Time-Space Efficient Regression Testing for Configurable Systems

Sabrina Souto[a], Marcelo d'Amorim[b]

[a] *Universidade Estadual da Paraíba, Campina Grande, PB, Brazil*
[b] *Universidade Federal de Pernambuco, Recife, PE, Brazil*

Abstract

Configurable systems are those that can be adapted from a set of input options, reflected in code in form of variations. Testing these systems is challenging because of the vast array of configuration possibilities where bugs can hide. In the context of evolution, testing becomes even more challenging — not only code but also the set of plausible configurations can change across versions.

This paper proposes `EvoSPLat`, a regression testing technique for configurable systems that explores all dynamically reachable configurations from a test. `EvoSPLat` supports two important application scenarios of regression testing. In the RCS scenario `EvoSPLat` prunes configurations (not tests) that are not impacted by changes. In the RTS scenario `EvoSPLat` prunes tests (not configurations) which are not impacted by changes.

To evaluate `EvoSPLat` under the RCS scenario we used a selection of configurable Java programs. Results indicate that `EvoSPLat` reduced time by $\sim$22% and reduced the number of configurations tested by $\sim$45%. To evaluate `EvoSPLat` under the RTS scenario we used GCC. Results indicate that `EvoSPLat` reduced time to run tests by $\sim$35%. Overall, results suggest that `EvoSPLat` is a promising technique to test configurable systems in the prevalent scenario of evolution.

1. Introduction

Configurable systems are those that can be adapted through input options, reflected in code in the form of variations. Large software systems often provide some level of configurability to users or to developers. The intuition is that the ability to reason about these variations facilitates maintainability and reduces time-to-market [1]. Configurable systems are prevalent and the amount of variation they offer can be very high [2, 3]. Examples of configurable systems include the Firefox web browser [4], the Linux kernel [5], the GCC compiler infrastructure [6], and the deals-recommendation web service Groupon [7].

Testing configurable systems is an important problem that continues to attract a lot of attention from the research community [8]. Conceptually, the large space of possible configurations makes the introduction of errors easier and testing for those errors more challenging. For the sake of cost, it is not uncommon practice to test the system against a single configuration[1] [9, 10]. For instance, Groupon [11] and GCC [9] follow this practice. At another extreme, exhaustively testing all configurations is unacceptably expensive. Large software systems typically offer hundreds of configuration options, leading to a combinatorial blowup in the number of possible configurations to test.

This work focuses on the problem of *regression testing configurable systems*. Regression testing is an important quality-assurance activity realized during software evolution to reduce the chances of defects escaping to production. It consists of repeatedly executing a test suite and monitoring its effects with the goal of anticipating the observation of errors. Regression testing is notoriously expensive and continues to receive huge attention from researchers and practitioners [12]. In the context of configurable systems, regression testing becomes even more expensive as each test needs to be executed against several different configurations. Despite the interest in regression testing from the research community and the prevalence of configurable systems in practice, research on this problem, which intersects two very-active areas, is surprisingly scarce.

To optimize regression testing, a technique intuitively needs to identify the impact of evolutionary changes on the execution of each test. Unfortunately, statically finding a sound yet small set of relevant configurations for each test is challenging – computing accurate static approximations of dynamic impact sets precisely and efficiently for non-configurable system is already a challenging problem [13, 14]; adding the configuration dimension does not make the problem any simpler [15, 16, 17]. Most previous research on this problem focused on the proposal of heuristics to find configurations seemingly-related to changes [18, 19, 20, 21]. The strategy scales but can lead to error misses as relevant configurations can be ignored.

This paper proposes `EvoSPLat`, a lightweight dynamic technique for efficient regression testing of configurable systems. `EvoSPLat` builds on our previously-developed technique, `SPLat` [11], which explores all configurations of a system that are dynamically reachable from an input test. `EvoSPLat` optimizes both time and space to explore change-impacted configurations and tests. Our approach is tuned for two scenarios of use: Regression Configuration Selection (RCS) and Regression Test Selection (RTS). `EvoSPLat` for RCS prunes configurations (not tests) that are not impacted by changes whereas `EvoSPLat` for

Email addresses: `sfs@cin.ufpe.br` (Sabrina Souto), `damorim@cin.ufpe.br` (Marcelo d'Amorim)

[1] Such configuration is often referred to as the "default" configuration and includes the features which are more typical to build the system.

RTS prunes tests (not configurations) which are not impacted by changes. EvoSPLat uses lightweight static analysis to observe impact based on saved information from previous runs.

Both modes of execution are important in regression testing of configurable systems. The RCS scenario is useful for configurable systems with a wide variety of tests, including unit and integration tests. In those cases, each test typically covers relatively small and diverse fractions of the code. For scenarios where tests cover large portions of the code when considering reachable configurations, RTS is likely a better fit. Such pattern is frequently observed when system tests are implemented by calling an executable through command-line parameters [22, 10].

The contributions of this paper are:

- A lightweight technique to alleviate cost of systematic testing in two important scenarios of use: RCS and RTS.
- An implementation of our technique that is publicly available at `https://sites.google.com/site/evosplat/`.
- An empirical evaluation, including software product lines (SPLs) and one large program (GCC). We evaluated EvoSPLat for RCS on twelve SPLs and observed significant reduction in time and space. We also evaluated EvoSPLat for RTS on GCC. Results show that EvoSPLat reduces time by 35%, on average, compared to SPLat. Compared to sampling techniques, namely *t-wise* [23] with *t*=2 and *t*=6, we observed that EvoSPLat retained the ability to detect faults as 6-wise but required much less configurations (time) to achieve that. Pairwise, in contrast, was significantly faster compared to EvoSPLat (and 6-wise) but missed 2 out of 5 real GCC bugs we analyzed.

2. Finding configurations for testing with SPLat

This section illustrates EvoSPLat on a small running example.

We call *configuration variables* (aka feature variables) those program variables whose purpose in code is to adapt behavior through variation. A *configurable system* is one that uses configuration variables to manage variability in code. A *configuration* is an assignment of values to configuration variables. In many cases, a *default configuration* exists (e.g., Groupon [11] and GCC [9]). Intuitively, the default configuration denotes typical behavior of the software. Variations are often controlled by users (not developers) from the command-line, through *configuration options*. These options are mapped to configuration variables in code. To simplify presentation, throughout the text we use the terms variables and options indistinctly as if the mapping was one-to-one. A test for a configurable system takes a configuration as an additional input. A *feature model (FM)* documents the variables of a configurable system and their relationships [24, 25], also called *constraints*. A *SAT solver* checks if a configuration is *legal* or not according to the FM constraints. The following two propositional constraints are part of the FM of GPL [26], a library of graph algorithms previously used in the evaluation of prior related work.

$(\texttt{MSTKRUSKAL} \lor \texttt{MSTPRIM}) \rightarrow (\texttt{UNDIR} \land \texttt{WEIGHTED})$
$\texttt{SHORTEST} \rightarrow (\texttt{DIRECTED} \land \texttt{WEIGHTED})$

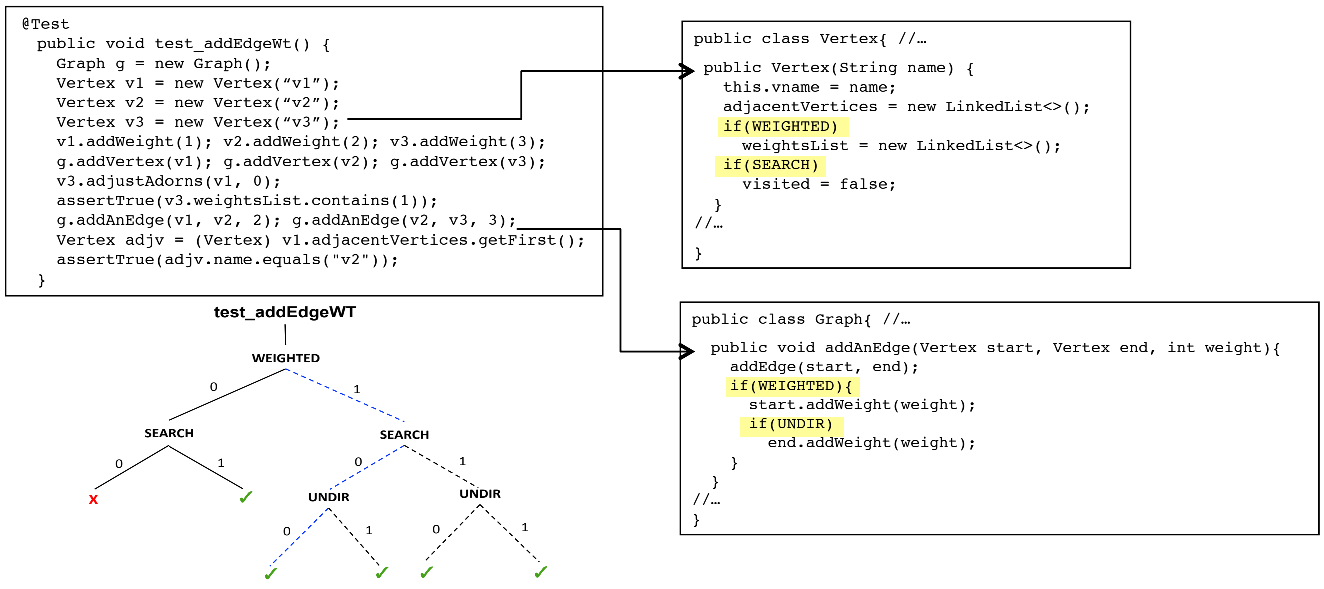
The first rule indicates that minimum spanning trees work on undirected weighted graphs. The second rule indicates that an algorithm for computing the shortest path between two nodes requires directed and weighted edges. FMs are optional in our framework as, in practice, they are not always available.

