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

Chang Hwan Peter Kim
University of Texas
Austin, TX

Darko Marinov
University of Illinois

Sarfraz Khurshid
University of Texas

Don Batory
University of Texas
Austin, TX

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

This paper presents SPLat, a novel technique that dynamically prunes such configurations. Our key insight is that the configurations that must be run for a test can be determined during test execution by monitoring accesses to configuration parameters. SPLat achieves an optimal reduction in the number of configurations and provides a lightweight, more widely applicable technique compared to previous approaches based on static analysis and heavyweight dynamic execution. Experimental results on seven SPLs written in Java show that SPLat substantially reduces the total test execution time in most cases. Moreover, we apply SPLat on a large industrial code written in Ruby on Rails.

1. INTRODUCTION
Many programs can be configured through dynamic and/or static selection of configuration parameters. For example, a family of programs can be structured as a software product line (SPL), where each program is defined by a combination of features, which represent increments of functionality that can be selected. SPLs in general and testing of SPLs in particular have attracted a lot of recent attention in research [1, 3, 5, 8, 9, 17]. As another example, many codebases that power modern websites are highly configurable. For instance, the code behind the groupon.com website has over 170 boolean configuration parameters 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 for each possible configuration or feature combination.1 Thus, one test does not simply encode one execution of a program, and the cost of running a test suite is proportional to the product of the size of the suite and the number of configurations. The traditional line of research addresses the high cost of testing such systems by (random or sophisticated) selection of configurations, e.g., to achieve pair-wise coverage [11]. However, such selection is typically test agnostic: it can run a test on several configurations for which the test is definitely equivalent (thus only increasing the test time without increasing the chance to find bugs), or it can miss to run the test on some configurations that can expose bugs [1]. A recent line of research is test aware and tries to identify for each test a more precise set of configurations that can affect the test outcome [1, 12, 21, 22, 24, 34]. This prior work relied on static analysis [22] or heavyweight dynamic analysis based on model checking [1, 12, 21, 24, 34].

This paper presents SPLat, a novel, lightweight technique to reduce the cost of systematic testing of highly configurable systems and SPLs. Given a test suite, SPLat determines, for each test in the suite, a smallest set of configurations on which the test must be run. At the risk of being too formal from the beginning, let us define equivalent configurations. Let $p_1 \ldots p_k$ (for $k \geq 1$) be the configuration parameters of a program. Let parameter $p_i$ ($1 \leq i \leq k$) take a value from a finite domain $D_i$—in the case of SPLs, each parameter takes a boolean value, which represents whether the corresponding feature is selected or not. A configuration $c = [p_1 = v_1, \ldots, p_k = v_k]$ (where $v_i \in D_i$) is not necessary for execution against a test $t$ if another configuration $c' = [p_1 = u_1, \ldots, p_k = u_k]$ (where $u_i \in D_i$) has already been executed against $t$ such that the execution path of $t$ against $c$ is identical to the execution path of $t$ against $c'$. Because the test $t$ fixes the input values (for non-configuration

1For ease of exposition, we will use the terms “configuration” and “feature combination” interchangeably.
This paper makes the following contributions:

- **Applicable technique for reducing combinatorics in testing**

SPLat achieves an optimal reduction in the number of configurations, i.e., for any test, SPLat runs only configurations that have a unique execution path for that test. Moreover, the practical uses of SPLat are not directly concerned with the reduction in the number of configurations but with the reduction in testing time, and the experimental results show that SPLat achieves a reduction in testing time proportional to the reduction in the number of configurations in most cases. Our insight into pruning the configurations is inspired by the Korat algorithm [4] for test-input generation, which introduced the idea of execution-driven pruning for solving data-structure invariants written as imperative code.

SPLat supports constraints among configuration parameters, which delimit valid configurations. For a SPL, these constraints are termed a feature model [20] that (1) provides a hierarchical arrangement of features and (2) defines allowed configurations (also known as programs and feature combinations), over all the features. SPLat does not use the constraints to filter invalid configurations a posteriori, but rather it solves the constraints using an off-the-shelf propositional satisfiability (SAT) solver to focus the execution solely on the valid configurations for a given feature model. Thus, SPLat uses SAT for pruning invalid configurations and in tandem uses execution-driven pruning to further remove the valid configurations that are unnecessary for execution of each test.

SPLat achieves its effectiveness by simply monitoring accesses of configuration parameters during test execution. The monitoring is lightweight—both in terms of its execution overhead as well as in terms of its cost of implementation. To demonstrate the feasibility of implementing SPLat for different runtime environments, we developed two implementations, one for Java and one for Ruby on Rails. The Java implementation of SPLat leveraged the publicly available Korat code [27]. The Ruby on Rails implementation of SPLat was developed from scratch and took the second co-author only one day to implement and be robust enough to run against a large, industrial codebase at Groupon, Inc. Overall, in comparison with previous approaches based on static analysis [22] and heavyweight dynamic analysis [1, 12, 21, 24, 34], SPLat provides a lightweight, more widely applicable technique for reducing combinatorics in testing configurable systems.

