iTree: Efficiently Discovering High-Coverage Configurations Using Interaction Trees

Charles Song, Adam Porter, and Jeffrey S. Foster
Computer Science Department, University of Maryland, College Park, U.S.A.
Email: {csfalcon,aporter,jfoster}@cs.umd.edu

***Abstract*—Software configurability has many benefits, but it also makes programs much harder to test, as in the worst case the program must be tested under every possible configuration. One potential remedy to this problem is combinatorial interaction testing (CIT), in which typically the developer selects a strength t and then computes a covering array containing all t-way configuration option combinations. However, in a prior study we showed that several programs have important high-strength interactions (combinations of a subset of configuration options) that CIT is highly unlikely to generate in practice. In this paper, we propose a new algorithm called *interaction tree discovery (iTree)* that aims to identify sets of configurations to test that are smaller than those generated by CIT, while also including important high-strength interactions missed by practical applications of CIT. On each iteration of iTree, we first use low-strength CIT to test the program under a set of configurations, and then apply machine learning techniques to discover new interactions that are potentially responsible for any new coverage seen. By repeating this process, iTree builds up a set of configurations likely to contain key high-strength interactions. We evaluated iTree by comparing the coverage it achieves versus covering arrays and randomly generated configuration sets. Our results strongly suggest that iTree can identify high-coverage sets of configurations more effectively than traditional CIT or random sampling.**

***Keywords*-Empirical Software Engineering, Software Configurations, Software Testing and Analysis**

I. Introduction

Many modern software systems are highly configurable. While this increases extensibility, reusability, and portability, it also greatly complicates many software engineering tasks, such as software testing. This is because the number of possible configurations grows exponentially in the number of configuration options, and in the worst case, each configuration may require separate treatment. Specifically, since any configuration might harbor a distinct error, each configuration should, in theory, be tested separately—something that is impossible in practice.

To alleviate this problem, researchers have proposed *combinatorial interaction testing* (CIT) [7], [2], [23], which identifies a small but systematic set of configurations under which to test. For example, with one CIT approach, developers choose an *interaction strength* t and compute a *covering array*, which is a set of configurations such that all possible t-way combinations of option settings appear at least once. The assumption underlying CIT is that configuration sets constructed in this way are small in size while providing good coverage of the program's behavior. Thus the approach cost-effectively increases the likelihood of finding faults.

However, our prior work [29] challenges this assumption in several ways. Specifically, we hypothesized that, in practice, a system's *effective configuration space*—the minimal set of configurations needed to achieve a specific goal—typically comprises only a tiny subset of the full configuration space, and that subset of configurations is not well approximated by t-way covering arrays. To test this hypothesis, we used symbolic execution [18], [15], [6] to discover a subject program's *interactions*, which are conjunctions of option settings needed to achieve specific testing goals, given a particular test suite. In our case, the testing goal was particular forms of coverage (line, block, edge, and condition). To illustrate the concept, consider a hypothetical program with four binary-valued configuration options: a, b, c, and d. Let's assume that when the goal is line coverage, this program has exactly three interactions: $\mathsf{a} \wedge \mathsf{b} \wedge \neg \mathsf{c}$, $\mathsf{a} \wedge \mathsf{b}$, and $\mathsf{a} \wedge \mathsf{d}$. Each interaction is associated with line coverage that is guaranteed to occur in any configuration that contains the interaction. For example, suppose $\mathsf{a} \wedge \mathsf{b} \wedge \neg \mathsf{c}$ guarantees coverage of lines 5–10. In that case the configuration (a setting of all options) $\mathsf{a} \wedge \mathsf{b} \wedge \neg \mathsf{c} \wedge \mathsf{d}$ is guaranteed to cover at least those same lines. Additionally, the union of the lines covered by all three interactions is the maximal coverage achievable across all possible configurations. See our prior publication for more details [29].

Among other things, our prior work found that for our subject programs and test suites, (a) most of the interactions needed to achieve maximum coverage were strength 3 (involved 3 option settings) or greater, and (b) the largest interactions needed to achieve maximum coverage were strength 7. These findings suggest CIT approaches, which are typically applied at $t = 2$ or $t = 3$ [9], are likely missing key high-strength interactions. Put another way, while CIT at low strength can yield significant coverage, it is very unlikely to achieve the maximum possible coverage—and achieving such coverage with CIT would require prohibitively large covering arrays (e.g., strength 7).

Another finding from our prior work was that interactions were quite rare. Only a handful of specific options setting combinations had to be exercised to maximize coverage, even under a comprehensive criteria, such as path coverage. This suggests CIT's insistence on testing every t-way combination of option settings may be unnecessarily expensive.

To improve the current situation, in this paper we propose a new algorithm that addresses the shortcomings of traditional CIT. Our algorithm aims to discover sets of configurations to test that are smaller than those chosen by CIT, while also having higher coverage. To achieve this aim, we developed iTree, an *interaction tree discovery algorithm* that combines low-strength covering arrays, runtime instrumentation, and machine learning (ML) techniques to construct an *interaction tree* for the subject program. An interaction tree is a hierarchical representation of what we call *proto-interactions*, which are potential interactions or subsets of potential interactions. In an interaction tree, each node is labeled with one or more option settings, and a node represents the proto-interaction given by the conjunction of all labels on the path from the root to the node.

iTree constructs an interaction tree as follows. Initially, the tree consists of one root node containing the empty interaction *true*. At each step, we pick a leaf node; use CIT at low strength to generate a systematic sample of configurations consistent with the proto-interaction for that node; run the system's test suite under each of those configurations; and identify any newly covered program entities. If there are any, we use ML to heuristically identify new proto-interactions that are likely responsible for that new coverage, which we then add to the interaction tree. The process continues until one of several stopping criteria is met. (See Section III.)

The key intuition behind iTree stems from a third finding from our prior work. There we observed that higher strength interactions were usually just lower strength interactions with one or more additional constraints. iTree exploits this observation by performing an iterative, search-based process in which the current iteration's sample configurations are based on the last iteration's proto-interactions. In this way, the set of configurations constructed as iTree executes have the potential to provide higher coverage than correspondingly sized configuration sets produced from traditional CIT.

