ABSTRACT

Title of dissertation: UNDERSTANDING, DISCOVERING AND LEVERAGING A SOFTWARE SYSTEM'S EFFECTIVE CONFIGURATION SPACE

Charles Song
Doctor of Philosophy
2011

Dissertation directed by: Professor Adam Porter
Department of Computer Science

Many modern software systems are highly configurable. While a high degree of configurability has many benefits, such as extensibility, reusability and portability, it also has its costs. In the worst case, the *full* configuration space of a system is the exponentially large combination of all possible option settings and every configuration can potentially produce unique behavior in the software system. Therefore, this *software configuration space explosion* problem adds combinatorial complexity to many already difficult software engineering tasks.

To date, much of the research in this area has tackled this problem using black-box techniques, such as *combinatorial interaction testing (CIT)*. Although these techniques are promising in systematizing the testing and analysis of configurable systems, they ignore a system's internal structure, which we think is a huge missed opportunity. We hypothesize that systems are often structured such that their *effective configuration spaces* – the set of configurations needed to achieve a specific

goal – are often much smaller than their full configuration spaces. If we can efficiently identify or approximate the effective configuration spaces, then we can use that information to greatly improve various software engineering tasks.

To understand the effective configuration spaces of software systems, we used *symbolic evaluation*, a white-box analysis, to capture all executions a system can take under any configuration. The symbolic evaluation results confirmed that the effective configuration spaces are in fact the composition of many small, self-contained groupings of options. We also developed analysis techniques to succinctly characterize how configurations interact with a system's internal structures. We showed that while the majority of a system's interactions are relatively low strength, some important high-strength interactions do exist, and existing approaches such as CIT are highly unlikely to generate them in practice.

Results from our in-depth investigations serve as the foundation for developing new approaches to efficiently discover effective configuration spaces. We proposed a new algorithm called *interaction tree discovery (iTree)* that aims to identify sets of configurations 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 covering arrays to test the system under, 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 and our results strongly suggest that iTree can identify high-coverage sets of configurations more effectively than traditional CIT or random

sampling.

We next developed an *interaction learning* approach that estimates the configuration interactions by building classification models for iTree execution results. This approach is light-weight, yet produces accurate estimations for the interactions, making leveraging effective configuration spaces practical for many software engineering tasks. Using this approach, we were able to approximate the effective configuration space of the ∼1M-LOC MySQL at very low cost, something that is infeasible using existing techniques.

UNDERSTANDING, DISCOVERING AND LEVERAGING A SOFTWARE SYSTEM'S EFFECTIVE CONFIGURATION SPACE

by

Charles Song

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2011

Advisory Committee:
Dr. Adam Porter, Chair/Advisor
Dr. Jeff S. Foster
Dr. Hal Daume' III
Dr. Rance Cleaveland
Dr. David Barbe

Dedication

This dissertation is dedicated to my loving and supportive parents:

Huafu Song and Mingxun Cai

Acknowledgments

This dissertation and my graduate career would not have been possible without the assistance and support of many. I would like to extend my thanks to all of them.

First, I would like to thank my advisor, Adam Porter. The research presented in this dissertation was only possible because he taught me how to do research, write papers, and give talks. I admire his ability to develop new ideas and to explain complex concepts in such elegant ways. Adam has been one of the most important mentors in my life and I thank him for teaching me essential lessons that I will need not only in research but in life as well.

I also would like to thank Jeff Foster. It has been my pleasure to collaborate with Jeff on my research. I am grateful for his ready willingness to provide guidance and I value his astute observations on research problems that helped me focus my research agenda.

I thank Hal Daume' III for providing his expertise in machine learning to my research. Whenever I needed help, Hal has always been there patiently giving valuable information that ultimately improved my work.

I thank the members of my committee: Adam Porter, Jeff Foster, Hal Daume' III, Rance Cleaveland and David Barbe. I am very grateful for your support and effort. Your insightful feedback helped me improve and refine this dissertation.

I would like to give special thanks to Elnatan Reisner and Kin Keung Ma for their collaboration on a vital part of my research. I am fortunate to have had the experience of working with these intelligent and inquisitive researchers.

I would like to thank Mikael Lindvall and Chris Ackermann from Fraunhofer USA CESE for being great colleagues and friends. It has been wonderful working with both of them and I have learned a great deal from conversations with them.

I would like to thank Vibha Sazawal. In the beginning of my graduate career, I had the pleasure of working with Vihba and her encouraging words motivated me to pursuit a doctorate degree and academic research.

Finally, I thank my parents, Huafu Song and Mingxun Cai, for their support and encouragement throughout my life. Their words of encouraging pushes for my pursuits and progress still ring in my ears. I am forever in debt to their sacrifices, and I know I would not have many accomplishments in life without their love and support.

Table of Contents

List of Figures

Chapter 1

Introduction

As modern software systems grow in size and complexity, they are increasingly designed and built as flexible combinations of components that can be configured in a multitude of different ways. The ability to configure a software system to run in various environments, to include specific feature sets and to have certain quality of service makes the system more portable, reusable, and extensible. For example, the most popular web server on the Internet, the Apache HTTP Server, has hundreds of both run-time and compile-time options to configure it to run on a number of different operating systems, to include various optional features, and to tune it for specific performance requirements.

While it is clear that a high degree of configurability offers many benefits, it also greatly complicates the tasks of designing, building, and maintaining configurable software. The complication stems from the fact that developers are no longer dealing with a single system anymore; instead the system is a family of related configured instances. The sheer number of possible configurations can be tremendous, as in the worst case, the *full* configuration space of a system is the exponentially large combination of all possible option settings. Every configuration can potentially produce unique behavior in the system. We call this the *software configuration space explosion* problem, and it adds combinatorial complexity to many already difficult

software engineering tasks. For instance, during software testing, 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)* [15, 8, 46]. In the context of configurable systems, CIT typically involves developers manually modeling the system's *configuration space* — all the ways in which it can be configured — and then using the resulting model to define coverage criteria which then drive the configuration-aware testing processes. 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 system's behavior. Thus this approach cost-effectively increases the likelihood of finding faults.

Covering arrays and other interaction testing techniques have been used in organizations such as Microsoft [17], Phillips [70] and NASA [71]. Although a number of research results show these techniques produce good structural coverage and detect software faults [8, 19, 42], there is only weak scientific understanding of why and how well they work in a more absolute sense. For instance, a covering array parametrized by a single integer strength t is clearly a gross approximation of a software system's internal structure.

1.1 Research Hypotheses and Contributions

This dissertation challenges CIT's assumptions. We think that *configuration interactions*, which are conjunctions of configuration option settings needed to achieve specific behavior in the software systems, are rare. The structure of a software system limits the ways that configuration options can interact with each other. More specifically, we think that systems are often structured such that their *effective configuration spaces* — the set of configurations needed to achieve a specific goal — are often much smaller than their full configuration spaces. For example, complete line coverage might be achievable by running only a small number of carefully chosen configurations. If our hypothesis is true, then we can greatly improve various software engineering tasks for a given configurable system by leveraging its effective configuration space appropriate for the specific software engineering task.

We have identified three primary research hypotheses:

1. For many practical tasks, a system's effective configuration space is a small subset of its full configuration space.

2. We can efficiently discover or approximate the effective configuration space of a software system.

3. We can greatly improved numerous software engineering tasks by leveraging a system's effective configuration space.

The research work in this dissertation is conducted in three parts, each part addressing one of the primary research hypotheses.

In the first part, we undertook a series of thorough studies on the configuration spaces of some medium-sized subject systems. We first used symbolic evaluation [40, 33, 12], a white-box analysis, to generate all the executions our subject systems can take under any configuration under a given test suite. The symbolic evaluation results showed that the total number of paths executed is dramatically smaller than the number of all possible configuration option combinations. We next developed techniques to calculate the configuration interactions of these systems under the test suite. In our case, the testing goal was particular forms of coverage (line, block, edge, and condition). We found that the interactions were quite rare; only a handful of specific option setting combinations had to be exercised to maximize coverage, even under a comprehensive criteria, such as the path coverage. This suggests CIT's insistence on testing every t-way combination of option settings may be unnecessarily expensive. We also found that for our subject systems and test suites, most of the interactions needed to achieve maximum coverage were low strength (involved 4 option settings or fewer), and the very few largest interactions needed to achieve maximum coverage were higher strength (involved 7 option settings). These findings suggest CIT approaches, which are typically applied at $t = 2$ or $t = 3$ [17], are likely missing key high-strength interactions. Finally, we observed that higher strength interactions were usually just lower strength interactions with one or more additional constraints.

In the second part of this work, we created 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 achieving

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 software system. An interaction tree is a hierarchical representation of what we call *proto-interactions*, which are potential interactions or subsets of potential interactions. The key intuition behind iTree stems from our observation that higher strength interactions were usually build on top of the lower strength ones. iTree exploits this observation by performing an iterative, search-based process in which the current iteration's sampled configurations are based on the last iteration's proto-interactions. In this way, the sets of configurations constructed as iTree executes have the potential to provide higher coverage than correspondingly sized configuration sets produced from traditional CIT. We compared iTree against traditional CIT and against similarly sized sets of randomly selected configurations. Our results show that iTree is more likely to find high-coverage configuration sets, and it does so more rapidly than the other approaches. We also found that iTree can easily scale up to large software systems such as the ∼1M-LOC MySQL database and was again more efficient and effective than either CIT or random sampling. We then developed a technique that can efficiently and accurately estimate configuration interactions using the iTree execution results with decision tree classifiers. Using this technique, we were able to approximate the configuration interactions of MySQL in minutes.

Finally, in the third part of this work, we developed tools and techniques that can leverage the knowledge of effective configuration spaces to improve testing of configurable systems. We created an iTree-based automated distributed testing

framework that parallelizes the execution of highly effective configurations selected by iTree on distributed computing resources. We also studied configuration-aware regression testing and developed an algorithm that can generate a small set of configurations that, for a given set of program changes, execute every change under all configurations affected. Our results demonstrated that leveraging the knowledge of effective configuration space can greatly improve the cost-effectiveness of the development and maintenance of configurable software systems.

1.2 Organization

The following outlines the organization of this dissertation. Chapter 2 discusses works that are related to our research; we consider works on configurable software, CIT, symbolic evaluation, and machine learning. Chapter 3 presents our studies on software systems' configuration spaces. Chapter 4 describes in detail our iTree approach to discover effective configurations for specific software engineering goals. Chapter 5 demonstrates tools and techniques for practical applications of effective configuration spaces. Chapter 6 presents contributions and future work.

Chapter 2

Background and Related Work

In this research, we address the problem of testing highly configurable software systems. Our work uses and improves upon many ideas and techniques from existing research work across several areas of computer science. We have categorized this existing work into four broad categories. In the sections below, we describe each category in more detail. Section 2.1 presents the concepts of highly configurable software systems. We discuss work related to the design, implementation and verification of such systems and the challenges they present to software engineering, specifically to software testing. Section 2.2 discusses work related to *combinatorial interaction testing CIT*. We introduce CIT techniques and discuss their applications in the development, testing and maintenance of configurable software systems. Section 3.1 discusses work related to *symbolic evaluation*. We present the different designs and implementations of various symbolic evaluators and their intended applications. As far as we know, we are the first to use symbolic evaluation to study configurable systems. Section 2.4 discusses work related to machine learning techniques and their application to dynamic analysis of software systems.

2.1 Highly Configurable Systems

Today's software systems are increasingly shifting from individual programs to families of related programs, but the concept of program families is not new. Parnas defined program families in his 1976 paper [50] as: "Sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members." Significant reuse can be achieved by implementing the set of common properties of these systems as one integrated highly configurable system. The process of *configuration* then binds the optional properties (or variations) of a program family to configure the instances in order to produce a specific software system.

There are numerous techniques that can be used to implement variability in program families [28], including conditional compilation, dynamic class loading, design patterns, aspect-oriented programming (AOP) [75], etc. These techniques differ in both the code-level mechanisms used and the exact binding time of the configurable features. At one extreme, there are dynamically reconfigurable systems [67, 21] in which the feature binding steps may repeatedly take place at runtime. At the other extreme, there are software product lines (SPLs) [14] that define an architectural model to implement a family of software products that share common features while allowing for variability in functionality, performance and level of service. But the most common configurable software systems are programs such as desktop applications, web servers, and databases, that allow users to modify a set of pre-defined configuration options via the command-line or configuration files and

then run the programs with the user specified option settings.

Highly configurable systems lower the overall development and maintenance costs of multiple systems with commonly sharable capabilities [4], but they also present significant challenges for their design, development and testing processes. In this dissertation, we focus on software testing. The already challenging problem of testing a single software system has been replaced with an even harder problem of testing a set of configured instances that can be produced by all of the different possible combinations of features. The *software configuration space explosion* problem arises because the number of possible configured instances of such a system can be tremendous. In the worst case, the configuration space is exponentially large combination of all possible option settings. Many research results suggest that faults can appear in some configured instances but not in others [43, 82, 57]. Therefore, during software testing, each configured instance should be tested, but that is impossible in practice.

Instead, researchers and practitioners opt to sample a subset of all the possible configured instances of a system in order to provide some confidence in software quality. One of the most prominent configuration space sampling techniques used is CIT.

2.2 Combinatorial Interaction Testing

CIT [15, 8, 46] was originally designed for testing traditional program inputs. It computes a small set of test inputs guaranteed to contain certain combinations of

the input values. In the context of configurable systems, CIT typically involves developers manually modeling the system's *configuration space* — all the ways in which it can be configured — and then using the resulting model to define coverage criteria which then drive the configuration-aware testing processes. One popular implementation of CIT takes a parameter t, called the *interaction strength*, and computes a *covering array* – a small set of configurations such that all possible t-way combinations of the configuration option settings appear in at least one configuration. Testing and analysis are then done on each of the covering array's configurations.

For software testing, several studies have shown that high statement and block coverage can be achieved with low strength covering arrays (t=2 or 3), while slightly higher strengths may be needed for edge or path coverage or for fault detection [8, 19, 42]. Therefore, even at low strength, covering arrays should be an effective selection technique based on structural coverage. Qu et al. [57], studied whether 2-way covering arrays could effectively support regression testing on configurable systems. Their basic findings were that individual program changes affected different configurations differently, and therefore, systematically covering system configurations was an effective heuristic for configuration-aware regression testing. Yilmaz et al. [82] and Fouche et al. [25] extended covering array test results to fault characterization. That is, they used covering arrays to generate test data and then fed this data to machine learning algorithms to automatically classify the likely causes of failure. Both efforts were able to detect failures and to accurately determine which specific option settings were responsible for inducing the failures.

Several research have suggested the application of CIT approaches to software

product lines (SPLs) [16]. SPLs can instantiate enormous number of valid software product instances, which creates many challenges for selecting product instances for validation and testing. Oster et al. [49] used SPL feature models to extract valid feature pairs. Then product instances are generated such that any feature pair is covered by at least one product instance. Their selection algorithm allowed pre-selection of product instances and built additional product instances to cover the uncovered feature pairs. Perrouin et al. [51] defined product instance generation as a Constraint Satisfaction Problem (CSP) and focused on using SAT solvers to generate valid product instances that satisfies all t feature interactions.

There are numerous existing techniques developed to address the cost-effectiveness of CIT techniques. Garvin et al. [30] developed on a meta-heuristic algorithm for covering array generation called simulated annealing. By reorganizing the search space base on the CIT problem structure, their algorithm reduces both the running time of the generation process as well as the resulting sample size. Bryce et al. [10] developed an optimization framework for constructing prioritized covering arrays. Given a user-defined objective function, they construct covering arrays whose configurations are ordered so that the more "important" configurations can be tested before less important ones. Fouche et al. [24] created an incremental covering array technique that incrementally builds covering array schedules. This approach relieves the developers from picking the correct t-strength; it begins at a low strength, and then iteratively increases strength as resources allow. At each stage the previously tested configurations are reused in the higher strength covering arrays, thus avoiding duplication of work.

To date, much of the research on CIT has taken a black-box approach. While these efforts has produced promising results, there is only a weak scientific understanding of how well CIT really works. We feel this is a huge missed opportunity to improve the testing of highly configurable software systems.

2.3 Symbolic Evaluation

For our research work, we use symbolic evaluation to undertake white-box, code-level analyses of configurable systems to discover detailed information about how their configuration spaces are structured.

Symbolic evaluation has been around for more than 30 years. In the mid 1970's, King was one of the first to propose symbolic evaluation as an aid to program testing [40]. Theorem provers at that time, however, were fairly simple, limiting the approach's practical potential. Recent years have seen remarkable advances in Satisfiability Modulo Theory and SAT solvers, which has enabled symbolic evaluation to scale to more practical problems. Some recent symbolic evaluators include DART [33], CUTE [69], KLEE [12], Pex [74], Splat [80], and Java PathFinder [1] etc. There are important technical differences between these systems, however, at a high level, the basic idea is the same: The developer marks values as symbolic, and the symbolic evaluator explores all possible program paths reachable under arbitrary assignments to those symbolic values.