This paper makes the following contributions:

- **Lightweight analysis of configurable programs.** We introduce the idea of lightweight monitoring for highly configurable systems to facilitate test execution.
be run against any valid assignment since the test only initializes the non-feature variables. The test instantiates an object of the Notepad class and creates a toolbar for it. We use the automated GUI testing framework Fixtures for Easy Software Testing (FEST) [14] 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, requires that the text area contains exactly what was entered, and finally 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 of the configurations need not be executed. First, analyzing just the feature model, two configurations are invalid: MTW = 000 and MTW = 001, where M, T, W stand for Menubar, Toolbar, and WordCount respectively. Therefore, no more than 6 configurations need to run.

SPLat further reduces that number by (dynamically) analyzing the code that the test executes. For example, executing the test against the configuration $c := MTW = 100$ exercises the same path, and the same sequence of computations, as executing against the configuration $c' := MTW = 101$. Therefore, having executed $c$, execution of $c'$ is not necessary. For this example, SPLat runs the test for only three configurations ($MTW = 000$, $MTW = 010$, and $MTW = 011$), which is the optimal number.

3. TECHNIQUE

Given a test written against configurable code, SPLat determines all relevant configurations on which the test should be run to potentially produce different outcomes. SPLat is dynamic in that it runs the test on one configuration, observes the execution to record accesses to configuration parameters, and uses the observed information to determine which configurations can be safely pruned. SPLat repeats this process until it explores all relevant configurations or until it reaches a given bound on the number of configurations. Recall that configurations can be constrained by a feature model. We first describe the feature model abstraction and then the core algorithm.

3.1 Feature Model Abstraction

SPLat assumes a feature model implementation that conforms to the interface shown in Figure 3. A feature assignment (also called a configuration) assigns (boolean) values to features. 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 all the constraints in the model. Given an assignment $a$, the method enumerate, using a SAT solver like SAT4J [35], returns the set of all complete assignments that (1) agree with $a$ on the values of feature variables in $a$ and (2) assign the values of the remaining feature variables to make the complete assignment valid. If such a set is not empty for some assignment $a$, then we say that $a$ is satisfiable; the method isSatisfiable checks this. In addition, a feature model can check if a feature is mandatory, and if it is, return the mandatory boolean value.
interface FeatureModel {
    Boolean isSatisfiable
    (Map<FeatureVariable, Boolean>
        featureAssignment);
    Set<Map<FeatureVariable, Boolean>> enumerate
        (Map<FeatureVariable, Boolean>
            featureAssignment);
    Boolean isMandatory(FeatureVariable);
    Boolean getMandatoryValue(FeatureVariable);
}

Figure 3: Feature Model Interface not true java; boolean not capitalized, move declarations outside for easier understanding

3.2 Main Algorithm
Figure 4 lists the core SPLat algorithm. It takes as input a test $t$ for a program with the given feature model $fm$. To enable exploration, the algorithm maintains a state that keeps the values of feature variables (line 2) and a stack of feature variables that are accessed during test execution (line 1). Besides this, SPLat uses stateless exploration and does not need to save, compare, or modify program state as done in stateful model checking [11, 12, 21, 24, 34]; Korat only needs to be able to set the values of feature variables, to observe their accesses by the code under test, and to re-execute the test from the beginning. sounds stateful to me

The algorithm first initializes the values of feature variables (lines 8-16) using the feature model interface introduced in the previous subsection. Mandatory features are set to the only value they can have. Optional features are initially all set to false. A careful reader may note that this initial assignment may be unsatisfiable for the given feature model. For example, this initial assignment would be unsatisfiable for our Notepad example test. However, we describe later on how SPLat enforces satisfiability during execution; SPLat can adjust the assignment before the test would execute any code based on the wrong assignment (and potentially report a “false alarm” test failure that is due to running an unsatisfiable configuration rather than finding an actual bug in the code under test). This initialization and other invocations of state.put() not only map a feature variable to a boolean value in state but also assign the value to the feature variable in the code under test. not sure I follow

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

SPLat next runs the the test (line 24) which, during its execution, call notifyFeatureAccess() whenever it is about to access a feature variable. the accessed feature variable is

An important step occurs at this point (line 51). The value that was initially assigned to the feature variable may make the configuration unsatisfiable. Therefore, SPLat checks whether the partial assignment of the values of the feature variables on the stack is satisfiable for the given feature model. If the assignment is unsatisfiable, then SPLat changes the value of the feature variable about to be accessed so that it becomes true, which guarantees that the new assignment is satisfiable (before the execution could have observed the old, unsatisfiable value). The argument for why this change of value is possible to keep the assignment satisfiable follows from the overall correctness of the SPLat algorithm: it explores only satisfiable partial assignments in line 42, and it checks if the assignment is satisfiable on every access (line 51), so if a smaller partial assignment was satisfiable for all variables on the stack, then it must be possible to extend that assignment with at least one value for the new feature variable that is being added to the stack. And that new feature variable currently has the value false (because every feature variable is initialized to false), so if it is not satisfiable, then we can change the value to true (line 53). I wonder if this could be said more clearly

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 the state and stack and (2) satisfies the feature model (line 28).

