). The algorithm sorts these configurations using one of the strategies discussed in Section 4.3.

Finally, the algorithm picks the highest ranked configuration and asks the user to label it. If the configuration is legal, this scenario reveals a real problem, and the user should stop the inspection of additional configurations to fix it. If the configuration is illegal, it is

added to the set of illegal configurations, and the inspection for this test proceeds to the next configuration. Note that the set of failing unknown configurations (Δ) is updated accordingly. The inspection finishes after the user finds a legal configuration (as that is a clear indication of a real problem in code or test) or inspects all the illegal configurations for each failing test.

The call to `update` (line 46) updates the counts of passing and failing configurations (i.e., P_t and F_t for each test t from T) after every configuration labeling. In `DYNAMIC` mode, this update can potentially modify the relative ranking of each test in $\mathcal{R}$.

5. EVALUATION

We pose the following research questions to evaluate `SPLif`:

RQ1 How well does `SPLif` rank faulty tests for inspection?

RQ2 How well does `SPLif` rank configurations (of selected tests) for inspection?

RQ3 How well does `SPLif` scale to real code?

5.1 Subjects

We *initially* evaluated `SPLif` using five small subjects (size range: 1.7–3.6KLOC) that were used to test SPLs previously. Figure 4 tabulates, for each subject, the source of the subject, the number of feature variables, the number of valid (complete) configurations, and the code size.

Subject	*Source*	*# Features*	*# Valid Configs.*	*LOC*
`Companies`	[34]	10	192	2,059
`DesktopSearcher`	[41]	16	462	3,779
`GPL`	[43]	13	73	1,713
`Notepad`	[38]	17	256	2,074
`ZipMe`	[9]	13	24	3,650

Figure 4: Target SPLs used in `SPLif` evaluation.

5.2 Setup

Tests analyzed. We asked 5 students to create tests. We assigned two subjects per student and two students per subject, and provided instructions on how to create tests. In brief, we instructed students to create a test suite that achieves maximum feature coverage. Detailed instructions that students received for creating tests can be found elsewhere [24]. We provided support to students on how to use the FEST framework [25] to create GUI tests for `DesktopSearcher` and `Notepad`.

Figure 5 shows the number of tests and the number of configurations those tests dynamically reached for each SPL. Column "*All*" is the total number of reachable configurations and column "*≥1 Failing*" is the number of reachable configurations that cause at least one test to fail. Values in parentheses denote the number of legal configurations in the corresponding set.

		Reachable Configs.	
Subject	*# Tests*	*All*	*≥1 Failing*
`Companies`	19	152 (64)	72 (4)
`GPL`	25	1,268 (137)	497 (49)
`Notepad`	62	3,273 (66)	265 (6)
`DesktopSearcher`	44	254 (160)	125 (83)
`ZipMe`	62	2,431 (153)	261 (12)

Figure 5: Tests and Reachable Configurations. Values in parentheses show the subset of legal configurations.

Figure 6 shows the distributions of legal configurations (column "*LCs*") that fail for some number of failing tests (column "*FTs*"). For example, 14 configurations of `GPL` fail in only one test (but not necessarily the same test).

Figure 7 shows, for each SPL and test, the number of configurations that make the test run fail (column "*F*"), the number of

`Companies`

LCs	*FTs*
4	1

`GPL`

LCs	*FTs*
14	1
35	2

`Notepad`

LCs	*FTs*
5	1
1	3

`DesktopSearcher`

LCs	*FTs*
66	1
9	2
7	3
1	4

`ZipMe`

LCs	*FTs*
12	1

Figure 6: Distribution of number of legal configurations (LCs) per number of failing tests (FTs).

`Companies`

t_i	F_i	P_i	FL_i
13	16	16	0
14	24	8	0
15	48	16	0
16	48	16	0
17	1	1	1
18	7	5	1
19	12	4	2

`GPL`

t_i	F_i	P_i	FL_i
18	4	4	0
19	4	4	0
20	33	78	0
21	192	64	0
22	7	1	1
23	26	26	2
24	248	8	35
25	256	256	46

`Notepad`

t_i	F_i	P_i	FL_i
29	67	62	1
30	100	138	4
31	133	94	3

`DesktopSearcher`

t_i	F_i	P_i	FL_i
10	1	4	0
11	1	8	0
12	1	8	0
13	1	2	0
14	1	8	0
15	1	8	0
16	1	8	0
17	1	8	0
18	1	32	0
19	1	8	0
20	1	32	0
21	1	4	0
22	1	8	0
23	1	32	0
24	1	0	0
25	1	0	0
26	2	8	0
27	3	8	0
28	3	8	0
29	3	2	0
30	3	8	0
31	1	25	1
32	3	1	3
33	3	1	3
34	3	1	3
35	4	0	3
36	5	0	4
37	7	1	7
38	7	1	7
39	14	0	8
40	14	1	13
41	15	2	14
42	17	1	16
43	28	3	27