Figure 1 shows the test addEdgeWt from GPL. This test builds a graph with 3 connected vertices and checks some properties on the graph. For example, the first assertion checks if the weight 1 can be found through the vertex v3. This should be the case if the weight list associated with v3 was correctly updated with the call to `v3.adjustAdorns(v1, 0)`. In the following, we will refer to three configuration options of GPL: WEIGHTED, SEARCH, and UNDIR. For the sake of illustration, let us assume that the tester uses a default configuration that has the option WEIGHTED set and all other options unset. Considering a single configuration, test execution will cover one distinct path in code where only branches associated with options which are set will be traversed. For this pair of test and configuration execution *passes*.

Given a test for a configurable system, SPLat [11] determines a set of configurations on which the test should be run. SPLat finds configurations as follows. It executes the test on one configuration, observes the values of configuration variables, and uses these values to determine the next configuration that the test should be run against. It repeats this process until it explores all relevant configurations or until it reaches a specified bound on the number of configurations. This exploration effectively produces a *decision tree* with nodes corresponding to configuration variables and edges corresponding to the different values these variables can hold. Leaf nodes indicate complete paths traversed by the test. Figure 1 shows the decision tree obtained with a complete run of SPLat on test addEdgeWt.

Considering the decision tree for test test_addEdgeWt, a path where variable WEIGHTED is set indicates that graph edges have weights and a path where variable UNDIR is set indicates that the graph is undirected. SPLat accepts categorical types. Discretization methods (e.g., domain partitioning [27]) can be used to handle non-categorical domains. SPLat starts execution without assigning concrete values to variables; it assigns values on demand, when variables are covered. SPLat finds a *sound set* of configurations to execute the test. This means that, provided that no bounds on time or number of configurations exist, SPLat cannot miss fault-revealing configurations on a given input test. Alternative selection strategies exist (e.g., combinatorial interaction testing [23]) but they do not provide soundness guarantees, in contrast to SPLat.

When the user provides on input a feature model constraining the set of legal configurations, SPLat can safely prune illegal configurations with the help of a SAT solver. We emphasize that this model is optional for SPLat. Back to the example, the configuration `[(WEIGHTED,0),(SEARCH,0)]` is the first one that SPLat explores. Note that the Vertex constructor covers variables WEIGHTED and SEARCH, however variable UNDIR is not covered in this example. SPLat discards this configuration as it is illegal (✗) according to the GPL configuration constraints, provided on input for this case. The cross under the leftmost path in the decision tree indicates that. Then SPLat re-executes the test on a new configuration. It assigns another value to



```
@Test
  public void test_addEdgeWt() {
    Graph g = new Graph();
    Vertex v1 = new Vertex("v1");
    Vertex v2 = new Vertex("v2");
    Vertex v3 = new Vertex("v3");
    v1.addWeight(1); v2.addWeight(2); v3.addWeight(3);
    g.addVertex(v1); g.addVertex(v2); g.addVertex(v3);
    v3.adjustAdorns(v1, 0);
    assertTrue(v3.weightsList.contains(1));
    g.addAnEdge(v1, v2, 2); g.addAnEdge(v2, v3, 3);
    Vertex adjv = (Vertex) v1.adjacentVertices.getFirst();
    assertTrue(adjv.name.equals("v2"));
  }
```

```
public class Vertex{ //…
 public Vertex(String name) {
   this.vname = name;
   adjacentVertices = new LinkedList<>();
   if(WEIGHTED)
     weightsList = new LinkedList<>();
   if(SEARCH)
     visited = false;
  }
//…
}
```

```
public class Graph{ //…
  public void addAnEdge(Vertex start, Vertex end, int weight){
    addEdge(start, end);
    if(WEIGHTED){
      start.addWeight(weight);
      if(UNDIR)
        end.addWeight(weight);
    }
  }
//…
}
```

Figure 1: Test, corresponding decision tree, and fragments of GPL code that illustrate how variables are reached through test.

variable SEARCH and re-runs the test against the configuration [(WEIGHTED,0),(SEARCH,1)], which is legal (✓). This process continues until all reachable configurations are explored.

3. EvoSPLat in a nutshell

SPLat showed that there are scenarios where exploring all reachable configurations is feasible. For example, SPLat was able to explore all dynamically reachable configurations in Groupon PWA[2] for several test cases [11]. Unfortunately, there are important scenarios where the number of configurations to explore is still very high. EvoSPLat addresses this inherent scalability issue of SPLat. In summary, EvoSPLat leverages information from previous runs to optimize time and space.

EvoSPLat provides two modes of execution: Regression Configuration Selection (RCS) and Regression Test Selection (RTS). In RCS mode (Section 3.1), EvoSPLat restricts the set of configurations to run on each test; no restrictions apply to the test set. In contrast, in RTS mode (Section 3.2), EvoSPLat restricts the set of tests that will be executed; no restrictions apply to the configurations associated to each test. EvoSPLat for RCS benefits the most in a scenario where, for a given test, distinct configurations cover highly-different sets of functions. In that scenario, if a function changes, only few configurations would be impacted and only few re-executions would be necessary for a given test. If, on the contrary, most functions were covered across reachable configurations, RCS would have little effect as the test would need to be re-executed in all those configurations. EvoSPLat for RTS is a better fit for the later scenario. Although it does not prune test-reachable configurations, it is more lightweight compared to RCS. The decision on when to apply RCS or RTS depends on the kinds of tests of the application. Typically, unit and integration test suites are better fit for RCS whereas system tests are better fit for RTS.

[2]Groupon PWA is the name of the codebase that powers the groupon.com website. See Section 4.2 from [11].

3.1. *EvoSPLat for RCS*

EvoSPLat for RCS proceeds as follows. Initially, SPLat is used to bootstrap the process of discovering configurations to run on a given test. As result, it produces the entire decision tree for that test. When there is a change, a lightweight static analysis is used to detect which subtrees of the decision tree have been affected. Then, a separate execution of SPLat runs on each of these subtrees, potentially inducing changes in the tree structure. Ideally, when code changes, only a small number of subtrees and paths will be affected. To sum, EvoSPLat for RCS *saves space* by reducing the number of configurations to explore and it *saves time* by optimizing constraint solving time.

Back to the example, let us consider the scenario where the developer changes the function addAnEdge. In this case, EvoSPLat first detects that only the right subtree of the decision tree was affected by the change and then it spawns a new execution of SPLat, rooted in the subtree associated with configuration [(WEIGHTED, 1)]. The value of variable WEIGHTED is fixed in that execution. Hence, EvoSPLat will only explore four of the six paths. The right subtree in the tree from Figure 1 shows those paths. The reduction can be higher or lower depending on the amount of code changes made during evolution and the size of the decision tree.

EvoSPLat also accelerates test execution by caching results of SAT solver calls. The hypothesis is that more often than not the execution of a test on (Evo)SPLat will produce highly-similar subtrees in consecutive runs and make similar SAT calls. Note that it is not possible to completely avoid SAT calls as code changes can result in changes in the tree structure, leading to new constraints. However, to account for different orderings of accessed variables, EvoSPLat stores the constraints in canonical form, which exists for the language of propositions we used.

3.2. `EvoSPLat` for RTS

Unfortunately, `SPLat` for RCS will not scale for systems where most tests reach most configurations; a scenario that occurs often when testing is performed mostly against a single function, typically the main function of the system. In that case, the entire code under testing is reachable through that function. This is what happens, for example, with GCC, where a test consists of a source file augmented with testing directives indicating what "features" of the compiler should be executed (see Figure 5). In GCC the tests execute the same codebase modulo the variations induced from these input directives. As result, most functions are called in most configurations. For cases like GCC, re-running `SPLat` upon evolutionary changes is prohibitively expensive. In a nutshell, `EvoSPLat` for RTS identifies which tests have been impacted by changes and only re-runs `SPLat` on those tests. `EvoSPLat` maintains a map from tests to functions covered by those tests to identify impacted tests. At a high-level, this approach to test selection is similar to that used in Ekstazi [28].