DART uses *concolic execution* [68, 45], which mixes concrete execution and symbolic evaluation. This system associates each symbolic input to a concrete value,

and the program is executed as usual with these values. At the same time, DART collects a list of symbolic constraints over the symbolic inputs, one at each branch point (i.e., if-statement) along the concrete execution path. After the execution finishes, DART carefully picks a branch point and replaces the symbolic constraint it generates with the negation. The new list of symbolic constraints is then solved by a constraint solver to get a satisfying assignment that will direct the program to another path whose prefix is the same as the previous one, but branches differently at the chosen branch point. This process is repeated until all branch points have been chosen, or the maximum number of allowed iterations has been reached.

EXE [13] instruments the program by adding code that maintains constraints along execution paths, consults the constraint solver when a conditional is hit, and calls fork() to branch the execution if the conditional is unresolvable. The instrumented program is then compiled and run natively. When there is an assertion failure, the constraint solver will yield a set of concrete inputs, based on the current path condition, that will drive the uninstrumented program along the same path and hit the assertion failure. KLEE [12], the successor to EXE, performs symbolic evaluation in a similar manner. However, instead of instrumenting the program and running it natively, KLEE interprets it. The main advantage of this over calling fork() is that the latter requires duplication of memory, which is expensive in both time and space (although fork() does copy-on-write, it is likely that any branch will modify the memory which triggers the copy). KLEE avoids this by modeling memory as a persistent map so that portions of the heap can be shared among multiple executions efficiently.

To address the path explosion problem during symbolic evaluation, Xie et al. [79] proposed a search strategy called Fitnex. This approach uses a fitness function to measure the distance from an already discovered execution path to the target branch. The fitness value generated by this function is used to guide the future exploration of execution paths. In thir work, they implemented Fitnex in the Pex [74] tool and found their approach to be effective for achieving high code coverage faster than existing search strategies. Person et al. [52] addresses the scalability issue of symbolic evaluation with a technique called differential symbolic execution. This technique exploits the fact that structures of changed programs are mostly the same as the previous versions. Therefore, instead of performing full symbolic evaluation on the programs each time, they efficiently calculate deltas (or changes) in program behaviors after modifications and perform partial symbolic evaluation on part of the program.

Symbolic evaluation is not limited to program variables either. Hu et al. [37] used symbolic evaluation to study conditional compilation using C/C++ preprocessor directives. In this work, they were able to use their tool to analyze the Linux kernel source code to find the simplest directive conditions to enable the compilation of a line of code.

2.4 Machine Learning

Several researchers have applied machine learning techniques to testing and analysis of software systems.

Wegener et al. [76] created a tool environment to apply evolution testing to C programs. Their approach uses evolutionary computation algorithms to generate test data that fulfils a given structural testing criteria using fitness functions that are based on branching conditions of uncovered program paths. Bueno et al. [11] used genetic algorithms to identify potentially infeasible program paths; they proposed that monitoring the progress of genetic search could identify an infeasible path. They developed a "path similarity metric" fitness function that uses control and data flow information to guide the search. Kasik et al. [39] focused on generating graphical user interface (GUI) test cases that mimicked novice user behavior. They used genetic algorithms as a repeatable technique to generate unexpected user events. The fitness values are assigned to the events according to how much they resemble novice-like behavior using a special reward system that was built based on observations.

Podgurski et al. [54] used cluster analysis and random sampling to improve the accuracy of software reliability estimates. Their approach involved collecting execution profiles and applying automatic cluster analysis to these profiles to partition the executions based on dissimilarity. A stratified random sample of executions is then selected to reduce the number of program executions that need to be checked manually for conformance to requirements. Dickinson et al. [18] presented a technique called cluster filtering that takes a choice of dissimilarity metric, cluster count, and sampling method to filter failure predicting test cases. Podgurski et al. [53] presented a semi-automated strategy for classifying software failures. They applied both supervised and unsupervised pattern classification techniques and multivariate

visualization techniques to execution profiles in order to prioritize software failure reports. Francis et al. [26] developed two tree-based strategies to group together software failures with similar causes. Their first approach used dendrograms [72], which are tree-like diagrams used to represent the results of hierarchical cluster analysis, to refine an initial failure clustering. Their second approach applied classification trees to classify failures and to refine the classification of these failures.

Haran et al. [35] developed three techniques – association trees, random forests and adaptive sampling association trees – to automatically classify fielded software system executions. The general approach of association trees is to collect execution data from lightly instrumented instances and models from both in-house and in-the-field training sets to predicted execution outcomes. Random forests lowered instrumentation requirements and improved prediction accuracy by building numerous lightweight association trees to vote on the execution outcome. Also, adaptive sampling uses earlier execution data to determine which data to collect in future instances, thus improving the quality of execution data collected while reducing the instrumentation overhead. Burn et al. [9] developed a fault invariant classifier based on two different machine learning techniques. Their decision tree approach uses an external invariant detector, Daikon [23], to isolate faults revealing properties within the test subjects. Bowring et al. [5] used Markov models to build a classifier for program executions. Their approach used instrumentation of all branches within the subject programs and model a particular class of branches – event transitions – to improve the accuracy of the classifiers.

Our research work uses machine learning techniques to select configurations

that should be tested. This mostly involves classification learning techniques, and we are specifically interested in techniques that enable the extraction of knowledge from the classification models [61].

Chapter 3

Understanding the Effective Configuration Space

In this chapter, we look at our first research hypothesis: For many practical tasks, a system's effective configuration space is a small subset of its full configuration space. To help us explore this hypothesis we formed four general research questions:

1. How does configuration affect the behavior of software systems?
2. How can we characterize systems' effective configuration spaces?
3. Are the existing approaches effectively sampling the configuration spaces?
4. Can we improve configuration space sampling using knowledge of effective configuration spaces?

To provide answer to the first question, we used *symbolic evaluation*, a white-box analysis technique, to empirically study and understand the configuration spaces of two subject systems. Symbolic evaluation enables us to capture *all* executions a system can take under any configuration. The data captured from symbolic evaluation are execution trees that contain all possible paths executed under any combinations of configuration option settings. By definition, each path in the execution trees is distinct from all other paths, thus every configuration option combination given by a path is unique. We found that, for our subject systems, the total number of

paths executed is dramatically smaller than the number of all possible configuration option combinations. These results indicate that the *effective configuration spaces* are indeed much smaller than the full configuration spaces.

To answer the second question, we developed new analysis techniques to characterize the relationship between configuration and system behavior. Without further analysis, the execution paths from symbolic evaluation tell us only a little about the a system's effective configuration space. Therefore, we next developed two analysis techniques, the *guaranteed coverage* and the *execution conditions* analyses, that can project the execution paths onto different types of structural coverage. These techniques allowed us to calculate the *configuration interactions*, which succinctly characterize the relationship between configuration options and the internal structures of a software system. We found that, if the goal of a specific software engineering task is measured by more abstract properties, then the execution paths are no longer unique, and the effective configuration space collapses further. For example, to reach complete line coverage during testing, the most of the execution paths are actually redundant.

Using the configuration interaction data, we can answer the third research question. To understand how well existing configuration space sampling approaches worked, we evaluated two techniques, the covering arrays and the random sampling, on their ability to achieve high structural coverage during software testing. Although these techniques produce relatively good results in practice, we found that the covering arrays are doing too much work covering all interactions of a set t strength, but at the same time, they often miss higher strength interactions needed to achieve

the software engineering goals. And random sampling, lacking a systematic way to determine the sample size, depends on the probability of including the right interactions in the configuration samples. Based on our evaluation results, we think a more effective sampling approach should focus on the coverage of actual interactions of a software system instead all potential interactions in the full configuration space.

Finally, to provide the answer to the forth question, we investigated whether we can use the configuration interactions to generate small configuration sets that are more effective than those generated by existing sampling approaches. We developed a greedy algorithm that packs the interactions together, aiming to find a minimal configuration set that still achieves the same structural coverage as the full set of execution paths. For our subject systems, the resulting *minimal covering sets* range in size from only 5 to 10 configurations regardless of the coverage criteria, which is much smaller than the covering arrays and the random samples. This suggests the effective configuration space looks more like a union of disjoint interactions rather than a monolithic cross-product of all configuration option settings.

The remainder of this chapter is organized as follows. Section 3.1 describes our experiments using symbolic evaluation to study systems' configuration space. Section 3.2 describes the two analysis techniques we used to calculate the configuration interactions. Section 3.3 presents our evaluations of existing sampling approaches using the configuration interaction data and the implications on more effective sampling approaches. Section 3.3.2 describes our new sampling approach that generates a small set of configurations called the *minimal covering set* using a greedy algorithm to pack together configuration interactions.

We note that some material in this chapter is collaborative work from a previous publication [62] with Elnatan Reisner and Kin Keung Ma. Specifically, Section 3.1, Section 3.2.1 and Section 3.3.2. In Section 3.1, the symbolic evaluator used during our empirical studies was built by Kin Keung Ma. Elnatan Raisner and Kin Keung Ma also prepared the two subject systems for symbolic evaluation and created their test suites. In Section 3.2.1, the algorithm of the guaranteed coverage analysis was implemented by Elnatan Raisner with the assistance of Kin Keung Ma. In Section 3.3.2, Elnatan Raisner implemented the algorithm for minimal covering set generation. We shared the responsibility of analyzing the results in these sections.

3.1 Using Symbolic Evaluation to Study Configurable Software

3.1.1 Configurable Software

In this dissertation, we define a configurable software system as a generic code base and a set of mechanisms for implementing pre-planned variations in the code base's structure and behavior. These variations are wide-ranging, covering hardware and operating system platforms (e.g., Windows vs. UNIX), run-time features (e.g., enable/disable SSL encryption), performance tuning (e.g., maximum number of concurrent clients) and many others. In practice, these pre-planned variations can be implemented using a variety of programming constructs, such as run-time conditional expressions (e.g., if-then statements), conditional compilation (e.g., C++ preprocessor directives), dynamic executable loading (e.g., Java Reflection). In this

chapter, we focus on the run-time configuration options, which are usually given via configuration files and have their values read into program variables.

We also define a *configuration* as a mapping of the configuration options to their settings. Every combination of option settings is a distinct configuration, and all possible configurations a software system can take on make up the system's *configuration space*.

To understand the internal structures of these software systems with respect to configuration, we need to first capture the affects configuration has on the run-time behavior of the systems under all configurations. However, due to the *configuration space explosion* problem, we cannot get this information by directly executing the software systems under every possible configuration. Instead, we opted to undertake white-box, code-level analyses to discover details about software systems' internal structures with respect to their configurations.

Figure 3.1 illustrates several ways that run-time configuration options can be used in the source code, and explains why understanding their usage requires fairly sophisticated technology. All of these examples are taken from our subject systems and the configuration options are shown in boldface.

The example in Figure 3.1(a) shows a section of vsftpd's command loop, which receives a command and then uses a long sequence of conditionals to interpret the command and carry out the appropriate actions. The example shows two such conditionals that also depend on boolean configuration options. In this case, the configuration options enable certain commands, and the enabling condition can either be simply the current setting of the option (as on line 2) or may involve an

```
1       ...
2       else if (tunable_pasv_enable &&
3               str_equal_text(&p_sess->ftp_cmd_str, "EPSV")) {
4                       handle_pasv(p_sess, 1);
5               }
6       ...
7       else if (tunable_write_enable &&
8               (tunable_anon_mkdir_write_enable || !p_sess->is_anonymous) &&
9               (str_equal_text(&p_sess->ftp_cmd_str, "MKD") ||
10              str_equal_text(&p_sess->ftp_cmd_str, "XMKD"))) {
11                      handle_mkd(p_sess);
12      }
```

(a) Boolean configuration options (vsftpd)

```
13      if ((Conf_MaxJoins > 0) &&
14      (Channel_CountForUser(Client) >= Conf_MaxJoins))
15              return IRC_WriteStrClient(Client,
16                      ERR_TOOMANYCHANNELS_MSG,
17          Client_ID(Client), channame);
```

(b) Integer-valued configuration options (ngIRCd)

```
18      else if(Conf_OperCanMode) {
19              /* IRC-Operators can use MODE as well */
20              if (Client_OperByMe(Origin)) {
21                      modeok = true;
22                      if (Conf_OperServerMode)
23                              use_servermode = true; /* Change Origin to Server */
24              }
25      }
26      ...
27      if (use_servermode)
28              Origin = Client_ThisServer();
```

(c) Nested conditionals (ngIRCd)

```
29      remote_fd = vsf_sysutil_accept_timeout(p_sess->pasv_listen_fd, p_accept_addr, tunable_accept_timeout);
30      ...
31      vsf_sysutil_accept_timeout(int fd, struct vsf_sysutil_sockaddr* p_sockaddr, unsigned int wait_seconds) {
32              ...
33              if (wait_seconds > 0) {
34                      ...
35              }
36      }
```

(d) Options being passed through the program (vsftpd)

Figure 3.1: Examples of configuration options being used in our subject systems.

interaction between multiple options (as on lines 7–8).

Not all options need to be booleans, of course. Figure 3.1(b) shows code from ngIRCd in which Conf_MaxJoins is an integer option that, if positive (line 13), gives the maximum number of channels a user can join (line 14). In this example, error processing occurs if the user tries to join too many channels.

Figure 3.1(c) shows a different example in which two configuration options are tested in nested conditionals. This illustrates that it is insufficient to look at tests of configuration options in isolation; we also need to understand how they may interact based on the system's structure. Moreover, in this example, if both options are enabled then use_servermode is set on line 23, and its value is then tested on line 27. This shows that the values of configuration options can be indirectly carried through the state of a system.

Figure 3.1(d) shows another example of using configuration options indirectly. Here wait_seconds in the vsf_sysutil_accept_timeout function is assigned the value of a configuration option through one of its parameters, and the value of this parameter is then used in the conditional (lines 33) to control the execution of some lines of code.

As we saw above, simple approaches such as searching for the option names will be insufficient, because configuration options can be used in complex ways in the systems' source code. Instead, we propose to use *symbolic evaluation* to capture all executions that a system can take under any configuration.

3.1.2 Symbolic Evaluator

There existing a bevy of symbolic evaluators built for vast number of different applications. The choices of their designs and implementations greatly impact their applicability to our specific use case. Many of these evaluators include techniques to guide the evaluator to "interesting" execution paths, under the assumption that if arbitrary program inputs are made symbolic, then there are too many paths to explore in a reasonable amount of time [32]. In contrast, for our studies we need to explore *all* execution paths. The symbolic evaluator we picked for our studies, Otter, was designed and built by the University of Maryland programming language group with the application of studying configurable software systems in mind.

3.1.2.1 Otter's Design and Implementation

Otter is essentially a C source code interpreter, with one key difference; it allows some data to be designated as *symbolic*, meaning their values represent unknowns that may take on any value. Otter tracks these values as they flow through a program, and conceptually forks execution if a conditional depends on a symbolic value. Thus, if it runs to completion, Otter will have simulated all execution paths through the program that are reachable for any values that the symbolic data can take on. For our work, we treat only the configuration options as symbolic, therefore on successful exit, Otter would have simulated all paths reachable by the systems under any possible configuration.

Otter's general approach closely mimics the implementation of KLEE [12],

which performs pure symbolic evaluation, as oppose to concolic execution used by DART [33]. Otter's simulated environment, which models memory as a persistent map so that portions of the heap can be shared among multiple executions efficiently, allows it to search through the configuration space of a software system much faster. But the lack of real execution also means, Otter does not perform actual testing of the software systems.

Otter is written in OCaml, it uses CIL [47] as a front end to parse C programs and transform them into an easier-to-use intermediate representation. In addition, the use of CIL also enables easy instrumentation and measurement of the symbolic evaluation results. As Otter executes, it records the execution paths explored so that we can later compute the structural coverages such as line, block, edge, and condition. Note that the definitions of these metrics are for CIL's representation of the input program, which is simplified to use only a subset of the full C programming language. However, the precision of these metrics is sufficient for our investigations.

To compute line coverage, we record which CIL statements Otter executes and project that back to the original source code lines using a mapping maintained by CIL. For block and edge coverage, we group CIL statements into basic blocks, which are sequences of statements that start at a function entry or a join point; do not contain any join point after the first statement; end in a function call, `return`, `goto`, or conditional; or fall through to a join point. Normally, CIL expands short-circuiting logical operators `&&` and `||` into sequences of branches. However, for line, block, and edge coverage, we disable that expansion as long as the right operand has no side effect, so that both operands are computed in the same basic block.