`ZipMe`

t_i	F_i	P_i	FL_i
30	1	1	0
31	1	1	0
32	1	3	0
33	1	3	0
34	1	8	0
35	1	2	0
36	1	3	0
37	1	3	0
38	1	3	0
39	1	3	0
40	1	3	0
41	1	2	0
42	1	1	0
43	1	3	0
44	1	3	0
45	2	12	0
46	2	8	0
47	2	6	0
48	2	2	0
49	4	33	0
50	5	7	0
51	10	38	0
52	11	43	0
53	12	12	0
54	31	201	0
55	32	198	0
56	61	399	0
57	61	1244	0
58	1	2	1
59	3	1	1
60	3	3	2
61	4	5	2
62	136	132	6

Figure 7: Counts of passing and failing executions per test and SPL. t_i is the test id. F_i is the number of failures of t_i. P_i is the number of passing executions of t_i. FL_i is the number of legal configurations in which t_i fails. We omit test entries without failing runs.

configurations that the test passes (column "*P*"), and the number of legal configurations that produce failure (column "*FL*"). Each table contains two sections: the first includes entries where *FL=0* and the second where *FL≠0*. We omit test entries that only pass, i.e., entries where $F = 0$.

Initial Feature Model and Ground Truth. In the experiments, we initialized `SPLif` with an empty feature model, *i.e.*, the execution of `SPLat` at line 10 had no illegal configuration. To classify configurations (and hence model the user), we used pre-existing feature models as reference to label legality of configurations.

5.3 Ranking Tests Using Suspiciousness Score

The purpose of test ranking is to speed up discovery of bugs through the prioritization of tests that fail on legal configurations. Figure 8 shows, for each SPL, the ranking of tests from Figure 7. Column *R* shows the rank of test t_i. The *suspiciousness score* S_i for that test is computed by: $S_i = F_i/(F_i + P_i)$.

Recall that Section 4.2 states that the ranking of tests is obtained by lexicographically sorting the triples $\langle FL_i,\ S_i,\ F_i \rangle$. However, at this stage no configurations are known to be (il)legal. For this reason, we used the pair $\langle S_i,\ F_i \rangle$ and reported only S_i.

We observed that for the cases with a relatively small number of failing tests (`Companies`, `GPL`, and `Notepad`) classification of tests is not very helpful. Although the first tests with legal configuration appear high in the ranking (positions 3, 1, and 1, respectively), overall other tests that do not fail on legal configurations are also highly ranked. For the cases with relatively many failing tests

Companies

R	t_i	S_i
1	15	0.75
2	16	0.75
3	19	0.75
4	14	0.75
5	18	0.58
6	13	0.50
7	17	0.50

GPL

R	t_i	S_i
1	24	0.97
2	22	0.88
3	21	0.75
4	18	0.50
5	19	0.50
6	25	0.50
7	23	0.50
8	20	0.30

Notepad

R	t_i	S_i
1	31	0.59
2	29	0.52
3	30	0.42

DesktopSearcher

R	t_i	S_i
1	24	1.00
2	25	1.00
3	39	1.00
4	35	1.00
5	36	1.00
6	42	0.94
7	40	0.93
8	43	0.90
9	41	0.88
10	37	0.88
11	38	0.88
12	32	0.75
13	33	0.75
14	34	0.75
15	29	0.60
16	13	0.33
17	27	0.27
18	28	0.27
19	30	0.27
20	10	0.20
21	26	0.20
22	21	0.20
23	11	0.11
24	12	0.11
25	14	0.11
26	15	0.11
27	16	0.11
28	17	0.11
29	19	0.11
30	22	0.11
31	31	0.04
32	18	0.03
33	20	0.03
34	23	0.03

ZipMe

R	t_i	S_i
1	59	0.75
2	62	0.51
3	30	0.50
4	31	0.50
5	60	0.50
6	42	0.50
7	53	0.50
8	48	0.50
9	61	0.44
10	50	0.42
11	58	0.33
12	35	0.33
13	41	0.33
14	32	0.25
15	33	0.25
16	36	0.25
17	37	0.25
18	38	0.25
19	39	0.25
20	40	0.25
21	47	0.25
22	43	0.25
23	44	0.25
24	51	0.21
25	52	0.20
26	46	0.20
27	45	0.14
28	55	0.14
29	54	0.13
30	56	0.13
31	34	0.11
32	49	0.11
33	57	0.05

Figure 8: Ranking of tests. Column R shows the rank of test t_i from Figure 7; S_i shows the suspiciousness score of t_i. A row in gray color indicates a test for which at least one failing configuration it reaches is legal.

