SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems

Chang Hwan Peter Kim
University of Texas
Austin, TX, USA

Darko Marinov[1,2]
[1]University of Illinois
and [2]Groupon, Inc., USA

Sarfraz Khurshid   Don Batory
University of Texas
Austin, TX, USA

Sabrina Souto   Paulo Barros   Marcelo d'Amorim
Federal University of Pernambuco
Recife, PE, Brazil

ABSTRACT

Many programs can be configured through dynamic and/or static selection of configuration variables. A *software product line (SPL)*, for example, specifies a family of programs where each program is defined by a unique combination of features. Systematically testing SPL programs is expensive as it can require running each test against a combinatorial number of configurations. Fortunately, a test is often independent of many configuration variables and need not be run against every combination. Configurations that are not required for a test can be pruned from execution.

This paper presents *SPLat*, a new way to dynamically prune irrelevant configurations: the configurations to run for a test can be determined during test execution by monitoring accesses to configuration variables. `SPLat` achieves an optimal reduction in the number of configurations and is lightweight compared to prior work that used static analysis and heavyweight dynamic execution. Experimental results on 10 SPLs written in Java show that `SPLat` substantially reduces the total test execution time in many cases. Moreover, we demonstrate the scalability of `SPLat` by applying it to a large industrial code base written in Ruby on Rails.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Verification

Keywords

Software Product Lines, Configurable Systems, Software Testing, Efficiency

1. INTRODUCTION

Many programs can be configured through selection of configuration variables. A *software product line (SPL)*, for example, specifies a family of programs where each program is defined by a unique combination of features (increments in program functionality). Many codebases that power modern websites are also highly configurable. For instance, the code behind the `groupon.com` website has over 170 boolean configuration variables and can, in theory, be deployed in over 2^{170} different configurations.

Systematically testing configurable systems and SPLs is challenging because running each test can, in principle, require many actual executions—one execution for each possible configuration or feature combination.[1] Thus, one test does not simply encode one execution of a program; the cost of running a test suite is proportional to the number of tests times the number of configurations. Current techniques for handling this combinatorial problem can be divided into sampling and exhaustive exploration. Sampling uses a random or sophisticated selection of configurations, *eg* pair-wise coverage [9]. However, such selections can run a test on several configurations for which the test executions are provably equivalent (thus only increasing the test time without increasing the chance to find bugs), or it can fail to examine configurations that can expose bugs [2]. Exhaustive exploration techniques consider all configurations, but use optimization techniques to prune redundant configurations that need not be explored. [2, 10, 19, 20, 22, 33]. Such works use either static analysis [20] or heavyweight dynamic analysis based on model checking [2, 10, 19, 22, 33].

This paper presents *`SPLat`*, a new lightweight technique to reduce the cost of systematic testing of SPLs and highly configurable systems. Because a test typically exercises a subset of the code base, it is likely that some of the features will never even be encountered during the test execution, no matter how the test is executed. Combinations of such unreachable features yields many runs of a test that have the same *trace* or sequence of bytecode instructions executed by the test. `SPLat` determines for each test the set of unique traces and hence the smallest set of configurations to run. Specifically, let $p_1 \ldots p_k$ (for $k \geq 1$) be the configuration variables for a program. Each p_i takes a value from a finite domain D_i—in the case of SPLs, each variable takes a boolean value that represents whether the feature is selected or not. Let c and c' be two different configurations to run on a test t, and let τ and τ' be their traces. In both runs, t

[1]For ease of exposition, we use the terms "configuration", "feature combination" and "program" interchangeably.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491459

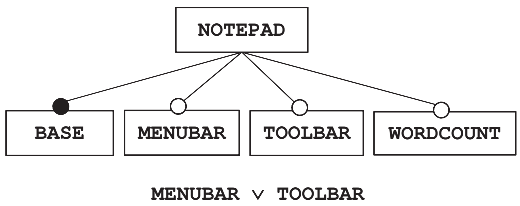

Figure 1: Feature Model

fixes the input values for non-configuration variables. Configuration c' is unnecessary to execute if c has been executed and $\tau' = \tau$. The set of all configurations with unique traces forms a *revealing subdomain* for t [39].

Our insight is that the configurations to run against a test can be determined *during* test execution, rather than using an up-front static analysis. SPLat achieves an optimal reduction in the number of configurations, *ie* for any test, SPLat runs only configurations that have a unique trace. Experimental results show that SPLat yields a reduction in testing time that is proportional to the reduction in the number of configurations. Our insight into pruning configurations was inspired by the Korat algorithm for test-input generation [4], which introduced the idea of *execution-driven pruning* for solving data-structure invariants written as imperative code.

SPLat supports constraints among configuration variables, which defines the valid configurations. For a SPL, these constraints are expressed through a *feature model* [18] that (1) provides a hierarchical arrangement of features and (2) defines allowed configurations. SPLat uses SAT to prune invalid configurations and in tandem uses execution-driven pruning to further remove the valid configurations that are unnecessary for execution of each test.

SPLat is effective because it monitors the accesses of configuration variables during test execution. Monitoring is lightweight—both in terms of its execution overhead and in terms of its implementation effort. We developed two implementations, one for Java and one for Ruby on Rails. The Java implementation of SPLat leveraged the publicly available Korat code [25]. The Ruby on Rails implementation of SPLat was developed from scratch and took only two days to implement while being robust enough to run against a large, industrial codebase at Groupon, Inc.

This paper makes the following contributions:

- **Lightweight analysis of configurable programs:** We introduce the idea of lightweight monitoring for highly configurable systems to speed up test execution. SPLat instantiates this idea and can be easily implemented in different run-time environments.
- **Implementation:** We describe two implementations of SPLat, one for Java and one for Ruby on Rails.
- **Evaluation:** We evaluate SPLat on 10 Java SPLs. Experimental results show that SPLat effectively identifies relevant configurations with a low overhead. We also apply SPLat on a large configurable system (with over 171KLOC in Ruby on Rails). The system uses over 170 configuration variables and contains over 19K tests (with over 231KLOC in Ruby on Rails). To our knowledge, this is the largest and most complex industrial codebase used in research on testing SPLs [2, 10, 19, 20, 22, 33], and SPLat scales to this size without much engineering effort.

```
class Notepad extends JFrame {
 Notepad() {
  getContentPane().add(newJTextArea());
 }

 void createToolBar() {
  if(TOOLBAR) {
   JToolBar toolBar = new JToolBar();
   getContentPane().add
              ("North", toolBar);
   if(WORDCOUNT) {
    JButton button = new
          JButton("wordcount.gif");
    toolBar.add(button);
   }
  }
 }

 void createMenuBar() {
  if(MENUBAR) {
   JMenuBar menuBar = new JMenuBar();
   setJMenuBar(menuBar);
   if(WORDCOUNT) {
    JMenu menu = new
            JMenu("Word Count");
    menuBar.add(menu);
   }
  }
 }
}
```