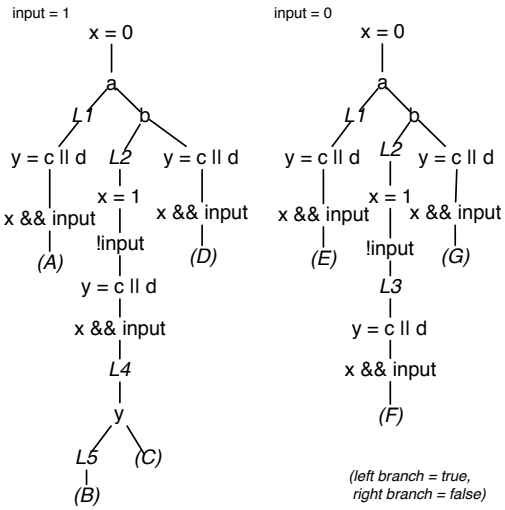
Then to compute block coverage, we record which basic blocks are executed, and to compute edge coverage, we compute which control-flow edges between basic blocks are traversed. Lastly, for condition coverage, we enable expansion of `&&` and `||`, so that each part of a compound condition is always in its own basic block. We then compute how many conditions — that is, how many branches — are taken in the expanded program.

A symbolic value in Otter represents a sequence of untyped bits, e.g., a 32-bit symbolic integer is treated as a vector with 32 symbolic bits in Otter. This low-level representation is important because many C programs perform bit manipulations that must be modeled accurately. When a symbolic expression has to be evaluated, Otter invokes STP [29], an Satisfiability Modulo Theory (SMT) solver optimized for bit vectors and arrays. These implementation choices allow us to model most types of configuration options used by our subject systems.

Otter supports all features of C we found necessary for the investigations of the configuration spaces of our subject systems, including pointer arithmetic, arrays, function pointers, variadic functions, and type casts. Loops, which can cause symbolic evaluation to "get stuck" as it tries to unroll the loop an unbounded or extremely large number of times, were not a major obstacle in our investigations: Configuration options almost never influence loop bounds, so all loops were executed in the usual way, with the concrete test cases determining the number of loop iterations. Otter currently does not handle dereferencing symbolic pointer values, floating-point arithmetic, in-line assembly or multiple processes. In all cases, these features either do not appear in our subject systems or do not affect the results of

```
1   int a=α, b=β, c=γ, d=δ; // symbolic
2   int input=...; // concrete
3   int x = 0;
4
5   if (a) {
6     /* L1 */
7   } else if (b) {
8     /* L2 */
9     x = 1;
10    if (!input) {
11      /* L3 */
12    }
13  }
14
15  int y = c || d;
16
17  if (x && input) {
18    /* L4 */
19    if (y) {
20      /* L5 */
21    }
22  }
```

(a) Example program



(b) Full execution trees

Figure 3.2: Example symbolic evaluation using Otter.

our investigations.

All of our subject systems interact with the operating system in some way. Thus, we developed "mock" libraries that simulate a file system, network, and other needed operating system components in Otter's symbolic evaluation environment. Our libraries also allow test cases to control the contents of files, data sent over the network, and so on. These mock library functions are written in C and are executed by Otter just like any other code.

3.1.2.2 Example Symbolic Evaluation

To illustrate how Otter performs configuration space investigation, consider the example C code in Figure 3.2(a). This program includes input variables a, b, c, d, and input. The first four are intended to represent run-time configuration options, and we initialize them on line 1 with *symbolic values* α, β, γ, and δ, respectively. The last variable, input, represents program inputs other than configuration options. Thus we leave it concrete, and it must be supplied by the user (e.g., as part of argv (not shown)).

We have indicated five lines, represented by comments /* *L1–L5* */, whose coverage we are interested in. Figure 3.2(b) shows the sets of paths explored by Otter as *execution trees* for the two concrete "test cases" for this program: The tree for input=1 is on the left, and the tree for input=0 is on the right. Here nodes correspond to program statements, and branches represent places where Otter has a choice and hence "forks," exploring both possible paths. For the sake of simplicity, we will assume that all symbolic values may only represent 0 and 1, but Otter fully models symbolic integers as arbitrary 32-bit quantities.

For example, consider the tree with input=1. All executions begin by setting x to 0 and then testing the value of a, which at this point contains α. Since there are no constraints on α, both branches are possible. Otter forks its execution at the test of a. First, it assumes $\alpha = 1$ and reaches *L1* (left branch). It then falls through to line 15 (the assignment to y) and performs the test on line 17 (x && input). This test is false, since x was set to 0 earlier, hence Otter does not fork. We label this path

through the execution tree as (A). Notice that as Otter explored path (A), it made some decisions about the settings of symbolic values, specifically that $\alpha = 1$. We call this and any other constraints placed on the symbolic values a *path condition*.

Here, path (A) covers `L1`, and so any configuration that sets `a`=1 (corresponding to $\alpha = 1$), with arbitrary choices for β, γ, and δ, will cover `L1`. This is what makes symbolic evaluation so powerful: With a single predicate we characterized the behavior of many possible concrete choices of symbolic inputs. Otter continues by returning to the last place it forked and tries to explore the other path. In this case, it returns to the conditional on line 5, adds $\alpha = 0$ to the path condition, and continues exploring the execution tree. Each time Otter encounters a conditional, it calls the SMT solver to determine which branches (possibly both) of the conditional are possible based on the current path condition.

In total, there are four paths that can be explored when `input`=1, and three paths when `input`=0. However, there are 2^5 possible assignments to the 5 input variables. Hence using symbolic evaluation for these test cases enables us to gather full coverage information with only 7 paths, rather than the 32 runs required if we had tried all possible combinations of symbolic and concrete inputs.

3.1.3 Configuration Space Study

Using Otter, we explored the configuration spaces of two subject systems: vsftpd, a widely used secure FTP daemon and ngIRCd, the "next generation IRC daemon". Both subject systems are written in C and each has multiple configuration

	vsftpd	ngIRCd
Version	2.0.7	0.12.0
# Lines (sloccount)	10,482	13,601
# Lines (executable)	4,112	4,387
# Basic Blocks	4,584	6,742
# Edges	5,033	7,374
# Conditions	2,528	3,432
# Test Cases	64	142
# Symbolic Config Opts	30	13
Boolean/Integer	20/10	5/8
# Concrete Config Opts	65	16
Full Config, Test Space	1.4×10^{11}	4.2×10^{7}

Figure 3.3: Program statistics for vsftpd and ngIRCd for the symbolic evaluation experiments.

options that can be set in system configuration files.

3.1.3.1 Subject Systems

Figure 3.3 gives descriptive statistics for each subject system. The top two rows list the system version numbers and lines of code as computed by sloccount [78]. The next group of rows lists the number of executable lines, basic blocks, edges, and conditions; these four metrics are the structural coverages we measure in our investigations. To get more accurate measurements, we removed some unreachable code.

For example, we eliminated 3 files in vsftpd that are reachable only in two-process mode, which we disabled because Otter does not support multiprocess symbolic evaluation.

We note that, all these statistics except for sloccount metric are based on the CIL representation of the program after preprocessing. We also note that there are more basic blocks than executable lines of code in both subject systems. This occurs because, in many cases, single lines form multiple blocks. For example, a line that contains a for loop will have at least two blocks (for the initializer and the guard), and lines with multiple function calls are broken into separate blocks according to our definition.

The next row in Figure 3.3 lists the number of test cases. In creating these test cases, we attempted both to cover the major functionality of the systems and to maximize overall line coverage. Neither subject system come with its own test suite. For vsftpd, we developed test cases to exercise its major functionality such as logging in; listing, downloading, and renaming files; asking for system information; and handling invalid commands. For ngIRCd, we created test cases based on the IRC functionality defined in RFCs 1459, 2812 and 2813. Our test suite for ngIRCd cover most of the client-server commands and a few of the server-server commands, with both valid and invalid inputs.

We stopped creating new test cases when we reached a point of diminishing returns for our efforts. For example, much of the code left uncovered by our test suites handles system call failures, such as malloc() returning NULL; modeling these failures would have greatly increased the number of execution paths (and hence

analysis time) without shedding extra light on the configuration spaces of these systems. Other uncovered code would have required significantly extending Otter — e.g., to handle asynchronous signals — which was beyond of the scope of these studies.

Figure 3.3 next shows the counts of the configuration options. We give the total number of analyzed configuration options and also break them down by type, boolean or integer. We also list the number of configuration options we left as concrete. We decided to leave some options concrete based on two criteria: Whether the option was likely to expose meaningful behavior, and our desire to limit total analysis effort.

Finally, Figure 3.3 shows how many executions would be required if we ran every test case under every possible configuration, given the number of distinct values each symbolic option could take. We will contrast these extremely large numbers with the results of symbolic evaluation.

3.1.3.2 Symbolic Evaluation Results

We ran the subject systems using our test suites in Otter, with symbolic configuration options. Figure 3.4 summarizes these symbolic evaluation runs. The first two rows show the number of paths executed by Otter, summed across all test cases. Clearly, this is dramatically smaller than the number of executions that would have been necessary had we instead naively run each test case under all possible configuration option combinations: Only 0.0001% of the naive count for vsftpd and

	vsftpd	ngIRCd
# Paths		
Line, Block, Edge	30,304	53,205
Condition	136,320	95,637
Average # Paths		
Line, Block, Edge	474	375
Condition	2,130	674
Coverage		
Line	62%	73%
Block	63%	66%
Edge	56%	61%
Condition	49%	57%
# Examined / Total Opts		
Line, Block, Edge	22/30	13/13
Condition	24/30	13/13

Figure 3.4: Summary of the symbolic evaluation experiments.

0.2% for ngIRCd. Also, recall that these are actually *all* possible execution paths for these test suites under any settings of the symbolic configuration options, given Otter's simulated environment.

Notice that condition coverage, which has logical operators expanded into sequences of conditionals as discussed above, has many more execution paths. This effect is most pronounced for vsftpd, which more than quadruples the number of

paths because it contains many logical expressions that test multiple configuration options at once. For example, **if** (x || y || z) would yield at most two paths before expansion, but four paths after.

To aid comparison across our subject systems, we next show the total number of execution paths averaged over all test cases.

Figure 3.4 then lists the coverage achieved by these paths, the maximum coverage achievable by these test suites considering all possible configurations using the options and settings we analyzed. If we adjust for the error handling and unreachable code, our test suites' line coverage is near or exceeds 80% for both subject systems. Covering the remaining code would in many cases have required adding new mocked libraries, adding more symbolic configuration options, etc. Overall, however, based on our analysis of these systems, we believe that the test cases are reasonably comprehensive and are sufficient to expose much of the configurable behavior of the subject systems.

The next group of rows shows the number of configuration options that appear in at least one path condition (i.e., were constrained in at least one path and thus distinguish different execution paths) versus the total number of options set symbolic. Notice that Otter constrains two more options with condition coverage than under the other metrics. This occurs because of the expansion of logical operators into sequences of conditionals. For example, under line, block, and edge coverage, the condition **if** (x || 1) would be treated as a single branch that Otter would treat as always true. But under condition coverage, the conditional would be expanded, and Otter would see **if** (x) first, causing it to branch on x.

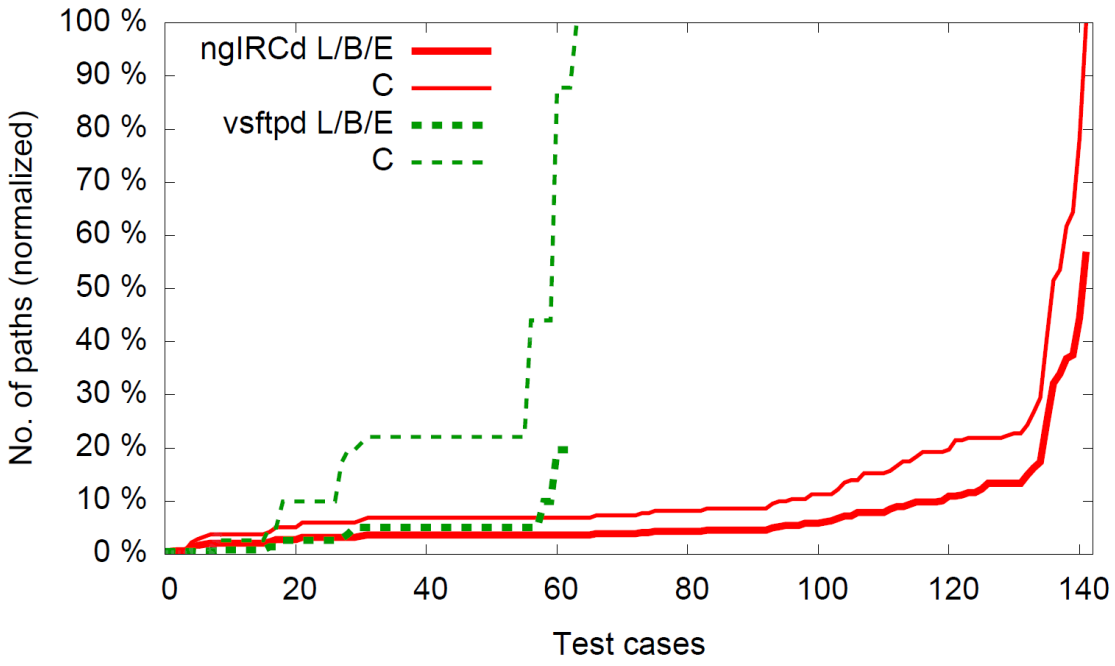


Figure 3.5: Number of paths executed by each test case during symbolic evaluation (L/B/E=line/block/edge, C=condition).

Figure 3.5 plots the number of paths executed by each test case for each system, both with unexpanded logical operators (line/block/edge) and expanded (condition). The x-axis is sorted from the fewest to the most paths, and the y-axis is the percentage of execution paths relative to the highest number of paths seen in any test case for the expanded (condition) version of the system.

In Figure 3.5, we see that majority of the test cases did not execute anywhere near the maximum number of paths. This means that test cases also limit the possible configuration option combinations; many of vsftpd and ngIRCd's options are not necessarily used in every test case. This can be seen clearly in the figure: Only a handful of vsftpd and ngIRCd's test cases exercise more than 25% of the execution paths.

One interesting feature of Figure 3.5 is that, for vsftpd, the numbers of paths of different test cases cluster into a handful of groups (indicated by the plateaus in the graph). This suggests that within a group, the test cases branch on the configuration options in essentially the same manner (most likely because the vsftpd employs common segments of code to test the configuration options). In ngIRCd, this clustering also appears but is less pronounced.

The data from *symbolic evaluation* offers strong support for our first hypothesis. We found that, to achieve high structural coverage, the total number of execution paths executed is dramatically smaller than the number of all possible configuration option combinations. This suggests that, our subject systems are structured in ways that not only limits how configuration options can combine, but also limits the number of options that can combine to exercise new behavior. In other words, the effective configuration spaces of these systems are indeed much smaller than their full configuration spaces.

3.2 Understanding the Configuration Interactions

The raw output from symbolic evaluation are execution trees containing all paths executed under all configuration option settings for a given test suite. Without further analysis, however, these paths tell us only a little about our subject systems. By definition, each execution path explored is distinct from all other paths. Thus with no abstraction, every configuration option combination given by an execution path is unique. However, if the testing goals are measured by more abstract proper-

ties of the system, such as structural coverage, then the paths are no longer unique, and the effective configuration space collapses further. To better understand the effective configuration spaces of our subject systems, we chose to project the execution paths onto line, block, edge, and condition coverage to succinctly characterize the relationship between the configuration options and these abstract system properties.

3.2.1 Guaranteed Coverage Analysis

One technique that we developed to do this is the *guaranteed coverage* analysis.

Definition 1 *Given a test suite and a coverage criterion, we say that a predicate p over the (initial settings of the) configuration options* guarantees coverage *of program entity X if there exists some test case in the test suite such that any configuration satisfying p is guaranteed to cover X.*

For example, from Figure 4.1(b) we can see that any configuration satisfying $\alpha = 0 \wedge \beta = 1$ (i.e., `a=0`, `b=1`) is guaranteed to cover *L2*, no matter the choice of γ and δ.

We can use symbolic evaluation's output to compute the guaranteed coverage for a predicate p, which we will write as $Cov(p)$. We first find $Cov^T(p)$, the coverage guaranteed under p by test case T, for each test case; then, $Cov(p) = \bigcup_T Cov^T(p)$. To compute $Cov^T(p)$, let p_i^T be the path conditions from T's symbolic evaluation, and let $C^T(p_i^T)$ be the covered lines, blocks, edges, or conditions that occur in that path. Then $Cov^T(p)$ is

$$Consistent^T(p) = \{p_i^T \mid SAT(p_i^T \wedge p)\}$$

$$Cov^T(p) = \bigcap_{q \in Consistent^T(p)} C^T(q)$$

In words, first we compute the set of path conditions p_i^T such that p and p_i^T are consistent. If this holds for p_i^T, the items in $C^T(p_i^T)$ *may* be covered if p is true. Since our symbolic evaluator explores all possible execution paths, the intersection of these sets for all such p_i^T is the set guaranteed to be covered if p is true.