We evaluated iTree in several ways. First, we compared iTree's performance under various combinations of ML algorithms and CIT sampling criteria. In each case, we computed how quickly iTree reached maximum possible coverage on two subject programs, vsftpd and ngIRCd, studied in our prior work [29]; by using symbolic execution and significant computing power, the prior work was able to determine all possible program executions achievable for vsftpd and ngIRCd while varying up to 30 configuration options under a fixed test suite. We found that the best choice for iTree is to use a voting protocol to combine multiple ML classifiers, and an adaptive sampling approach that uses higher-strength covering arrays initially, and lower-strength covering arrays below the top level of the interaction tree. (See Section IV.)

Second, we compared iTree against traditional CIT and against similarly sized sets of randomly selected configurations. Again, we used vsftpd and ngIRCd as subject programs. Our results show that iTree is more likely to find high-coverage configuration sets, and it does so more rapidly than the other approaches. (See Section V.)

Note that while symbolic execution gave us a powerful and precise baseline for the first two experiments, that technology is not scalable to large systems. For example, for the 10K-LOC systems we studied, complete analysis used 40 computers working round the clock for several days.

In a final experiment, we evaluated the scalability of iTree to a large-scale system for which symbolic evaluation is infeasible, specifically the ∼1M-LOC MySQL database system. We found that iTree easily scaled up to MySQL and was again more efficient and effective than either CIT or random sampling (See Section VI).

These results suggest iTree is an important advance in the testing of highly configurable systems. Our approach will enable developers to test their systems to higher levels of coverage at less cost than currently possible with traditional CIT. We also believe the iTree process can be adapted to document a system's configuration-related structure to better support a range of software engineering tasks, such as impact analysis, reverse engineering, bug isolation, and more.

II. Effective Configuration Spaces

Our previous work [29] empirically studied how and why the execution behavior of two configurable systems, vsftpd and ngIRCd, changes in relation to how those systems are configured. As discussed earlier, the results of our previous study strongly suggest that, for a given test suite, maximal levels of coverage can be achieved with a very small number of carefully chosen configurations, i.e., systems' effective configuration spaces with respect to maximal coverage are indeed small. For developers to exploit this insight, however, they need cost-effective and time-sensitive techniques for choosing the specific configurations to test. Unfortunately, we currently know of no such techniques; the approaches we used in our previous work are computationally very expensive, and symbolic execution cannot be run to exhaustion on large, practical systems.

Key Observations: Thus, the goal of the current work is to begin creating a practical method to identify or approximate a system's effective configuration space. Our particular focus is on finding small sets of configurations for testing that yield a high degree of coverage, but we believe our approach generalizes to other software engineering tasks.

Toward this aim, we reexamined the specific interactions we found in our initial study, and made a number of observations that suggest the design of such a method. We illustrate our observations using the code in Figure 1, which contains a highly simplified snippet of vsftpd's source code. The code includes two traditional program variables, `dsa_cert_file` and `one_process_mode`, which are initialized on lines 1 and 2. In practice, `dsa_cert_file` is a program input whose value would come from a test case, but we have hard-coded it here for simplicity. The program also contains

```
1  int* dsa_cert_file=NULL; /* test input */
2  int one_process_mode=1;
3  if (listen} {
4      if (accept_timeout) {
5          /* R1: listen ∧ accept_timeout */
6      } else {
7          /* R2: listen ∧ ¬accept_timeout */
8      }
9  } else {
10     /* R3: ¬listen */
11 }
12 if (ssl_enable) {
13     if (!dsa_cert_file)
14         die();
15 }
16 /* R4: ¬ssl_enable */
17 if (one_process_mode) {
18     if (local_enable || ssl_enable)
19         die();
20 }
21 /* R5: ¬ssl_enable ∧ ¬local_enable */
22 if (!local_enable && !anonymous_enable)
23     die();
24 /* R6 (lots of code) : ¬ssl_enable∧
25     ¬local_enable ∧ anonymous_enable */
26 if (dual_log_enable) {
27     /* R7: ¬ssl_enable ∧ ¬local_enable∧
28         anonymous_enable ∧ dual_log_enable */
29 } else {
30     /* R8: ¬ssl_enable ∧ ¬local_enable∧
31         anonymous_enable ∧ ¬dual_log_enable */
32 }
```

Figure 1. An example program and its interactions.

six binary configuration options, highlighted in bold, whose values depend on the system's runtime configuration. In the actual source code, each option's name is prefixed with tunable_, but we have omitted the prefixes to save space.

Figure 1 includes eight regions of code, marked */* R1–8 */*, whose coverage we are interested in. The coverage of these regions, of course, depends on the values of the configuration options and program variables. For each region, we list the interaction that controls coverage of that region for this particular test case. For example, at the beginning of the program, the coverage of *R1–R3* depends on configuration variables listen and accept_timeout.

More interestingly, for execution to reach the large amount of code in *R6*, several options must be set in specific ways. First, to reach *R4* and any code thereafter, ssl_enable must be set to 0, because this test case sets dsa_cert_file to be NULL. Next, consider reaching *R5*. Since one_process_mode is set to 1, to reach *R5* the condition on line 18 must be false; and since as just discussed ssl_enable is 0 if we reach this line, local_enable must also be 0. Finally, to continue on to reach *R6*, we need the condition on line 22 to be false, and since local_enable $= 0$ if we reach that line, we must have anonymous_enable $= 1$. Putting this together, any configuration that reaches *R6* for this test case needs at least ssl_enable $= 0$, local_enable $= 0$, and anonymous_enable $= 1$. Finally, the coverage of *R7* and *R8* also depends on the value of dual_log_enable.

Note that although in this example we were able to reach all code regions, and coverage of each was guaranteed by a distinct interaction, in practice this is not usually the case. In actual systems some regions are unreachable with the given test suite, and some regions have more than one interaction that guarantees their coverage.

We found that the configuration option patterns just described are common in vsftpd and ngIRCd. From these patterns, we make three observations:

Observation 1: Interactions are relatively rare. The code shown in Figure 1 includes six binary options, so in the worst case there could be 639 different interactions[1]. In the example code, however, there are only eight interactions. Since some of these interactions can be simultaneously satisfied in a single configuration, only three configurations are needed to cover all eight regions. For vsftpd and ngIRCd respectively, there were only 43 and 435 interactions.