(a) Code

```
public void test() {
 Notepad n = new Notepad();
 n.createToolBar();

 // Automated GUI testing
 FrameFixture f = newFixture(n);
 f.show();
 String text = "Hello";
 f.textBox().enterText(text);
 f.textBox().requireText(text);
 f.cleanUp();
}
```

(b) Test

Figure 2: Notepad SPL and Example Test

2. MOTIVATING EXAMPLE

To illustrate the testing process, we use a simple Notepad product line and introduce SPL concepts as needed. Figure 1 shows the feature model of Notepad. This model has two mandatory features—the root NOTEPAD and BASE (non-root mandatory features are indicated by filled-in dots)—and three optional features—MENUBAR, TOOLBAR, and WORDCOUNT (indicated by unfilled dots). A mandatory feature is always true; it is present in every configuration.

An optional feature may be set to `true` or `false`. A *configuration* is an assignment of values to all feature variables. A configuration is *valid* iff it satisfies all constraints the feature model expresses. A feature model can also include additional propositional logic constraints. In this example, every Notepad configuration must have a `MENUBAR` or `TOOLBAR`. For example, assigning `false` to both `TOOLBAR` and `MENUBAR` would violate the disjunction constraint and therefore be invalid. In contrast, assigning `false` to one of these two features and `true` to the other feature is valid.

For the SPLs we consider in this paper, a feature is a boolean variable within the code. Figure 2(a) shows the code for Notepad. `BASE` (clear color) represents the core functionality, which, in this case, corresponds to constructing a `Notepad` with a `JTextArea` that the user types into. `TOOLBAR` (green color) adds a `JToolBar` to the frame. `MENUBAR` (red color) sets a `JMenuBar` against the frame. `WORDCOUNT` (blue color) adds its toolbar icon if the toolbar is present or its menubar item if the menubar is present.

Figure 2(b) shows an example test that instantiates the `Notepad` class and creates a toolbar for it. Note that test does *not* call the `createMenuBar()` method. To be able to execute a test, each variable in the test, except the feature variables, must be given a value.

We use the automated GUI testing framework FEST [12] to run the test. The helper method `newFixture()` is not shown for simplicity. The test execution launches the frame, simulates a user entering some text into the `JTextArea` of the frame, checks that the text area contains exactly what was entered, and closes the frame.

Without analyzing the feature model or the code, this test would need to be run on all 8 combinations of the 3 optional features, to check all potential test outcomes. However, some configurations need not be run. Analyzing the feature model, we note that two configurations are *invalid*: $MTW = 000$ and $MTW = 001$, where M, T, and W stand for `MENUBAR`, `TOOLBAR`, and `WORDCOUNT` respectively. Hence, no more than 6 configurations need to be run.

`SPLat` further reduces that number by dynamically analyzing the code that the test executes. For example, executing the test against the configuration $c := MTW = 100$ executes the same trace as configuration $c' := MTW = 101$. The reason is that the test only calls `createToolBar()`, which is empty in both configurations c and c' since `TOOLBAR` is false in both configurations. Although the code in `createMenuBar()` is different in c and c', the test never executes it. Therefore, having executed c, execution of c' is unnecessary. We will show in Section 3.3 that `SPLat` runs this test for only three configurations (*eg* $MTW = 010$, $MTW = 011$, $MTW = 100$).

3. TECHNIQUE

Given a test for a configurable system, `SPLat` determines all relevant configurations on which the test should be run. Each configuration run executes a unique trace of the test. `SPLat` executes the test on one configuration, observes the values of configuration variables, and uses these values to determine which configurations can be safely pruned. `SPLat` repeats this process until it explores all relevant configurations or until it reaches a specified bound on the number of configurations. We first describe the feature model interface and then the core algorithm.

```
class FeatureVar {...}
class VarAssign { ...
 Map<FeatureVar, boolean> map; ...}
interface FeatureModel {
 Set<Assign> getValid(Assign a);
 boolean isSatisfiable(Assign a);
 boolean isMandatory(FeatureVar v);
 boolean getMandatoryValue(FeatureVar v);
}
```

Figure 3: Feature Model Interface

3.1 Feature Model Interface

Figure 3 shows the code snippet that defines the `FeatureModel` interface. The type `FeatureVar` denotes a feature variable. A `VarAssign` object encodes an assignment of boolean values to feature variables. An assignment can be *complete*, assigning values to all the features, or *partial*, assigning values to a subset of the features. A complete assignment is *valid* if it satisfies the constraints of the feature model. A partial assignment is *satisfiable* if it can be extended to a valid complete assignment.

The `FeatureModel` interface provides queries for determining the validity of feature assignments, obtaining valid configurations, and checking if particular informed features are mandatory. Given an assignment α, the method `getValid()` returns the set of all complete assignments that (1) agree with α on the values of feature variables in α and (2) assign the values of the remaining feature variables to make the complete assignment valid. If the set is not empty for α, we say that α is *satisfiable*; the method `isSatisfiable()` checks this. The method `isMandatory()` checks if a feature is mandatory according to the feature model and the method `getMandatoryValue()` returns the mandatory value for the informed feature. We build on a SAT solver (SAT4J [34]) to implement these feature model operations.

3.2 Main Algorithm

Figure 4 lists the `SPLat` algorithm. It takes as input a test `t` for a configurable system and a feature model `fm`. To enable exploration, the algorithm maintains a `state` that stores the values of feature variables (line 2) and a `stack` of feature variables that are read during the latest test execution (line 1). `SPLat` performs a mostly stateless exploration of paths: it does not store, restore, or compare program states as done in stateful model checking [2, 10, 19, 22, 33]; instead, `SPLat` stores only the feature decisions made along one path and re-executes the code to explore different program paths, which corresponds to valid and dynamically reachable configurations. To that end, `SPLat` needs to be able to set the values of feature variables, to observe the accesses to feature variables during a test run, and to re-execute the test from the beginning.