Note that RCS generalizes RTS. With RCS one can model a test that should be ignored by associating an empty set of configurations with that test. RTS, however, does not generalize RCS as it is unable to partially restrict configurations of tests as RCS does. Despite this subsumption relationship, in practice, RTS remains important as it enables optimizations to `EvoSPLat`.

4. `EvoSPLat` in detail

This section describes in detail `EvoSPLat`, a technique to reduce cost of testing for configurable systems in the dominant scenario of evolution. `EvoSPLat` offers two execution modes, reflecting important scenarios of evolution. In the following, we describe these two modes.

4.1. `EvoSPLat` for RCS

`EvoSPLat` takes on input a feature model, a test, and a list of methods of interest denoting changes. It explores relevant configurations and reports test verdicts on output. `EvoSPLat` executes differently depending on whether it is executing for the first time or not. We call by "First Run" the initial execution and by "Next Runs" the subsequent runs of `EvoSPLat`. In the following, we present the interface `EvoSPLat` uses to check satisfiability of configurations and then detail the algorithm.

Figure 2 shows the classes and interfaces that `EvoSPLat` uses to access the feature model to check legality of configurations. 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 use the SAT4J [29] SAT solver to implement these feature model operations.

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

Figure 2: Feature Model Interface

First Run

Figure 3 shows the pseudo-code of `SPLat`, modified to support evolution. `SPLat` takes on input a feature model, a test, and a stack denoting what decision subtree should be explored (line 9 from Figure 3). Figure 4 shows the pseudo-code of `EvoSPLat`, which calls `SPLat`. `EvoSPLat` takes as input a feature model, a test, and a method list, empty on the first run, denoting those methods that have been impacted by evolutionary changes (line 10 from Figure 4). The first run of `EvoSPLat` invokes `SPLat` passing an empty stack on input (line 23 from Figure 4). In fact, this initial call is equivalent to invoking the non-evolutionary version of `SPLat`.

The statements modified in the original version of the `SPLat` algorithm [11] appear underlined in Figure 3. `SPLat` maintains a `state` (line 5) that stores the values of feature variables, and a `stack` (line 4) of `Entry` objects (line 1), representing methods and feature variables accessed during the test run. For each read of an optional feature variable, `SPLat` calls `notifyFeatureRead()` (line 55) to update both the `stack` with the new feature variable read and the `state` with a satisfiable value. For each method read, the algorithm calls `notifyMethodVisited()` (line 64) to update the `stack` with the new invoked method. At line 16, `SPLat` executes the test on a valid partial assignment. After finishing execution, `SPLat` determines the next partial configuration to execute (lines 21–36). If the last read feature holds `true`, then `SPLat` has explored both values of that feature, and it is popped off the stack (lines 24–29). If the size of the stack becomes smaller than the size of the stack passed on input `st`, it means the subtree of interest has been fully explored and `SPLat` should terminate. If the last read feature holds `false`, then `SPLat` has explored only the `false` value, and the feature should be set to `true` (lines 29–35). Another important step occurs now (line 32). While backtracking over the `stack` found a partial 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 assignment to run. If no such assignment is found, the `stack` becomes empty, and `SPLat` terminates.

Assuming that the subject under test provides a feature model

```
1  interface Entry {};
2  // Method and FeatureVar are subtypes of Entry
3
4  Stack<Entry> stack;
5  Map<FeatureVar, Boolean> state;
6  List<CacheEntry> cache;
7
8  /* run SPLat on a given subtree */
9  void SPLat(FeatureModel fm, Test t, Stack st) {
10  cache = new ArrayList<CacheEntry>();
11
12  do {
13   // Repeatedly run the test
14   stack = new Stack();
15   stack.addAll(st);
16   t.runInstrumentedTest();
17   cache.add(new CacheEntry(stack, state));
18
19
20   // Find new configuration
21   while (!stack.isEmpty()) {
22    while (stack.top() instanceof Method) stack.pop();
23    FeatureVar f = stack.top();
24    if (state.get(f)) {
25     state.put(f, false); // Restore
26     stack.pop();
27     // don't execute beyond the input subtree
28     if (stack.size() <= st.size()) return;
29    } else {
30     state.put(f, true);
31     VarAssign pa = getPartialAssignment(state, stack);
32     if (isSAT(fm, pa)) {
33      break;
34     }
35    }
36   }
37
38  } while (!stack.isEmpty());
39   /* update caches */
40   caches.put(t, cache);
41   writeToFile(trie);
42 }
43
44 Trie<Boolean> trie;
45 boolean isSAT(FeatureModel fm, VarAssign vAs) {
46  if (trie == null) trie = loadTrieFromFile();
47  /* use mapping if it exists */
48  if(trie.contains(vAs)) return trie.get(vAs).booleanValue();
49  /* new configuration => update trie */
50  boolean res = fm.isSatisfiable(vAs);
51  trie.put(vAs, res);
52  return res;
53 }
54
55 void notifyFeatureRead(FeatureVar f) {
56  if (!stack.contains(f)) {
57   stack.push(f);
58   VarAssign pa = getPartialAssignment(state, stack);
59   if (!isSAT(fm, pa))
60    state.put(f, true);
61  } }
62
63 // called-back from test execution
64 void notifyMethodVisited(Method method) {
65  if (!stack.contains(method)) {
66   stack.push(method);
67  } }
```

Figure 3: SPLat algorithm modified. Changes to the original SPLat algorithm [11] appear underlined.

```
1  /* characterization of one test run */
2  class CacheEntry {
3   Stack<Entry> stack;
4   Map<FeatureVar, Boolean> state;
5  }
6
7  /* persistent part of EvoSPLat state */
8  Map<Test, Set<CacheEntry>> caches;
9
10 void EvoSPLat(FeatureModel fm, Test t, List<Method>
       changedMethods){
11
12  // Initialize features
13  state = new Map();
14  for (FeatureVar f: fm.getFeatureVariables())
15   state.put(f,
16    fm.isMandatory(f)?fm.getMandatoryValue(f):false);
17
18  // Instrument the code under test
19  instrumentOptionalFeatureAccesses();
20
21  Set<CacheEntry> cache = caches.get(t);
22  if (cache==null || cache.isEmpty())
23   SPLat(fm, t, new Stack()); // initial run of SPLat
24  else
25   Set<Stack> visited = new Set();
26   foreach CacheEntry:cel ∈ cache {
27    state.reset();
28    int hit = findMethod(cel.stack, changedMethods);
29    // Update stack and state
30    Stack st = new Stack();
31    for (int index = 0; index < hit; index++) {
32     // Stack contains changed method at position hit
33     Entry e = cel.stack.get(index);
34     // Reconstruct part of the original stack
35     st.push(e);
36     // Update feature variable assignment
37     if (e instanceof FeatureVar)
38      state.put(e, cel.state.get((FeatureVar) e));
39    } // end for
40    if (hit > 0) {
41     /* do not visit already visited subtrees */
42     if (!visited.contains(st)) {
43       visited.add(st);
44       /* Execute SPLat on affected subtree */
45       SPLat(fm, t, st);
46     }
47    }
48   } // end foreach
49
50 }
```

Figure 4: EvoSPLat algorithm (RCS). Calls to the SPLat modified algorithm (see Figure 3) appear underlined.

that constraints the set of legal configurations, EvoSPLat reduces execution cost by caching the results of SAT solver calls (see lines 32 and 59). This caching is the simplest memoization mechanism offered by EvoSPLat; it focuses on time reduction. To implement this mechanism EvoSPLat uses a map, where keys correspond to partial variable assignments and values correspond to booleans indicating whether or not the corresponding assignment is satisfiable. Given that these keys are potentially highly similar by construction, EvoSPLat uses a trie (aka prefix trees) [30] for faster storage and lookup of key-value pairs. Given the fact that generated constraints are of the form $\bigwedge a_i$, with $a_i = v$ or $a_i = \neg v$ for a symbolic variable v, canonicalization is obtained by pre-determining an ordering on symbolic variables. For example, EvoSPLat will represent the constraint $y \wedge x$ as $x \wedge y$, given the name order $x < y$.

Next Runs

Subsequent runs of EvoSPLat invoke SPLat passing nonempty stacks corresponding to subtrees of the exploration tree that need to be explored upon changes. In the following, we elaborate how EvoSPLat works on the first run and subsequent runs. Our current version of EvoSPLat for RCS assumes that potentially fault-revealing changes occur inside method bodies. Considering that all tests passed in the previous evolution cycle, a call to a changed method is a necessary condition to activate potential failures. Under this assumption, EvoSPLat for RCS

```
1  /* { dg-do compile } */
2  /* { dg-options "-O2 -fdump-tree-optimized" } */
3  int g(_Complex int*);
4  int f(void){
5   _Complex int t = 0; int i, j;
6   __real__ t += 2;
7   __imag__ t += 2;
8    return g(&t);
9  }
10 /* { dg-final { scan-tree-dump-times "__complex__" 0 "
       optimized" { xfail *-*-* } } } */
```

Figure 5: An example of test (complex-4.c from GCC test suite).

applies a lightweight change-impact analysis that monitors calls to changed methods to decide what configurations need to be re-executed on each test.