(DesktopSearcher and ZipMe), we found that ranking tests was helpful: most tests that fail on legal configurations appear at top positions in the ranking.

RQ1. Based on these results we conclude the following:

> The use of a suspiciousness score based on pass-fail ratios of test runs is a good predictor for labeling tests that fail on legal configurations when the number of failures is relatively high.

5.4 Ranking Configurations

Once a test is selected, the tester needs to identify the configurations more likely to be legal out of those that trigger failure. Such task can be overwhelming: a test run can expose many distinct failures and there can be many tests. We considered 4 prioritization techniques for inspecting configurations, defined as follows.

- ***Random*** is our comparison baseline. It randomly orders failing configurations associated to a test.
- ***UpdateFM*** is a variation of *Random* that memoizes previously labeled configurations.
- ***Weighted*** is a variation of *UpdateFM* that ranks the configurations for inspection according to their weights.
- ***Adaptive*** is a variant of *Weighted* that re-ranks tests on-the-fly.

We obtain these techniques by setting the option flags (STATIC, UPDATEFM, WEIGHTED, and DYNAMIC) in the SPLif algorithm.

When the tester finds a legal configuration for a failing test, SPLif skips to the next test, and when he/she finishes inspecting all tests, he/she repairs the test/code for all recognized bugs, *i.e.*, for failures of pairs of legal configuration and test. Section 5.5 discusses another scenario where the tester stops to repair the test/code after finding a failure of a pair of legal configuration and test.

Figure 9 shows the number of configuration inspections needed to find the first legal configuration for each test. In *Random* mode, this number is >= to the number of tests. In the other modes, the number is smaller because SPLif recognizes already inspected configurations. If they appear again, SPLif skips to the next test.

Mode	Companies	GPL	Notepad	DesktopSearcher	ZipMe
Random	146	257	90	44	269
UpdateFM	69	211	40	30	45
Weighted and *Adaptive*	69	223	10	34	49

Figure 9: Total number of inspections considering the 4 variants: *Random*, *UpdateFM*, *Weighted* and *Adaptive*.

From the results, we recommend two techniques be used to inspect configurations, depending on the scenario the tester may have:

1. *UpdateFM* is more appropriate when there is a considerable number of tests failing on illegal configurations that are common among them; and
2. *Weighted* and *Adaptive* are more appropriate when there is a considerable number of tests failing on legal configurations that are common among them.

Discussion. Results indicate that for DesktopSearcher, GPL and Notepad the use of variant *Random* revealed more legal configurations than any other variant. Recall that *Random* only uses SPLif to rank tests. This is expected as *Random* does not memoize already-labeled configurations, increasing the number of inspections (of legal configurations or not). The other variants record already-labeled configurations (to detect bugs in other failing tests faster) and hence may not visit all failing configurations. However, for all subjects, the total number of configuration inspections necessary to inspect all tests was much higher in *Random* compared to other variants. This highlights the importance of SPLif to filter out tests that fail on legal reachable configurations.

UpdateFM is the simplest variant of SPLif that ranks both tests *and* configurations. In this variant, SPLif memoizes configuration labelings in symbolic models. Intuitively, this enables faster classification of other buggy tests that fail for legal configurations and helps to ignore failures on illegal configurations (lines 24–29 in Figure 3). We found that this variant helps to speed up classification of buggy tests when a large number of configurations repeat across tests.

Weighted and *Adaptive* also use memory (*UpdateFM*) of previous labelings. Results indicate that *UpdateFM* performs better compared to these variants for most of the subjects. The reason for this can be attributed to the observation that when there is a considerable number of tests failing on common illegal configurations, then it helps if these illegal configurations are identified soon. Such a characteristic appears in ZipMe (27 tests failing on illegal configurations vs. just 5 tests failing on legal configurations), in DesktopSearcher (30 tests failing on illegal configurations vs. 13 tests failing on legal configurations), and in Companies. The large number of common illegal configurations bring about the difference for GPL (see Figure 8).

In contrast, for Notepad, all failing tests have failures due to legal configurations. Furthermore, there is 1 legal configuration common amongst all failing tests. The *Weighted* (which performs heuristic ranking of configurations based on configurations previously labeled) and *Adaptive* (which re-ranks tests after every labeling) variants help detect this common legal configuration in just 10 configuration inspections. This leads to the detection of the other buggy tests without requiring any further inspection of configurations. *UpdateFM*, which randomly ranks configurations brings to the top a different legal configuration (unique to a single test), and then ranks the legal configuration common to the other tests after 40 inspections.