Observation 2: Most coverage is explained by low-strength interactions. In the example, five of the eight interactions involve only one or two option settings, one interaction involves three settings, and the remaining two involve four settings each. While the example is highly simplified, we found the same trend in the actual systems. For the systems and test suites we examined, over 94% of the achievable coverage could be achieved with lower-strength (size four or less) interactions. Full coverage, however, required a handful of higher-strength interactions (up to strength seven).

Observation 3: Higher-strength interactions tend to be built on lower-strength ones. As shown in the example, the higher strength interactions guaranteeing coverage of *R7* and *R8* are refinements of the interaction at *R6*, which is itself a refinement of *R5*. In implementation terms, interactions tend to arise because control-flow guards effectively stack up on each other, not because complex guards appear directly in the code. That is, higher-strength interactions often add additional constraints to existing lower-strength interactions.

III. Interaction Tree Discovery

Based on the insights discussed in Section II, we developed the interaction tree discovery algorithm (iTree). iTree's goal is to automatically discover and test a small set of high-coverage configurations. iTree works as follows. First, it instruments the system under test to measure some desired type of coverage. This paper focuses on line coverage, but the algorithm should apply to any type of coverage. Next, iTree repeats the following steps until particular stopping criteria are met. First, it computes a small sample of configurations under which to test the system. As we

[1]Computed as $1+\sum_{i=1}^{6} C(6,i)\cdot 2^i$, i.e., the sum of all ways of picking option subsets times the number of settings, plus the interaction *true*.

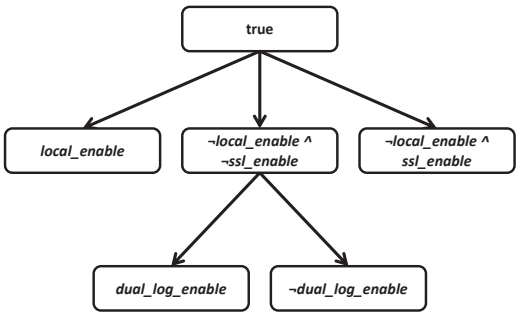

Figure 2. The interaction tree for the example program.

```
1  iTree = /* tree containing root 'true' */
2  runs = ∅ /* (config × coverage) set */
3  do {
4      node = findBestLeafNode(iTree, runs);
5      configSet = generateConfigSet(node.proto_interaction);
6      newruns = executeConfigSet(configSet);
7      if cov(newruns) ⊆ cov(runs)
8          continue;
9      runs = runs ∪ newruns
10     interactions = discoverProtoInters(node.proto_interaction, runs);
11     if !(interactions.empty())
12         /* add newly discovered interactions to tree */
13         updateTree(iTree, node, interactions);
14 } while (!stoppingCriteriaMet());
```

Figure 3. Pseudocode for the interaction tree discovery.

shall see later, the sample is chosen to select configurations that are likely to exercise previously uncovered entities. Next, iTree runs the system's test suite on each of the sampled configurations and records coverage information. Using this coverage data, iTree then attempts to learn *proto-interactions*—conjunctions of option settings—that cause the new coverage and that may warrant further exploration in the next iteration of iTree.

We represent iTree's behavior as an *interaction tree*, which is a hierarchical representation of proto-interactions. The nodes of the tree represent proto-interactions rather than interactions because they may not, in fact, be full-fledged interactions; because iTree is heuristic in nature, some nodes may represent only portions of interactions, or may represent full interactions with additional constraints.

Figure 2 shows the interaction tree for Figure 1. Each node is labeled with a set of option settings, with *true* at the root node (corresponding to the empty setting). A node represents the proto-interaction that is the conjunction of settings along the path from the root to the node. For example, the interaction $\neg$local_enable $\wedge$ $\neg$ssl_enable $\wedge$ dual_log_enable is represented by the left node on the lowest level of the tree. We also see that $\neg$local_enable $\wedge$ ssl_enable is in the interaction tree, but does not correspond to an actual interaction that guarantees coverage of particular code regions. Thus, in this case, iTree has created a proto-interaction that will not lead to useful higher-strength interactions.

Algorithm: Figure 3 gives the pseudocode for the iTree algorithm. iTree runs in a loop, iterating until developer-supplied stopping criteria are met (e.g., no more coverage is achieved or a time limit has expired). The iTree algorithm begins with an interaction tree iTree containing just one node, *true*. As the iTree progresses, it also records in runs the set of all configurations executed so far and their corresponding coverage information. At the beginning of each iteration, findBestLeafNode uses various heuristics to pick a leaf node to explore next. (This heuristic is important because we do not expect to fully explore the interaction tree, as that would be too expensive.)

Next, the proto-interaction represented by the path to the selected node is passed to generateConfigSet. This method creates a sample set of configurations that are consistent with the proto-interaction represented by the selected node, while the set of configurations broadly samples all options not participating in the proto-interaction. Currently, iTree leverages CIT for this step, but other sampling techniques could be substituted.

Next, executeConfigSet compiles, instruments, and executes the system's test suite under each configuration in the sample. The data from the resulting executions is then added to runs. Then runs and node.proto_interaction, the proto-interaction represented by node, are passed to discoverProtoInters, which uses machine learning to discover additional proto-interactions that account for any newly covered entities. Note that, by design, any proto-interactions discovered at this step must include the settings in node.proto_interaction. Finally, updateTree adds the newly discovered proto-interactions to the interaction tree as children of the current node.