`EvoSPLat` explores configurations as follows. It first initializes the values of feature variables (lines 12–16). The algorithm then instruments (line 19) the code under test to observe both feature variables read and methods calls. Next, it reads the list of traces (line 21) observed during the previous run of `EvoSPLat` for that test. If the list is empty, it means no history is available and `SPLat` is called for a full-run (line 23). Otherwise, `EvoSPLat` calls `SPLat` to explore all impacted configurations (lines 26-48). Each iteration of this loop calls `SPLat` on one decision subtree that has been affected by changes. The parameter `st` in the call at line 45 denotes such subtree; the object `st` encapsulates a partial assignment of feature variables to values. For example, the assignment `[(WEIGHTED,1)]` denotes the right subtree from the example in Section 3.1.

4.2. `EvoSPLat` for RTS

In contrast to RCS, RTS does not restrict configurations for execution in each test. Instead, RTS restricts the tests that will be executed in each evolution cycle. For those tests, all reachable configurations should be executed. Note that RTS is a particular case of RCS – one could model the effects of RTS with RCS by assigning an empty set of configurations to those tests that should be ignored and retaining the original set of configurations for selected (non-ignored) tests. Despite the generality of RCS, there are scenarios where configuration selection has limited effectiveness. RTS comes as an alternative for those cases. More precisely, reduction in number of configurations with RCS is limited when most configurations explored in tests cover most functions. In those cases, it is possible that the changed function(s), if covered by the test at all, will likely be covered by any of its reachable configurations. Situations like this arise more frequently in systems whose tests exercise a single main function [10]. GCC is a case in point (see Section 5.2).

To support RTS, `EvoSPLat` first identifies what tests have been impacted by changes and then performs a full execution with `SPLat` on those tests. `EvoSPLat` maintains a map from tests to set of functions that have been covered by any configuration reachable from that test. Recall that, in this scenario, most configurations of a given test cover most functions. At test selection time, a test will be considered for testing if a changed function is included in the function set associated to that test.

4.2.1. *Implementation for GCC*

Figure 5 shows an example of GCC test written in the DejaGnu [31] testing framework. A GCC test runs against a single configuration and is comprised of the following parts. The DejaGnu directive *dg-do* specifies the type of action to run against the test. For a compiler such as GCC, possible options are: preprocess, compile, assemble, link, and run. For example, the directive `dg-do compile` at line 1 instructs DejaGnu to only compile functions `g` and `f`. No other operations (e.g., assemble, link, run, etc.) will be executed in this case. The DejaGnu directive *dg-options* specifies a subset of options that must be present to run the test. For this example, the directive at line 2 indicates that this test must be executed with at least the options "O2" and "-fdump-tree-optimized" enabled. The body of the test (lines 3 to 10) corresponds to C/C++ code. Finally, the instruction at line 10 describes the oracle to determine whether the test should pass or fail.

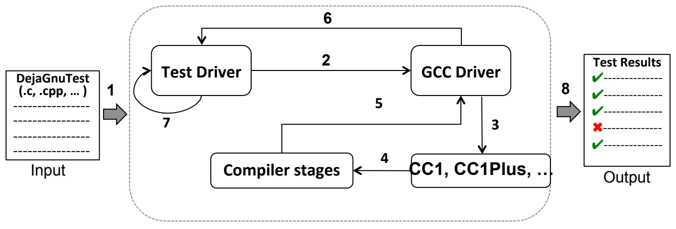


Figure 6: GCC test execution workflow.

Figure 6 illustrates the test execution workflow for GCC. The input of the workflow is a test (as in Figure 5) and the output is the report of test results. After reading the test (1), the *Test Driver* calls the *GCC Driver* passing the commands interpreted from the test file (2). Based on the test input, the *GCC Driver* calls the `main function` of the compiler (e.g., CC1 for C, CC1Plus for C++) passing compilation parameters (3). Different compiler stages (scan, parse, etc.) will be triggered depending on the "Actions" of the test (4). The compiler sends results back to the *GCC Driver* when compilation finishes (5) and the *GCC Driver* forwards the output to the *Test Driver* (6), which checks these outputs (7) and elaborates a test report (8).

Other configurable systems use primarily system tests [10] for testing, similarly to GCC. This is important as to decide which reduction strategy to use in regression testing (see Section 5).

5. Evaluation

Table 1 details the structure of this section, which is organized in two parts to reflect our different scenarios of evaluation. Section 5.1 evaluates `EvoSPLat` under the RCS scenario on twelve Software Product Lines (SPLs) written in Java. Section 5.2 evaluates `EvoSPLat` under the scenario of RTS on GCC, a large configurable system built over the course of 3 decades. Recall that in Regression Configuration Selection (RCS) mode `EvoSPLat` aims to reduce the number of configurations reachable by each test but with the test set fixed (see Section 4.1) whereas in Regression Test Selection (RTS) mode `EvoSPLat` aims to reduce the number of tests executed but with configurations of tests fixed (see Section 4.2).

Section	Scenario	RQs	Subject Programs	Type of Tests
5.1	RCS	R1, R2	SPLs	Unit/System/Integration
5.2	RTS	R3, R4, R5	GCC	System

Table 1: Organization of Section 5.

Although both SPLs and configurable systems use variability as key principle, the characteristics of the tests in these programs are different. As observed in recent empirical studies [22, 10], most tests in configurable systems are system tests, which often cover large portions of the code. This limits the ability of RCS to prune configurations. The intuition is that `EvoSPLat` for RCS would benefit the most in a scenario where, for a given test, distinct configurations cover highly-different sets of functions. In that scenario, if a function changes, only few configurations would be impacted and only few re-executions would be necessary for a given test. If, on the contrary, most functions were covered across most configurations, RCS would have little effect as the test would need to be re-executed in all those configurations. In this later scenario, `EvoSPLat` for RTS is preferable. It is unable to prune configuration but is more lightweight.

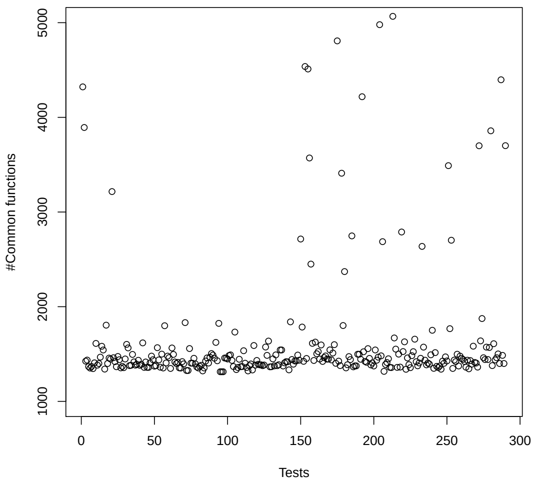


Figure 7: Distribution of common functions across configurations for each test.

Prior to evaluating our proposed techniques, we conducted a preliminary limitation study to understand how successful RCS would be in a configurable system like GCC. To that end, we studied the variance in the distribution of functions covered across configurations of each test in GCC. Figure 7 illustrates the distribution of functions that are common across all reachable configurations. The average amount of functions per configuration lies between one and two thousand. A closer inspection reveals that most configurations of a given test cover about the same set of functions. This happens because the test in GCC exercises a main function, which dynamically reaches most functions of the compiler (see Section 4.2.1). This is common practice in configurable systems where configuration options are passed on the command line. We also observed a high percentage of common functions across configurations and across tests, 72% of functions are called by all tests. These observations substantiate our decision to evaluate `EvoSPLat` for RTS with GCC and evaluate `EvoSPLat` for RCS with SPLs.

5.1. Regression Configuration Selection (RCS)

Considering the scenario of RCS we posed the following research questions:

RQ1. What are the time savings obtained with the caching of constraint solver calls?

RQ2. What are the space savings obtained with the recording of decision trees?

For this experiment, we selected 12 subjects of various sizes that have been previously used in other studies. Table 2 provides details about these subjects, including number of features, number of valid configurations that can be generated combining these features, number of methods, and code size.

Subject	*Source*	*#Features*	*#Valid Confs.*	*#Methods*	*LOC*
101Companies	[32]	10	192	307	2,059
DesktopSearcher	[33]	16	462	113	3,779
Elevator	[34]	5	20	121	1,046
Email	[35]	8	40	227	1,233
GPL	[26]	13	73	95	1,713
JTopas	[36]	5	32	578	2,031
MinePump	[37]	6	64	99	580
Notepad	[38]	17	256	136	2,074
Prevayler	[39]	5	32	523	2,844
Sudoku	[40]	6	20	99	853
XStream	[41]	7	128	2,318	14,480
ZipMe	[42]	13	24	338	3,650

Table 2: SPLs used.