The algorithm first initializes the values of feature variables (lines 8–13) using the feature model interface. Mandatory features are set to the only value they can have, and optional features are initially set to `false`. Note that initial assignment may be invalid for the given feature model. For example, initially setting feature variables to `false` would violate the constraint in our Notepad example. We describe later how `SPLat` enforces satisfiability *during* execution (in line 50). It adjusts the assignment of values to feature variables *before* test execution gets to exercise code based on an invalid configuration. Such scenario could potentially lead to a "false alarm" test failure as opposed to revealing an actual bug in the code under test. Note that the calls to `state.put()` both in the initialization block and elsewhere not only map a feature variable to a boolean value in the state maintained by `SPLat` but also set the value of the feature variable referred to by the code under test.

`SPLat` then instruments (line 16) the code under test to observe feature variable reads. Conceptually, for each read of an optional feature variable (*eg* reading variable `TOOLBAR` in the code `if(TOOLBAR)` from Figure 2), `SPLat` replaces the read with a call to the `notifyFeatureRead()` method shown in Figure 4. The reads are statically instrumented so that they can be intercepted just before they happen during test execution. Mandatory feature variable reads need not be instrumented because the accessed values remain constant for all configurations.

`SPLat` next runs the test (line 22). The test execution calls the method `notifyFeatureRead()` whenever it is about to read a feature variable. When that happens, `SPLat` pushes the feature variable being read on the `stack` if it is not already there, effectively recording the order of the first reads of variables. This `stack` enables backtracking over the values of read feature variables. An important step occurs during the call to `notifyFeatureRead()` (line 50). The initial value assigned to the reached feature variable may make the configuration unsatisfiable. More precisely, at the beginning of the exploration, `SPLat` sets an optional feature value to `false`. When the code is about to read the optional feature, `SPLat` checks whether the `false` value is consistent with the feature model, *ie* whether the *partial* assignment of values to feature variables on the `stack` is satisfiable for the given feature model. If it is, `SPLat` leaves the feature as is. If not, `SPLat` changes the feature to `true`.

Note that updating a feature variable to `true` *guarantees* that the new partial assignment is satisfiable. The update occurs *before* execution could have observed the old value which would make the assignment unsatisfiable. The reason why this change of value keeps the assignment satisfiable follows from the overall correctness of the `SPLat` algorithm: it explores only satisfiable partial assignments (line 36), and it checks if the assignment is satisfiable in *every* variable read (line 50); thus, if a partial assignment was satisfiable considering all features on the `stack`, then it must be possible to extend that assignment with at least one value for the new feature that was not on the `stack` but is being added. If the variable associated with the new feature stores `false` at the moment execution accesses that variable, and if the partial assignment including that feature variable is *not* satisfiable, then we can change the value to `true` (line 51). Recall that optional feature variables are initialized to `false`.

After finishing one test execution for one specific configuration, `SPLat` effectively covers a set of configurations. This set can be determined by enumerating every complete assignment that (1) has the same values as the partial assignment specified by variables `state` and `stack` (lines 23–24) and (2) is valid according to the feature model (line 26).

`SPLat` then determines the next configuration to execute by backtracking on the `stack` (lines 28–39). If the last read feature has value `true`, then `SPLat` has explored both values of that feature, and it is popped off the stack (lines 30–33). If the last read feature has value `false`, then `SPLat` has explored only the `false` value, and the feature should be set to `true` (lines 33–38). Another important step occurs now (line 36). While the backtracking over the `stack` found a partial assignment to explore, it can be the case that this assignment is not satisfiable for the feature model. In that case, `SPLat` keeps searching for the next satisfiable assignment to run. If no such assignment is found, the `stack` becomes empty, and `SPLat` terminates.

```
1  Stack<FeatureVar> stack;
2  Map<FeatureVar, Boolean> state;
3
4  // input, shared with instrumented code
5  FeatureModel fm;
6
7  void SPLat(Test t) {
8   // Initialize features
9   state = new Map();
10  for (FeatureVar f: fm.getFeatureVariables())
11   state.put(f, fm.isMandatory(f) ?
12               fm.getMandatoryValue(f) :
13               false);
14
15   // Instrument the code under test
16   instrumentOptionalFeatureAccesses();
17
18   do {
19
20    // Repeatedly run the test
21    stack = new Stack();
22    t.runInstrumentedTest();
23    VarAssign pa =
24     getPartialAssignment(state, stack);
25    print("configs covered: ");
26    print(fm.getValid(pa));
27
28    while (!stack.isEmpty()) {
29     FeatureVar f = stack.top();
30     if (state.get(f)) {
31      state.put(f, false); // Restore
32      stack.pop();
33     } else {
34      state.put(f, true);
35      pa = getPartialAssignment(state, stack);
36      if (fm.isSatisfiable(pa))
37       break;
38     }
39    }
40
41   } while (!stack.isEmpty());
42 }
43
44 // called-back from test execution
45 void notifyFeatureRead(FeatureVar f) {
46  if (!stack.contains(f)) {
47   stack.push(f);
48   VarAssign pa =
49    getPartialAssignment(state, stack);
50   if (!fm.isSatisfiable(pa))
51    state.put(f, true);
52  }
53 }
```

Figure 4: SPLat Algorithm

3.3 Example Run

We demonstrate `SPLat` on the example from Figure 2. According to the feature model (Figure 1), `NOTEPAD` and

BASE are the only mandatory features and are set to true. The other three feature variables are optional and therefore SPLat instruments their reads (Figure 2(a), lines 7, 11, 20, and 23). Conceptually, the exploration starts from the configuration $MTW = 000$.

When the test begins execution, notifyFeatureRead() is first called when TOOLBAR is read. TOOLBAR is pushed on the stack, and because its assignment to false is satisfiable for the feature model, its value remains unchanged (*ie* stays false as initialized). Had the feature model required TOOLBAR to be true, the feature's value would have been set to true at this point.

With TOOLBAR set to false, no other feature variables are read before the test execution finishes. (In particular, WORDCOUNT on line 11 is not read because that line is not executed when TOOLBAR is false.) Therefore, this one execution covers configurations $MTW = -0-$ where $-$ denotes a "don't care" value. However, configurations $MTW = 00-$ are invalid for the given feature model, so this one execution covers two valid configurations where TOOLBAR is false and MENUBAR is true (MTW=10-). Note that even though the value of WORDCOUNT does not matter here, it is given a value nonetheless for an execution because in an execution, each variable must have a concrete value. So let us say MTW=100 here.