RQ2. Based on these results we conclude the following:

> The overall number of inspections needed to find problems in tests or source code is fewer when SPLif uses an inferred feature model (*UpdateFM*) as opposed to randomly selecting configurations (*Random*). Re-ranking of tests (*Adaptive*) and ranking of configurations (*Weighted*) seem to help when the number of tests failing on common legal configurations is high.

5.5 Incremental Runs of SPLif

Re-execution of SPLif for one test (or the entire test suite) after every test/code repair can be expensive, and potentially wasteful if the repair had corrected all faults in the inspected test and the change did not impact any other test. One approach to deal with this cost is to optimistically assume that the repair in the test corrects all its related faults and, in such a case, not re-execute SPLif. Instead, this approach removes the repaired test from the list and control proceeds to the next test in the ranking. The rationale for this approach is to inspect tests quicker looking for new faults.

After all failing tests have been inspected once, SPLif is *re-started* to run on the new version of the repaired tests alone. Note that all configurations for the other tests must have been inspected in previous runs of SPLif. This incremental pass acts as a validation phase for the test repairs and also helps to identify other faults that could have been exposed by other legal configurations. This is exactly what we did in the experiments of Section 5.4. We elaborate on the subsequent passes in the following. We observed that, except for GPL, none of the repairs modified application code. For all subjects, except DesktopSearcher, SPLif did not reveal any other faults in the second pass. For DesktopSearcher, the first pass discovered 13 tests failing on legal configurations (out of 34 reported in SPLif's ranking). The second pass found 6 of the 13 repaired tests to fail again. These tests failed for unseen legal configurations that had not been explored earlier. The third pass again revealed a failure in one of the 6 tests for another unseen configuration.

5.6 Case Study: GCC

We next considered the GNU Compiler Collection (GCC) [28], a large system with hundreds of configuration options [30]. The GCC testing infrastructure runs each test for only one configuration; we used SPLat to run each test for multiple configurations, reachable from tests, and then we applied SPLif to rank both the failing tests and configurations, as we did in Section 5. Our primary goal was to see how SPLif could scale to such a large system and what kind of failures it could find. With the recent work on testing GCC [15, 40, 58], we did not expect to find new bugs, but SPLif did. As we do not have extensive GCC knowledge to properly classify general failures, we focused our inspection of failures on crashes, which provide a stronger indication of a real bug in code. We first give a high-level summary of setup and results, and then more details of applying SPLif on GCC.

5.6.1 Setup

Test execution. Test execution in GCC is time consuming. It takes ∼45min to run all 2,608 tests from the dg-gcc test suite considering 1 configuration per test and our running environment. This corresponds to roughly 1s per test run. To deal with the high cost of test execution, we focused on a selection of test suites, ran SPLat using a limited number of options, limited the number of configurations per test to 50, and randomized the execution of SPLat to sample different, dynamically reachable, configurations.

Tests analyzed. Overall, we analyzed a total of 4,108 tests from three different test suites: 2,608 from gcc-dg, 548 from dg-torture, and 952 from tree-ssa. We focused on these suites for this experiment because we observed from bug-reports that the incidence of bugs revealed with these suites was higher.

Options analyzed. To make the runtime reasonable, we restricted the number of options that SPLat considers (see [30]). We used the 40 most frequently cited options in the GCC bug reports (from the month of July 2014). The rationale was that more bugs could be found close to where some bugs have been recently reported (we noticed that the top options do not change much across months).

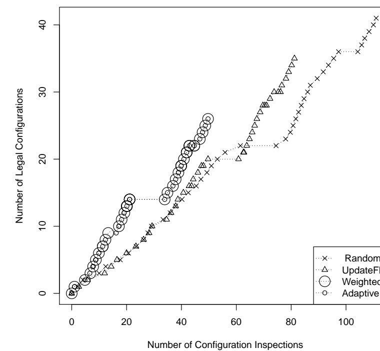

Figure 10: Configuration inspection progress for GCC.

Initial Feature Model and Ground Truth. In this experiment we initialized SPLif with an empty feature model (*i.e.*, the execution of SPLat at line 10 is unconstrained) and used an existing feature model of GCC [27] as the ground truth to model developer knowledge for classifying configurations. Other choices of initial model and ground truth are possible.

For the ground truth we built on the work of Garvin *et al.* [27] that documented 110 constraints from GCC. We augmented this model with constraints that we manually extracted from the online documentation of GCC [29]; we found a total of 136 new constraints. For example, we found that the option -fsel-sched-pipelining enables software pipelining of the innermost loops during selective scheduling and has no effect unless the options -fselective-scheduling or -fselective-scheduling2 are turned on.