We classify execution of `EvoSPLat` in one of three categories. In one limit, a *complete execution* is necessary to bootstrap `EvoSPLat`. In this case, all tests and configurations have to be executed. A *partial execution* is necessary when evolutionary changes affect some tests (and configurations) making it necessary to re-execute tests (against some configurations). In another limit, it is possible that no test execution is necessary because *no impact* was detected. Our evaluation on SPLs considers the first two scenarios of regression testing — complete execution and partial execution. In principle, there is no cost associated with running `EvoSPLat` under the "no impact" scenario.

5.1.1. Answering RQ1 — Complete execution (Time Reduction)

A complete re-execution may be necessary during evolution. Consider, for example, the first run of the test suite using `EvoSPLat` or the scenario where all tests and configurations have been impacted by changes. This experiment evaluates whether `EvoSPLat` can help even in those cases.

`EvoSPLat` optimizes the cost of SAT solving. Recall that `SPLat` can use a SAT solver to discard invalid configurations. On the one hand, this is important because often only a small fraction of the $2^{\#Features}$ configurations are valid. See columns "*#Features*" and "*#Valid Confs.*" in Table 2. This indicates that using a SAT solver conceptually pays off. On the other hand, the cost of SAT solving can be substantial. This indicates that caching results of SAT solver calls may be important.

This experiment evaluates the distinct benefit of caching the calls to the SAT solver (lines 44–53 from Figure 3). Recall that `EvoSPLat` uses a trie, shared across all tests, to store the outcomes of feasibility checks and that trie keys correspond to feature constraints. To reduce noise in measurements, we discarded subjects from Table 2 whose test suite takes less than five seconds to run.

To isolate cost of SAT solving, this experiment simulates the scenario where code changes would not influence feature decisions. Under these circumstances the decision trees generated in consecutive runs of `EvoSPLat` on the same test would be the same. This means that only one, potentially expensive, `SPLat` run is necessary until the point where a change in the decision tree is observed. Until that point, it suffices to execute a test against each of the configurations obtained with the first execution of `SPLat` on that test. In this setup, it is possible to isolate the benefits of an optimal cache hit ratio. Although we did not measure how prevalent this scenario is to the practice, we conjecture that it happens often. If, on the contrary, decision trees were to always change during evolution, that would certainly mean a lack of understanding about the purpose of options.

We considered three techniques for comparison:

- "`SPLat`" corresponds to the baseline technique.
- "FirstRun" corresponds to the technique that runs `EvoSPLat` for the first time, with an empty cache, as described in Section 4.1.
- "NextRuns" corresponds to the technique that runs `EvoSPLat` with a filled cache, as described in Section 4.1, but assuming the decision tree does not change.

Subject	`SPLat`			`EvoSPLat(%)`	
	#SAT	#UNSAT	TotalTime (CS%)	FirstRun	NextRuns
101Companies	10,187	642	933 (0.60)	0.27	0.31
DesktopSearcher	4,942	272	1,261 (14.80)	5.80	14.35
Email	5,446	432	6.7 (73.00)	19.40	46.27
GPL	4,652	420	7.18 (89.00)	5.30	53.14
JTopas	788	0	139 (9.85)	3.70	5.17
Notepad	168,549	18,695	16,025.5 (30.00)	6.30	22.50
XStream	486	0	93.8 (15.00)	10.80	12.00
AVG	-	-	- (33.17)	7.36	21.96

Table 3: Savings in SAT calls.

Table 3 summarizes results for this experiment. Recall that `EvoSPLat` stores constraints in a trie and that, to maximize cache hit ratio, the keys in the trie are stored in a canonical form (Section 4.1). This figure shows the breakdown of SAT queries that the baseline technique – `SPLat` – generates and land in SAT or UNSAT. Note that the number of SAT calls can be very high in some cases. That happens because (Evo)`SPLat` checks consistency of partial configurations incrementally. Exploration backtracks only when the solver answers UNSAT to a query, which means it is no longer possible to reach a valid configuration. The other solver calls will result in a SAT answer.

Column "TotalTime (CS%)" shows the overall time in seconds. This comprehends time spent running all tests on all (valid) configurations, and the time spent with constraint solving. The percentage of time spent with constraint solving alone "(CS%)" appears in parentheses. Column "FirstRun" reports the reduction in time obtained with the initial execution of `EvoSPLat`. Saving of time happens even in this case because `EvoSPLat` starts using the trie as it is filled. Column "NextRuns" reports the reduction in time obtained with the subsequent executions of `EvoSPLat`. Recall that, in this case, `EvoSPLat` makes no calls to the SAT solver. Note from the table the difference between the savings obtained and the maximum possible savings (column "%CS"). For instance, considering XStream, the savings of both "FirstRun" and "NextRuns", albeit significant (10% and 12%, respectively), are still below the maximum possible (15%).

Considering all subjects, tests, and configurations that we analyzed, results show an average time reduction of 21.96%. Results also show that this value represents 66.2% (21.96/33.17) of the maximum possible reduction, which is the total time spent with constraint solving in `SPLat`. It should be possible to reduce this gap further by using more efficient canonicalizations and trie implementations.

Time savings seem to depend on several factors such as complexity of constraints, number of paths, and length and complexity of tests. As expected, we noted smaller improvements for the cases where relatively more time is spent executing test code as opposed to calling the SAT solver. For example, the tests for the subjects 101Companies, DesktopSearcher, and Notepad, spend a considerable time accessing the graphical user interface and the tests for the subjects JTopas and XStream spend a significant amount of time accessing files.

Answering RQ1: Results indicate that caching results of SAT solver calls should be done, although performance of caching depends on the characteristics of the application.

5.1.2. Answering RQ2 — Partial Execution (Space Reduction)

A partial execution is necessary when changes affect only some configurations reachable from given tests. In this case, `EvoSPLat` reconstructs only the parts of the decision tree that may have been affected by changes. This experiment evaluates how much `EvoSPLat` can reduce space required to explore configurations. More specifically, we want to find out how much less configurations, compared to a complete run (as in Section 5.1.1), one would need to re-run in a scenario of change.

This experiment assumes that changes occur only in methods and, within method bodies, changes occur in application code, not in feature decisions, and that feature variables are not re-assigned within code (as in the SPLs we analyzed). Under these circumstances, the decision tree, obtained in previous runs, is not affected. Intuitively, this setup enables one to filter subtrees – inducing a set of configurations – affected by changes. Furthermore, to limit the scope of the experiment, we are assuming changes in only one method per evolution cycle. The rationale for that is that the size of the SPL subjects we analyzed is small (see Table 2). Note that, albeit small, these are subjects typically used to study SPL testing. Section 5.2 reports results of `EvoSPLat` on a much larger software system.

We used the following procedure to measure configuration reduction. We run `SPLat` once on each test and measure, for each method that appears in the trace, the fraction of configurations whose test execution trace does not contain that method.

Subject	[0-0.1[	[0.1-0.2[	[0.2-0.3[	[0.3-0.4[	[0.4-0.5[	[0.5-0.6[	[0.6-0.7[	**Total**
101Companies	9	2	4	4	3	-	-	22
DesktopSearcher	1	-	5	8	30	-	-	44
Elevator	1	2	-	-	-	-	-	3
Email	1	-	6	7	1	-	-	15
GPL	8	4	1	1	7	3	1	25
JTopas	28	-	-	-	-	-	-	28
MinePump	-	-	2	1	-	-	-	3
Notepad	-	-	42	1	-	-	-	43
Prevayler	-	2	3	1	2	-	-	8
Sudoku	-	1	-	-	4	-	1	6
XStream	7	-	-	-	-	-	-	7
ZipMe	14	18	7	3	5	-	-	47
Total	69	29	70	26	52	3	2	-

Table 4: Distribution of tests per subject over increasing intervals of configuration reduction. A configuration reduction for a given test is a measure of how much `EvoSPLat` reduces the number of configurations explored (and hence cost of exploration) compared to running the test again with `SPLat`. For example, of the 25 tests of GPL, 7 tests lie in the [0.4-0.5[interval. This means that the number of configurations `EvoSPLat` explores would be reduced by a factor of ∼45% for those tests. The rightmost interval clusters the test cases that are most beneficial for `EvoSPLat` in number of configuration savings whereas the leftmost interval clusters the worst-performing cases. Note that this setup ignores changes that could affect the decision tree (and produce less/more traces).

The intuition is that it is not necessary to run the test on a configuration that does *not* call the changed method. For a given test case, we save the average saving across all methods. For the sake of illustration, let us assume that a test explores only two configurations, c_1 and c_2, that the code contains only three methods m_1, m_2, and m_3, and that configuration c_1 covers m_1 and m_2 whereas configuration c_2 covers m_1 and m_3. For this setup, a change (only) in method m_1 results in no reduction as it is always referred in the traces while a change in either m_2 or m_3 results in 50% reduction (in both cases a single configuration is eliminated). An average reduction of 33% will be observed considering the three methods.