SPLat then determines the next configuration to execute by backtracking on the stack (lines 30-44). If the lastly accessed feature variable has only had its false value explored, then it can be set to true (lines 34-37). If the lastly accessed feature variable has had both values explored, then it is popped off the stack (lines 37-40) either until a feature variable is found that needs its true value explored or until the stack is empty.

Another important step occurs at this point (line 42). While the backtracking over the stack found a configuration to explore, it can be the case that this configuration is not satisfiable for the feature model. In that case, SPLat keeps searching for the next satisfiable configuration to run. If no such configuration is found and the stack becomes empty, the algorithm terminates. One output of the algorithm is the set of configurations covered, but the side effect of the algorithm is that the test has been executed for all configurations that could lead to different outcomes and thus a failure, so another output is the set of tests that fail for certain configurations. another output is the set of configurations for which the test fails?

3.3 Example Run
We demonstrate SPLat on the example from Figure 2. According to the feature model reference/move figure1 here??, "feature" because the algorithm need not handle mandatory features further.
Stack<FeatureVariable> stack;
Map<FeatureVariable, Boolean> state;

// input, shared with instrumented code
FeatureModel fm;

void splKorat(Test t) {
    // Initialize features
    state = new Map();
    for (FeatureVariable f:
        fm.getFeatureVariables()) {
        if (fm.isMandatory(f))
            state.put(f, fm.getMandatoryValue(f));
        else
            state.put(f, false);
    }

    // Instrument the code under test
    instrumentOptionalFeatureAccesses();

    // Repeatedly run the test
    do {
        stack = new Stack();
        t.runInstrumentedTest();
        print("configs covered: " + fm.enumerate
            (state.getAll(stack)));
        boolean satisfiable = false;
        while (!satisfiable) {
            while (!stack.isEmpty()) {
                FeatureVariable f = stack.top();
                if (!state.get(f)) {
                    stack.pop();
                } else {
                    state.put(f, true);
                    break;
                }
            }
            if (!state.get(f)) {
                state.put(f, false);
            }
            satisfiable = fm.isSatisfiable
                (state.getAll(stack));
        }
    } while (!stack.isEmpty());
}

void notifyFeatureAccess(FeatureVariable f) {
    if (!stack.contains(f)) {
        stack.push(f);
    } else {
        state.put(f, true);
    }
}

Figure 4: SPLat Algorithm

Because SPLat re-executes the test from the beginning and does store and restore the program state to backtrack at feature variable access, code can be redundantly executed between test executions. For example, the code from the start of the test until the (first) TOOLBAR access is executed in each of the 3 test executions, whereas a stateful exploration technique need execute this portion only once. However, the existing state exploration techniques cannot save and restore program state efficiently in general (especially for programs using I/O, Graphical User Interface (GUI), native code) and for this example because it uses native GUI code. In contrast, SPLat works for this example, which makes it more widely applicable, and additionally our evaluation shows that SPLat is efficient despite the redundant execution because most of the saving in execution time is due to not running the irrelevant configurations.

3.4 Reset Function

Although a stateless exploration technique such as SPLat does not need to save and restore program state, Ah hah! program state! in the middle of exploration like a stateful exploration technique does, the former does need to be able to restart a new execution with a program state unaffected by the previous execution. Put another way, every execution of the program needs to start with the same program state. Restarting an execution with a new runtime (e.g., spawning a new Java Virtual Machine (JVM) in Java) is the simplest solution, but it can be both inefficient and unsound. It can be unnecessarily inefficient because, even without restarting the runtime, the different executions may be able to share a runtime and still have identical initial program states. how?