For our example program in Figure 4.1, below are some predicates and the coverage they guarantee given the test cases input=1 and input=0. We abbreviate $\alpha = 1$ as α and $\alpha = 0$ as $\neg\alpha$.

p	$Consistent(p)$ (input = 1)	$Consistent(p)$ (input = 0)	$Cov(p)$
α	(A)	(E)	$\{L1\}$
β	$(A), (B), (C)$	$(E), (F)$	$\emptyset$
$\neg\alpha$	$(B), (C), (D)$	$(F), (G)$	$\emptyset$
$\neg\alpha \wedge \beta$	$(B), (C)$	(F)	$\{L2, L3, L4\}$
$\neg\alpha \wedge \beta \wedge \gamma$	(B)	(F)	$\{L2, L3, L4, L5\}$

We can also use the guaranteed coverage to find interactions among the configuration options.

Definition 2 *An* interaction *is a conjunction of option settings* $S = \bigwedge_i (x_i = v_i)$ *that guarantees coverage that is not guaranteed by any subset of (the conjuncts of)* S.

For example, $Cov(\alpha = 0 \wedge \beta = 1)$ is a strict super set of $Cov(\alpha = 0) \cup Cov(\beta = 1)$, so $\alpha = 0 \wedge \beta = 1$ is an interaction. Informally, interactions indicate option combinations that are "interesting" because they guarantee some new coverage.

Definition 3 *The* strength *of an interaction is the number of option settings it contains.*

For example, $\alpha = 0 \wedge \beta = 1$ has strength 2. Lower-strength interactions place fewer constraints on configurations, whereas higher-strength interactions require more options to be set in particular ways to achieve their coverage.

Using the definition of $Cov(p)$, we performed the guaranteed coverage analysis on the symbolic evaluation outputs of our two subject systems. First, we computed $Cov(true)$, which we call *guaranteed* 0*-way coverage*. These are coverage elements that are guaranteed to be covered for any choice of options. Here, when we refer to *t-way coverage*, t is the interaction strength. Then for every possible option setting $\mathsf{x} = \mathsf{v}$, we computed $Cov(\mathsf{x} = \mathsf{v})$. The union of these sets is the *guaranteed* 1*-way coverage*, and it captures what coverage elements will definitely be covered by 1-way interactions. Next, we computed $Cov(\mathsf{x1} = \mathsf{v1} \wedge \mathsf{x2} = \mathsf{v2})$ for all possible pairs of option settings, which is *guaranteed* 2*-way coverage*. Similarly, we continue to increase the number of options in the interactions until $Cov(\mathsf{x1} = \mathsf{v1} \wedge \mathsf{x2} = \mathsf{v2} \wedge ...)$ reaches the maximum possible coverage.

For boolean options, the possible settings are clearly 0 and 1. For integer-valued options that we constrained, we used those chosen values; for the remaining integer options, we solved the path conditions discovered by symbolic evaluation and

manually inspected the code to find appropriate concrete settings. For example, if the path condition was x>=0, then the solver might choose $x = 0$ as a possible concrete setting. Because there are multiple path conditions, we sometimes found that different concrete settings were generated by the SMT solver for the same options. In these cases we used our judgement and code examination to select appropriate values. On the other hand, ranges for some integer options depend on how the system is executed. In these cases we examined the test suites to determine the possible values for such options in our test runs.

3.2.1.1 Analysis Results

The results from the guaranteed coverage analysis allow us to explore which configuration interactions are actually required to achieve the line, block, edge, and condition coverage reported in Figure 3.4.

Figure 3.6 shows the number of configuration interactions at each interaction strength. The first thing to notice is that the maximum interaction strength is always seven or less. This is significantly lower than the number of options in each subject system. We also see that the number of interactions is quite small relative to total number of interactions that are theoretically possible. This observation supports our hypothesis that the interactions between configuration options are not complete; only small groups of options interact and only with subset of the values.

For ngIRCd, there are significantly more interactions at higher strength than for vsftpd. This is because almost all of ngIRCd's integer options can take on

	t=1	t=2	t=3	t=4	t=5	t=6	t=7
vsftpd							
Line	7	4	3	16	5	6	2
Block	7	4	3	16	6	6	2
Edge	9	4	4	27	7	7	2
Condition	9	4	4	32	14	9	2
ngIRCd							
Line	11	19	33	117	144	111	-
Block	15	25	33	118	147	111	-
Edge	17	29	37	125	159	111	-
Condition	17	33	37	131	174	111	-

Figure 3.6: Number of configuration interactions at each t strength for line, block, edge and condition coverage criteria.

many different values across our test suite, magnifying the number of interactions. This is an artifact of the integer-valued options we chose for ngIRCd; there are many cases where several different integer values for a particular option interact identically with other options, thereby increasing the number of interactions by a multiplicative factor.

Also notice that there is little variation across different coverage criteria — they have remarkably similar numbers of interactions. We investigated further and found that the majority of interactions are actually identical across all four criteria. This is encouraging, because it indicates that many interactions are insensitive to

	vsftpd	**ngIRCd**
Line & Block & Edge & Cond	43	427
Line & Block & Edge	-	-
Block & Edge & Cond	1	17
Block & Edge	-	-
Edge & Cond	15	22
Edge	1	-
Cond	15	25
Total	78	493

Figure 3.7: Configuration interactions shared among the different coverage criteria.

the particular coverage criterion.

Figure 3.7 shows the number of configuration interactions shared by just one, two, three or all of the coverage criteria. There are interactions that differ among the coverage criteria. Line coverage has the least amount of interactions; almost all of the interactions for line coverage were also present in the other three coverage criteria. Block coverage's interactions were almost a subset of those of edge and condition coverage. And condition coverage required the most number of interactions. This data is consistent with the relative complexity of each coverage criterion. Despite the differences in number of interactions, interactions shared by all coverage criteria make up the majority of all the interactions for all subject systems. This shows that the more complex coverage criteria did not significantly alter the way configuration options interact with each other.

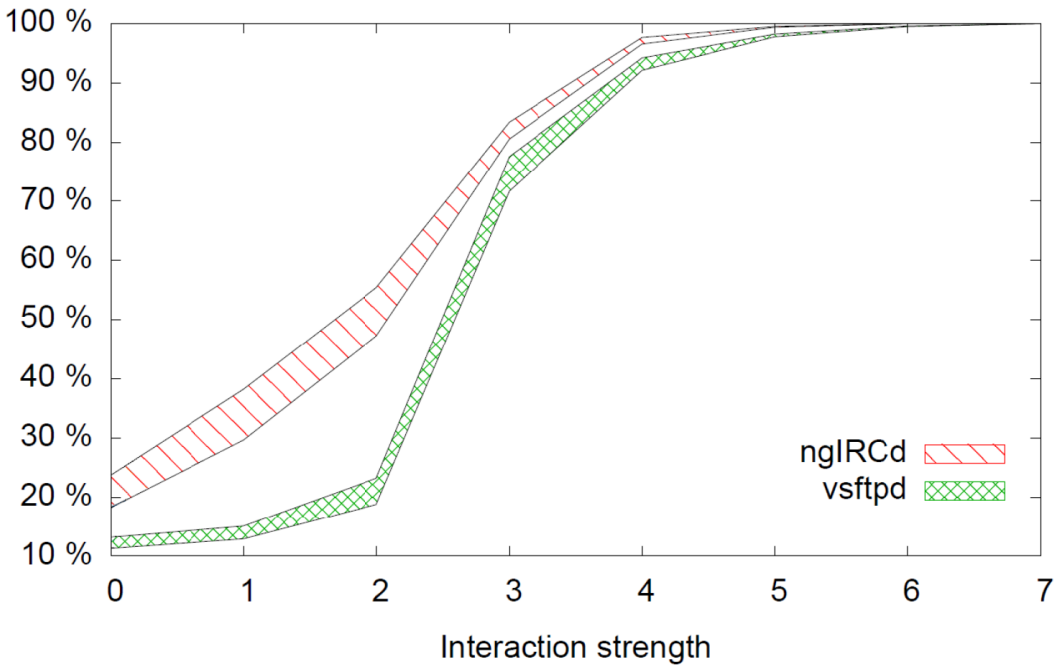


Figure 3.8: vsftpd and ngIRCd's cumulative guaranteed coverage at each interaction strength.

Next, we looked at the amount of coverage guaranteed by the interactions of each coverage criterion. We found for all subject systems, the interactions that contributed the most amount of coverages were shared among all coverage criteria. The interactions required by the more complex coverage criteria guaranteed very little coverage. In fact, all interactions that guaranteed more than 0.6% of total coverage for any coverage criterion were shared among all coverage criteria. For our subject systems at least, the important interactions for the least demanding coverage criteria, is also important for the most demanding ones.

Figure 3.8 presents the configuration interactions in terms of cumulative coverage. The x-axis is the t-way interaction strength, and the y-axis is the percentage of the maximum possible coverage. Note that higher strength interaction coverage

always includes the lower strength coverage, e.g., if a line is covered no matter what the settings are (0-way), then it is certainly covered under particular settings (1-way or higher). As it turns out, the trend lines for line, block, edge or condition coverage criteria are essentially the same for a given subject system, and so the plot shows a region enclosing each data set. In ngIRCd, with some slight variation, line coverage corresponds to the upper boundary of the region, and edge, block, and condition coverage to the lower boundary.

We also notice in this figure that the right-most portion of each region adds little to the overall coverage. Thus, for these subject systems and test suites, high strength interactions are not needed to cover most of the code. We can also see that vsftpd gains coverage slowly but then spikes with 3-way interactions. This suggests the presence of *enabling options*, which must be set a certain way for the system to exhibit large parts of its behavior. For example, for vsftpd (in single-process mode), the enabling options must ensure local logins and SSL support are turned off, and anonymous logins are turned on. ngIRCd also has enabling options that account for the increasing coverage up to interaction strength three, but the effect of these options are less pronounced. In ngIRCd, setting $\mathsf{Conf\_ListenIPv4} = 1$ ensures some amount of coverage, then setting $\mathsf{Conf\_PongTimeout} >= 20$ ensures more coverage, and finally setting $\mathsf{Conf\_MaxNickLength} >= 5$ ensures yet more coverage. This chain accounts for the coverage of ngIRCd up to interaction strength three.

These enabling options also show up in Figure 3.6. In that figure we can see that the number of interactions peak around $t = 4$ for vsftpd, and $t = 4$ or $t = 5$ for ngIRCd. For both systems, most of the interactions that are strength $t = 4$ or

greater generally involve the enabling options plus additional options.

Examining the configuration interactions in detail, we observed that certain option settings reappear in multiple interactions. In fact, many of the higher strength interactions actually subsume lower strength ones. For example, vsftpd's enabling options, ssl_enable = 0, local_enable = 0 and anonymous_enable = 1 each appeared in 37, 32 and 30 out of the 43 line coverage interactions, respectively. More interestingly, these 3 option settings appeared together in 30 interactions. NgIRCd's interactions had the similar behavior. Since ngIRCd's options have numerous possible values, however, we grouped multiple values of some options during our investigation. We found that PongTimeout > 1, ListenIPv4 = 1, MaxNickLength > 5 and MaxConnectionsIP != 1 each appeared in 410, 404, 351 and 243 out of the 435 line coverage interactions respectively. PongTimeout > 1 and ListenIPv4 = 1 appeared together in 398; PongTimeout >1, ListenIPv4 = 1 and MaxNickLength > 5 appeared together in 343; and all 4 option settings appeared together in 222 interactions.

We think these patterns in the interactions are due to the structure of the systems' source code. Because certain interactions, especially the enabling options, are combined with a small number of option settings to form other higher strength interactions, the interactions of our subject systems form hierarchical structures that resembles trees.

The results from the *guaranteed coverage* analysis explained why the effective configuration spaces are much smaller than the full configuration spaces. The reasons are:

1. The number of actual interactions is very small relative to theoretically possible interactions and the interactions involves only small number of options and subsets of their values.

2. Most of the coverage was accounted for by the lower-strength interactions.

3. Higher strength interactions were usually just lower strength interactions with one or more additional constraints.

3.2.2 Execution Conditions Analysis

The *guaranteed coverage* was a powerful tool to study the configuration spaces of software systems, however, it also has a major drawback. This analysis uses a brute force algorithm to calculate the coverage guaranteed to be covered by each potential interaction. This is extremely expensive, and thus not practical for most software engineering tasks. For vsftpd and ngIRCd, it took 40 machines running for several days to compute the configuration interaction data for each coverage criterion.

To address this drawback, we developed a more efficient technique, called the *execution conditions* analysis, that outputs a set of predicates defined over the configuration options for each test case such that whenever any one of predicates is true for a given configuration, the test case is guaranteed to execute certain coverage.

To illustrate the operations of the execution conditions analysis, we describe the process of computing the interactions of a system's line coverage. For each line of code and test case, we collect from the symbolic evaluation data, the path conditions

associated with every path that executes the line. Next, we compute all satisfying assignments for these path conditions. For this we created a tool that exhaustively queries the STP theorem prover for these values from a system's configuration space model.

Once we have computed all the satisfying assignments we treat them as a truth table for describing the operation of a digital logic circuit. Each option setting can be either On (the path condition has this setting), Off (the path condition does not have this setting) or Don't Care (the path condition does not reference this option setting). The circuit's output is then either True (line of code was executed) or False (line of code was not executed). We then feed this truth table to a boolean logic minimization tool, called Espresso [64], which produces an logically equivalent, but minimized logical expression that succinctly captures the conditions under which a given test case executes a given line of code.

The output of the analysis is a logical formula in disjunctive normal form, where each disjunction expresses a unique interaction – a predicate over configurations that when true implies that the given test case will execute the given line of code. For example, consider the simple program in Figure 3.2, running the test case where input=1. Our analysis determines that *L4* is executed by the given test cases whenever $a = 0 \wedge b = 1$. The values of c and d, which do appear in the path condition for the one path leading to *L4*, are removed by Espresso during the minimization because they have no influence on whether *L4* will be executed.

Configuration Interactions	vsftpd	ngIRCd
Strength 1	7	11
Strength 2	4	21
Strength 3	3	33
Strength 4	16	112
Strength 5	5	138
Strength 6	6	111
Strength 7	2	90
Total	43	516

Figure 3.9: Number and strength of configuration interactions discovered using the execution conditions analysis.

3.2.2.1 Analysis Results

To demonstrate the speed advantage of the execution conditions analysis, we performed the process described above to calculate the configuration interactions needed for line coverage for both vsftpd and ngIRCd. We forgo the block, edge and condition coverage criteria because we have determined that all four had mostly the same interactions.

Where the *guaranteed coverage* analysis needed multiple machines running for days to compute the configuration interaction data, the execution conditions analysis took only hours on a single machine to produce the equivalent data. This drastic improvement in speed not only improves the studying of configuration software systems, but also makes leveraging the knowledge of effective configuration spaces

more practical for numerous software engineering tasks.

Figure 3.9 summarizes the configuration interactions generated using the execution conditions analysis. This figure shows the number of unique interactions at each strength. We compare these interactions against the ones calculated using guaranteed coverage analysis (Figure 3.6). For vsftpd, the number of interactions matched exactly. But for ngIRCd, the number of interactions differed slightly at strength 2, 4, 5 and 7.

To explain the differences in results, we compared the interactions of the same lines of code generated by the two analysis techniques. We found that for most lines, the interactions matched exactly, but on rare occasions, the execution conditions analysis can produce interactions that are not minimum; some extra option settings might be attached to the actual interactions.

To ensure the configuration interactions generated using the execution conditions analysis are still safe in terms of guaranteeing coverage, we performed verifications for both vsftpd and ngIRCd. Since we have the complete execution paths information obtained through symbolic evaluation for these systems, we verified whether every path consistent with the interactions actually executes the lines of code guaranteed by the interactions. Our verification confirmed that every line is indeed executed if these interactions are present in a configuration.

3.3 Understanding Configuration Space Sampling

To cope with the *software configuration space explosion* problem, a lot of research has been done to develop configuration space sampling techniques. However, most of the previous research took the black-box approach. While black-box approaches have many strengths, they also have real limitations. One key problem is that their assumptions about the configuration spaces may not accurately reflect the structure of the software systems. When this occurs developers will expend valuable resources in inefficient ways, testing and analyzing configurations that don't expose new behavior or failing to consider configurations that do.

As far as we know, we are the first to apply white-box techniques to study a software system's configuration space. And our analysis techniques take into account of a system's internal structure, therefore, should be able to provide more accurate assumptions about its configuration space. In this section, we compare our analysis results against the commonly accepted assumptions about the configuration spaces that the existing configuration space sampling approaches are based on.

3.3.1 Analysis of Existing Approaches

There are numerous configuration space sampling approaches. One such approach is the CIT which systematically generate t-way covering arrays, in which all possible t-way combinations of option settings appear in at least one configuration in the samples. Studies have suggested that testing with even relatively low interaction strength (2- or 3-way) covering arrays tends to yield good structural

coverage and good fault detection [8, 19, 42]. Another popular approach is to randomly select configuration samples from a system's full configuration space. Even though, random sampling does not have a constructive method for choosing the size of the configuration samples, it has proven itself as an effective testing technique in practice [20].