SPLat next re-executes the test with TOOLBAR set to true, as it is satisfiable for the feature model. WORDCOUNT is encountered this time, but it can remain false, and the execution completes, covering MTW=-10 (again, for an execution, all variables need to be set, so let us say that MTW=010 is what actually executes). SPLat then sets WORDCOUNT to true, and the execution completes, covering MTW=-11 (let us say MTW=011 was used). SPLat finally pops off WORDCOUNT from the stack because both its values have been explored, and pops off TOOLBAR for the same reason, so the exploration finishes because the stack is empty. In summary, the test's first execution covers MTW=10- (MTW=100 is executed), second execution covers MTW=-10 (MTW=010 is executed) and third execution covers MTW=-11 (MTW=011 is executed). Therefore, the technique covers all 6 valid configurations by executing just three configurations.

3.4 Reset Function

While a stateless exploration technique such as SPLat does not need to store and restore program state in the middle of execution like a stateful exploration technique does, the stateless exploration does need to be able to restart a new execution from the initial program state unaffected by the previous execution. Restarting an execution with a new runtime (*eg* spawning a new *Java Virtual Machine (JVM)* in Java) is the simplest solution, but it can be both inefficient and unsound. It is inefficient because even without restarting the runtime, the different executions may be able to share a runtime and still have identical initial program states, *eg* if the test does not update any static variables in the JVM state. It can be unsound because a new runtime may not reset the program state changes made by the previous executions (*eg* previous executions having sent messages to other computers or having performed I/O operations such as database updates). We address these issues by sharing the runtime between executions and requiring the user to provide a *reset function* that can be called at the beginning of the test.

Our sharing of the runtime between executions means that settings that would normally be reset automatically by creating a new runtime must now be manually reset. For example, Java static initializers must now be called from the reset function because classes are loaded only once. However, we believe that the benefit of saving time by reusing the runtime outweighs the cost of this effort, which could be alleviated by a program analysis tool. Moreover, for the Groupon code used in our evaluation, the testing infrastructure was already using the reset function (developed independently and years before this research); between any test execution, the state (of both memory and database) is reset (by rolling back the database transaction from the previous test and overwriting the state changes in the tearDown and/or setUp blocks after/before each test).

3.5 Potential Optimization

The algorithm in Figure 4 is not optimized in how it interfaces with the feature model. The feature model is treated as a blackbox, read-only artifact that is oblivious to the exploration state consisting of the state and stack. Consequently, the isSatisfiable() and getValid() methods are executed as if the exploration state was completely new every time, even if it just incrementally differs from the previous exploration state. For example, when running the test from Figure 2, SPLat asks the feature model if $MTW = -1-$ is satisfiable (line 36 of the SPLat algorithm) after the assignment $MTW = -0-$. The feature model replies true as it can find a configuration with the feature TOOLBAR set to true. Then when WORDCOUNT is encountered while TOOLBAR=true, SPLat asks the feature model if the assignment $MTW = -10$ (TOOLBAR=true *and* WORDCOUNT=false) is satisfiable (line 50 of the SPLat algorithm). Note that the feature model is not aware of the similarity between the consecutive calls for $MTW = -1-$ and $MTW = -10$. But if it were, it would only have to check the satisfiability of WORDCOUNT=false.

The change to the algorithm to enable this synchronization between the exploration state and the feature model is simple: every time a feature variable is pushed on the stack, constrain the feature model with the feature's value, and every time a feature variable is popped off the stack, remove the corresponding feature assignment from the feature model. A feature model that can be updated implies that it should support incremental solving, *ie* a feature model should not have to always be solved in its entirety. Our current SPLat tool for Java does not exploit incremental solving, meaning that the tool has not reached the limits of the underlying technique and can be made even faster.

3.6 Implementation

We implemented two versions of SPLat, one for Java and one for Ruby on Rails. We selected these two languages motivated by the subject programs used in our experiments (Section 4).

For Java, we implemented SPLat on top of the publicly available Korat solver for imperative predicates [25]. Korat already provides code instrumentation (based on the BCEL library for Java bytecode manipulation) to monitor field accesses, and provides basic backtracking over the accessed fields. The feature variables in our Java subjects were already represented as fields. The main extension for `SPLat` was to integrate Korat with a SAT solver for checking satisfiability of partial assignments with respect to feature models. As mentioned earlier, we used SAT4J [34].

For Ruby on Rails, we have an even simpler implementation that only monitors accesses to feature variables. We did not integrate a SAT solver, because the subject code did not have a formal feature model and thus we treated all combinations of feature variables as valid.

4. EVALUATION

Our evaluation addresses the following research questions:

RQ1 How does `SPLat`'s efficiency compare with alternative techniques for analyzing SPL tests?

RQ2 What is the overhead of `SPLat`?

RQ3 Does `SPLat` scale to real code?

In Section 4.1, we compare `SPLat` with related techniques using 10 SPLs. In Section 4.2, we report on the evaluation of `SPLat` using an industrial configurable system implemented in Ruby on Rails.

4.1 Software Product Lines

We evaluate our approach with 10 SPLs listed in Table 1.[2] Note that most of these subjects are configurable programs that have been converted into SPLs. A brief description for each is below:

- **101Companies** [16] is a human-resource management system. Features include various forms to calculate salary and to give access to the users.
- **Email** [14] is an email application. Features include message encryption, automatic forwarding, and use of message signatures.
- **Elevator** [31] is an application to control an elevator. Features include prevention of the elevator from moving when it is empty and a priority service to the executive floor.
- **GPL** [29] is a product line of graph algorithms that can be applied to a graph.
- **JTopas** [17] is a text tokenizer. Features include support for particular languages such as Java and the ability to encode additional information in a token.
- **MinePump** [26] simulates an application to control water pumps used in a mining operation. Features include sensors for detecting varying levels of water.
- **Notepad** [21] is a GUI application based on Java Swing that provides different combinations of end-user features, such as windows for saving/opening/printing files, menu and tool bars, etc. It was developed for a graduate-level course on software product lines.
- **Prevayler** [27] is a library for object persistence. Features include the ability to take snapshots of data, to compress data, and to replicate stored data.
- **Sudoku** [32] is a traditional puzzle game. Features include a logical solver and a configuration generator.

[2]All subjects except 101Companies have been used in previous studies on testing/analyzing SPLs, including GPL by [2, 5], Elevator, Email, MinePump by [2], JTopas by [6], Notepad by [21, 20], XStream by [11, 35] Prevayler by [36, 1], and Sudoku by [1].