We now discuss each step of the algorithm in more detail.

findBestLeafNode: Since iTree aims to find high-coverage configurations, findBestLeafNode prioritizes nodes by the amount of coverage achieved by configurations containing the node's proto-interaction. The assumption is that proto-interactions corresponding to high-coverage configurations are more likely to lead to uncovered code with further exploration. iTree computes a node's priority as follows. First, let $\mathit{Conf}(\mathsf{runs}, \mathsf{node})$ be the subset of runs whose configurations are consistent with node's proto-interaction. For a run $r \in \mathit{Conf}(\mathsf{runs}, \mathsf{node})$, define $\mathit{cov}(r)$ as the number of entities covered by r. Then node's priority is given by

$$\mathit{priority}(\mathsf{node}) = \frac{\sum_{r \in \mathit{Conf}(\mathsf{runs},\mathsf{node})} \mathit{cov}(r)}{|\mathit{Conf}(\mathsf{runs}, \mathsf{node})| + 1}$$

and the highest-priority node is chosen. The formula simply computes a slightly biased average coverage for all configurations that are consistent with the node's proto-interaction. The bias of one added in the denominator means that nodes corresponding to few runs will have lower priority than their average, but has little effect on nodes corresponding

	ssl	loc	lis	acc	anon	dual
C1	1	1	0	1	0	1
C2	1	0	1	1	1	1
C3	0	0	0	1	0	0
C4	0	1	1	0	0	0
C5	0	0	0	0	1	1
C6	1	1	1	0	1	0

(a) Initial covering array

	ssl	**loc**	lis	acc	anon	dual
C7	**0**	**0**	1	1	0	0
C8	**0**	**0**	0	0	0	1
C9	**0**	**0**	1	0	1	1
C10	**0**	**0**	0	0	1	0
C11	**0**	**0**	0	1	1	1

(b) Covering array with ssl = loc = 0

ssl=ssl_enable	loc=local_enable	lis=listen
acc=accept_timeout	anon=anonymous_enable	dual=dual_log_enable

Figure 4. Example 2-way covering arrays.

to many runs (since then $|Conf(\mathsf{runs}, \mathsf{node})|$ is high). We found this adjustment useful in that it leads to a slight, but beneficial, preference for nodes that correspond to multiple, high-coverage configurations, over nodes that correspond to fewer, high-coverage configurations.

generateConfigSet: This function generates a sample set of configurations, each of which is consistent with its parameter node.proto_interaction. To do this we use a CIT tool called CASA [8] to generate a low-strength covering array over only the remaining options. We then combine those partial configurations with the settings from node.proto_interaction. In our experiments, we used both 2- and 3-way covering arrays in this step, and found the performance was not sensitive to this choice.

Figure 4 shows two covering arrays created by generateConfigSet as iTree discovered the interaction tree in Figure 2. In this case we chose to generate 2-way covering arrays. Figure 4(a) gives the covering array picked in the first iteration of iTree. Interestingly, our 2-way covering array happened to include both the 3-way interaction (see Figure 1) $\neg$ssl_enable $\wedge$ $\neg$local_enable $\wedge$ anonymous_enable (in C5) needed to reach *R6* and beyond, and the 4-way interaction $\neg$ssl_enable $\wedge$ $\neg$local_enable $\wedge$ anonymous_enable $\wedge$ dual_log_enable (also in C5) needed to reach *R7*. After the data from these configurations was analyzed, iTree added the three children of *true* shown in Figure 2.

The next iteration of iTree expanded the middle of the three nodes (since this node covered *R6*, which contains many lines), and generateConfigSet created the covering array shown in Figure 4(b). Note that in this covering array, ssl_enable and local_enable are fixed. As a result, the 2-way covering array of the remaining options is very effective, and includes both 4-way interactions (the one mentioned, plus the one needed to reach *R8*, in C7 and C10). At this point, iTree has covered all the marked regions of the program.

executeConfigSet: In this step we instrument the system under test and execute its test suite on each configuration in the sample. We compute line coverage with gcov. We ran the instrumented programs on Skoll, a distributed, continuous quality assurance system running on a grid comprising 120 CPUs [25]. As we will see in Section VI, Skoll allowed us to scale up iTree, running and analyzing many jobs at once.

discoverProtoInters: Finally, we use a two step process to discover proto-interactions to add to the interaction tree: First, we statistically cluster configurations according to their coverage data, and second, we try to find proto-interactions responsible for differences in execution.

In the first step, we find all runs involving configurations consistent with the proto-interaction we are exploring. Note that we extract this subset from all of runs, not just those newly explored in the current iteration—this way we get better information as iTree progresses. We then cluster these runs using Weka's [16] implementation of CLOPE [34], a clustering algorithm that groups together similar transactional data records with high dimensionality. As input to CLOPE, we represent each line as a boolean attribute set to *true* if covered in a run and *false* otherwise. Then we use CLOPE to cluster together configurations that execute many of the same lines.

In the second step, we use decision trees [28] to discover commonalities among configurations in each of the clusters. In our implementation, each configuration option is an attribute, and the cluster that a configuration belongs to is the class. The decision tree algorithm then builds a model for classifying the cluster to which a configuration belongs based on the configuration's option settings. If the resulting model identifies specific option settings that predict cluster membership, then we treat them as new proto-interactions and append them to the interaction tree to form higher-strength proto-interactions. Otherwise no new proto-interactions are added, and exploration of this path stops. In our experiments we evaluated several decision tree algorithms and found each to be adequate for this task.

We should mention that CLOPE requires a special parameter called repulsion, which ranges from 0.5 to 4.0, and which controls the ease with which clusters form. To make iTree completely automated, we implemented a voting system to adaptively select an appropriate repulsion value. Each time discoverProtoInters is called, CLOPE is run multiple times with repulsion values ranging from 0.5 to 4.0 in increments of 0.5. We perform the second step of the interaction discovery process using the clusters generated under each repulsion value. At the end of discoverProtoInters, we keep the most frequently occurring unique proto-interactions generated under the range of repulsion values.

stoppingCriteriaMet: iTree allows its users to plug in their own criteria for determining when to halt execution. Our default is to halt execution when the interaction tree

	vsftpd	ngIRCd
Version	2.0.7	0.12.0
# Lines (sloccount)	10,482	13,601
# Run-time opts.	30	13
Boolean/Enum	20/10	5/8
Full config space	2.1×10^9	2.9×10^5
# Test cases	64	141
# Max coverage	2,549	3,193

Figure 5. Program statistics for vsftpd and ngIRCd. Note that we removed some unreachable code before measuring lines of code.

has no more unexplored proto-interactions. The experiments in the following sections include other criteria as well, e.g., in some experiments, we stop execution when a maximum number of configurations have already been tested. Another possibility is to use wall clock time as a stopping criteria, e.g., when doing nightly testing.