However, not much research was done to scientifically understand how well these approaches really work or if their assumptions about the underlying configuration spaces are correct. So far our analysis results have confirmed some of what researchers and practitioners have long suspected, that a software system's effective configuration space is much smaller than its full configuration space and that low strength interactions can achieve good coverage. However, our results do not completely agree with all of existing assumptions either. For instance, we found that the configuration interactions are actually rare and that the configuration options do not fully interact with each other at a set t strength. We hypothesize that, for many practical tasks, these existing sampling approaches are both inefficient and ineffective at generating configuration samples that can achieve complete coverage during testing.

To support our hypothesis, we leverage the symbolic evaluation and configuration interaction data to perform two studies for evaluating the cost-effectiveness of covering arrays and random sampling. Knowledge gained from these evaluations can serve as the foundation for developing more effective sampling approaches.

3.3.1.1 Experimental Design

For each subject system, we generated multiple samples of the configuration space, which define the concrete configurations that are used to execute the system under. Multiple samples ensure that any one good or bad sample will not skew our analysis results. For covering array sampling, we generated 30 sets of 2- and 3-way covering arrays at each t strength, for the configuration options we analyzed earlier, using the Covering Arrays by Simulated Annealing (CASA) [30] tool. Normally, when using covering arrays, the possible settings of non-enumerated (i.e., integer) options are manually selected by developers. However, the settings we used came from our symbolic evaluation results in which we determined key settings that maximized path coverage for our subjects systems and test suites. This may make our covering array samples more cost-effective than those used in practice. The covering arrays we generated for vsftpd had 10–12 configurations for the 2-way samples and 32–35 configurations for the 3-way samples. For ngIRCd, the 2- and 3-way covering arrays contained 32 and 128–132 configurations, respectively. For each covering array we generated, we also generated an equally-sized randomly sampled set of configurations.

3.3.1.2 Structural Coverage Evaluation

In our first study we analyze the line coverage achieved by the sampling approaches for both subject systems. For this study we collected execution information for every line of code reachable by our test suites under any configuration. We want

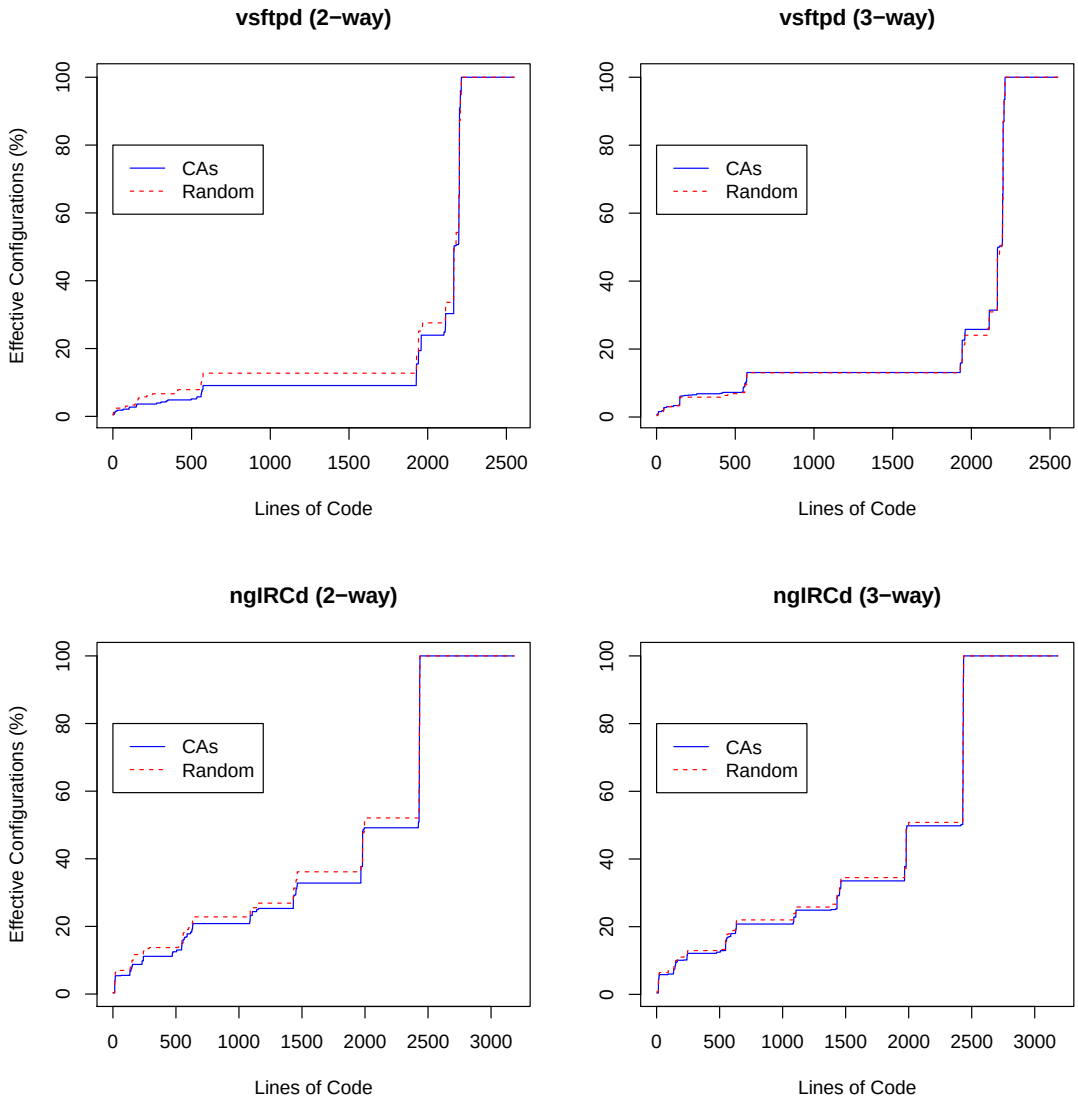


Figure 3.10: Percentage of effective configurations for every reachable line of code in various configuration samples.

to count the average number of effective configurations, configurations that are able to execute any given line in at least one test case, for each sample size. Figure 3.10 depicts this data.

From the figure, we clearly see that the data sets in subplots are practically identical regardless of the sampling approach used. Next we note that, for every

sampling set, some lines had an effective configuration set size of zero. That means there are some lines of code, that are reachable in at least one configuration, but were not covered by the configuration samples. Vsftpd's covering arrays and random samples missed an average of 1.18% of the reachable lines. NgIRCd's covering arrays and random samples missed on average 0.44% of the reachable lines. Through manual inspection we verified that these lines of code can only be executed when 4 or more configuration options take on specific settings. These settings are not guaranteed to appear in a 2- or 3-way covering array and in this case did not occur by chance either in the samples. On the other extreme, there are 13.18% and 23.15% of the reachable lines that are always executed regardless of configuration for vsftpd and ngIRCd, respectively.

The remaining lines lie somewhere in the middle; they are covered by some, but not all configurations. Moreover, the observed sizes tend to be small. For vsftpd, almost 80% of the reachable lines had effective configuration set smaller than 20% of the sampling set. And for ngIRCd, 65% of the reachable lines of code had the effective set smaller than 40% of the sampling set. A likely explanation for this is that the patterns in the underlying configurations, many involving a few specific option setting, are guarding the execution of these lines.

We see in the figure that these lines cluster into groups with the same effective configuration set size – this creates the "stairstep" pattern in the graph. For vsftpd, almost half of the lines had the same effective configuration size. We think these lines of code all required the crucial 3-way interaction which enabled the major functionality of vsftpd. The plots for ngIRCd had a distinctive laddering effect for

both t-strengths. This also can be explained by ngIRCd's enabling options that iteratively enable more behavior as additional option settings are included.

The data on a system's actual configuration interactions is important because, for example, if there are many unique interactions then a covering-based approach might be cost-effective. On the other hand, if there are very few unique interactions, then a more selective approach might be warranted. Using only the coverage data, we cannot determine the exact interactions involved in the configuration samples, thus, we turn to the configuration interaction data.

3.3.1.3 Interaction Coverage Evaluation

In our second study, we analyze the covering arrays and random samples for the configuration interactions they covered. For this study we measured the effectiveness of the sampling approaches by the degree they included the subject systems' configuration interactions.

Figure 3.11 shows the percentage of unique configuration interactions that are covered by each of the sampling methods for vsftpd and ngIRCd. These results can be explained by observing that the t-way covering array samples guarantee to cover the t-way interactions but may cover some more complex interactions by chance. For vsftpd, roughly 55% of the configuration interactions are covered by the 2-way samples and 80% are covered by the 3-way samples. However, for ngIRCd, which has more complex configuration interactions, the coverage is lower; slightly higher than 20% for the 2-way samples and about 40% for the 3-way samples. The

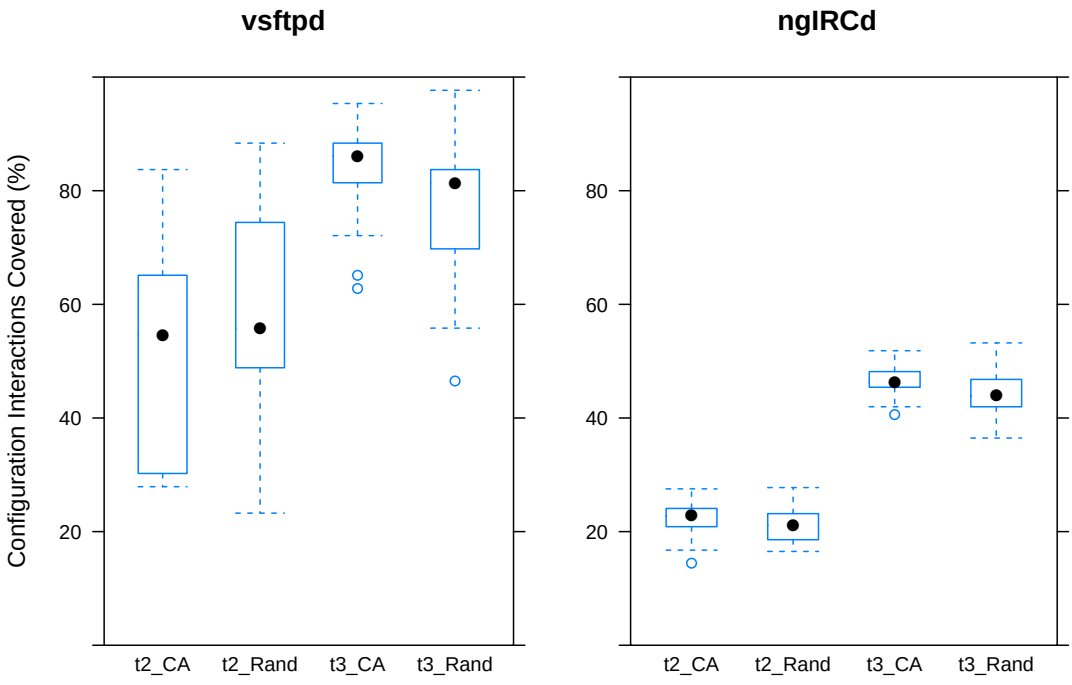


Figure 3.11: Percentage of configuration interactions covered by various configuration samples.

random samples, having the same size as the covering array samples, covered similar percentages of the interactions for both subject systems. This is consistent with the results from our previous study on coverage.

Even though both sampling methods covered similar percentage of the configuration interactions, however, our findings suggest both covering arrays and random sampling are inefficient at doing so for our subject systems. Due to the sparseness of configuration interactions, covering arrays, which are designed to cover all interactions of a given strength, covered the actual interactions only occasionally. And random sampling, lacking a systematic technique, depended on luck. And unless specific interaction patterns, especially the enabling options, are included by the configuration samples, majority of the lines cannot be executed. We confirmed that

most of the sampled configurations did not include the crucial enabling options of vsftpd and ngIRCd. This explains why covering arrays are inefficient at covering the required interactions.

We also see that, the 3-way samples covered significantly more of the interactions for both systems. Since the 3-way samples did not have significantly more overall coverage, we can conclude that many of the configuration interactions are actually redundant when projected to line coverage. However, even at 3-way strength, none of the sampling sets covered all of the interactions. This result again shows that configuration interactions can be redundant for line coverage.

In order to reach full coverage using covering arrays, the number of configurations would increase exponentially as the t strength increases. However, we know that the number of higher strength interactions do not increase exponentially as t strength increases; in fact, there are fewer actual interactions when $t = 4$ or higher for our subject systems. This suggests that covering arrays are doing too much work covering all interactions of a set t strength, and at the same time, they are not doing enough work covering some higher strength interactions.

The results from these studies is partially consistent with existing research results, that even relatively low interaction strength covering arrays yielded good structural coverage for our subject systems, but, they do not always guarantee the execution of every line of code. The random samples with similar sampling sizes performed on par with the covering arrays. However, random sampling which lacks the systematic method to determine the sampling size would rely on developer intuition in practice. From these results we conclude that a more effective sampling

method should focus on the coverage of actual interactions of the software systems instead all potential interactions in the full configuration spaces.

3.3.2 Minimal Covering Sets

Next, we want to investigate whether we can use the configuration interactions to generate more effective configuration samples that are also smaller than the samples generated by the covering arrays and random sampling.

To do this, we developed a greedy algorithm that packs interactions together to form a minimal set of configurations that achieves the maximum possible coverage of our subject systems. We want to pack consistent interactions together to form complete configurations, which assign values to all configuration options. For example, 1-way interactions `a=0` and `b=0` are consistent and can be packed into the same configuration, but `a=0` and `a=1` are contradictory and must go in different configurations.

We begin with the empty list of configurations. At each step of the algorithm, we pick the interaction that (if we also include the coverage of all subsets of that interaction) guarantees the most previously uncovered lines, blocks, etc. Then, we scan through the list to find a configuration that is consistent with our pick. We merge the picked interaction with the first such configuration we find in the list, or append this interaction to the list as a new configuration if it is inconsistent with all existing configurations. This algorithm will always terminate and cover all lines, blocks, etc., though it is not guaranteed to find the actual minimum configuration

Config #	1	2	3	4	5	6	7	8	9
vsftpd									
Line	2,521	18	8	1	1	-	-	-	-
Block	2,853	25	9	1	1	-	-	-	-
Edge	2,731	50	17	6	1	1	1	-	-
Condition	1,132	71	14	9	2	1	1	1	1
ngIRCd									
Line	3,148	30	6	6	1	1	1	-	-
Block	4,401	50	8	7	4	1	1	-	-
Edge	4,390	62	14	8	6	2	2	2	-
Condition	1,881	27	23	5	4	1	1	1	1

Figure 3.12: Additional coverage achieved by each configuration in the minimal covering sets.

set.

3.3.2.1 Data and Analysis

Figure 3.12 summarizes the results of our algorithm. The column labeled 1 shows how many lines, blocks, edges, or conditions are covered by the first configuration in the list. Then column n (for $n > 1$) shows the additional coverage achieved by the nth configuration over configurations $1..(n-1)$. Notice that minimal covering sets range in size from 5 to 10, which is much smaller than the number of possible configurations. We inspected these minimal covering sets and, for some coverage

metrics, we discovered that the results were in fact minimum. For the others, we simply verified that there was no obvious way to generate smaller configuration sets.

This suggests that when we abstract in terms of coverage, in fact the configuration space looks more like a union of disjoint interactions (that can be efficiently packed together) rather than a monolithic cross-product. Our algorithm was able to generate configuration sets with such small sizes because many of the interactions are consistent with others. These consistent interactions exist because of two main reasons:

1. The configuration options do not interact fully; there are clusters of options interact independently of each other.

2. Many of the higher strength interactions actually subsumes the lower strength ones, therefore, their values do not contradict.

We can also see that each subject system follows the same general trend, with most coverage achieved by just the first configuration. And the last several configurations often add only one additional coverage element. This last finding again confirms that not every interaction offers the same level of coverage.

Finally, we also used this algorithm to compute a set of configurations which ensures that every realizable *path* is executed at least once. These effective configuration spaces of vsftpd and ngIRCd contained 3,092 and 3,518 configurations, respectively. While significantly larger than for the simpler coverage criteria, these numbers are still far smaller than the size of the full configuration space.

3.4 Summary

In this chapter, we performed several empirical studies to understand the configuration spaces of two medium-sized subject systems and confirmed our first research hypothesis that for many practical tasks, the effective configuration space of a software system is much smaller then its full configuration space.

First, we used *symbolic evaluation*, a white-box analysis technique, to generate all execution paths these subject systems can take on under any configuration. The symbolic evaluation results showed that the possible execution paths are only tiny subsets of these systems' full configuration spaces. We also found evidence that suggest systems are structured in ways that not only limits how configuration options can combine, but also limits the number of options that can combine to exercise new behavior.

Next, we developed two techniques, the *guaranteed coverage* analysis and the *execution conditions* analysis, and calculated the *interactions* between the configuration options of these subject systems. We did this by projecting the execution paths, obtained via symbolic evaluation, onto different structural coverage criteria and found that when using more abstract properties, the effective configuration spaces can be collapsed even further. We gained three key insights about the effective configuration spaces:

1. Configuration interactions were quite rare; only a handful of specific options setting combinations had to be exercised to maximize coverage.