Table 4 presents the distribution of tests per subject over different intervals of configuration reduction. The higher the interval a test is allocated the better. We omitted columns where no test cases felt under. For example, considering this setup, `EvoSPLat` reduces the number of configurations explored by an average of ∼45% (cf. interval "[0.4,0.5[") for 7 tests of GPL.

Answering RQ2: Considering all subjects in this experimental setup, results indicate that it is possible to obtain a significant reduction in number of configurations explored. However, this highly depends on the functions changes and the impact set associated to those functions.

5.2. Regression Test Selection (RTS)

The goal of this study is to assess how `EvoSPLat` performs on a scenario of RTS using a highly-configurable real system. We pose the following research questions:

RQ3. How efficient `EvoSPLat` is compared to `SPLat`?
RQ4. How efficient `EvoSPLat` is compared to sampling techniques?
RQ5. How effective `EvoSPLat` is (w.r.t. bug finding) compared to alternative techniques?

In this experiment, we used the GNU Compiler Collection (GCC) [43], a large system with more than 7 million lines of code, more than 17k tests, with hundreds of configuration options [1], and 2,015 feature variables. GCC has been developed for almost three decades, receiving contribution from over 500 developers [44].

5.2.1. Techniques, Metrics, and Methodology

We evaluated `EvoSPLat` against `SPLat` and sampling techniques, namely *t-wise* sampling for *t*=2 and *t*=6. These values of *t* have been recently used by Medeiros et al. [10] to evaluate the impact of these interaction degrees in testing configurable systems. The metrics we used to evaluate efficiency of techniques are *number of tests* selected for re-execution upon an evolution cycle and *time reduction* The metric used to evaluate efficacy is *fault detection*. For that, we considered five bugs we previously found in GCC version 4.8 [9].

To bootstrap RTS one needs to first map the functions covered by each test on any reachable configuration. In a subsequent iteration, if any of the functions covered by a given test changes, `EvoSPLat` for RTS (`EvoSPLat` for short) re-runs that test against *all its configurations*. Otherwise, `EvoSPLat` skips the test. For example, `SPLat` reaches 192 configurations for the test `gcc-dg.pr58145-2.c`. Executing this test against these configurations covers 4,129 functions. Analyzing the evolution cycle the day after August 21, 2015 (our day of reference), we observed that the changes impacted all the 192 configurations. More precisely, for any given reachable configuration of the test, at least one of the 31 modified functions in this evolution cycle appeared in its corresponding execution trace. As result, `EvoSPLat` re-executed the test on all 192 configurations.

5.2.2. Experimental Setup

Test execution in GCC is time-consuming. Considering one configuration per test and our running environment, it takes roughly 2s to run each test. In this experiment, we focused on the `gcc-dg` test suite, which contains 3,343 tests. The rationale for choosing this test suite was that we observed from bug reports a higher incidence of bugs revealed with this suite.

In contrast to the simulated scenario of evolution from Section 5.1, this experiment evaluates `EvoSPLat` under real evolution scenarios. To that end, we needed to make some simplifications. We considered daily changes of GCC as it represents an important scenario of evolution. We analyzed the build files that the GCC team posts to their web site every day. Due to a relatively high cost to prepare the execution environment (e.g., download, instrument code, and build), we considered only seven evolution cycles, the period of a week. Table 5 shows the amount of functions that change over that period, starting on August 21, 2015. We use the notation "+*n*" to indicate an evolution cycle from day *n* to day $n+1$. For example, "+1" denotes the period between the starting date and next day. Note that the amount of changed functions is very small relative to the total number of functions in GCC (as per Figure 7). The stability of GCC justifies that. To further reduce exploration time, we restricted the number of configuration options that `SPLat` con-

Evolution	*Functions Changed*	*Tests Exec. (%)*	EvoSPLat		(Evo)Pair-wise	(Evo)Six-wise
			Confs/Test	Time (h)	Time (h)	Time (h)
+1	31	100	134	6.5	0.67	27.0
+2	4	100	134	6.5	0.67	27.0
+3	4	-	-	-	-	-
+4	15	5	248	0.7	0.04	1.4
+5	13	-	-	-	-	-
+6	1	57	169	4.8	0.38	15.6
+7	8	100	134	6.5	0.67	27.0
AVG	10.85	51.7	117	3.6	0.35	14.0

Table 5: RTS Results. *#Tests Exec. (%)*: the percentage of tests executed; *#Confs/Test*: the average on the number of configurations executed per test; *Time (h)*: the time spent to run the tests and configurations. (Evo)Pair-wise generated 13 Confs/Test; (Evo)Six-wise generated 553 Confs/Test.

siders (see [1]). We used the 33 most-frequently cited options from the GCC bug reports in the week we focused our analysis.

Recall that SPLat is able to constrain the set of configurations to explore when a formally-specified feature model exists. Unfortunately, GCC does *not* provide such model. In this study, we enriched the feature model constructed by Garvin et al. [45] that documents 110 constraints from GCC. We augmented the model with constraints that we manually extracted from the online documentation [46]. We found 136 new constraints not included in Garvin et al.'s model, resulting in a total of 246 constraints of which 86% relate to the 33 options we analyzed.

5.2.3. Answering RQ3

Considering a single configuration per test, it takes ∼1h18m to run only the gcc-dg test suite. To reduce overall time for running our experiments, we specified an upper bound of 100 configurations to explore per test. Of the 3,343 tests from gcc-dg, 290 tests do not reach this bound. We considered only those tests as to enable EvoSPLat to explore the entire decision tree on a rerun. Without this scheme, it would be possible to bias the search by narrowing the configuration space in the tree. The purpose of this restriction is to facilitate analysis of results; EvoSPLat does support cases that reach configuration bounds.

Daily Build	*Baseline*	EvoSPLat	
	Time(h)	Tests Exec.(%)	Time(h)
+1	10.15	100	10.15
+2	10.15	100	10.15
+3	10.15	-	-
+4	10.15	7.6	0.78
+5	10.15	100	10.15
+6	10.15	47	5.1
+7	10.15	100	10.15
AVG	10.15	65	6.64

Table 6: SPLat vs. EvoSPLat.

Table 6 shows results comparing EvoSPLat with our baseline, SPLat. Results show that EvoSPLat is either very beneficial or not at all. In 4 out of 7 cases EvoSPLat obtained no reduction. For the rest of the cases, a significant reduction was obtained. On average, results indicate a reduction of ∼35% in the number of tests required for re-execution and of running time. The cases were all tests required re-execution (see 100% under column "*Tests Exec. %*") reveal that during the week under consideration, a "basic" function was changed. This function is shared across all tests and configurations. For example, the function compile_file from the file toplev.c was modified in the first evolution cycle and that function is called by every test for every configuration.

Answering RQ3: Results obtained in a scenario of evolution (a week of evolution in GCC release 4.8) shows that running EvoSPLat (as opposed to rerunning SPLat from scratch) saves time. The average reduction of time considering this scenario was nearly 35%.

5.2.4. Answering RQ4

In contrast to the previous experiment, this experiment does not restrict the number of configurations that EvoSPLat explores. The rationale is that limiting the number of configurations would obviously favor EvoSPLat compared to high values of t (with t denoting the width of configuration vectors in *t-wise* testing) and disfavor EvoSPLat compared to low values of t. For that, SPLat is not restricted to configuration bounds in this experiment. However, to reduce experimentation time to a reasonable figure, we limited the number *of tests* to 100. We randomly selected those from the 3,343 tests of gcc-dg.

Table 5 reports efficiency results. Column *Evolution* shows the evolution cycle id. An evolution cycle refers to the changes between two consecutive daily builds. Column *Functions Changed* shows the number of functions that changed in a cycle. We omitted results for the baseline as it is virtually constant across different cycles: it takes nearly 6.5h to execute all tests against all reachable configurations. The columns EvoSPLat, pairwise, and six-wise show results for each of the techniques. Column *Tests Exec.(%)* shows the percentage of tests in the test suite that required execution, it is the same for all techniques. A higher number in this column indicates low reduction. Columns "Time (h)" show the runtime cost of running each of the techniques. For EvoSPLat, column "Confs/Test" shows the *average* number of configurations explored per test. For pair-wise and six-wise, which are black-box approaches, this value is constant. Pair-wise explores 13 configurations per test and six-wise explores 553 configurations per test. Note that when there are code changes which are not covered by tests, it it not necessary to run the techniques. This happens in evolution cycles "+3" and "+5".

We generated *t-wise* configurations with the ACTS tool [47]. (Evo)Pair-wise and (Evo)Six-wise work similarly to EvoSPLat but note that, as black-box techniques, the number of configurations they consider is fixed. As EvoSPLat, they initially map the functions covered by each test on any reachable configuration (we used SPLat to obtain this information). Then, if any of the functions covered by a given test changes, (Evo)Pair-wise and (Evo)Six-wise re-run that test against *all precomputed configurations*. Otherwise, they discard the test.