Table 1: Subject SPLs

SPL	*Features*	*Confs*	*LOC*
101Companies	11	192	2,059
Elevator	5	20	1,046
Email	8	40	1,233
GPL	14	73	1,713
JTopas	5	32	2,031
MinePump	6	64	580
Notepad	23	144	2.074
Prevayler	5	32	2,844
Sudoku	6	20	853
XStream	7	128	14,480

- **XStream** [28] is a library for (de)serializing objects to XML (and from it). Features include the ability to omit selected fields and to produce concise XML.

Table 1 shows the number of optional features (we do not count the mandatory features because they have constant values), the number of valid configurations, and the code size for each subject SPL. More details of the subjects and results are available at our website [23].

4.1.1 Tests

We prepared three different tests for each subject SPL. The first test, referred as `LOW`, represents an optimistic scenario where the test needs to be run only on a small number of configurations. The second test, referred as `MED` (for MEDIUM), represents the average scenario, where the test needs to be run on some configurations. The third test, referred as `HIGH`, represents a pessimistic scenario, where the test needs to be run on most configurations.

To prepare the LOW, MED, and HIGH tests, we modified existing tests, when available, or wrote new tests because we could not easily find tests that could be used without modification. Because some subjects were too simple, tests would finish too quickly for meaningful time measurement if test code only had one sequence of method calls. Therefore, we used loops to increase running times when necessary. Each test fixes all inputs except the feature variables. The tool, test suites and subjects are available on the project website [23].

4.1.2 Comparable Techniques

We compared `SPLat` with different approaches for test execution. We considered two naïve approaches that run tests against *all* valid configurations: NewJVM and ReuseJVM. The ***NewJVM*** approach spawns a new JVM for each distinct test run. Each test run executes only *one* valid configuration of the SPL. It is important to note that the cost of this approach includes the cost of spawning a new JVM. The ***ReuseJVM*** approach uses the same JVM across several test runs, thus avoiding the overhead of repeatedly spawning JVMs for each different test and configuration. This approach requires the tester to explicitly provide a reset function (Section 3.4). Because the tester likely has to write a reset function anyway, we conjecture that this approach is a viable alternative to save runtime cost. For example, the tester may already need to restore parts of the state stored outside the JVM such as files or database.

We also compared `SPLat` with a simplified version of a previously proposed static analysis [20] for pruning configurations. Whereas [20] performs reachability analysis, control-flow and data-flow analyses, the simplified version, which we call ***SRA*** (Static Reachability Analysis), only performs the reachability analysis to determine which configurations are reachable from a given test and thus can be seen as the static analysis counterpart to `SPLat`. SRA builds a call graph using inter-procedural, context-insensitive, flow-insensitive, and path-insensitive points-to analysis and collects the features syntactically present in the methods of the call graph. Only the valid combinations of these *reachable* features from a test need to be run for that test.

Finally, we compared `SPLat` with an artificial technique that has zero cost to compute the set of configurations on which each test need to run. More precisely, we use a technique that gives the same results as `SPLat` but only counts the cost of executing tests for these configurations, not the cost of computing these configurations. We call this technique ***Ideal***. The overhead of `SPLat` is the difference between the overall cost of `SPLat` explorations and the cost of executing tests for Ideal.

4.1.3 Results

Table 2 shows our results. We performed the experiments on a machine with X86_64 architecture, Ubuntu operating system, 240696 MIPS, 8 cores, with each core having an Intel Xeon CPU E3-1270 V2 at 3.50GHz processor, and 16 GB memory. All times are listed in seconds. Our feature model implementation solves the feature model upfront to obtain all valid configurations; because this solving needs to be done for every feature model (regardless of using `SPLat` or otherwise), and because it takes a fraction of test execution time, we do not include it.

Here is a description of each column in Table 2:

- **Test** refers to one of the three categories of tests described earlier.
- **All Valid** identifies the techniques that run the test against all valid configurations, namely **NewJVM** and **ReuseJVM**. **ReuseJVM** shows time absolutely and as a percentage of **NewJVM** duration.
- Columns under **`SPLat`** details information for `SPLat`:
  - **Confs** shows the number of configurations that `SPLat` runs for a particular test.
  - **SPLatTime** shows the time it takes to run a test using `SPLat`. `SPLat` reuses the same JVM for different executions, like ReuseJVM. The time is shown absolutely and as a percentage of **ReuseJVM** (not **NewJVM**).
  - **IdealTime** shows the time in seconds for running `SPLat` without considering the cost to determine which configurations to run for the test; therefore, this number excludes instrumentation, monitoring, and constraint solving.
  - **Overhead** shows the overhead of `SPLat`, calculated by subtracting **IdealTime** from **SPLatTime**, and dividing it by **IdealTime**.
- Columns under **Static Reachability (SRA)** show results for our static analysis:
  - **Confs** shows the number of configurations reachable with such analysis,
  - **Overhead** shows the time taken to perform the static reachability analysis, and
  - **Time** shows the time taken to run the configurations determined by this analysis.

Efficiency. The **ReuseJVM** column shows that reusing JVM saves a considerable amount of time compared to spawning a new JVM for each test run. For example, for half of the tests, reusing JVM saves over 50% of the time, because running these tests does not take much longer than starting up the JVM. For tests that take considerably longer than starting up the JVM, such saving is not possible.

`SPLat` further reduces the execution time over **ReuseJVM** by determining the reachable configurations. For example, for the `LOW` test for `Notepad`, reusing the JVM takes 34% of the time to run without reusing the JVM, and with `SPLat`, it takes just 2% of the already reduced time. In fact, the table shows that in most cases, as long as `SPLat` can reduce the number of configurations to test (*ie* **Confs** is lower than the total number of configurations), it runs faster than running each configuration (*ie* less than 100% of **ReuseJVM**).

Comparison with Static Reachability Analysis. The static reachability analysis yields less precise results compared to `SPLat`: the number of configurations in the column **Confs** is larger than the number of configurations in the corresponding column for `SPLat`. In fact, for `JTopas`, `Notepad` and `MinePump`, the SRA reports all features as being accessed from the call graph, and therefore reports that all valid configurations have to be tested. For example, for `JTopas`, this is due to its tests invoking the `main` method of the SPL, from which all feature variable accesses may be reached using different input values, which the analysis is insensitive to. For `Notepad`, this is due to the the use of the `FEST` automated GUI testing framework, which relies heavily on reflection. Because the method being invoked through reflection cannot necessarily be determined statically, the analysis yields a conservative result. For `MinePump`, each test happens to exercise a sequence of methods that together reach all feature variable accesses.