It can be unsound because a new runtime may not reset the program state changes made by the previous execution (for example, previous executions having sent messages to other computers or having performed I/O operations). We address both of these issues by sharing the runtime between executions and requiring each test to call a reset function 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 man-
ually reset. For example, for Java subjects, 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 can be automated by a program analysis tool. 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 Possible Optimizations for Interfacing with the Feature Model

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, FeatureModel.isSatisfiable() and FeatureModel.enumerate() are executed as if the exploration state is completely new every time, even if it is just incrementally different from the previous exploration state. For example, when running the test from Figure 2, after TOOBAR=false finishes, the algorithm asks the feature model if TOOBAR=true is satisfiable, to which the feature model replies true. Then when WORDCOUNT is encountered while TOOBAR=true, the algorithm asks the feature model if the assignment WORDCOUNT=false and TOOBAR=true is satisfiable, although TOOLBAR is already known to be true. The feature model can be aware of this and only incrementally 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 fm 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, i.e., a feature model should not have to always be solved in entirety. The feature model implementation that our current SPLat tool for Java uses does not support incremental solving, which means that our timing results could be even better.

4. EVALUATION

This section describes an experimental evaluation using two implementations of SPLat, one for Java using seven subject product lines (Section 4.1), and one for Ruby on Rails using the Groupon codebase, which is a highly configurable system (Section 4.2).

4.1 Evaluation: Software Product Lines

We implemented SPLat on top of the Korat solver for imperative predicates [4] and integrated it with SAT4J [35]. We evaluate our approach using seven subject product lines: Graph Product Line (GPL) [31], Notepad [23], XStream [30], Email [16], MinePump [28], Elevator [33], and JTopas [19]. All but Notepad have also been used for evaluating testing techniques by other groups (including GPL by [1] [6], Elevator, Email, MinePump by [1], JTopas by [7] and XStream by [13][36]). Notepad was developed years before this paper was written for a graduate-level course on software product lines. We give a brief description of each SPL:

- **GPL** represents a set of programs that implement different combinations of graph algorithms. Its features vary algorithms and structures of the graph (e.g. directed/undirected and weighted/unweighted).
- **Notepad** represents a set of GUI applications based on Java Swing that implement different combinations of end-user features, such as saving/opening files, printing and user interface support (e.g. menu bar or toolbar).
- **XStream** represents a set of programs for serializing objects to XML and back that implement different combinations of features, such as the ability to alias class names and field names, omit selected fields, and produce concise XML for readability. XStream was converted into a product line from an open-source Java program of the same name [30] by turning boolean configuration flags into features.
- **Email** represents a set of email programs, whose features include encryption, automatic forwarding, and e-mail signatures.
- **MinePump** represents a set of water pump systems used in a mining operation. The bottom of the mine shaft is kept dry by a pump, which requires deactivation when combustible methane gas is detected in the mine. Features include sensors for detecting varying levels of water.
- **Elevator** represents a set of elevator systems, whose features include preventing the elevator from moving when it is empty and priority service for the executive floor.
- **JTopas** represents a set of programs for tokenizing text, whose features include support for tokenizing Java code (such as tokenizing block and line comments) and for encoding varying degrees of information in a token. Like XStream, JTopas was converted into a product line from an open-source program [19].

Table 1 shows the number of features (note that we only count the number of optional features since the mandatory features have constant values), the number of configurations, and the code size for each subject SPL. More details of the subjects and results are available at our website [25].

<table>
<thead>
<tr>
<th>SPL</th>
<th>Features</th>
<th>Configs</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPL</td>
<td>14</td>
<td>75</td>
<td>1713</td>
</tr>
<tr>
<td>Notepad</td>
<td>23</td>
<td>144</td>
<td>2074</td>
</tr>
<tr>
<td>XStream</td>
<td>7</td>
<td>128</td>
<td>14380</td>
</tr>
<tr>
<td>Email</td>
<td>8</td>
<td>40</td>
<td>1233</td>
</tr>
<tr>
<td>MinePump</td>
<td>8</td>
<td>34</td>
<td>580</td>
</tr>
<tr>
<td>Elevator</td>
<td>5</td>
<td>20</td>
<td>1046</td>
</tr>
<tr>
<td>JTopas</td>
<td>5</td>
<td>32</td>
<td>2034</td>
</tr>
</tbody>
</table>
Table 2: Evaluation