5.6.2 Results

Ranking tests and configurations. The main goal of ranking both tests and configurations is to find a legal configuration that fails quickly. In total, 4,108 tests failed in 3,986 configurations, due to crash or not. Recall that we ran each test against 50 reachable configurations. Considering only crashes, a total of 43 tests manifested crashes in 268 pairs of test and configurations. From a total of 43 crashing tests, only 2 tests had all crashing configurations illegal. These bad cases ranked lower, at positions 23 and 38 respectively.

Figure 10 illustrates the progress in number of legal configurations found with the progress in number of configurations inspected, using the four alternative variants for inspecting configurations presented in Section 5.4. Using the *Weighted* or *Adaptive* mode, from a total of 268 failing configurations (218 distinct), only 45 configurations needed inspection (26 of which were legal). This indicates that with a relatively low number of inspections SPLif enables one to find a legal failing configuration in all 43 crash-revealing tests.

New bugs found. To provide informative bug reports to the GCC team we needed to simplify observed failures. For that we grouped failing test-configuration pairs in clusters and minimized configurations (from each pair) inside clusters. We clustered failing pairs according to the GCC error message and the code location responsible for the failure/crash, when it is available. For example, all crashes with the message "int_cst_value" corresponding to location tree.c:10625 were grouped in one cluster. Figure 11 shows all clusters defined according to these heuristics, and lists the crashes we found in the main trunk of GCC. Column "*Properties*" lists the properties, column "*Tests*" shows the number of crashing tests, column "*Pairs*" lists the number of test-configuration pairs that crashes, column "*Id*" denotes the id of the bug as reported in the GCC bug-tracking system, column "*Date Confirmed*" shows the date the team confirmed the bug as new. A bug report is initially given the status "unconfirmed". Column "*Current Status*" shows the current status of

Cluster data			*Bug report data*			
Properties	*#Tests*	*#Pairs*	*Id*	*Confirmed*	*Fixed*	*Status*
`compute_affine_dependence, tree-data-ref.c: 4233`	34	223	61980	Aug.1, 2014	-	NEW
`int_cst_value, tree.c: 10625`	4	34	62069	Aug.8, 2014	-	NEW
`verify_ssa failed, tree-ssa.c: 1056`	1	6	62070	Aug.8, 2014	Aug.11, 2014	RESOLVED FIXED
`build2_stat, tree.c: 4265`	1	4	62140	Aug.14, 2014	Oct.16, 2014	RESOLVED FIXED
`Segmentation fault: 11`	1	1	62141	Aug.14, 2014	Nov.19, 2014	RESOLVED FIXED

Figure 11: GCC bugs. Details at: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=<Column"Id">.

the bug, and finally column "*Date Fixed*" shows the date the GCC team fixed the bug.

Unfortunately, there may be configuration options that do not reproduce the crash. For that reason, we manually applied delta-debugging [59] to simplify configurations inside each bucket. More specifically, we selectively removed options from the input configuration and re-executed the test in separate. We repeated this process until the test would no longer reveal the crash.

We applied these simplification mechanisms as follows. For each cluster we randomly picked one pair of test and configuration and minimized the configuration to reproduce the crash at the same location. Then we re-run *all tests* from the same cluster with the minimized configuration to confirm that we would be able to reproduce the crash. The average number of options before minimization was 7. With minimization we found that only 1 or 2 options were relevant to reproduce crashes. With the minimized configuration at hand, we filed a bug-report, one per cluster. When we observe that the GCC team fixed 3 of the reported bugs (case of bugs: 62070, 62140, 62141) we validated the fix by re-running all pairs of tests and configurations inside the corresponding cluster for the updated version of GCC. If this validation fails it would likely indicate a different problem which would demand the split of the cluster and the report of a new bug. That did not happen for those 3 bugs, *i.e.*, apparently, the fixes were effective.

RQ3. Based on these results we found the following:

> Our proposed technique helped to reveal new bugs in GCC, a large configurable system that has been under active development for almost three decades.

5.6.3 *Details of applying* `SPLif` *on GCC*

GCC testing infrastructure. We briefly introduce DejaGnu [23], the GCC testing framework. The code snippet below shows an example DejaGnu test.

```
/*{ dg-do compile }*/
/*{ dg-options "-std=gnu99 -Wformat"}*/
#include format.h
void vbar(char **sp, wchar_t **lsp) { ... }
```

This test is for the C front-end of GCC. The directive `dg-do` instructs DejaGnu to only compile the function `vbar`. Other directives can run other tasks (*e.g.*, preprocess, assemble, link, and run) on this test and combine these tasks. The directive `dg-options` instructs DejaGnu to override the default option values with the assignments that follow. In this example, the code will be compiled according to two options: "`-std=gnu99`" (uses ANSI C dialect) and "`-Wformat`" (checks format of string arguments of several functions). DejaGnu determines test verdicts by matching specified regular expressions with the outputs of test runs.