Note that the SRA approach first statically determines the configurations to run (which takes the time in column **SRA Overhead**) and afterwards dynamically runs them one by one (which takes the time in column **SRA Time**). Comparing just the static analysis time (**SRA Overhead**) with the `SPLat` overhead (**SPLat Overhead**) shows that SRA has a considerably larger overhead, in some cases two orders of magnitude larger. Although the static analysis overhead can be offset by (re)using the reachable configurations it determines against tests that have the same code base but have different inputs, in general, it would require a very large number of such tests for the approach to have a smaller overhead than `SPLat`. Moreover, comparing just the time to execute the configurations computed by SRA (column **SRA Time** with the time to execute the configurations computed by `SPLat` (column **IdealTime**) shows that SRA again takes longer because SRA computes a higher number of configurations than `SPLat` due to the conservative nature of static analysis.

RQ1. Based on the comparison with **NewJVM**, **ReuseJVM**, and **SRA**, we conclude the following:

> `SPLat` is more efficient than the techniques that run all valid configurations for tests of SPLs or prune reachable configurations using static analysis. Moreover, compared with static analysis, `SPLat` not only gives results faster but also gives more precise results.

Table 2: Experimental Results for Various Techniques

	All Valid			SPLat							Static Reachability (SRA)		
Test	*NewJVM*	*ReuseJVM*		*Confs*		*SPLatTime*		*IdealTime*	*Overhead*		*Confs*	*Overhead*	*Time*
101Companies (192 configs)													
LOW	35.46	2.13	(6%)	32	(16%)	1.64	(77%)	0.72	0.92	(127%)	96	84.04	1.28
MED	49.37	3.90	(7%)	160	(83%)	6.84	(175%)	3.58	3.26	(91%)	192	82.54	3.99
HIGH	283.69	45.26	(15%)	176	(91%)	47.6	(105%)	41.59	6.01	(14%)	192	81.93	45.16
Elevator (20 configs)													
LOW	10.74	5.17	(48%)	2	(10%)	1.33	(25%)	0.71	0.62	(87%)	2	23.29	0.76
MED	50.97	46.65	(91%)	10	(50%)	23.62	(50%)	23.14	0.48	(2%)	20	23.74	46.17
HIGH	62.57	59.48	(95%)	20	(100%)	60.71	(102%)	59.28	1.43	(2%)	20	24.38	60.43
Email (40 configs)													
LOW	40.63	10.74	(26%)	1	(2%)	1.00	(9%)	0.87	0.13	(14%)	1	23.62	0.87
MED	57.56	48.87	(84%)	30	(75%)	36.99	(75%)	37.14	-0.15	(0%)	40	22.81	49.02
HIGH	58.02	48.93	(84%)	40	(100%)	48.96	(100%)	49.26	-0.31	(0%)	40	23.84	49.16
GPL (73 configs)													
LOW	19.21	2.23	(11%)	6	(8%)	0.79	(35%)	0.29	0.49	(168%)	6	104.97	0.30
MED	190.53	171.62	(90%)	55	(75%)	130.87	(76%)	128.52	2.35	(1%)	55	99.41	128.69
HIGH	314.20	285.89	(90%)	70	(95%)	278.77	(97%)	277.48	1.29	(0%)	73	103.52	286.28
JTopas (32 configs)													
LOW	26.59	16.83	(63%)	8	(25%)	6.29	(37%)	4.49	1.80	(40%)	32	86.87	16.44
MED	29.04	18.55	(63%)	16	(50%)	13.16	(70%)	9.71	3.46	(35%)	32	86.87	18.70
HIGH	28.92	18.93	(65%)	32	(100%)	25.31	(133%)	18.43	6.88	(37%)	32	86.87	18.48
MinePump (64 configs)													
LOW	23.71	7.53	(31%)	9	(14%)	3.65	(48%)	1.90	1.75	(91%)	64	22.69	7.49
MED	59.72	14.78	(24%)	24	(37%)	10.43	(70%)	6.26	4.17	(66%)	64	22.38	15.35
HIGH	13.72	5.75	(41%)	48	(75%)	37.80	(657%)	4.81	32.99	(685%)	64	22.18	5.77
Notepad (144 configs)													
LOW	398.22	135.60	(34%)	2	(1%)	3.06	(2%)	2.45	0.61	(24%)	144	80.40	135.47
MED	418.23	156.27	(37%)	96	(66%)	104.95	(67%)	104.91	0.04	(0%)	144	80.62	156.35
HIGH	419.99	153.39	(36%)	144	(100%)	153.11	(99%)	152.16	0.94	(0%)	144	81.29	151.94
Prevayler (32 configs)													
LOW	65.34	40.23	(61%)	12	(37%)	22.49	(55%)	22.8	-0.31	(-1%)	32	205.54	45.39
MED	121.38	96.50	(79%)	24	(75%)	102.49	(106%)	105.86	-3.37	(-3%)	32	214.67	111.37
HIGH	149.08	120.7	(80%)	32	(100%)	127.17	(105%)	131.37	-4.20	(-3%)	32	290.66	135.61
Sudoku (20 configs)													
LOW	51.11	48.10	(94%)	4	(20%)	42.72	(88%)	24.12	18.6	(77%)	10	31.87	24.28
MED	118.14	105.67	(89%)	10	(50%)	58.31	(55%)	54.16	4.15	(7%)	10	31.75	53.67
HIGH	489.60	334.82	(68%)	20	(100%)	316.47	(94%)	332.36	-15.89	(-4%)	20	31.74	338.48
Xstream (128 configs)													
LOW	111.26	30.04	(27%)	2	(1%)	1.57	(5%)	1.08	0.49	(45%)	2	106.50	1.06
MED	105.10	9.04	(8%)	64	(50%)	5.77	(63%)	5.26	0.51	(9%)	64	109.22	5.14
HIGH	101.66	8.68	(8%)	128	(100%)	9.16	(105%)	8.59	0.57	(6%)	128	105.68	8.74

Overhead. Table 2 also shows the overhead that SPLat has over the Ideal technique (column **SPLat Overhead**). The overhead is generally small, except for the LOW tests and tests for several subjects (*eg* JTopas and Mine). The overhead is high for the LOW tests because these tests finish quickly (under 7 seconds, often under 1 second), meaning that instrumentation, monitoring and feature model interaction take a larger fraction of time than they would for a longer executing test. The overhead is high for JTopas because the feature variables are accessed many times because they are accessed within the tokenizing loop. The overhead is high for MinePump because feature accesses and their instrumentation take relatively longer to execute for this particular test as the subject is very small.