<table>
<thead>
<tr>
<th>Test</th>
<th>NewJVM</th>
<th>ReuseJVM</th>
<th>Reachable</th>
<th>SPLatTime</th>
<th>SPLatTime2</th>
<th>Overhead</th>
<th>S.A. Conf.</th>
<th>S.A. Overhead</th>
<th>S.A. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOW</td>
<td>26.55</td>
<td>16.83 (63.31%)</td>
<td>8 (32%)</td>
<td>6.29 (47.3%)</td>
<td>4.89</td>
<td>1.3 (40%)</td>
<td>32</td>
<td>86.87</td>
<td>16.94</td>
</tr>
<tr>
<td>M</td>
<td>20.01</td>
<td>18.95 (93.89%)</td>
<td>10 (100%)</td>
<td>13.46 (70.58%)</td>
<td>6.73</td>
<td>1.36 (46%)</td>
<td>32</td>
<td>86.87</td>
<td>18.57</td>
</tr>
<tr>
<td>HIGH</td>
<td>28.92</td>
<td>18.94 (65.46%)</td>
<td>40 (100%)</td>
<td>28.31 (133.29%)</td>
<td>18.54</td>
<td>6.98 (37.34%)</td>
<td>32</td>
<td>86.87</td>
<td>18.57</td>
</tr>
<tr>
<td>LOW</td>
<td>40.63</td>
<td>10.74 (26.44%)</td>
<td>1.2 (55%)</td>
<td>1.9 (3.5%)</td>
<td>0.87</td>
<td>0.13 (6.69%)</td>
<td>1</td>
<td>23.92</td>
<td>0.87</td>
</tr>
<tr>
<td>M</td>
<td>37.56</td>
<td>10.84 (47.86%)</td>
<td>50 (100%)</td>
<td>22.97 (77.51%)</td>
<td>11.74</td>
<td>0.24 (4.86%)</td>
<td>10</td>
<td>24.84</td>
<td>8.07</td>
</tr>
<tr>
<td>HIGH</td>
<td>58.92</td>
<td>19.9 (34.39%)</td>
<td>50 (100%)</td>
<td>38.96 (103.2%)</td>
<td>36.26</td>
<td>0.24 (4.86%)</td>
<td>40</td>
<td>24.84</td>
<td>39.16</td>
</tr>
<tr>
<td>LOW</td>
<td>50.77</td>
<td>36.08 (71.74%)</td>
<td>10 (100%)</td>
<td>24.92 (50.38%)</td>
<td>23.12</td>
<td>0.38 (7.66%)</td>
<td>20</td>
<td>23.74</td>
<td>34.17</td>
</tr>
<tr>
<td>M</td>
<td>60.67</td>
<td>36.08 (59.59%)</td>
<td>20 (100%)</td>
<td>40.74 (102.68%)</td>
<td>40.26</td>
<td>1.04 (25.31%)</td>
<td>40</td>
<td>24.84</td>
<td>58.72</td>
</tr>
<tr>
<td>HIGH</td>
<td>112.26</td>
<td>21.58 (37.47%)</td>
<td>2 (55%)</td>
<td>1.57 (22.85%)</td>
<td>1.06</td>
<td>0.48 (40.21%)</td>
<td>2</td>
<td>23.92</td>
<td>1.96</td>
</tr>
<tr>
<td>LOW</td>
<td>105.1</td>
<td>9.04 (8.65%)</td>
<td>64 (100%)</td>
<td>5.77 (63.88%)</td>
<td>3.26</td>
<td>0.51 (8.76%)</td>
<td>64</td>
<td>109.22</td>
<td>5.19</td>
</tr>
<tr>
<td>M</td>
<td>101.66</td>
<td>8.3 (8.5%)</td>
<td>128 (100%)</td>
<td>9.18 (85.82%)</td>
<td>8.59</td>
<td>0.57 (6.4%)</td>
<td>128</td>
<td>105.68</td>
<td>8.74</td>
</tr>
<tr>
<td>HIGH</td>
<td>398.22</td>
<td>27.99 (11.74%)</td>
<td>11 (30%)</td>
<td>2.36 (26.06%)</td>
<td>2.85</td>
<td>0.21 (44.14%)</td>
<td>14</td>
<td>86.94</td>
<td>13.57</td>
</tr>
<tr>
<td>LOW</td>
<td>418.47</td>
<td>25.49 (20.65%)</td>
<td>46 (83%)</td>
<td>10.35 (15.14%)</td>
<td>10.24</td>
<td>0.01 (4.22%)</td>
<td>14</td>
<td>86.94</td>
<td>15.46</td>
</tr>
<tr>
<td>M</td>
<td>419.95</td>
<td>25.49 (20.65%)</td>
<td>144 (100%)</td>
<td>15.54 (39.81%)</td>
<td>154.16</td>
<td>0.04 (0.96%)</td>
<td>14</td>
<td>86.94</td>
<td>15.46</td>
</tr>
<tr>
<td>HIGH</td>
<td>19.21</td>
<td>2.23 (11.62%)</td>
<td>6 (32%)</td>
<td>0.79 (52.29%)</td>
<td>0.29</td>
<td>0.48 (158.75%)</td>
<td>6</td>
<td>104.97</td>
<td>0.5</td>
</tr>
<tr>
<td>LOW</td>
<td>196.53</td>
<td>171.62 (90.68%)</td>
<td>55 (100%)</td>
<td>130.87 (76.26%)</td>
<td>128.52</td>
<td>2.35 (18.86%)</td>
<td>55</td>
<td>98.41</td>
<td>128.69</td>
</tr>
<tr>
<td>M</td>
<td>344.2</td>
<td>265.85 (76.99%)</td>
<td>100 (100%)</td>
<td>399.77 (109.51%)</td>
<td>397.38</td>
<td>1.27 (108.86%)</td>
<td>100</td>
<td>103.52</td>
<td>256.21</td>
</tr>
<tr>
<td>HIGH</td>
<td>23.71</td>
<td>7.5 (31.74%)</td>
<td>4 (40%)</td>
<td>3.15 (134.89%)</td>
<td>1.9</td>
<td>1.15 (58.61%)</td>
<td>4</td>
<td>22.59</td>
<td>7.49</td>
</tr>
<tr>
<td>LOW</td>
<td>59.74</td>
<td>14.18 (41.72%)</td>
<td>24 (73.5%)</td>
<td>15.83 (46.51%)</td>
<td>6.26</td>
<td>4.11 (66.06%)</td>
<td>64</td>
<td>22.35</td>
<td>15.25</td>
</tr>
<tr>
<td>M</td>
<td>13.12</td>
<td>5.5 (41.91%)</td>
<td>48 (100%)</td>
<td>37.8 (200.3%)</td>
<td>8.94</td>
<td>34.09 (205.41%)</td>
<td>64</td>
<td>22.35</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Table 2 shows the results. All times are listed in seconds.3 The table is structured as follows:

- **Test** identifies the test case for the row entries. For each subject SPL, we took 3 independent tests, where a test has all inputs fixed except the feature variables. The first test, labeled LOW, represents the ideal scenario, where the test needs to be run only on a small number of configurations. The second test, labeled MEDIUM, represents the average scenario, where the test needs to be run on some configurations. The third test, labeled HIGH, represents the unsatisfactory scenario, where the test needs to be run on most of the configurations.

- In **NewJVM** approach, we run each configuration of the product line against a test with a new JVM. This approach does not require writing a reset function to reset Java runtime settings, but incurs the overhead of constructing a new runtime. The corresponding column shows the duration.

- In **ReuseJVM** approach, we run each configuration of the product line with the same runtime to avoid the cost of creating a new runtime. (The JVM is started up only once and the test entry is invoked for each configuration.) This approach requires the tester to explicitly reset what normally would be reset automatically by a new runtime, such as putting static initializers in the main function. Because the tester likely has to write a reset function anyway for settings beyond a runtime (such as resetting I/O), writing additional code to reset the JVM is a viable option to save the cost of a new runtime. The corresponding column shows the time taken in seconds (and as a percentage of the NewJVM duration).

- **Reachable** shows the number of configurations to run as determined by SPLat (for the particular test).

- In **SPLatTime** approach, we use SPLat to run the test, reusing the same runtime. The corresponding column shows the duration in seconds (and as a percentage of that for ReuseJVM).

- **SPLatTime2** shows the time taken in seconds for the theoretically ideal (oracular) scenario of running the minimum number of configurations without incurring the cost of any analysis technique to determine those configurations. Therefore, this number excludes the instrumentation, monitoring and feature model interaction overhead incurred by SPLat.

- **Overhead** shows the overhead of SPLat, calculated by subtracting SPLatTime2 from SPLatTime, and dividing it by SPLatTime2.

- We also compare SPLat's performance against static reachability analysis performed in our previous work on static analysis [22] (Section 4.1.1). Static reachability analysis 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 combinations of these reachable features need to be run. **S.A. Conf.** shows the number of such configurations. **S.A. Overhead** shows the time taken to perform the static reachability analysis. **S.A. Time** shows the time taken to run the configurations determined by the static reachability analysis.

Test cases. We wrote new tests as well as modified existing tests (when available) for each of the subject product lines. Some of the tests used loops to increase the running time because they finished too quickly for meaningful time measurement. The exact test suites are available on the project website [25].

Effectiveness. Table 2 shows that reusing JVM saves a considerable amount of time. For example, for over half of
the tests, reusing JVM saves over 50% of execution time. This is because these tests do 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. And even if it were possible, SPLat further reduces the execution time by determining the reachable configurations. In particular, by reusing the JVM for LOW test for Notepad, the test takes 34% of the time to run without reusing the JVM, but with SPLat, the test takes just 2% of the already reduced time. In fact, the table shows that except for Mine’s HIGH test, as long as SPLat can reduce the number of configurations to test (i.e., Reachable is lower than the total number of configurations), it runs faster than running each configuration (i.e., less than 100% of Reuse/JVM).