IV. Evaluating iTree Parameters

We explored iTree's cost-effectiveness in a series of experiments, described in this and the next two sections. Our first experiment, presented next, aims to determine two key algorithmic parameters: the covering array strength to use in generateConfigSet, and the decision tree implementation to use in discoverProtoInters. Our second experiment explores modifying iTree to adaptively select covering array strength and use multiple decision tree approaches simultaneously. Our remaining experiments compare the coverage achieved by iTree, random sampling, and a single, high-stength covering array (Sections V and VI).

A. Experimental Setup

Subject Programs: For our first experiment, we used vsftpd, a widely used secure FTP daemon, and ngIRCd, the "next generation IRC daemon." We studied these systems extensively in prior work [29]. From our prior work, we have test suites for these programs and detailed information about the programs' configuration spaces with respect to those test suites. Figure 5 gives descriptive statistics for each system. They have roughly 10–13KLOC, and are written in C. The figure details the total number of configuration options we analyzed, broken down by type (boolean or integer). This is the same set of options and settings we used in our prior work. The values we used for the integer options also came from our previous work, and were chosen to maximize path coverage for these subject programs and test cases. Finally, the last rows list the size of the full configuration space for the options (the total number of different possible configurations); the number of test cases in our test suite; and the maximum possible number of lines covered if we execute every test case under every possible configuration.

Covering Array Strengths: Each iTree iteration begins by creating a sample of configurations, derived from a t-way covering array. The value of t determines the size of each sample, and may also influence the speed with which iTree terminates. In this study, we use either $t = 2$ or $t = 3$.

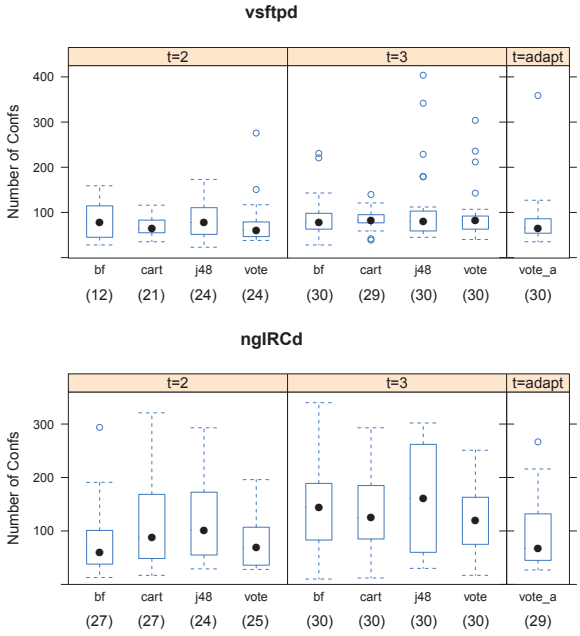

Figure 6. Interaction tree effort for different iTree parameters.

Decision Tree Algorithms: Many different decision tree classifiers have been proposed in the machine learning literature. We used three algorithms in our experiment: J48, CART, and Best-First, all as implemented in Weka [16]. We picked J48 and CART because they are the most popular decision tree implementations. We chose Best-First because it is designed to produce compact classifications, which may be well-suited to iTree's incremental search approach.

Experimental Design: We ran iTree 30 times on both subject programs under each possible combination of decision tree and covering array strength. For each run, we continued execution until we reached the maximum possible coverage (as determined from our prior work [29]). The number of configurations executed is our metric for finding the best parameter settings—the lower the number, the faster the algorithm achieves full coverage. Note that in these experiments, rather than actually run the executeConfigSet step we instead used the code coverage data we had already computed in our prior work (which gave us a mapping from configurations to their coverage).

B. Data And Analysis

Figure 6 shows boxplots of our experimental results for vsftpd and ngIRCd. The left half of each chart shows the results of the decision trees for $t = 2$, and the right half shows the results for $t = 3$. The y-axis reports the number of configurations required to achieve full coverage. The number in parentheses under each box indicates the number of runs, out of 30, in which full coverage was achieved. We defer discussion of vote and vote_a to Section IV-B3.

1) Covering Array Strengths: Figure 6 shows that increasing the t strength of the covering arrays did not greatly

change the cost of running iTree for vsftpd. It did have some effect for ngIRCd, where the average size of the configuration sets increased across all decision tree algorithms. However, we also see that the number of runs in which iTree reached maximal coverage is substantially higher when $t = 3$ than when $t = 2$. We looked in more detail at the individual runs and observed that, at each iteration, both the likelihood of discovering proto-interactions and the accuracy of the discovered proto-interactions dramatically improved as t increased. However, there was a trade off: while increased sample size meant more cost at each iteration, it also resulted in fewer overall iterations for our subject systems. In the end, the total cost did increase for ngIRCd.

We note that variance in the number of configurations tested appears unrelated to covering array strength. Instead, it seems more tied to the system tested. In particular, for vsftpd, the range of the number of configurations tested is fairly stable, while for ngIRCd it fluctuates considerably. Based on further analysis, we believe this occurs because the configuration space model for ngIRCd, taken from our previous work, contained many redundant option settings from the perspective of line coverage, i.e., a given line of code could be hit in many different configurations. Therefore, in a sense ngIRCd's 2-way covering arrays already enjoyed the benefits of larger configuration samples.

2) Decision Trees: In Figure 6, the first three columns for each t-value show the effect of the Best-First, CART, and J48 decision trees on iTree. The data shows no systematic performance differences across the algorithms. Looking at the individual iterations of iTree, however, we did find some differences: Best-First and CART fail to discover any proto-interactions in the configuration sample more often than J48 does, but J48 tends to produce less accurate classifications—it more often includes option settings that are not part of the actual interactions. Both situations can have negative consequences. For instance, failing to discover a real interaction will cause iTree to improperly abandon the currently selected node and continue with a lower-priority proto-interaction. This can delay or even prevent coverage of some necessary higher strength interactions. Discovering inaccurate proto-interactions, on the other hand, can be worse. Because iTree builds higher-strength proto-interactions on top of lower-strength ones, inaccurate option settings can propagate through an iTree path, and ultimately could prevent iTree from achieving complete coverage.