2. Most of the interactions needed to achieve maximum coverage were of low

strength but higher strength interactions are needed to achieve the maximum coverage.

3. Higher strength interactions were usually just lower strength interactions with one or more additional constraints.

Then, we used the configuration interaction data and performed empirical evaluations on two popular configuration space sampling approaches, the covering arrays and the random sampling. We showed that these two sampling approaches are quite inefficient and ineffective at achieving full coverage because they do not precisely cover the required configuration interactions. What we found suggests that a more effective sampling approach should focus on the coverage of actual interactions of the software systems instead all possible interactions in the configuration spaces.

Finally, we developed a more selective sampling approach that uses a greedy algorithm to pack the configuration interactions into small configuration samples we call *minimal covering sets*. The minimal covering sets generated for our subject systems are very small (with only 5-10 configurations) but they can more effectively achieve full coverage during software testing than the existing approaches.

Chapter 4

Discovering the Effective Configuration Space

The results from our analyses in the previous chapter gave us great insights to the effective configuration spaces of software systems. We can exploit these analysis results to dramatically improve the effectiveness of many configuration-aware software engineering tasks. However, the techniques we developed to generate these results are computationally very expensive. For most practical tasks, the developers need more cost-effective and time-sensitive techniques to analyze a system's configuration space. In this chapter we look at our second research hypothesis: We can efficiently discover or approximate the effective configuration space of a software system. We aim to provide answers to the following two research questions:

1. Can we discover effective configurations using cost-effective and time-sensitive techniques?

2. Can the configuration interactions be discovered or estimated without the complete execution paths of a software system?

To address the first question, we developed an new approach that is much lighter-weight than symbolic evaluation, yet still effectively explores the configuration space of a software system. This approach, we call iTree – an *interaction tree discovery algorithm*, can discover sets of configurations that achieve better coverage

during software testing while also has fewer configurations than those chosen by traditional CIT. The iTree approach combines low-strength covering arrays, runtime instrumentation, and machine learning (ML) techniques to construct an *interaction tree* for the software system. An interaction tree is a hierarchical representation of what we call *proto-interactions*, which are potential interactions or subsets of potential interactions. And the iTree performs an iterative, search-based process in which the current iteration's configuration samples are based on the proto-interactions in the interaction tree. We conducted several experiments designed to evaluate the performance of the iTree algorithm and found that this new approach can quickly achieve full coverage. We also evaluated the scalability of iTree to a large-scale system, specifically the ∼1M-LOC MySQL database system, for which symbolic evaluation is infeasible. The evaluation results show that iTree can easily scale up to practical industrial systems.

To address the second question, we developed the *interaction learning* approach that can quickly estimate the configuration interactions of a software system. This approach uses decision tree classifiers to "learn", from the execution results of the configurations discovered by iTree, the likely option setting combinations responsible for the coverage of the program entities. We evaluated the accuracy of the estimated configuration interactions of vsftpd and ngIRCd by comparing them to the interactions calculated using the symbolic evaluation data, we found that the estimations are almost exactly the same as the actual interactions. We then used this approach to estimate the configuration interactions of the much larger MySQL database system and showed that it can easily scale to large software systems; in

fact, the entire process took just minutes to complete on a single machine.

The following sections describe these new approaches in more detail. Section 4.1 illustrates the observations on effective configuration spaces that motivated the iTree's design and presents the implementation choices of the iTree discovery algorithm. Section 4.2 presents a series of experiments we conducted to improve the performance of iTree by fine tuning its parameters and heuristics. Section 4.3 presents several empirical evaluations of iTree's performance by comparing it to existing configuration space sampling approaches, including a scalability evaluation experiment using the MySQL database. Section 4.4 discusses an approach to extract an even smaller configuration set from an iTree run for subsequent testing tasks. Section 4.5 describes the interaction learning approach and analyzes the estimated configuration interactions for vsftpd, ngIRCd, and MySQL.

4.1 Using iTree to Discover Effective Configurations

Although we began our work focusing on exploring a hypothesis, our ultimate goal is to use the knowledge of a system's effective configuration space to make software engineering tasks more cost-effective in practice. Symbolic evaluation, which all of our configuration space analysis techniques from Chapter 3 relied on has several practical limitations:

1. The existing implementations can only be used to analyze run-time configuration options, not compile-time options. This would severely limit the types of systems and configuration spaces that can be analyzed.

2. The approach is computationally expensive and does not scale. As an example, our subject systems vsftpd and ngIRCd were both about 10K LOC and had no more than 30 configuration options. The analysis of these systems, each required 40 client machines running for several days. For practical systems with 100K to 1M LOC and 100+ configuration option, symbolic evaluation will simply be infeasible.

3. Symbolic evaluation analyzes the software systems in a simulated environment. It requires special modifications to the systems to run in that environment and it does not perform actual testing of the systems. Therefore, after the analysis steps, additional steps must be taken to perform testing.

In this chapter, our goal is to create a more practical approach that can address these limitations. This approach should efficiently discover a software system's effective configuration space without the reliance on developer intuitions. To reach this goal, we opted to use dynamic analysis, which can handle both run-time and compile-time configuration options, to perform actual execution of the systems without special environments or modifications to the source code. The result of our efforts is the iTree, an *interaction tree discovery algorithm.*

4.1.1 iTree Design Motivation

The key motivation and intuition behind the iTree approach stem from our observations of the configuration interactions that we made in the previous chapter. To better understand these observations that motivated the design of such

```
1   int* dsa_cert_file=NULL; /* test input */
2   int one_process_mode=1;
3
4   if (tunable_listen} {
5       if (tunable_accept_timeout) {
6           /* L1: tunable_listen ∧ tunable_accept_timeout */
7       } else {
8           /* L2: tunable_listen ∧ ¬tunable_accept_timeout */
9       }
10  } else {
11      /* L3: ¬tunable_listen */
12  }
13
14  if (tunable_ssl_enable) {
15      if (!dsa_cert_file)
16          die();
17  }
18  /* L4: ¬tunable_ssl_enable */
19
20  if (one_process_mode) {
21      if (tunable_local_enable || tunable_ssl_enable)
22          die();
23  }
24  /* L5: ¬tunable_ssl_enable ∧ ¬tunable_local_enable */
25
26  if (!tunable_local_enable && !tunable_anonymous_enable)
27      die();
28  /* L6 (lots of code) : ¬tunable_ssl_enable ∧ ¬tunable_local_enable
29                          ∧tunable_anonymous_enable */
30
31  if (tunable_dual_log_enable) {
32      /* L7: ¬tunable_ssl_enable ∧ ¬tunable_local_enable
33             ∧tunable_anonymous_enable ∧ tunable_dual_log_enable */
34  } else {
35      /* L8: ¬tunable_ssl_enable ∧ ¬tunable_local_enable∧
36             tunable_anonymous_enable ∧ ¬tunable_dual_log_enable */
37  }
```

Figure 4.1: A simplified snippet of vsftpd's source code and its configuration interactions.

an approach, we illustrate them using the example program in Figure 4.1. This example contains a highly simplified snippet of vsftpd's server startup 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 its value here for simplicity. This example program also contains six binary configuration options, highlighted in bold, whose values depend on the system's runtime configuration.

Figure 4.1 includes eight regions of code, marked */* *L1–L8* **/*, in whose coverage we are interested in. The coverage of these regions, of course, depends on the values of the configuration options and the program variables. For each region, we list the configuration interaction that controls the coverage of that line for this particular test case. For example, at the beginning of the program, the coverage of *L1*–*L3* depends on the values of the configuration variables tunable_listen and tunable_accept_timeout.

More interestingly, for the execution to reach the large amount of code in *L6*, several options must be set in specific ways. First, to reach *L4* and any code thereafter, tunable_ssl_enable must be set to 0, because this test case sets dsa_cert_file to be NULL. Next, consider reaching *L5*. Since one_process_mode is set to true, to reach *L5* the condition on line 21 must be false; and since as just discussed tunable_ssl_enable is 0 if we reach this line, then tunable_local_enable must also be 0. Finally, to continue on to reach *L6*, we need the condition on line 26 to be false, and since tunable_local_enable is 0 if we reach that line, we must set tunable_anonymous_enable to 1. Putting this together, any configuration that reaches *L6* for this test case needs at least tunable_ssl_enable = 0, tunable_local_enable = 0, and tunable_anonymous_enable = 1, the enabling options of vsftpd. Finally, the coverage of *L7* and *L8* also depends on the value of tunable_dual_log_enable.

Note that although in this example we were able to reach all of the code regions, and coverage of each region 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 both vsftpd and ngIRCd. From these patterns, we make three observations:

1. Configuration interactions are relatively rare. The code shown in Figure 4.1 includes six binary options, so in the worst case there could be 639 different interactions; 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*. In the example program, 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 code regions. We observed that, for vsftpd and ngIRCd, there were only 43 and 435 configuration interactions respectively.

2. Most coverage can be explained by lower-strength configuration interactions. In the example program, five of the eight interactions involve only one or two option settings. One more interaction involves three settings, and the remaining two involve four option settings each. While this example is highly simplified, we found the same trend in the actual systems. For the subject systems and test suites we examined, over 94% of the achievable coverage

could be achieved with lower-strength configuration interactions (i.e., with four or fewer option settings). Full coverage, however, required a handful of higher-strength interactions (up to strength seven).

3. Higher-strength configuration interactions tend to be built on top of lower-strength ones. As shown in the example, the higher strength interactions guaranteeing coverage of `L7` and `L8` are refinements of the interaction at `L6`, which is itself a refinement of `L5`'s interaction. 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 source code. That is, the higher-strength interactions often add additional constraints to the existing lower-strength interactions.

4.1.2 Algorithm and Implementation

Based on the observations just discussed, we developed the interaction tree discovery algorithm (iTree). iTree's goal is to automatically discover and execute a small set of highly effective (e.g., high coverage) configurations. iTree works as follows. First, it instruments the system under test to measure some desired type of coverage. This chapter focuses on line coverage, but the algorithm should apply to any type of coverage — it only requires that coverage of a configuration can be expressed as a mapping between a program entity and a boolean indicating whether it has been covered or not. Next, iTree repeats the following steps until a stopping criteria is met. First, it computes a small sample of configurations under which to

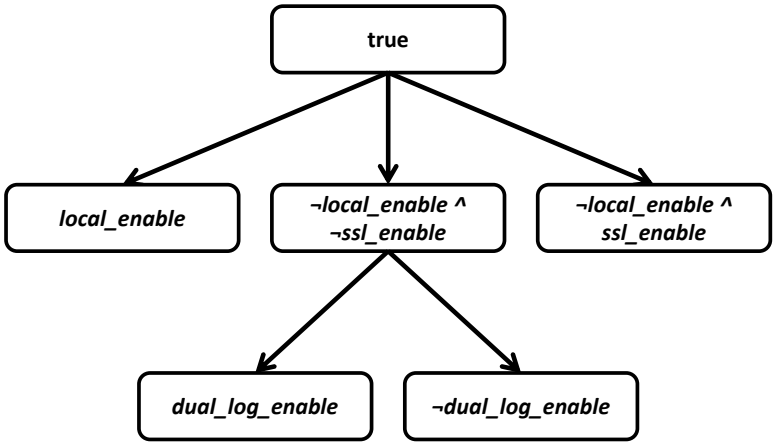


Figure 4.2: An interaction tree for the example program in Figure 4.1.

test the system. As we shall see later, the goal of using a sampling approach is to select configurations that are likely to execute previously uncovered program entities. Next, iTree runs the system's test suite on each of the sampled configurations and captures coverage information from those runs. Using this coverage data, iTree then attempts to discover *proto-interactions* — conjunctions of option settings — that cause the new coverage and that may warrant further exploration in the future iterations of iTree.

We represent iTree's behavior as an *interaction tree*, which is a hierarchical representation of the proto-interactions it discovered. The nodes of the interaction 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 some nodes may represent full interactions with additional constraints. Figure 4.2 shows the interaction tree for the example program from Figure 4.1. Each node is labeled with a set of option settings, with

```
1  iTree = /* tree containing root 'true' */
2  runs = ∅ /* (config × coverage) set */
3
4  do {
5      node = findBestLeafNode(iTree, runs);
6      configSet = generateConfigSet(node.proto_interaction);
7
8      newruns = executeConfigSet(configSet);
9      if cov(newruns) ⊆ cov(runs)
10       continue;
11     runs = runs ∪ newruns
12
13     interactions = discoverProtoInters(node.proto_interaction, runs);
14     if !(interactions.empty())
15       /* Add newly discovered interactions to tree */
16       updateTree(iTree, node, interactions);
17
18 } while (!stoppingCriteriaMet());
```

Figure 4.3: Pseudocode for the interaction tree discovery algorithm.

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 proto-interaction ¬tunable_local_enable∧¬tunable_ssl_enable∧ tunable_dual_log_enable is represented by the left node on the lowest level of the tree. We also see that ¬tunable_local_enable ∧ tunable_ssl_enable is in the interaction tree, but does not correspond to an actual interaction that guarantees coverage of any particular line of code. Thus, in this case, iTree has created a proto-interaction that will not lead to useful higher-strength interactions.

Figure 4.3 gives the pseudocode for the iTree algorithm. iTree runs in a loop, iterating until a particular stopping criteria is met (e.g., no more coverage is achieved or a developer-specified 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() selects from the interaction tree a leaf node to explore next. Since we might not be able to fully explore an interaction tree (which would be too expensive), findBestLeafNode() uses various heuristics to pick the most promising node, according to the coverage information in runs, to explore.

Next, the proto-interaction represented by the path to the selected node is passed into generateConfigSet(). This method creates a sample set of configurations in which every configuration is consistent with the proto-interaction represented by the selected node, while the set of configurations broadly samples all the other options not participating in the proto-interaction. In our implementation, iTree leverages CIT for this step, but other sampling techniques could be substituted.

After this, 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 techniques to identify further proto-interactions that account for any newly-covered program entities. Note that, by design, any proto-interactions discovered at this step must include the option settings in node.proto_interaction. Finally, updateTree() adds the newly discovered proto-interactions to the interaction tree as children of the currently selected node. We now discuss each step of the algorithm in more detail.

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

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

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 adding one in the denominator means that nodes corresponding to fewer runs will have lower priority than their average coverage, but it has little effect on nodes corresponding to many runs (since then $|Conf(\mathsf{runs}, \mathsf{node})|$ is high). We found this adjustment to be useful in that it leads to a slight, but beneficial, preference for nodes that correspond to multiple, high-coverage configurations, over nodes which 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 [30] to generate a low-strength covering array over only the remaining options. We then combine those partial configurations with the

	ssl	local	listen	accept	anonymous	dual_log
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	**local**	listen	accept	anonymous	dual_log
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 = 0 and local = 0

ssl=tunable_ssl_enable	local=tunable_local_enable	listen=tunable_listen
accept=tunable_accept_timeout	anonymous=tunable_anonymous_enable	dual_log=tunable_dual_log_enable

Figure 4.4: Example 2-way covering arrays generated during an iTree run.

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 very sensitive to this choice.

Figure 4.4 shows two covering arrays created by generateConfigSet() as iTree discovered the interaction tree in Figure 4.2. In this case we chose to generate 2-way covering arrays. Figure 4.4(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 4.1) ¬tunable_ssl_enable ∧ ¬tunable_local_enable ∧ tunable_anonymous_enable (in C5) needed to reach *L6* and beyond, and the 4-way interaction ¬tunable_ssl_enable ∧ ¬tunable_local_enable ∧ tunable_anonymous_enable ∧ tunable_dual_log_enable (also in C5) needed to reach *L7*. After the coverage data from these configurations was analyzed, iTree added the three children of the *true* node shown in Figure 4.2.

The next iteration of iTree expanded the middle of the three leaf nodes, which has the highest priority score (since this node covered *L6* that contains many lines of code), and generateConfigSet() then created the 2-way covering array shown in Figure 4.4(b). Note that in this covering array, the values of tunable_ssl_enable and tunable_local_enable are fixed. As a result, this 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 *L8*, in C7 and C10). At this point, iTree has covered all the marked lines of the example program.

executeConfigSet(): This function instruments the system under test, executes its test suite under each configuration in the sample and collects the coverage information. Different implementations can be developed to handle different programing languages, instrumentation tools, and execution environments. In our implementa-

tion, we compute the line coverage with gcov [31], the GNU coverage profiling tool for C and C++. We execute the instrumented systems on Skoll [55], a distributed, continuous quality assurance system running on a grid comprising 120 machines. As we will discuss in Section 5.1, using Skoll allowed us to easily scale up iTree to test and analyze large scale software systems under many configurations 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 the proto-interactions responsible for the differences in execution.