**Overhead.** Table 2 also shows a small overhead (single digit percentage) compared to the oracular technique, except for LOW tests and tests for JTopas and Mine. The overhead is high for LOW tests because these tests finish quickly (under 7 seconds), meaning that instrumentation, monitoring and feature model interaction takes a larger fraction of time than they would for a longer executing test. The overhead is high for JTopas likely because the feature variables are accessed many times since they are accessed within the tokenizing loop. The overhead is high for Mine because feature accesses and their instrumentations take relatively longer to execute for this particular test case since the subject is very small (580 lines of code). SPLat, due to its cost, cannot execute a test faster than knowing the reachable configurations upfront and running the test only on those configurations. Thus, the two small negative overheads for Email are likely due to randomness in executing the programs. It is important to note that effectiveness and overhead are orthogonal. As long as the reduction in time due to the reduction in configurations is larger than the overhead, we are saving time. To illustrate, GPL’s LOW test incurs over 168% overhead but the reduction in configurations outweighs the overhead and SPLat takes only 35% of running each configuration with the same JVM.

**Reset.** Our results do not take into account the cost of writing a reset function. NewJVM, ReuseJVM and SPLat all require settings outside of the runtime to be explicitly reset, but NewJVM automatically resets runtime-specific settings by creating a new runtime. The cost of writing the reset function, which only needs to be done once, is likely to be amortized over multiple runs of a test.

### 4.1.1 Comparison with Static Reachability Analysis

The static reachability analysis yields results that are more imprecise than SPLat (Column S.A.Confs entries are larger than the corresponding entries in Column Reachable). In fact, for JTopas, Notepad, and Mine, the static reachability analysis reports all features as being accessed from the call graph and therefore reports all configurations as having to be tested. For JTopas, this is due to its tests invoking the main method of the product line, 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 Mine, each test happens to exercise a sequence of methods that together reach all feature variable accesses. Note that the static reachability analysis approach first determines configurations to run (which takes the time in column S.A.Overhead) and afterwards runs them one by one (which takes the time in column S.A.Time). Therefore, the time to run the static reachability analysis is its overhead and can be compared to SPLat’s overhead (column Overhead). Note that the static reachability analysis has a considerably larger overhead, in some cases two orders of magnitude longer, than that of SPLat. Although static reachability analysis overhead can be offset by 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.

### 4.2 Evaluation: 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 171,000 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 fraction 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 231,000 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 [38].)

Groupon PWA has over 17,000 Rspec (unit and integration) tests. (The particular code version we used had exactly 17,276 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 takes under 10 minutes on a cluster of 4 computers with 24 cores each.

**SPLat application.** We implemented SPLat for Ruby on Rails to apply it to Groupon PWA. We did not have to implement any 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 feature model was present so the feature model constraints did not need to be solved.

We set out to evaluate how many configurations each test could cover if we allow to vary 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 allow us to enumerate all the configurations. Therefore, we set the limit on the number of configurations
### Table 3: Reachable configurations

<table>
<thead>
<tr>
<th>Vars</th>
<th>Tests</th>
<th>Vars</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10167</td>
<td>1</td>
<td>1649</td>
</tr>
<tr>
<td>3</td>
<td>1167</td>
<td>4</td>
<td>561</td>
</tr>
<tr>
<td>6</td>
<td>399</td>
<td>7</td>
<td>285</td>
</tr>
<tr>
<td>9</td>
<td>261</td>
<td>10</td>
<td>118</td>
</tr>
<tr>
<td>12</td>
<td>66</td>
<td>13</td>
<td>31</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>19</td>
<td>14</td>
</tr>
<tr>
<td>21</td>
<td>109</td>
<td>22</td>
<td>44</td>
</tr>
<tr>
<td>24</td>
<td>22</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>27</td>
<td>14</td>
<td>28</td>
<td>17</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
<td>31</td>
<td>21</td>
</tr>
<tr>
<td>33</td>
<td>11</td>
<td>34</td>
<td>17</td>
</tr>
<tr>
<td>36</td>
<td>11</td>
<td>37</td>
<td>4</td>
</tr>
<tr>
<td>39</td>
<td>0</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4: Accessed features

<table>
<thead>
<tr>
<th>Vars</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10167</td>
</tr>
<tr>
<td>3</td>
<td>1167</td>
</tr>
<tr>
<td>6</td>
<td>399</td>
</tr>
<tr>
<td>9</td>
<td>261</td>
</tr>
<tr>
<td>12</td>
<td>66</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>109</td>
</tr>
<tr>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
</tr>
<tr>
<td>33</td>
<td>11</td>
</tr>
<tr>
<td>36</td>
<td>11</td>
</tr>
<tr>
<td>39</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
</tr>
</tbody>
</table>

5. **RELATED WORK**

We describe the most closely related projects to SPLat.

#### 5.1 Dynamic Analysis

**Korat.** SPLat was inspired by the Korat algorithm [4] for test-input generation using imperative constraints, which are written as 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; while this would allow Korat to directly apply on configurations, it would lead Korat to essentially explore the entire space of feature combinations (up to $2^N$ combinations for $N$ features) before every test execution. In contrast, SPLat provides a novel way to support feature models while retaining the effectiveness of Korat based on execution-driven pruning by applying it with SAT in tandem. Additionally, SPLat can change the configuration being run during the test execution (line 53 in Figure 4), which Korat did not do for data structures.