3) Hybrid Approaches: When we examined the worst-performing runs from the previous experiments, we found they suffered from inaccurate early classification that rippled through subsequent iterations of iTree. To avoid this problem, we took two steps. First, we developed a voting system, similar to bagging [1], that creates an ensemble classifier out of several simple classifiers. The voting algorithm filters out option setting combinations unless at least 2 out of 3 decision trees produce them as classifiers. The results using the voting system are shown in the `vote` column.

Second, we developed an adaptive sampling approach, in which we create three, 2-way covering arrays in the first iteration, and then one, 2-way covering array in each remaining iteration. Also, if an interaction tree path is about to terminate because no new proto-interactions have been found, we generate and test one new covering array before abandoning that path. The results using both voting and adaptive sampling are shown in the `vote_a` column.

We can see from the figure that `vote_a` is an attractive choice overall—its average cost is lower or only slightly worse than the best of the other algorithms, and it yields full coverage on every or almost every run.

V. Comparing iTree to Other Approaches

As mentioned earlier, both CIT and random sampling are popular approaches for generating configuration samples and produce relatively good results in practice. To better understand how iTree compares with these existing approaches, we conducted a series of experiments.

Experimental Design: For these experiments, we again used vsftpd and ngIRCd and ran each technique 30 times. One problem with CIT and random sampling is that developers cannot know *a priori* how large a sample is necessary. For CIT, developers must pick a t value, and for random sampling developers must guess a sample size. In this experiment, we created covering arrays using a range of different t strengths. For each strength, testing ran until either the maximum possible coverage was achieved or until no more configurations remained. Using 5- and 4-way covering arrays for vsftpd and ngIRCd, respectively, often produced maximal coverage, so we used those as our largest sample sizes. We next tested the systems with random samples sized equal to the average size of these largest covering arrays.

We also tested the systems using iTree. For this experiment, we used `vote_a` as described in the previous section, and set iTree to stop when either no new coverage is achieved on any path or the number of configurations tested exceeds the size of the random configuration samples. We measure performance using two criteria: (1) whether complete coverage was reached and (2) if so, the number of configurations needed to reach the complete coverage. We ignore the time needed to generate the configurations because in our experiments this time is dramatically smaller than the time required to run the test cases.

Data and Analysis: Figure 7 shows the results of these experiments. The x-axis is the number of configurations tested so far in each run, and the y-axis is the median number of lines covered at that point across all runs. Here we are assuming that configurations are tested in the order they are generated, although in actuality the work can be done in parallel across multiple CPUs. The 10 data points plotted in each figure divide the time line into equal epochs, corresponding to 36 or 53 configurations tested for vsftpd

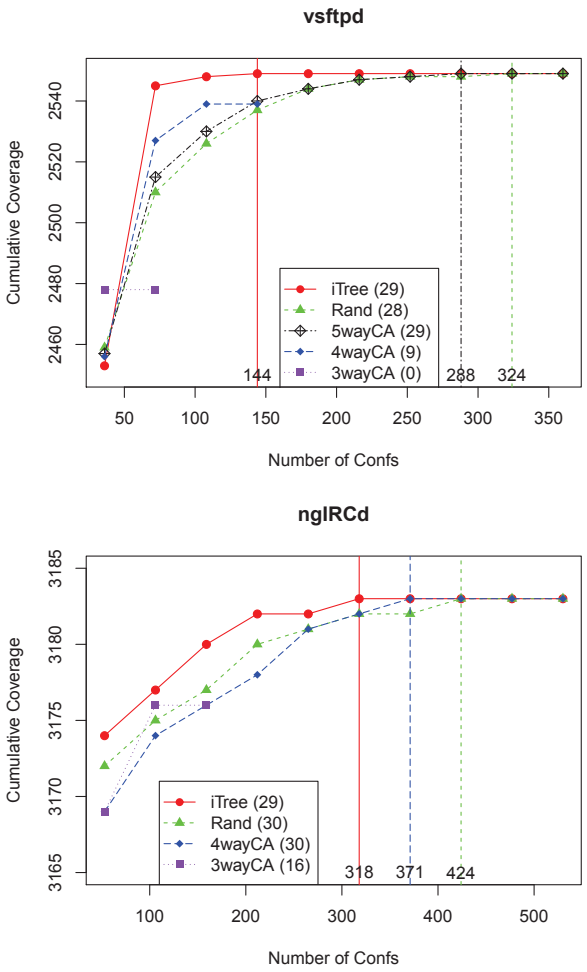

Figure 7. Comparing iTree against covering arrays and random sampling.

and ngIRCd, respectively. Note that iTree runs can terminate before executing the number of configurations of the other methods. In that case, we simply treat subsequent time points as unchanged from the previous time point. The figures also include a vertical line indicating the epoch in which 90% of the runs achieved maximal coverage. The numbers in parentheses in the legend indicate the total number of runs, out of 30 each, that reached full coverage for each approach.

In the top portion of Figure 7 showing the vsftpd results, we see that iTree, 5-way covering arrays, and random sampling eventually reached full coverage in almost all runs (29, 28, and 29, respectively), but 4-way only reached full coverage in a third of the runs, and 3-way never reached full coverage. Moreover, looking at the vertical lines, we see that 90% of the iTree runs reached full coverage with just over half the number of configurations, on average, of 5-way covering arrays, which themselves did noticeably better than random sampling. We see a similar trend for ngIRCd in the bottom figure, for which iTree, 4-way covering arrays, and random sampling achieved full coverage in all or almost all runs (29, 30, and 30, respectively), but 3-way covering

	MySQL
Version	5.1
# Lines (sloccount)	939,842
# Compile-time opts.	8
Boolean/Enum	8/0
# Run-time opts.	8
Boolean/Enum	4/4
Full config space	5.9×10^5
# Test cases	1244

Figure 8. MySQL program statistics.

arrays only reached full coverage in just over half the runs (16). Again, 90% of the iTree runs reached full coverage faster than 4-way covering arrays, 90% of which reached full coverage faster than random sampling. While iTree's benefit for ngIRCd is not quite as stark as for vsftpd, it still provides noticeable improvement over the other approaches.

4) Discussion: Overall, these results showed iTree performing better than t-way covering arrays and random sampling, at substantially less cost. This conclusion, of course, depends on how those approaches are actually used. For example, if developers used high strength covering arrays or large random samples, they would be likely to get most of the available coverage, but would do so at large cost. As we know from our previous research, this is not a very efficient approach, because few of those configurations are really necessary to achieve specific types of coverage, such as line coverage. For instance, it would require a 7-way covering array with thousands of configurations to guarantee complete line coverage for vsftpd and ngIRCd. If developers instead used a low-strength, 2-way covering array, the cost would be much lower, but so would the coverage.