SPLat, due to its cost in monitoring feature variables, should not execute a test faster than knowing the reachable configurations upfront and running the test only on those configurations. Thus, the occasional small negative overheads for Email, Prevayler, Sudoku are due to the experimental noise and/or the occasionally observed effect where an instrumented program runs faster than the non-instrumented program. It is important to note that efficiency and overhead are orthogonal. As long as the reduction in time due to the reduction in configurations is larger than the overhead, SPLat saves the overall time. To illustrate, the GPL's LOW test incurs over 168% overhead, but the reduction in configurations outweighs the overhead, and SPLat takes only 35% of running all valid configurations with the same JVM.

RQ2. Based on the discussion about overhead, we conclude the following:

> SPLat can have a large relative overhead for short-running tests, but the overhead is small for long-running tests.

4.2 Configurable Systems

Groupon. Groupon is a company that "features a daily deal on the best stuff to do, see, eat, and buy in 48 countries" (`http://www.groupon.com/about`). *Groupon PWA* is name of the codebase that powers the main groupon.com website. It has been developed for over 4.5 years with contributions from over 250 engineers. The server side is written in Ruby on Rails and has over 171K lines of Ruby code.

Groupon PWA code is highly configurable with over 170 (boolean) feature variables. In theory, there are over 2^{170} different configurations for the code. In practice, only a small number of these configurations are ever used in production, and there is one default configuration for the values of all feature variables.

Groupon PWA has an extensive regression testing infrastructure with several frameworks including Rspec, Cucumber, Selenium, and Jasmine. The test code itself has over 231K lines of Ruby code and additional code in other languages. (It is not uncommon for the test code to be larger than the code under test [37].)

Groupon PWA has over 19K Rspec (unit and integration) tests. A vast majority of these tests run the code only for the default configuration. A few tests run the code for a non-default configuration, typically changing the value for only one feature variable from the default value. Running all the Rspec tests on a cluster of 4 computers with 24 cores each takes under 10 minutes.

`SPLat` Application. We implemented `SPLat` for Ruby on Rails to apply it to Groupon PWA. We did not have to implement the reset function because it was already implemented by Groupon testers to make test execution feasible (due to the high cost of re-starting the system). Moreover, no explicit feature model was present, so feature model constraints did not need to be solved.

We set out to evaluate how many configurations each test could cover if we allow varying the values of all feature variables encountered during the test run. We expected that the number of configurations could get extremely high for some tests to be able to enumerate all the configurations. Therefore, we set the limit on the number of configurations to no more than 16, so that the experiments can finish in a reasonable time. This limit was reached by 2,695 tests. For the remaining 17,008 tests, Table 3 shows the breakdown of how many tests reached a given number of configurations. We can see that the most common cases are the number of configurations being powers of two, effectively indicating that many features are encountered independently rather than nested (as in Figure 2 where the read of `WORDCOUNT` is nested within the block controlled by the read of `TOOLBAR`).

We also evaluated the number of features encountered. It ranges from 1 up to 43. We found 43 is a high number in the absolute sense (indicating that a test may potentially cover 2^{43} different configurations), 43 is also a relatively low number in the relative sense compared to the total of over 170 features. Table 4 shows the breakdown of how many tests reached a given number of feature variables. Note that the numbers of configurations and feature variables may seem inconsistent at a glance, *eg* the number of tests that have 1 configuration is larger than the number of tests than have 0 feature variables. The reason is that some tests force certain values for feature variables such that setting the configuration gets overwritten by the forced value.

Table 3: Reachable Configurations

Configs	*Tests*	*Configs*	*Tests*
1	11,711	2	1,757
3	332	4	882
5	413	6	113
7	19	8	902
9	207	10	120
11	29	12	126
13	6	14	32
15	10	16	349
17	2,695	-	-

Table 4: Accessed Features

Vars	*Tests*	*Vars*	*Tests*	*Vars*	*Tests*
0	11,711	1	1,757	2	1,148
3	1,383	4	705	5	389
6	466	7	323	8	425
9	266	10	140	11	86
12	80	13	34	14	28
15	54	16	62	17	1
19	14	20	260	21	109
22	45	23	19	24	22
25	9	26	2	27	14
28	17	29	6	30	8
31	24	32	6	33	14
34	31	35	11	36	15
37	8	38	2	39	2
40	3	42	2	43	2

In summary, these results show that the existing tests for Groupon PWA can already achieve a high coverage of configurations, but running all the configurations for all the tests can be prohibitively expensive. We leave it as a future work to explore a good strategy to sample from these configurations [8, 9, 30].

RQ3. Moreover, based on the fact that we could run `SPLat` on the codebase as large as Groupon PWA, we conclude the following:

> `SPLat` scales to real, large industrial code. The implementation effort for `SPLat` is relatively low and the number of configurations covered by many real tests is relatively low.

4.3 Threats to Validity

The main threat is to external validity: we cannot generalize our timing results to all SPLs and configurable systems because our case studies may not be representative of all programs, and our tests may be covering an unusual number of configurations. To reduce this threat, we used multiple Java SPLs and one real, large industrial codebase. For SPLs, we designed tests that cover a spectrum of cases from `LOW` to `MED(IUM)` to `HIGH` number of configurations. For the Groupon codebase, we find that most real tests indeed cover a small number of configurations. Our study has the usual internal and construct threats to validity.

We believe that `SPLat` is a helpful technique that can be used in practice to improve SPL testing. An important threat to this conclusion is that our results do not take into account the cost of writing a reset function. Although other techniques that use stateless exploration also require reset functions (*eg* VeriSoft [13]), the cost of developing such functions could affect practicality. NewJVM, ReuseJVM, and `SPLat` all require the state outside of the JVM to be explicitly reset, but only NewJVM automatically resets JVM-specific state by spawning a new JVM for each test run.

5. RELATED WORK

5.1 Dynamic Analysis

Korat. `SPLat` was inspired by Korat [4], a test-input generation technique based on Java predicates. Korat instruments accesses to object fields used in the predicate, monitors the accesses to prune the input space of the predicate, and enumerates those inputs for which the predicate returns true. Directly applying Korat to the problem of reducing the combinatorics in testing configurable systems is not feasible because the feature model encodes a *precondition* for running the configurable system, which must be accounted for. In theory, one could automatically translate a (declarative) feature model into an imperative constraint and then execute it before the code under test, but it could lead Korat to explore the *entire* space of feature combinations (up to 2^N combinations for N features) before *every* test execution. In contrast, `SPLat` exploits feature models while retaining the effectiveness of execution-driven pruning by applying it with SAT in tandem. Additionally, `SPLat` can change the configuration being run during the test execution (line 51 in Figure 4), which Korat did not do for data structures.