Instrumentation. It is important to distinguish *input option* from *feature variable*. Input options correspond to the configuration parameters passed to the system, typically on the command line (*e.g.*, "`-Wformat`") whereas feature variables are the program variables that reflect these options in code (*e.g.*, "`warn_format`"). To enable execution of `SPLat` on GCC (see Figure 3, line 10), we instrumented GCC to monitor all accesses to feature variables by wrapping the uses of accessor macros with a call to a function we defined. These macros are automatically generated from option definition files [1]. The execution of each test on instrumented GCC produces a log file containing the values that feature variables assume during execution.

Variables-to-option map construction. To determine new input vectors from variable accesses we constructed a map to express the correspondence between feature variables (with their values) and configuration options. It is important to note that the construction of this map can be automated (see [57]). This map expresses, for example, the correspondence of `warn_format=0` with the option `-Wno-format`. From the log file of accessed variables produced with the test run and the variable-option map, `SPLat` is able to generate new test inputs (*i.e.*, option vectors).

5.7 Threats to Validity

The selection of subjects constitutes an important threat to generalization. To mitigate this threat we considered not only open-source previously-used SPLs but also one real configurable system (GCC) that has been under active development (and testing) for almost three decades.

The selection of tests is another threat to generalization. For SPLs to reduce bias in the selection we did not provide details to the testers on how we would use the tests. For a configurable system like GCC we used all test suites (hundreds of tests) related to the features we tested.

Another threat is the potential high number of variables in incomplete configurations that testers may need to inspect. To mitigate this problem we ran our techniques on large code and observed that many variables appear undefined in several configurations to inspect, confirming our expectations (as discussed in previous paper [39]) that not all variables are accessed in every path. It is also important to note that `SPLif` favors configurations with more undefined variables for inspection (see "$?_c$" in Section 4.3).

Our selection of feature options to analyze in GCC could introduce bias. It is possible that on a larger set of options more configurations would fail, creating a different scenario for `SPLif`. We remain to evaluate how different heuristics (for selecting feature options) influences `SPLif` results.

Finally, another threat to generalization is the assumption that the labeling provided by the user is accurate. `SPLif` allows the user to skip the labeling of a configuration that he/she is not sure of. This problem could also be mitigated by checking the legality of each labeling with the feature model learned until that point.

6. RELATED WORK

Product Line Testing. Testing software product lines is an active area of research [10,12,14,16,26,36,37,39,46,51,53,55]. The main focus of prior work is to optimize test *execution*. Two approaches have been considered: (1) detection of relevant products to test and (2) optimization of execution of sets of products.

For the first part the focus is to find the set of products that a test must be run against. Kim *et al.* [37] proposed a static analysis to compute a conservative approximation for the set of *relevant* products to run a given test. SPLat [39] also computes a sound estimate for the set of relevant products to test but it uses a low-overhead dynamic analysis, specifically execution-driven monitoring [13], to

determine relevant products.

For the second part the focus is to reduce the cost of running a test against the products that must be executed. Kim *et al.*'s shared execution [36] allows sharing the results of computations that are common across different tests, thereby allowing certain results, which under traditional execution would be computed multiple times, to be computed just once. Kästner *et al.* [16] propose to model variability as non-deterministic choices and then using a model checker to run the tests – which shares computations common to different products and avoids the need to enumerate products for a test. Nguyen *et al.* [44] extends that work by applying previous technique [16] to applications that build on top of plugins. `SPLif` can benefit from test execution optimization by computing its results more quickly. Moreover, `SPLif`'s output could be used to guide the selection of subsequent configurations for execution.

Qu *et al.* [46] focus on regression testing of evolving configurable software systems. They present an empirical study about the impact of configuration selection heuristics used in regression testing on fault-detection capability. Their results highlight that a number of bugs may be missed if certain configurations are not tested and that prioritizing configurations allows for more effective testing. It is natural to consider the context of regression testing for applying `SPLif`; this context would allow failing and passing runs across different program versions to be compared and analyzed to more accurately identify causes of test failures and illegal configurations.

Uzuncaova's approach [55] addresses the test input generation problem for product lines. The approach generates input for a product by *augmenting* previously generated inputs for other products. Test generation techniques can enhance the usefulness of `SPLif`.

Al-Hajjaji *et al.* [6] propose a technique to speedup sampling based on configuration dissimilarity. The rationale is that similar products are likely to contain the same defects. Although `SPLif` uses similarity to rank configurations, the goal is different. We plan to investigate how dissimilarity can improve `SPLif` even further.