VI. Evaluating Scalability

Using iTree, we were able to achieve maximal coverage while executing on average about 100 configurations for both vsftpd and ngIRCd. This is encouraging, but after all, we had already solved this problem using symbolic execution, albeit at a far higher cost. However, ultimately our goal is to handle much larger systems, written in a variety of languages, with compile-time as well as run-time configuration options. None of these issues can currently be addressed using symbolic execution, but we believe that iTree may be the right tool for this problem.

To better understand this issue, we evaluated the scalability of iTree by running it on MySQL, a popular open source database. We are not aware of any current symbolic execution system that can fully handle this system. MySQL has more than 900K lines of code. It is written in a combination of C and C++, and its configuration space includes a large number of run-time as well as compile-time configuration options. As in our previous experiments in Section V, our evaluation compared iTree against covering arrays and randomly sampled configurations.

Subject Program: Figure 8 gives descriptive statistics for MySQL. The top two rows list the version we used and the lines of code it contains as computed by sloccount [33]. Next, the figure lists the number and types of configuration options we selected for our experiment. We give the numbers of compile-time and run-time configuration options separately, and each number is also broken down by type (boolean or enumeration). All told, we are focusing on 16 configuration options. We selected configuration options and settings that enabled the test suite to exercise the major configurable features of MySQL, such as default storage engines, SQL modes, and transaction isolation modes. All other MySQL options were left with their default values.

The next row in Figure 8 lists the number of unique configurations that can be generated given the number of distinct settings of configuration option. All told, the full configuration space given the subset of MySQL options we are considering includes roughly 600K configurations. Finally, the last row in the figure lists the number of test cases (1244) comprising the regression test suite that comes with MySQL, which we used for our experiment and focused on improving its line coverage. We should note that this is not a comprehensive high-coverage test suite, the default configuration achieves ∼90K LOC. We also note that not every test case runs in every configuration.

Experimental Design: Our experimental design is similar to that of Section V. Specifically, we compare 3-way covering arrays, 4-way covering arrays, random sampling, and iTree. On average, 3-way coverings contained 58 configurations, 4-way covering arrays contained 190 configurations, and random sampling also selected 190 configurations. We executed each approach 30 times and computed how much line coverage was achieved under each. For iTree we used the vote approach. One key difference between this experiment and the last is that we cannot know the maximal possible coverage achievable by the test suite, and so we only discuss observed line coverage.

We executed the experiment on the Skoll cluster using up to 90 CPUs at a time. Executing the MySQL test suite takes approximately 1.5 hours. The process involves downloading the MySQL source tree from source code repository; compiling an instance according to the compile-time option settings for the configuration to be tested; instrumenting the instances with gcov; starting the instance with the run-time option settings dictated by the configuration to be tested; running the test suite; and collecting the execution data.

Data and Analysis: Figure 9 summarizes the experimental results. The figure shows the growth in median coverage over time under each of the four approaches used, measured at 10 equally spaced intervals. The y-axis is the number of covered lines, and the x-axis indicates the number of configurations tested. We can see from these results that iTree covered more lines of code on average than the other methods after running the same number of configurations.

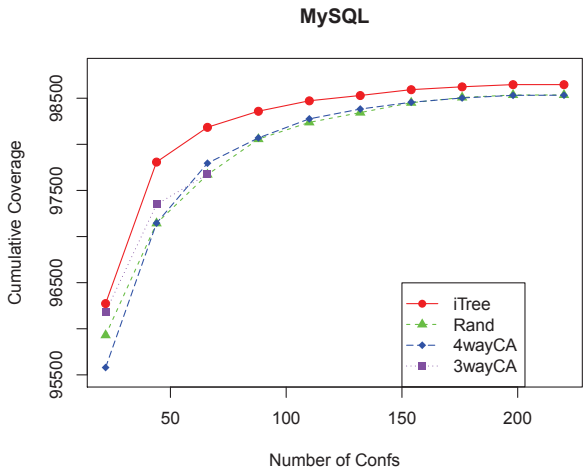

Figure 9. Comparing the number of configurations and coverage achieved using different testing approaches.

Interestingly, the traditional methods have very similar performance profiles. Thus, with respect to this data, it appears that at every level of effort, iTree-selected samples included configurations with unique coverage patterns that were not found by more traditional approaches.

The absolute difference in line coverage ranges from a high of around 0.7% (∼ 700 LOC) early on down to about 0.1% (∼ 115 LOC) near the end of the experiment. To better understand why these lines were found by iTree, but not by the other methods, we manually inspected MySQL's source code. We observed that the extra lines covered with iTree involved many small pockets of code scattered across numerous files, methods, and blocks and are apparently only executed in very specific circumstances. We further attempted to determine what interactions control the lines that are covered by iTree and not the other approaches, but were unable to decide this because of MySQL's size and complexity. However, we generated a 5-way covering array, executed its configurations, and found that none of these configurations covered those lines, either. This implies that the interactions controlling the lines in question are of strength 6 or higher.

VII. Threats to Validity

Like any empirical study, our observations and conclusions are limited by potential threats to validity. For example, in this work we used 3 widely used subject programs. Two are medium-sized; one is quite large. To balance greater external validity against higher costs and lessened experimental control, we focused on subsets of configuration options that we determined to be important. The size of these sets was substantial, but did not include every possible configuration option to keep our analyses tractable. The structural coverage criteria was line coverage. Other program behaviors such as data flows or fault detection might lead to different performance trade-offs. Our test suites taken together have reasonable, but not complete, coverage. Individually, the test

cases tend to focus on specific functionality, rather than combining multiple activities in a single test case. In that sense they are more like a typical regression suite than a customer acceptance suite. We intend to address each of these issues in future work.