Answering RQ4: Perhaps as expected, results indicate that (Evo)Pair-wise and (Evo)Six-wise offer, respectively, lower and upper bounds of comparison – EvoSPLat performed in between these two extremes. It ran four times faster than (Evo)Six-wise and ten times slower than (Evo)Pair-wise. Relative performance of techniques with respect to the number of configurations is similar.

5.2.5. *Answering RQ5*

The goal of this experiment is to evaluate the effectiveness, by comparing EvoSPLat with alternative techniques on the ability to detect real bugs. For that, we used the release 4.8.2 of GCC where the authors of this paper already found bugs[3] [9], and one next release with bugs fixed. We simulated regression by "inverting" the version history: we considered the fixed release as the initial version (without bugs) and the buggy version as the subsequent version. Similar methodology has been used in fault localization research [48].

In this setup, we used 29 tests that we knew a priori would reveal crashes in specific configurations. Overall, these crashes exposed five distinct bugs. On a crash, the GCC testing infrastructure reports an *"Internal Compiler Error (ICE)"* message followed by a specific error description which includes the statement that manifested the crash. We used these messages to identify the crash.

Technique	*Bug Id*				
	1	2	3	4	5
EvoSPLat	✓	✓	✓	✓	✓
(Evo)Pair-wise	✓	✓	✗	✓	✗
(Evo)Six-wise	✓	✓	✓	✓	✓

Table 7: Bugs detected in GCC version 4.8.2.

Table 7 reports the effectiveness results. Overall, we observed that EvoSPLat and (Evo)Six-wise detected all bugs while (Evo)Pair-wise could only reveal three bugs. The reason is that the two non-detected bugs are only manifested when a single buggy feature is enabled (and the others are disabled), and the configurations generated for pair-wise did not explore this kind of combination with this specific feature variable.

Answering RQ5: Results indicate that (Evo)Pair-wise missed two bugs that alternative techniques found. EvoSPLat was as effective as (Evo)Six-wise but it could find bugs much faster.

[3]All bugs were confirmed by the GCC team: `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=<x>`, where x=61980, 62069, 62070, 62140, 62141.

6. Discussion

We evaluated EvoSPLat in two scenarios: RCS (Regression Configuration Selection) and RTS (Regression Test Selection). The goals of RCS and RTS are the same: to reduce execution cost. However, the mechanisms to achieve the goal as well as the contexts of application vary. RCS focuses on reduction of configurations explored by each test (with test set fixed) whereas RTS focuses on reduction of the test set (with configurations reached in each test fixed).

Compared to our original technique SPLat, we found that EvoSPLat was beneficial in both scenarios, leading to significant reduction of space or time. Compared to sampling techniques, which are very popular for testing configurable systems [49, 50, 51, 23], EvoSPLat achieved a balance between time and efficiency. It is important to note that it is in general challenging to determine the best interaction degree to use for combinatorial testing. Considering the RTS scenario, we observed that, although (Evo)Pair-wise was by far the most efficient technique, it missed two bugs. In contrast, both EvoSPLat and (Evo)Six-wise caught all bugs but required significantly more time. EvoSPLat, however, detected all bugs four times faster than (Evo)Six-wise, on average.

Although performance largely depends on the code under analysis and code changes, overall, results indicate that our approach is lightweight and effective. EvoSPLat addresses an important gap that exists between two important fields of research and practice: configurable system's testing and regression testing.

7. Threats to Validity and Limitations

Considering internal validity, coding errors could invalidate our results. To mitigate this threat, we thoroughly checked our implementation and our experimental results, looking for discrepancies signaling potential errors. The bugs detected in our experiments were validated by other individuals. Considering GCC particularly, we used Bugzilla to report bugs to the GCC team who validated the bugs we found.

Considering external validity, the selection of subjects we used in the experiment with SPLs may not be representative of programs used in industry. These subjects, albeit previously-used in prior research, are small and may be tested differently compared to real software. To mitigate this threat we also considered one configurable system that has been under active development for decades: GCC. We found that a system like GCC is indeed tested differently (e.g., test suites include more system tests compared to integration and unit tests) and required adaptation of EvoSPLat. Another threat is related to the setup we used in our GCC experiments. We selected options, tests, code versions, and heuristics according to a specific criteria given that it is impractical to exhaustively consider all combinations of independent variables. We used most frequently cited options from the GCC bug reports (in the week we focused our analysis) and used a test suite with higher incidence of bugs, observed from bug reports. Additionally, we chose well known sampling heuristics that explored configuration vectors of strengths typically considered in the evaluation of prior related work (i.e., pair-wise and six-wise) and used a mature version of GCC (4.8.2) with bugs confirmed by developers.

It is worth mentioning that EvoSPLat currently only supports configurable systems with dynamically bound variables. Although several systems are implemented this way (e.g., Groupon web [7] and GCC [6]) there is also a number of systems that use pre-processor directives as a mechanism to implement variability (e.g., Linux Kernel [22]). It remains to investigate how EvoSPLat would perform on systems with #ifdef variability. In this work, we assume that only function declarations and function bodies change across versions. Although we noticed that most changes are of this kind, in practice, it is possible that constraints change across versions, changes occur outside functions and tests change across versions. EvoSPLat currently does not support those kinds of changes. We remain to investigate how EvoSPLat would perform under those scenarios.

8. Related Work

This section presents work most related to EvoSPLat.

8.1. Regression Testing for Non-Configurable Systems

Regression testing is a field of research that can directly impact in-house testing. One case that shows such impact in industry is Ekstazi [52], a recently-available Java library for regression testing that is used by Apache Camel, CXF, and Commons Math. Regression testing also continues to attract attention from the research community. Considering regression testing for non-configurable systems, recent research has focused on the the following key problems: test selection (e.g., [53, 54, 28, 55]), test-suite prioritization (e.g., [56, 57]), test-suite augmentation (e.g., [58, 59, 60]), and test-suite reduction (e.g., [61, 62]). In the following, we discuss some of these works.

Saha et al. [57] introduce a new approach to address the problem of regression test prioritization by reducing it to a standard Information Retrieval problem, such that it does not require any dynamic profiling or static program analysis to calculate the differences between two program versions. Elbaum et al. [55] present two new techniques for selecting and prioritizing regression tests in the context of continuous integration development environments. They use readily available test suite execution history data to determine what tests are worth executing and the priority of execution. Nanda et al. [54] propose an approach to select regression tests in systems that contain frequent changes in non-code artifacts. This technique focuses on changes to property files and databases, complementing code-centric approaches. Bohme et al. [60] introduce a new regression test generation technique that stress code where change interaction may occur, with the purpose of finding more errors.

None of these works consider the dimension variability to improve regression testing. This dimension is very important in many cases and accounts for many errors in large real software [2, 3, 10]. It remains open to explore how to combine these ideas with EvoSPLat in the context of configurable systems.

8.2. Regression Testing for Configurable Systems

Previous techniques to solve this problem apply heuristics to find configurations related to evolutionary changes [18, 19, 20, 21]. We shortly summarize and related some of these works in the following. Qu et al. [18, 19] focus on regression testing of evolving configurable software systems. They present an empirical study about the impact of configuration selection heuristics used in regression testing on fault-detection capability. Their results highlight that a number of bugs may be missed if certain configurations are not tested and that prioritizing configurations allows for more effective testing. Qu et al. [20] use slicing-based code change impact analysis to assist configuration selection. The technique consists of two steps. First, it looks for a set of variables to be re-tested from a static forward slice of the program. Then, from this set of variables, they use pair-wise CIT (Combinatorial Interaction Testing) to select configurations to be re-tested. The cost of running the impact analysis was high but experimental results showed promise when compared with random and exhaustive selection.

It is important to note that these techniques inherit the limitations of combinatorial testing: relevant configurations can be missed and irrelevant configurations can be captured. EvoSPLat discovers an approximation for the set of configurations that require execution. It is very important to note that if one knew a priori that evolutionary changes would definitely *not* affect the decision tree as obtained from the previous run of the technique, then it would be possible to safely re-execute the same configurations from the previous run without tooling support (and associated overhead). However, knowing in advance that the structure of the tree would be preserved requires more sophisticated analysis, which is often too expensive in this context [13, 14, 15, 16, 17].

8.3. Other

Zhang and Ernst [2, 3] propose a technique to troubleshoot configuration errors caused by configurable systems' evolution. They use dynamic profiling, execution trace comparison, and static analysis to link the undesired behavior to its root cause, a configuration option whose value can be changed to produce desired behavior from the new software version. This work starts from failures to find out what changes caused them. Diagnosis and repairing faults in configurable systems is an important problem. It remains to observe how EvoSPLat could assist in better diagnosis of configuration problems (i.e., in finding root causes of configuration problems).

The authors of this paper recently found [63] that SPLat can be combined with sampling to balance cost of exploration with fault-detection ability. As future work, we plan to apply the ideas from that work to improve regression testing of configurable systems even further.