Feature Model Extraction and Inference. There is a large body of work on inferring/extracting feature models [5,7,18,21,32,42,47,50,56,57] that include: static analysis to extract feature dependencies from code, information retrieval and data mining, evolutionary search, and algorithms based on propositional logic. She *et al.* [50], Rabkin and Katz [47], and Xu *et al.* [57] use static analysis to extract feature dependencies from code. Alves *et al.* [7] and Davril *et al.* [21] use information retrieval/data mining. Lopez-Herrejon *et al.* [42] use evolutionary search. Haslinger *et al.* [32] provide custom algorithms, they assume that the user provides a list of all legal configurations. Czarnecki and Wasowski [18] extract feature models from propositional formulas. Acher *et al.* [5] synthesize feature models by merging sets of product descriptions. Weston *et al.* [56] proposed a guided process to generate feature models, based on natural-language requirements documents, and represented these models in a way which details their semantic composition. None of the above cited works exploit tests and their executions. The techniques above can complement `SPLif` by making initial feature models more complete; hence reducing effort in user inspection.

Recent techniques have been proposed to analyze and validate feature models. Segura *et al.* [49] propose using metamorphic testing for the automated generation of test data for the analyses of feature models. Henard *et al.* [33] propose an automated approach to find and fix inconsistencies between system and re-engineered feature model. Any further development to improve feature models will benefit `SPLif`.

Fault Localization. Suspiciousness metrics have been proposed to rank code entities in terms of their likelihood of containing faults [2–4,19,22,35,48]. Tarantula [35] is a tool that marks a statement as possibly faulty if it is primarily executed by failing runs than by passing runs. These approaches are unaware of configurations.

To deal with configurations, Zhang and Zhang [62] and Ghandehari *et al.* [31] independently explored the same idea: given a pair of test and failing configurations they build new configurations in an attempt to better localize failure causes, *i.e.* options that provoke failure on given tests. They use suspiciousness metrics similar to those used in previous work. `SPLif` also discovers faulty configurations, but is aware of features constraints. In fact, it learns feature constraint during the classification process.

Configuration Troubleshooting. ConfDiagnoser [60] and ConfSuggester [61] propose a technique to troubleshoot configuration errors caused by configurable systems' evolution. They use dynamic profiling, execution trace comparison, and static analysis to link the undesired behavior to its root cause, a configuration option whose value can be changed to produce desired behavior from the new software version. Garvin *et al.* [27] try to predict future failures in configurable system based on the history of failures; they use configuration similarity to make this prediction. Swanson *et al.* [54] builds on Garvin *et al.*'s approach; they propose an automated approach to reconfigure configurable systems and hence avoid failures. These techniques complements `SPLif`, whose focus is on classifying tests and configurations for inspection.

7. CONCLUSIONS

Software Product Lines (SPLs) are an important design and implementation paradigm for controlling variability in families of related software products. Testing SPLs is important as the volume of research on that topic indicates. Prior research makes the assumption that SPLs come equipped with complete, formally specified feature models. Unfortunately, this assumption does not always hold.

We presented `SPLif`, a new approach for testing SPLs with incomplete feature models, or even no feature model at all. `SPLif` helps the user prioritize failing tests and configurations for inspection. Our experiments showed that `SPLif` is promising and can scale to large systems, such as GCC. In the near future, we plan to apply `SPLif` to other large configurable systems and to optimize execution by leveraging the similarities across multiple similar states [20,36,44].

Additional information about `SPLif` (*e.g.*, experimental data, instructions on how to reproduce results, etc.) can be found at `http://pan.cin.ufpe.br/splif`.

Acknowledgements. We thank Mateus Borges for the help with GCC. Sabrina is supported by the FACEPE fellowship BPG-0675-1.03/09. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-0845628, CCF-1012759, CCF-1212683, CCF-1319688, and CCF-1439957, and the Brazilian CNPq research agency under Grant No. 457756/2014-4.

8. REFERENCES

[1] . Option definition files. https://gcc.gnu.org/onlinedocs/gccint/Options.html.

[2] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *ASE*, 2009.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC*, pages 39–46, 2006.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *MUTATION*, 2007.

[5] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire. On extracting feature models from product descriptions. In *VaMoS*, pages 45–54, 2012.

[6] M. Al-Hajjaji, T. Thüm, J. Meinicke, M. Lochau, and G. Saake. Similarity-based Prioritization in Software Product-line Testing. In

SPLC, pages 197–206, 2014.

[7] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *SPLC*, pages 67–76, 2008.

[8] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. Efficient synthesis of feature models. In *SPLC*, pages 106–115, 2012.

[9] S. Apel and D. Beyer. Feature cohesion in software product lines: an exploratory study. In *ICSE*, pages 421–430, 2011.