**Shared execution.** There is a considerable amount of ongoing research on saving testing time by sharing execution between multiple configurations or inputs, in testing product lines [1, 21, 24, 34], model checking [9, 12], computer security [2, 26], and online patching [18, 39]. The basic idea is that multiple executions due to multiple configurations or inputs against the same codebase have much computation in common, which could be executed only once by sharing the program state between the executions. For example, the shared execution techniques in testing product lines run multiple configurations together and split execution when a feature variable is accessed (while checking the feature model to prune the invalid feature value), executing with one feature value after another and merging execution at a common execution point if possible to resume shared execution. All shared execution techniques, including those outside of product line research, are able to run a test only on reachable configurations that SPLat also finds. However, these techniques differ from SPLat in that they use *stateful* exploration, which requires a dedicated runtime for saving and restoring program state and only works on programs with such runtime support. Also, these techniques strive to maximize computation sharing by attempting to share control-flow, e.g., after execution diverges due to different configurations/inputs taking different paths, control-flows are merged into a single control-flow if the two control-flows arrive at the same execution point. This approach leads to these techniques having large runtime overheads (not because of simple engineering issues but because of fundamental challenges in approaches used), and these overheads are often not offset by the gains in saving shared computation. In contrast, SPLat uses *stateless* exploration [15] and never merges control-flow of different executions. Although our technique does not share computations between executions, it requires minimal runtime support and can be implemented very easily and quickly against almost any runtime system.
that allows variable values to be read and set during execution. For running tests that reach only a small number of configurations, SPLat’s lightweight approach seems far more appropriate than the heavyweight approach of shared execution techniques.

Spectrum of SPL testing techniques. [21] defines a spectrum of SPL testing techniques based on the degree of change required to a conventional runtime system to support running a test against a product line’s configurations. On the one end is the blackbox technique, which takes a conventional runtime system as is and runs the test for each configuration against it. NewJVM approach is such a technique. On the other end is the whitebox technique, which customizes a conventional runtime system extensively to make it product line-aware. Shared execution techniques discussed previously fall into this category. SPLat, which only requires runtime support for reading and writing to feature variables, is much closer to being a blackbox approach than a whitebox approach, but still provides an optimal reduction in the number of configurations to consider.

5.2 Static Analysis

We previously developed a static analysis to determine relevant features that can affect the outcome for a given test [22]. The test then needs to be run only on (all valid) combinations of these relevant features that satisfy the feature model. The static analysis, which uses an inter-procedural, context-insensitive, flow-insensitive and path-insensitive points-to analysis framework called Spark [29] (which is based on the Soot Java static analysis framework), can be divided into two analyses: reachability and effects. Reachability analysis determines the reachable features, whose code is reachable from the test’s entry point. Effects analysis determines the relevant features, a subset of the reachable features whose code includes a control-breaking statement (and thus can change the control-flow of other features’ code) or changes the data owned by other features’ code (and thus can change the data-flow of their code). 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 won’t even be reachable. And as Section 4 shows, SPLat determines reachable configurations with much greater precision and is likely to be considerably faster than the static analysis because it 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 (e.g., requiring user interaction). But such situations do not seem to arise often, especially for tests that exercise a small subset of the codebase.

5.3 Reducing Combinatorics for Program Analyses

Like test execution, performing a program analysis on a product line is challenging because the analysis has to check each program in the product line. Recent work [3, 5, 23] have focused on making a program analysis SPL-aware such that the analysis need not be applied redundantly against commonalities of the product line. In [23], runtime monitor elimination is made SPL-aware such that monitors are not inserted for configurations that can statically be determined to not trigger the monitor. [5] presents a technique for automatically converting any intra-procedural dataflow analysis into an SPL-aware analysis, such that analysis computations can be shared between configurations. [3] extends this work by performing the automatic conversion for inter-procedural, finite, distributive, subset problems. While these projects focus on the general problem of reducing combinatorics in checking SPL properties, SPLat focuses on program execution and is not tailored to static dataflow analyses.

5.4 Sampling

Sampling exploits domain knowledge, rather than program analysis results, 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 [10, 11, 32]. Our dynamic program analysis safely prunes feature combinations, while sampling approaches can miss problematic configurations [1].

6. CONCLUSION

This paper presented SPLat, a novel technique for reducing the combinatorics in testing configurable systems. SPLat dynamically prunes the space of configurations each test must be run against. SPLat achieves an optimal reduction in the number of configurations and provides a lightweight, more widely applicable technique compared to previous approaches based on static analysis and heavyweight dynamic execution. Experimental results on seven 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 the practicality of our approach.

7. REFERENCES