Shared execution. Starting from the work of d'Amorim *et al.* [10], there has been considerable ongoing research on saving testing time by sharing computations across similar test executions [2, 3, 7, 10, 15, 19, 22, 24, 33, 38]. The key observation is that repeated executions of a test have much computation in common. For example, Shared Execution [22] runs a test simultaneously against several SPL configurations. It uses a *set* of configurations to support test execution, and splits and merges this set according to the different decisions in control-flow made along execution. The execution-sharing techniques for testing SPLs differ from `SPLat` in that they use *stateful* exploration; they require a dedicated runtime for saving and restoring program state and only work on programs with such runtime support. Consequently, they have high runtime overhead not because of engineering issues but because of fundamental challenges in splitting and merging state sets at proper locations. In contrast, `SPLat` uses *stateless* exploration [13] and never merges control-flow of different executions. Although `SPLat` cannot share computations between executions, it requires minimal runtime support and can be implemented very easily and quickly against almost any runtime system that allows feature variables to be read and set during execution.

Sampling. Sampling exploits domain knowledge to select configurations to test. A tester may choose features for which all combinations must be examined, while for other features, only t-way (most commonly 2-way) interactions are tested [8, 9, 30]. Our dynamic program analysis *safely* prunes feature combinations, while sampling approaches can miss problematic configurations [2].

Spectrum of SPL testing techniques. Kästner *et al.* [19] define a spectrum of SPL testing techniques based on the amount of changes required to support testing. On the one end are black-box techniques that use a conventional runtime system to run the test for each configuration; `NewJVM` is such a technique. On the other end are white-box techniques that extensively change a runtime system to make it SPL-aware; shared execution is such a technique. `SPLat`, which only requires runtime support for reading and writing to feature variables, is a lightweight white-box technique that still provides an optimal reduction in the number of configurations to consider.

5.2 Static Analysis

We previously developed a static analysis that performs reachability, data-flow and control-flow checks to determine which features are relevant to the outcome of a test [20]. The analysis enables one to run a test only on (all valid) combinations of these relevant features that satisfy the feature model. `SPLat` is only concerned with reachability, so even if it encounters a feature whose code has no effect, it will still execute the test both with and without the feature. But a large portion of the reduction in configurations in running a test is simply due to the idea that many of the features are not even reachable. Indeed, as Section 4 shows, `SPLat` determines reachable configurations with much greater precision and is likely to be considerably faster than the static analysis because `SPLat` discovers the reachable configurations during execution. Static analysis may be faster if its cost can be offset against many tests (because it needs only be run once for one test code that allows different inputs), and if a test run takes a very long time to execute (*eg* requiring user interaction). But such situations do not seem to arise often, especially for tests that exercise a small subset of the codebase.

6. CONCLUSIONS

`SPLat` is a new technique for reducing the combinatorics in testing configurable systems. `SPLat` dynamically prunes the space of configurations that each test must be run against. `SPLat` achieves an optimal reduction in the number of configurations and does so in a lightweight way compared to previous approaches based on static analysis and heavyweight dynamic execution. Experimental results on 10 software product lines written in Java show that `SPLat` substantially reduces the total test execution time in most cases. Moreover, our application of `SPLat` on a large industrial code written in Ruby on Rails shows its scalability.

7. ACKNOWLEDGEMENTS

We thank Jeff Ayars, Jack Calzaretta, Surya Gaddipati, Dima Kovalenko, Seth Lochen, Michael Standley, and Victor Vitayaudom for discussions about this work and help with the `SPLat` experiments at Groupon. We also thank Mateus Borges for the help in testing earlier versions of `SPLat`. This material is based upon work partially supported by the US National Science Foundation under Grant Nos. CCF-1213091, CCF-1212683, CNS-0958231, CNS-0958199, CCF-0845628, and CCF-0746856.

8. REFERENCES

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

[2] 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.

[3] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.

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

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

[6] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In *ICSE*, pages 121–130, 2011.

[7] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *ICSE*, pages 335–344, 2010.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *ROSATEA*, pages 53–63, 2006.

[9] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA*, pages 129–139, 2007.

[10] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA*, pages 50–60, 2007.

[11] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *ISSTA*, pages 207–218, 2010.

[12] FEST. Fixtures for Easy Software Testing. `http://fest.easytesting.org/`.

[13] P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[14] R. J. Hall. Fundamental nonmodularity in electronic mail. *ASE Journal*, 12(1):41–79, 2005.

[15] P. Hosek and C. Cadar. Safe Software Updates via Multi-version Execution. In *ICSE*, 2013.

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

[17] Java tokenizer and parser tools. JTopas. `http://jtopas.sourceforge.net/jtopas/index.html`.

[18] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report 90-TR-21, CMU/SE, Nov. 1990.

[19] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *FOSD*, pages 1–8, 2012.

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

[21] 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.

[22] C. H. P. Kim, S. Khurshid, and D. Batory. Shared Execution for Efficiently Testing Product Lines. In *ISSRE*, pages 221–230, 2012.

[23] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat: Evaluation. `http://www.cs.utexas.edu/~chpkim/splat`, 2013.

[24] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Oakland*, 2012.

[25] Korat home page. `http://mir.cs.illinois.edu/korat/`.

[26] J. Kramer. Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1), 1983.

[27] Library for object persistence. *Prevayler*. `www.prevayler.org/`.

[28] Library to serialize objects to XML and back again. XStream. `http://xstream.codehaus.org/`.

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

[30] J. McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, CMU/SEI, Mar. 2001.

[31] M. Plath and M. Ryan. Feature integration using a feature construct. *SCP Journal*, 41(1):53–84, 2001.

[32] Puzzle game. *Sudoku*. `https://code.launchpad.net/~spl-devel/spl/default-branch`.

[33] A. V. Rhein, S. Apel, and F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *JPF Workshop*, 2011.

[34] SAT4J. `http://www.sat4j.org/`.

[35] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA*, pages 69–80, 2009.

[36] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In C. Consel and J. L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.

[37] N. Tillmann and W. Schulte. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. Technical Report MSR-TR-2005-153, Microsoft Research, November 2005.

[38] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *ASPLOS*, pages 193–204, 2009.

[39] E. Weyuker and T. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE TSE*, 6(3):236–246, 1980.