[10] S. Apel, A. von Rhein, P. Wendler, A. Groblinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *ICSE*, pages 482–491, 2013.

[11] D. S. Batory. Feature-oriented programming and the AHEAD tool suite. In *ICSE*, pages 702–703, 2004.

[12] P. Borba, M. B. Cohen, A. Legay, and A. Wasowski. Analysis, Test and Verification in the Presence of Variability (Dagstuhl Seminar 13091). *Dagstuhl Reports*, 3(2):144–170, 2013.

[13] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[14] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *SPLC*, pages 241–255, 2010.

[15] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *PLDI*, pages 197–208, 2013.

[16] C.Kastner, A.Rhein, S.Erdweg, J.Pusch, S.Apel, T.Rendel, and K.Ostermann. Toward variability-aware testing. FOSD'12, pages 1–8, 2012.

[17] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley, 2000.

[18] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC*, pages 23–34, 2007.

[19] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP*, 2005.

[20] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. *IEEE TSE*, 34(5):597–613, 2008.

[21] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *ESEC/FSE*, pages 290–300, 2013.

[22] J. C. de Campos, R. Abreu, G. Fraser, and M. d'Amorim. Entropy-based Test Generation for Improved Fault Localization. In *ASE*, pages 257–267, 2013.

[23] DejaGnu. GNU Testing Framework. gnu.org/s/dejagnu.

[24] Instructions to students on generating new tests for SPLs. http://www.cin.ufpe.br/~sfs/splif/experiments.html.

[25] FEST. Fixtures for Easy Software Testing. https://code.google.com/p/fest/.

[26] B. Garvin and M. Cohen. Feature interaction faults revisited: An exploratory study. In *ISSRE*, pages 90–99, 2011.

[27] B. Garvin, M. Cohen, and M. B. Dwyer. Failure avoidance in configurable systems through feature locality. In *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 266–296. 2013.

[28] GCC. GNU Compiler Collection. gcc.gnu.org.

[29] GCC Documentation. Options That Control Optimization. gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options.

[30] GCC Options. GCC Options. https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html.

[31] L. Ghandehari, Y. Lei, D. Kung, R. Kacker, and R. Kuhn. Fault localization based on failure-inducing combinations. In *ISSRE*, pages 168–177, Nov 2013.

[32] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. On extracting feature models from sets of valid feature combinations. In *FASE*, pages 53–67, 2013.

[33] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. Le Traon. Towards automated testing and fixing of re-engineered feature models. In *ICSE*, pages 1245–1248, 2013.

[34] Human-resource management system. 101Companies. http://101companies.org/wiki/@system.

[35] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[36] C. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *ISSRE*, pages 221–230, 2012.

[37] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *AOSD*, pages 57–68, 2011.

[38] C. H. P. Kim, E. Bodden, D. S. Batory, and S. Khurshid. Reducing Configurations to Monitor in a Software Product Line. In *RV*, pages 285–299, 2010.

[39] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *ESEC/FSE*, pages 257–267, 2013.

[40] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, pages 216–226, 2014.

[41] P. Lengauer, V. Bitto, F. Angerer, P. Grünbacher, and H. Mössenböck. Where has all my memory gone?: Determining memory characteristics of product variants using virtual-machine-level monitoring. In *VaMoS*, pages 1–8, 2013.

[42] R. E. Loepz-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *SSBSE*, pages 168–182, 2012.

[43] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *GPCE*, pages 10–24, 2001.

[44] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ICSE*, pages 907–918, 2014.

[45] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[46] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, pages 75–86, 2008.

[47] A. Rabkin and R. Katz. Static extraction of program configuration options. In *ICSE*, pages 131–140, 2011.

[48] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[49] S. Segura, R. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *ICST*, pages 35–44, April 2010.

[50] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE*, pages 461–470, 2011.

[51] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *FASE*, pages 270–284, 2012.

[52] Software Engineering Institute (SEI) SPL website. http://www.sei.cmu.edu/productlines/.

[53] C. Song, A. Porter, and J. S. Foster. itree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees. In *ICSE*, pages 903–913, 2012.

[54] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In *FSE*, pages 377–388, 2014.

[55] E. Uzuncaova. *Efficient Specification-based Testing Using Incremental Techniques*. PhD thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, Dec. 2008.

[56] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC*, pages 211–220, 2009.

[57] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *SOSP*, pages 244–259, 2013.

[58] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *PLDI*, pages 283–294, 2011.

[59] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *FSE*, 1999.

[60] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *ICSE*, pages 312–321, San Francisco, CA, USA, May 2013.

[61] S. Zhang and M. D. Ernst. Which Configuration Option Should I Change? In *ICSE*, pages 152–163, 2014.

[62] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *ISSTA*, pages 331–341, 2011.