9. Conclusions

Testing configurable systems is important and expensive. This paper presented a technique named EvoSPLat to alleviate cost of systematic testing these systems when evolutionary information is available. The key insight is that it is possible to use simple lightweight impact analysis to discard configurations and tests during an evolution cycle.

`EvoSPLat`makes a step forward in closing the gap between research and practice in testing configurable systems. We evaluated our technique on software product lines and on a large configurable systems. In both scenarios, results obtained were encouraging. A prototype implementation of our technique is available at `https://sites.google.com/site/evosplat/`.

Acknowledgments. Sabrina was supported by the FACEPE fellowship BPG-0675-1.03/09. This material is based upon work partially supported by the Brazilian CNPq research agency under Grant No. 457756/2014-4 and by a Microsoft SEIF'13 award.

References

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

[2] S. Zhang, M. D. Ernst, Automated Diagnosis of Software Configuration Errors, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, 2013, pp. 312–321.

[3] S. Zhang, M. D. Ernst, Which Configuration Option Should I Change?, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, 2014, pp. 152–163.

[4] Firefox, Web browser, `http://hg.mozilla.org`.

[5] Linux, kernel, `http://www.kernel.org`.

[6] GCC, Compiler infrastructure, `http://gcc.gnu.org`.

[7] Groupon, A global e-commerce marketplace, `http://groupon.com`.

[8] P. Borba, M. B. Cohen, A. Legay, A. Wasowski, Analysis, test and verification in the presence of variability (dagstuhl seminar 13091), Dagstuhl Reports 3 (2) (2013) 144–170. `doi:10.4230/DagRep.3.2.144`.
URL `https://doi.org/10.4230/DagRep.3.2.144`

[9] S. Souto, D. Gopinath, M. d'Amorim, D. Marinov, S. Khurshid, D. Batory, Faster Bug Detection for Software Product Lines with Incomplete Feature Models, in: Proceedings of the 19th International Conference on Software Product Lines, SPLC'15, 2015, pp. 151–160.

[10] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, S. Apel, A Comparison of 10 Sampling Algorithms for Configurable Systems, in: ICSE, 2016, pp. 643–654.

[11] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, M. d'Amorim, SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, 2013, pp. 257–267.

[12] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: A survey, Software Testing Verification and Reliability 22 (2) (2012) 67–120.

[13] J. Law, G. Rothermel, Whole program path-based dynamic impact analysis, in: ICSE, 2003, pp. 308–318.

[14] T. Apiwattanapong, A. Orso, M. J. Harrold, Efficient and precise dynamic impact analysis using execute-after sequences, in: ICSE, 2005, pp. 432–441.

[15] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, P. Borba, SPLlift - transparent and efficient reuse of IFDS-based static program analyses for software product lines, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2013.

[16] M. Lillack, C. Kästner, E. Bodden, Tracking load-time configuration options, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), 2014, pp. 445–456.

[17] F. Angerer, A. Grimmer, H. Prähofer, P. Grünbacher, Configuration-Aware Change Impact Analysis, in: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 385–395.

[18] X. Qu, M. Cohen, K. Woolf, Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization, in: IEEE International Conference on Software Maintenance, 2007. ICSM 2007., pp. 255–264.

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

[20] X. Qu, M. Acharya, B. Robinson, Configuration Selection Using Code Change Impact Analysis for Regression Testing, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 129–138.

[21] Z. Xu, M. B. Cohen, W. Motycka, G. Rothermel, Continuous test suite augmentation in software product lines, in: SPLC, 2013, pp. 52–61.

[22] I. Abal, C. Brabrand, A. Wasowski, 42 variability bugs in the linux kernel: A qualitative analysis, in: ASE, 2014, pp. 421–432.

[23] C. Nie, H. Leung, A survey of combinatorial testing, ACM Computing Surveys 43 (2).

[24] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (Nov. 1990).

[25] S. Apel, D. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines: Concepts and Implementation, Berlin/Heidelberg, 2013, 308 pages, ISBN 978-3-642-37520-0.
URL `http://www.springer.com/computer/swe/book/978-3-642-37520-0`

[26] R. E. Lopez-Herrejon, D. S. Batory, A Standard Problem for Evaluating Product-Line Methodologies, in: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering, GCSE '01, Springer, 2001, pp. 10–24.

[27] T. J. Ostrand, M. J. Balcer, The category-partition method for specifying and generating functional tests, Communications of ACM 31 (6) (1988) 676–686. `doi:http://doi.acm.org/10.1145/62959.62964`.

[28] M. Gligoric, L. Eloussi, D. Marinov, Practical regression test selection with dynamic file dependencies, in: ISSTA, 2015, pp. 211–222.

[29] SAT4J, `http://www.sat4j.org/`.

[30] E. Fredkin, Trie memory, Commun. ACM 3 (9) (1960) 490–499. `doi:10.1145/367390.367400`.
URL `http://doi.acm.org/10.1145/367390.367400`

[31] DejaGnu, DejaGnu,
`http://www.gnu.org/software/dejagnu/`.

[32] Human-resource management system, 101Companies,
`http://101companies.org/index.php/101companies:Project`.

[33] P. Lengauer, V. Bitto, F. Angerer, P. Grünbacher, H. Mössenböck, Where Has All My Memory Gone?: Determining Memory Characteristics of Product Variants Using Virtual-machine-level Monitoring, in: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS, 2013, pp. 1–8.

[34] M. Plath, M. Ryan, Feature Integration Using a Feature Construct, Sci. Comput. Program. 41 (1) (2001) 53–84.

[35] R. Hall, Fundamental Nonmodularity in Electronic Mail, Automated Software Engineering 12 (1) (2005) 41–79. `doi:10.1023/B:AUSE.0000049208.84702.84`.

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

[37] J. Kramer, J. Magee, M. Sloman, A. Lister, CONIC: an integrated approach to distributed computer control systems, Computers and Digital Techniques, IEE Proceedings E 130 (1).

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

[39] Library for object persistence, Prevayler,
`http://prevayler.org/`.

[40] S. P. L. website, Puzzle game,
`https://code.launchpad.net/~spl-devel/spl/default-branch`.

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

[42] S. Apel, D. Beyer, Feature Cohesion in Software Product Lines: An Exploratory Study, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, 2011, pp. 421–430.

[43] GCC, GCC, the GNU Compiler Collection,
`http://gcc.gnu.org/`.

[44] GCC Document, GCC Document, `https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc.pdf`.

[45] B. Garvin, M. Cohen, M. Dwyer, Failure Avoidance in Configurable Systems through Feature Locality, in: Assurances for Self-Adaptive Systems, Vol. 7740 of Lecture Notes in Computer Science, 2013, pp. 266–296.

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

[47] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, R. Kuhn, Combinatorial testing of acts: A case study, in: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012, pp. 591–600.

[48] J. C. de Campos, R. Abreu, G. Fraser, M. d'Amorim, Entropy-Based Test Generation for Improved Fault Localization, in: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, 2013, pp. 257–267.

[49] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, Constructing test suites for interaction testing, in: ICSE, 2003, pp. 38–48.

[50] D. R. Kuhn, D. R. Wallace, A. M. Gallo, Jr., Software Fault Interactions and Implications for Software Testing, IEEE TSE 30 (6) (2004) 418–421.

[51] D. R. Kuhn, R. N. Kacker, Y. Lei, Practical Combinatorial Testing, Tech. rep., Gaithersburg, MD, United States, sP 800-142 (2010).

[52] http://camel.apache.org/building.html. See reference "Executing unit tests using Ekstazi".

[53] A. Orso, T. Xie, Bert: Behavioral regression testing, in: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA '08, 2008, pp. 36–42.

[54] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, A. Orso, Regression Testing in the Presence of Non-code Changes, in: Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST), 2011, pp. 21–30.

[55] S. Elbaum, G. Rothermel, J. Penix, Techniques for Improving Regression Testing in Continuous Integration Development Environments, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 235–245.

[56] G. Rothermel, R. J. Untch, C. Chu, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (10) (2001) 929–948.

[57] R. K. Saha, L. Zhang, S. Khurshid, D. E. Perry, An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes, ICSE '15, pp. 268–279.

[58] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, M. J. Harrold, Test-suite augmentation for evolving software, in: ASE, IEEE, 2008, pp. 218–227.

[59] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, eXpress: Guided Path Exploration for Efficient Regression Test Generation, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis, 2011, pp. 1–11.

[60] M. Böhme, B. C. d. S. Oliveira, A. Roychoudhury, Regression tests to expose change interaction errors, in: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), 2013, pp. 334–344.

[61] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, D. Marinov, Balancing Trade-offs in Test-suite Reduction, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 246–256.

[62] J. Bell, G. Kaiser, Unit test virtualization with vmvm, in: ICSE, 2014, pp. 550–561.

[63] S. Souto, M. d'Amorim, R. Gheyi, Balancing soundness and efficiency for practical testing of configurable systems, in: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, 2017, pp. 632–642.
URL http://dl.acm.org/citation.cfm?id=3097444