VIII. Related Work

Combinatorial Interaction Testing. Covering array-based sampling for software testing is a specification-based technique that was originally proposed as a way to ensure even coverage of combinations of input parameters to programs [7], [2], [10], [5]. In more recent work, covering arrays have been used to model configurations that should be selected for testing [13], [27], [35], where the covering array defines a *test schedule* and each configuration is tested with an entire test suite. Covering arrays have also been used to test graphical user interfaces [37] and in model based testing [4].

Empirical research suggests testing with covering arrays with $t < 6$ can potentially find a large proportion of interaction faults [20]. Further studies suggest covering arrays can be effective in practice and can yield good structural coverage during testing [13], [26], [27], [35], [7], [12], [19].

Machine Learning in Software Engineering. Many researchers have proposed using dynamic analysis with machine learning techniques to analyze program executions. Haran et al. [17] developed three techniques—association trees, random forests, and adaptive sampling association trees—to automatically classify fielded software system executions. Podgurski and colleagues [11], [24], [14] used tree-based strategies and random sampling to classify program faults in order to prioritize software failure reports. Brun and Ernst [3] use machine learning to classify program invariants that manifest themselves in failing program runs. We are not aware of any other work that applies machine learning to testing software configurations.

Test Case Prioritization. The iTree approach is similar to some test prioritization techniques which try to find effective orderings of test cases that reveal faults earlier in the testing process. Many such techniques also utilize structural coverage as the surrogate criteria for prioritization [31], [32], [30]. Leon et al. [21] evaluated distribution-based cluster filtering on the execution profiles as prioritization scheme and found that it detects different faults than coverage-based prioritization. Yoo et al. [36] also applied clustering of test case dynamic runtime behavior to aid test prioritization. Li et al. [22] evaluated greedy, metaheuristic and evolutionary search algorithms for prioritization.

IX. Conclusions and Future Work

We have presented a new and scalable technique called iTree to support the testing of highly configurable systems. iTree's goal is to select a small set of configurations in which the execution of the system's test suite will achieve high coverage. This technique is based upon insights gained from our previous empirical studies in which we precisely quantified the relationships between software configuration and program execution behaviors. These insights led us to create a heuristic process that effectively searches out configurations in which high coverage is likely.

To evaluate iTree, we conducted several sets of experiments. Keeping existing threats to validity in mind, we tentatively drew several conclusions. All of these conclusions are specific to our programs, test suites, and configuration spaces; further work is needed to establish more general trends. The first set of studies evaluated the basic iTree approach and its parameters. Based on these efforts we developed several optimizations, such as adaptive voting, that improve robustness while also removing many issues that must be handled manually with current techniques. The second set of studies compared iTree with t-way covering arrays and random sampling, both existing techniques. The studies suggested that iTree produced higher coverage then the other techniques while testing fewer configurations. The third set of studies focused on scalability. This study applied iTree to MySQL, a large and popular database system. The results strongly suggested that iTree achieved higher coverage at lower cost than existing techniques. Taken together, our results strongly suggest that iTree is a promising technique that can scale to practical industrial systems.

Based on this initial work, we plan to pursue several research directions. First, we will extend our studies to include more systems with larger and more complex configuration spaces. Second, we plan to enhance iTree to incorporate new kinds of coverage. We will also examine how information gained as iTree operates might be incorporated into iTree's heuristics. Finally, we will explore post-processing the information and artifacts that iTree creates to support other software engineering tasks such as impact analysis, reverse engineering, and automatic architecture documentation.

Acknowledgments

This research was supported in part by NSF CCF-1116740 and NSF-CCF-0811284. The author Charles Song was partially supported by Fraunhofer CESE, USA.

References

[1] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996. 10.1007/BF00058655.

[2] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41–7, 1992.

[3] Y. Brun and M. D. Ernst. Finding Latent Code Errors via Machine Learning over Program Executions. In *ICSE*, pages 480–490, 2004.

[4] R. Bryce and C. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology*, pages 960–970, 2006.

[5] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *ICSE Analysis & Review*, 1998.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *Trans. Soft. Eng.*, 23(7):437–44, 1997.

[8] M. B. Cohen. Combinatorial interaction testing portal: Casa, 2009.

[9] J. Czerwonka. Pairwise testing in real world, practical extensions to test case generators. In *PNSQC*, 2006.

[10] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE*, pages 285–294, 1999.

[11] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE*, pages 339–348, 2001.

[12] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *ICSE*, pages 205–215, 1997.

[13] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *ISSTA*, pages 177–188, 2009.

[14] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-Based Methods for Classifying Software Failures. In *ISSRE*, pages 451–462, 2004.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[17] M. Haran, A. Karr, M. Last, A. Orso, A. Porter, A. Sanil, and S. Fouche. Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks. *Trans. Soft. Eng.*, 33(5):287–304, May 2007.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[19] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. In *NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[20] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *Trans. Soft. Eng.*, 30:418–421, 2004.

[21] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *ISSRE*, pages 442–453, 2003.

[22] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *Tarns. Soft. Eng.*, 33(4):225–237, 2007.

[23] R. Mandl. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054–1058, 1985.

[24] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated Support for Classifying Software Failure Reports. In *ISSRE*, page 465, 2003.

[25] A. Porter, C. Yilmaz, A. M. Memon, D. C. Schmidt, and B. Natarajan. Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. *Trans. Soft. Eng.*, 33(8):510–525, August, 2007.

[26] X. Qu, M. Cohen, and K. Woolf. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization. In *ICSM*, pages 255 –264, 2007.

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

[28] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986. 10.1007/BF00116251.

[29] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, pages 445–454, 2010.

[30] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. The impact of test suite granularity on the cost-effectiveness of regression testing. In *ICSE*, pages 130–140, 2002.

[31] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–188, 1999.

[32] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Trans. Soft. Eng.*, 27(10):929–948, 2001.

[33] D. A. Wheeler. Sloccount. http://www.dwheeler.com/sloccount/.

[34] Y. Yang, X. Guan, and J. You. CLOPE: a fast and effective clustering algorithm for transactional data. In *KDD*, pages 682–687, 2002.

[35] C. Yilmaz, M. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Trans. Soft. Eng.*, 31(1):20–34, 2006.

[36] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ISSTA*, pages 201–212, 2009.

[37] X. Yuan, M. Cohen, and A. M. Memon. Covering Array Sampling of Input Event Sequences for Automated GUI Testing. In *ASE*, 2007.