In the first step, we find all runs involving configurations consistent with the proto-interaction that iTree is exploring. Note that we extract this subset of configurations 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 [34] implementation of CLOPE [81], a clustering algorithm that groups together similar transactional data records with high dimensionality. Thus, we translate line coverage data from each run into an appropriate form: Every line of code is a boolean attribute, set to *true* if covered in a run and *false* otherwise. Then we use CLOPE to cluster together configurations that executed many of the same lines.

In the second step, we use decision tree classifiers [65] to discover commonalities of option settings among the configurations in each of the clusters. These common patterns are the proto-interactions that are responsible for the differences between

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 that a configuration belongs to based on its 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 interaction tree path stops. In our experiments we evaluated several decision tree algorithms and found each to be adequate for this task.

In the previous step we used CLOPE for clustering the configurations. CLOPE requires a special parameter called repulsion, which ranges from 0.5 to 4.0, to control 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 set of proto-interactions generated under the range of repulsion values.

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

4.2 Evaluating iTree Parameters

We explored iTree's cost-effectiveness in a series of experiments, described in this section. Our first experiment, presented next, aims to determine two key parameters to the algorithm: the covering array strength to use in generateConfigSet(), and the decision tree implementation to use in discoverProtoInters(). Our second experiment explores techniques that can reduce the size of the configuration set generated by an iTree run. And our third experiment explores modifying iTree to adaptively select configuration sample size and use multiple decision tree classifiers simultaneously to improve effectiveness.

4.2.1 Subject Systems

For these experiments, we use the two subject systems we have studied extensively in Chapter 3: vsftpd, and ngIRCd. In these experiments, we use the same test suites for these systems and detailed information about the systems' configuration spaces with respect to those test suites.

Figure 4.5 recaps the program statistics relevant to the current experiments. These systems have roughly 10-13K LOC, written in C. The figure details the total number of configuration options we analyzed, and the counts broken down

	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 4.5: Recap of relevant program statistics of vsftpd and ngIRCd for the iTree experiments.

by type (boolean or integer); this is the same set of options and settings that we used in Chapter 3. The values we used for the integer options also came from the analysis results of the previous chapter, and were chosen to maximize path coverage for these subject systems and their test suites. 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.

4.2.2 iTree Parameters

4.2.2.1 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, but may also influence the speed with which iTree terminates. In this study, we use either $t = 2$ or $t = 3$ at each iteration of the algorithm. In Section 4.2.5, we explore other ways to tweak the sizes of the configuration samples.

4.2.2.2 Decision Tree Algorithms

Many different decision tree classifiers have been proposed in the machine learning literature. We used two algorithms in our experiments: The C4.5 [59], an extension to the earlier ID3 [58] decision tree, that uses heuristics to attempt to generate simpler (smaller) decision trees; and the Classification and Regression Trees (CART) [7], which generates regression trees by finding rules based on variables values to split the data instances and prunes the resulting trees if possible. We picked these classifiers because they are the most popular decision tree implementations and they were designed to produce compact classifications, which may be well-suited to iTree's incremental search approach. We use Weka's [34] implementation of both of these decision tree algorithms.

The classification models generated by these two decision trees have important differences. Figure 4.6 shows the C4.5 and CART classification models generated for ngIRCd using the Waka implementations. As we can see the C4.5 model creates

```
Conf_ListenIPv4 = 1
| Conf_PongTimeout = 1: class0 (10.0)
| Conf_PongTimeout = 20: class1 (13.0/4.0)
| Conf_PongTimeout = 3600
| | Conf_MaxNickLength = 0: class0 (0.0)
| | Conf_MaxNickLength = 4: class0 (3.0)
| | Conf_MaxNickLength = 5: class0 (1.0)
| | Conf_MaxNickLength = 6: class1 (1.0)
| | Conf_MaxNickLength = 8: class0 (0.0)
| | Conf_MaxNickLength = 9: class1 (3.0)
| | Conf_MaxNickLength = 10: class0 (0.0)
| | Conf_MaxNickLength = 100: class0 (0.0)
Conf_ListenIPv4 = 0: class2 (33.0)
```

(a) C4.5 Decision Tree

```
Conf_ListenIPv4=(0): class2(33.0/0.0)
Conf_ListenIPv4!=(0)
| Conf_MaxNickLength=(4)|(5)|(0): class0(13.0/0.0)
| Conf_MaxNickLength!=(4)|(5)|(0)
| | Conf_PongTimeout=(3600)|(20): class1(13.0/0.0)
| | Conf_PongTimeout!=(3600)|(20): class0(5.0/0.0)
```

(b) CART Decision Tree

Figure 4.6: Classification models generated using C4.5 and CART decision trees.

a unique path for every option setting, even if some settings lead to the same classification. The CART model, on the other hand, branches on rules for the options' settings rather than the individual values of the options.

These differences affect the process of extracting proto-interactions from the classification models. From the C4.5 models, we simply parse for distinct paths and each path is treated as a proto-interaction. For the CART models, however, we must first reference the system's configuration space model to solve the inequalities in the branching rules. For example, $\mathsf{Conf\_MaxNickLength!} = (4)|(5)|(0)$ would transform

to Conf_MaxNickLength = (6)|(8)|(9)|(10)|(100) using ngIRCd's configuration space model. We then generate one proto-interaction for every unique combination of option settings in a decision tree path.

4.2.2.3 Iteration Retries

An iTree iteration may fail to discover any proto-interaction for a number of reasons. For instance, the decision tree algorithm failed to generate a classification model as a result of poor performing configuration samples. Failing to discover a real interaction will cause iTree to improperly abandon the currently selected node and to continue with a proto-interaction with lower priority score. This can delay or even prevent the coverage of some necessary higher strength interactions.

The obvious solution to deal with failed iterations is to perform a retry, that is, to generate more configuration samples and performed the classification again. However, not all failed iterations should be retried either, since some proto-interactions do not lead to actual interactions. Retrying every iteration can dramatically increase the number of configurations tested.

For our initial experiments, our iTree implementation only allows for retries on the very first iteration because without any proto-interactions the algorithm cannot proceed. In Section 4.2.5, we explore other conditions under which retries should be performed.

4.2.3 Initial Evaluation

In practice, software testing gets limited time budget and computing resources that might prevent iTree from explore all proto-interactions in the interaction tree. Therefore, the goal of these initial experiments is to determine the iTree parameters that direct the testing efforts to the most important proto-interactions first and yield a set of configurations that can achieve high coverage quickly. In this experiment, we ran iTree 30 separate times on both subject systems under each possible combination of decision tree and covering array t strength. For each run, we continued the execution until we reached the maximum possible coverage (as determined from our prior analysis results). 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 running the executeConfigSet() step, we instead used the line coverage data we had already computed from the previous chapter (which gave us a mapping from configurations to their line coverage).

4.2.3.1 Data and Analysis

Figure 4.7 shows the results of our experiments for vsftpd and ngIRCd. The data is depicted using box and whisker plots. The left half of each chart shows the results of the decision trees under covering array strength $t = 2$, and the right half shows the results for $t = 3$. The y-axis reports the number of configurations tested to achieve full coverage. The number in parentheses under each plot indicates the

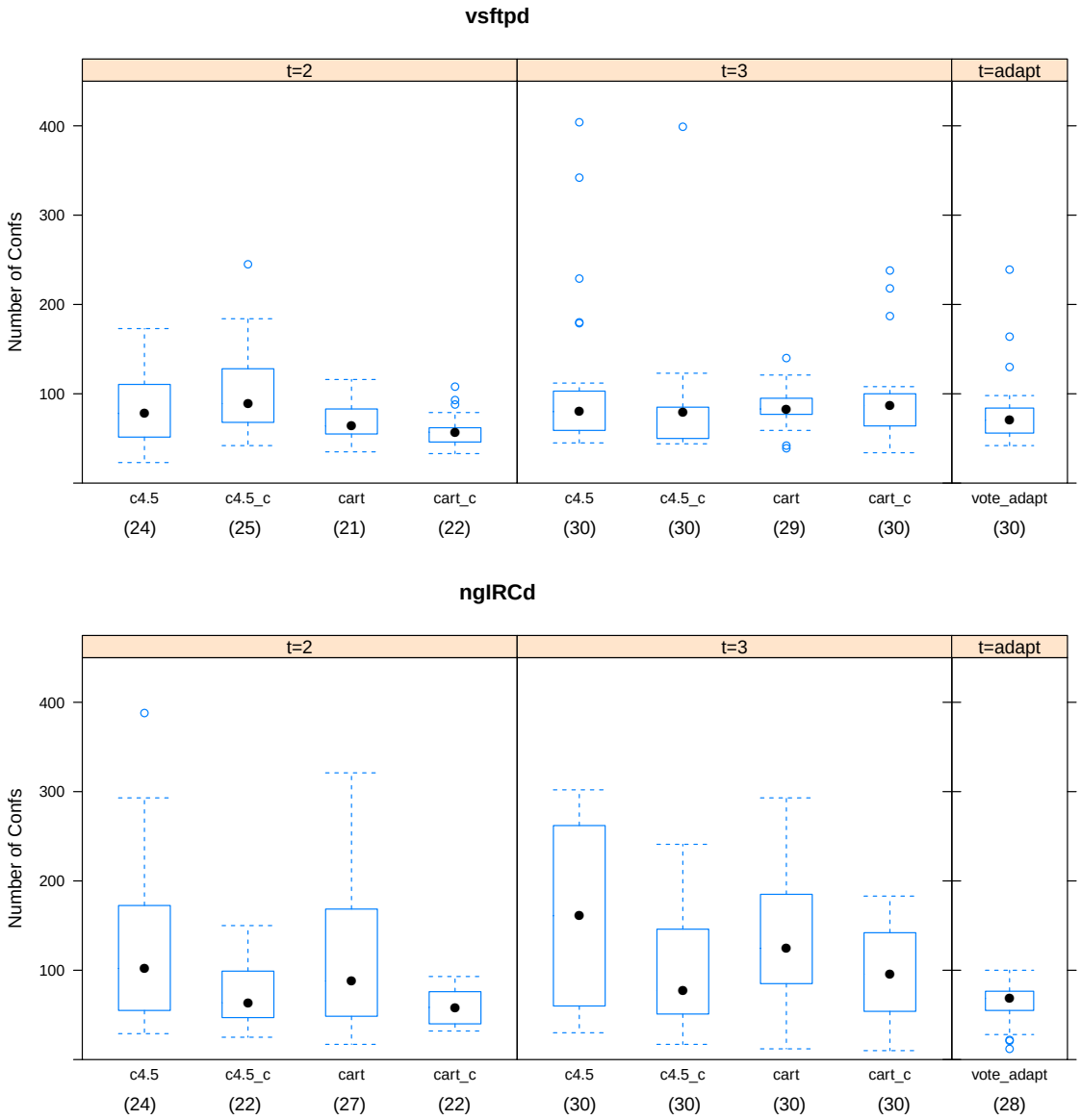


Figure 4.7: Interaction tree experiments using various iTree parameters and heuristics.

number of iTree runs, out of 30, in which full coverage was achieved. We defer discussion of c4.5_c, cart_c and vote_adapt to Section 4.2.4 and Section 4.2.5.

Covering Array Strengths: We see in Figure 4.7 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 the iTree algorithm reached maximal coverage is substantially higher when $t = 3$ than when $t = 2$, for both subject systems. We looked in more detail at the individual runs, and observed that both the likelihood of discovering proto-interactions and the accuracy of the discovered proto-interactions at each iteration dramatically improved as t increased. However, this resulted in a trade off. While increased sample size means 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 analysis results, contained many redundant option settings from the perspective of line coverage. This resulted in many equivalent proto-interactions being used to redundantly explore the same part of its effective configuration space and can delay iTree from reaching complete coverage. On the other hand, this also means ngIRCd's 2-way covering arrays experiments already enjoyed the benefits of larger configuration samples.

Decision Trees: In Figure 4.7, the c4.5 and cart columns for each t-value show the effect of the C4.5 and CART decision trees on iTree. The data shows no systematic differences in performance across the two classifiers. Looking at the individual iterations of iTree, however, we did find some differences: CART fail to discover any proto-interactions in the configuration samples more often than J48 does. Fortunately, this situation occurs mostly during the very first iteration and our retry heuristic guarantees that some proto-interactions will be discovered eventually, therefore, it does not impact the performance of iTree greatly.

In some of the runs, we find that both C4.5 and CART can discover proto-interactions that are not quite accurate. This usually results in poor iTree performance. We explore ways to improve the situation in Section 4.2.5.

4.2.4 Composite Proto-Interactions

Our iTree implementation so far treats every unique combination of option settings as a proto-interaction that should be explored further. However, it is evident from the ngIRCd's runs that some proto-interactions can be redundant and exploring them one at a time can dramatically increase the number of iTree iterations and the number of configurations tested. For example, to cover majority of ngIRCd's code, Conf_ListenIPv4 must be set to 1, but Conf_MaxNickLength can be set to either 20 or 3600, and Conf_MaxNickLength can be set to 6, 8, 9, 10, or 100; treating these proto-interactions individually would result in 10 iTree iterations. We cannot simpling ignore some of the option settings either, for instance, in order

to reach complete coverage for ngIRCd the 4-way interaction $\mathsf{Conf\_ListenIPv4} = 1 \wedge \mathsf{Conf\_MaxNickLength} = 6 \wedge \mathsf{Conf\_PongTimeout} = 3600 \wedge \mathsf{Conf\_PredefChannelsOnly} = 0$ must be covered by at least one configuration.

To efficiently handle multiple equivalent proto-interactions in a safe and efficient way, we created the *composite proto-interactions*. In a composite proto-interaction, each option can have more than one setting to represent all of the equivalent combinations. We modified generateConfigSet() to use a composite proto-interaction to generate covering array samples; instead of fixing every option that appeared in the proto-interaction to a single setting, the composite proto-interactions are used to reduce each option's possible settings that a CIT technique must cover. This way, a single iTree iteration can sample all t-way combinations of all the relevant option settings.

Some decision tree classifiers, such as the C4.5, create a unique path in its classification model for every equivalent branching value. To extract composite proto-interactions from such classification models, we use the following algorithm: First, we extract all the proto-interactions from the decision tree model and group them by their classifications. Then for each classification, we merge all the compatible proto-interactions. Proto-interactions are compatible if they constrain the exact same set of options. The merging process creates a composite proto-interaction by including, for each option, all the settings that appear in the compatible proto-interactions.

We note that, using this algorithm, the resulting composite proto-interactions can include combinations of option settings that do not actually belong to the same classification. But these composite proto-interactions are still safe for our use case

and do not occur frequently in practice.

4.2.4.1 Data and Analysis

Columns c4.5_c and cart_c in Figure 4.7 shows the performance of C4.5 and CART decision trees under each t-value using the composite proto-interactions. For vsftpd, using composite proto-interaction did not significantly change the performance of the iTree runs; this is understandable since most vsftpd's configuration options are boolean. But for ngIRCd, we see that both the number of iterations and the number of configurations needed to reach complete coverage decreased under both $t = 2$ and $t = 3$, this is especially significant for the $t = 3$ runs. This shows that using composite proto-interactions increased the likelihood of iTree covering all the interactions needed to reach complete coverage during an earlier iteration.

However, we also see that when $t = 2$, slightly fewer runs reached complete coverage for ngIRCd. We found the cause to be that a covering array generated (with the CASA tool) in a single iteration using a composite proto-interaction contains fewer configurations than several covering arrays generated in numerous iterations using several equivalent proto-interactions. In the case of $t = 2$, the smaller configuration samples were not as effective for executing new coverage and accurately discovering proto-interactions. The $t = 3$ runs, with larger samples, were not affected.

Overall, we find that using composite proto-interactions improved the performance and practicality of the iTree approach.

4.2.5 Adaptive Approach

When we examined the worst-performing runs from the previous experiments, we found that they suffered from inaccurate proto-interaction discovery. Both C4.5 and CART sometimes produce inaccurate classifications — they include option settings in the proto-interactions that are not part of the actual interactions. This situation can have great negative consequences, especially during the early stages of the iTree discovery, because the inaccuracies can propagate through an iTree path. This would effectively send iTree on a wild goose chase. Specifically, inaccurate proto-interactions may restrict configuration sampling to unimportant parts of the configuration space, thus preventing iTree from covering the interactions needed for complete coverage.

There are two main causes to the inaccurate proto-interaction discovery. The first cause is insufficient training examples provided to the decision tree classifiers. From our previous experiments we see that increasing the t-value improved the accuracy of the discovered proto-interactions. However, increasing the t-way also increased testing efforts. So, to increase the size of the configuration samples without dramatically increasing the test obligations, we decided to use two 2-way covering arrays for every iteration of iTree (with the CASA tool, 2-way covering arrays are about one third the size of 3-way covering arrays).

The second cause is inherent in the design of the decision tree classifiers. These classifiers are designed to minimize errors in classifying the training examples and this can cause the decision tree models to be overfitted for the configurations used

to discover the proto-interactions. We noticed, however, that C4.5 and CART uses different heuristics to reduce the error rate, and that the inaccuracies often involved different option settings. Thus, we developed an *aggregation classifier*, similar to bagging [6, 60], that creates an ensemble classifier out of C4.5 and CART decision trees. This aggregation classifier filters out option settings from proto-interactions unless both C4.5 and CART decision tree models produce them as classifiers. The assumption is option settings that appear in both models are more likely to be part of the actual interactions.

However, this aggregation classifier is also more likely to fail to discover proto-interactions; if the decision trees do not agree on any option setting during an iteration then iTree would be forced to abandon the current path. To alleviate this problem, iTree performs a retry of the current iteration if new coverage was executed during this iteration but not proto-interactions were discovered.

With the combination of aggregation classifier and the new retry condition, the iTree essentially produces either lower strength proto-interactions with option settings it is confident about or it increases the configuration sample size to produce more accurate classifications.

4.2.5.1 Data and Analysis

The results using the adaptive approach along with composite proto-interactions are shown in Figure 4.7's vote_adapt column. We can see from the figure that vote_adapt 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.

4.3 Performance Evaluation

4.3.1 Comparing iTree to Other Approaches

We now compare the performance of iTree against both traditional CIT and random sampling. As mentioned earlier, CIT and random sampling are popular approaches that produce relatively good results in practice. CIT generates a set of configurations that includes all possible interactions at a given t strength, and random sampling depends on the probability of including the right interactions in the configuration samples. To better understand how iTree compares with these existing approaches, we conducted a series of experiments.

4.3.1.1 Experimental Design

For these experiments, we again used vsftpd and ngIRCd and ran each approach 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 based on their experience or time constraints. In this experiment, we created covering arrays using a range of different t strengths. For each strength, testing ran until no more configurations remained in the sample sets. Using 5-way and 4-way covering arrays for vsftpd and ngIRCd, respectively often achieved the 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 these systems using iTree. For these experiments, we used vote_adapt as described in the previous section. Also, iTree is using its default stopping criteria. Using this stopping criteria does not require any inputs from the developers; iTree determines how much testing to perform by automatically stopping the process when no more proto-interactions are left unexplored in the interaction tree. We measure performance using two criteria: (1) whether complete coverage was reached by each approach and (2) if so, the number of configurations needed to reach the complete coverage.

4.3.1.2 Data and Analysis

Figure 4.8 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 30 runs. Here we are assuming that configurations are tested in the order they are generated by the respective approaches, although in actuality the testing process 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 and ngIRCd, respectively. We note that the largest covering arrays and random samples for vsftpd and ngIRCd contains on average 340 and 400 configurations respectively and the vote_adapt runs executed on average 255 and 345 configurations for vsftpd and ngIRCd respectively.

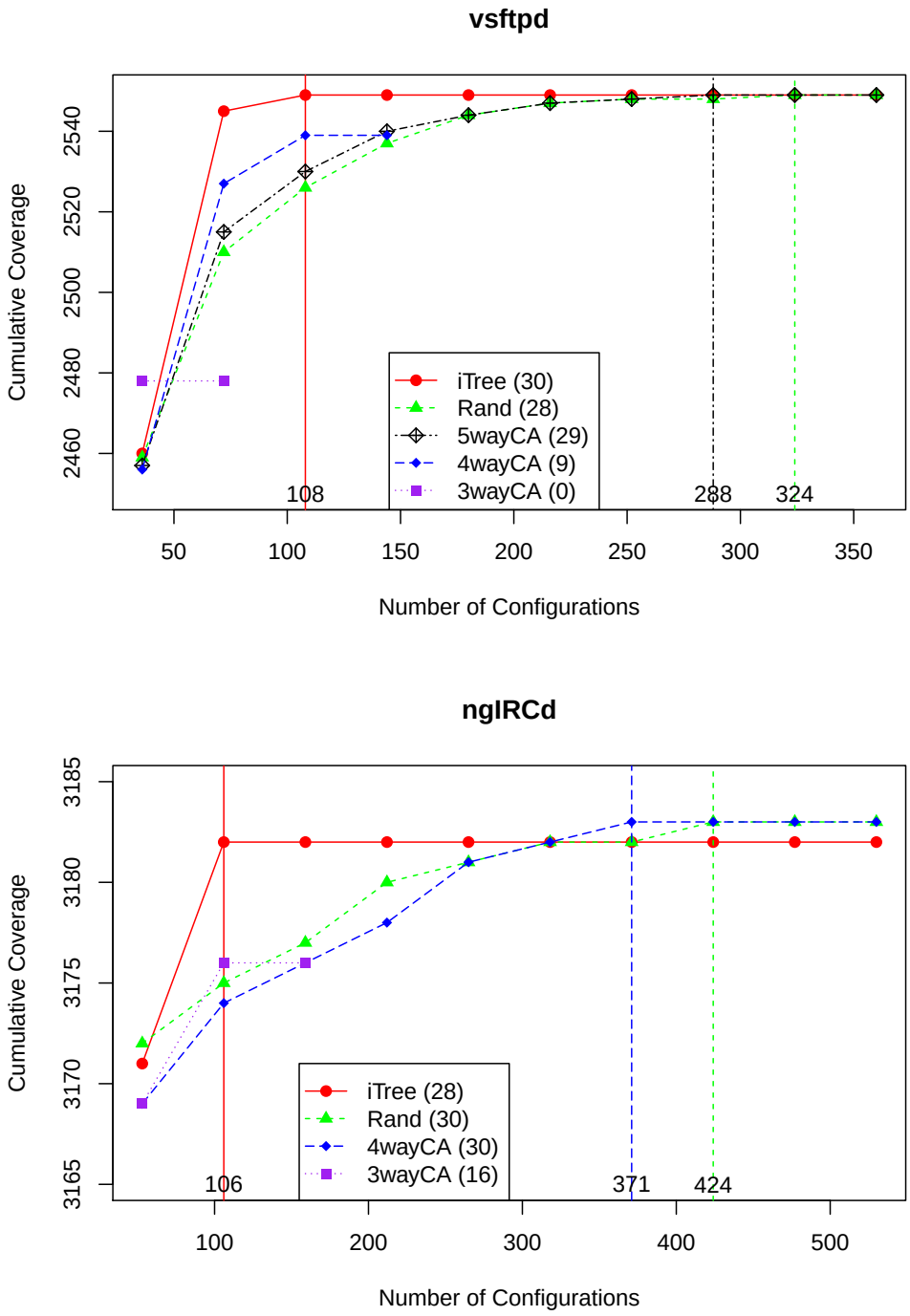


Figure 4.8: Comparing the number of configurations needed to reach complete coverage using iTree versus using covering arrays and random sampling.

Therefore, the iTree runs can terminate before executing the number of configurations of the other approaches. 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.

We see that for vsftpd, iTree, 5-way covering arrays, and random sampling eventually reached full coverage in almost all runs (30, 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 almost a third of 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, for which iTree, 4-way covering arrays, and random sampling achieved full coverage in all or almost all runs (28, 30, and 30, respectively), but 3-way covering arrays only reached full coverage in just over half the runs (16). Again, 90% of the iTree runs reached full with just a fraction of the number of configurations of 4-way covering arrays, which reached full coverage faster than random sampling. We note, because 2 of the iTree runs did not reach full coverage when it was terminated by the default stopping criteria, the median number of lines executed by these runs was 1 shy of the full coverage.

Overall, these results showed the iTree performing better than t-way covering arrays and random sampling, at substantially lower 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 the previous chapter, 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 7-way and 6-way covering arrays with thousands of configurations to guarantee complete line coverage for vsftpd and ngIRCd respectively. If developers instead used a low-strength, 2-way covering array, the cost would be much lower, but so would the coverage.

4.3.2 Scalability Evaluation

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 LOC as computed by sloccount [78]. It is written in a combination of C and C++, and its configuration space includes a large number of run-time as

	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 4.9: Program statistics of MySQL.

well as compile-time configuration options. As in our experiments in the previous section, our evaluation compare iTree against covering arrays and randomly sampled configurations.

4.3.2.1 Subject System

Figure 4.9 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 [78]. 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 4.9 lists the number of unique configurations that can be generated given the number of distinct settings of the configuration options; 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's source tree, which we used for our experiment. We should note that not every test case runs in every configuration.

4.3.2.2 Experimental Design

Our experimental design is similar to that of Section 4.3. 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 again used the vote_adapt approach. One key difference between this experiment and the last is that we cannot know the maximal possible coverage achievable by MySQL's test suite, and so we only discuss observed line coverage.

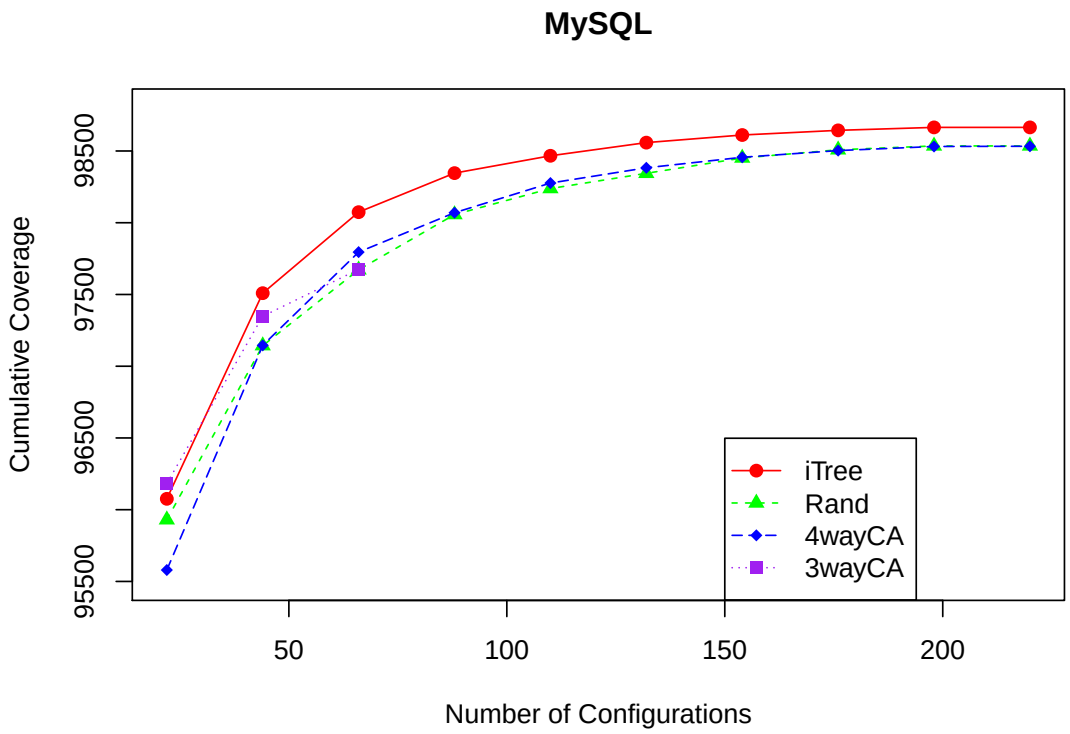


Figure 4.10: Comparing the number of configurations and coverage achieved using iTree against those achieved using other approaches.

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 for each configured instance. 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 [31]; 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.

4.3.2.3 Data and Analysis

Figure 4.10 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 so far. 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. 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 that included configurations with unique coverage patterns that were not found by the more traditional approaches.

The absolute difference in line coverage ranges from a high of around 0.5% ($\sim$ 500 LOC) early on down to about 0.1% ($\sim$ 132 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. We observed that the extra lines covered with iTree involved many small pockets of code scattered across numerous files, methods, and code 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 and executed its configurations, we 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.

The MySQL experiments once again showed that the iTree approach performs better than t-way covering arrays and random sampling. But more importantly, these experiments showed that iTree can handle large scale industry systems implemented in heterogeneous programming languages with configuration spaces that include both compile- and run-time configuration options.

4.4 Minimized iTree Sets

Next, we want to investigate whether we can generate even smaller configuration samples using the knowledge gained from the iTree's configuration space exploration process.

In Chapter 3, we developed a greedy algorithm that takes the configuration interactions of a system and packs them together to form a small set of configurations that can still achieve full coverage. We called these configuration sets the *minimal covering set*. In this algorithm, we greedily select an interaction that executes the most uncovered code at each iteration and pack it together with a consistent interaction already selected by a previous iteration. This process generates a small set of complete configurations, which assign values to all options, that can achieve high coverage.

On a high level, the iTree discovery algorithm is performing a similar algorithm to pack together configuration interactions. During each iteration of iTree, a high coverage proto-interaction is selected for further exploration. Configuration samples are generated by iTree to sample other combinations of option settings that can be

merged with this proto-interaction to achieve higher coverage. If some sampled configurations execute previously uncovered code, then new higher strength proto-interactions are generated for future iterations to explore in the same way. In this way, the process continuously packs together effective interactions to form higher strength ones until full coverage is reached or no more proto-interactions are left to explore.

We believe that a small configuration set, similar to the *minimal covering set*, can be generated by selecting a subset of the iTree execution. We call this set the *minimized iTree set*. To generate the minimized iTree set, we developed another greedy algorithm: Starting with no lines of code covered, the algorithm iteratively chooses the next configuration from the iTree execution data that covers the most currently uncovered lines and adds it to the set. The algorithm continues until every line of code executed by the iTree run is covered.

4.4.1 Data and Analysis

We performed the greedy algorithm on the vote_adapt iTree runs. For vsftpd, the minimized iTree sets contained between 5-8 configurations; very close to the size of the minimal covering set for line coverage (5 configurations). For ngIRCd, the minimized iTree sets contained between 5-6 configurations; interestingly, the minimized iTree sets are actually smaller than ngIRCd's minimal covering set (7 configurations). We think this is because the iTree's sampling approach makes many more attempts to merge potential interactions together, and thus it can be

Config #	1	2	3	4	5	6	7	8
vsftpd								
Mini iTree Sets (avg)	2,455	71	12	6	1	1	<1	<1
Mini Cov Set	2,521	18	8	1	1	-	-	-
ngIRCd								
Mini iTree Sets (avg)	3,135	30	9	6	1	<1	-	-
Mini Cov Set	3,148	30	6	6	1	1	1	-

Figure 4.11: Comparing additional coverage achieved by each configuration in the minimized iTree sets against those in the minimal covering sets.

more effective than the minimal covering set greedy algorithm.

Figure 4.11 compares the average additional coverage achieved by each configuration in the minimized iTree sets to those in the minimal covering sets. As in Figure 3.12, the column labeled 1 shows how many lines are covered by the first configuration in the configuration set. Then column n (for $n > 1$) shows the additional coverage achieved by the nth configuration over configurations $1..(n-1)$. We can see that both minimized iTree sets and minimal covering sets follow the same general trend, with most coverage achieved by just the first configuration and the last several configurations often add only one additional line.

We also used this algorithm to generate the minimized iTree sets for MySQL's vote_adapt runs. These configuration sets contained between 41-65 (on average 60) configurations. Figure 4.12 shows the cumulative coverage (averaged over all sets) achieved by each additional configuration. We see that, for MySQL, almost all of

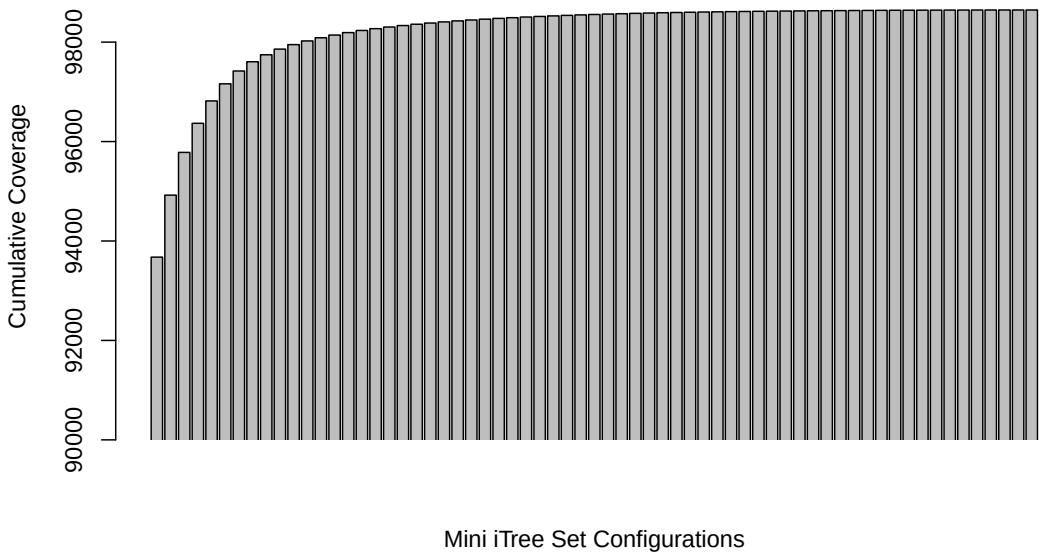


Figure 4.12: Cumulative coverage of MySQL's minimized iTree set.

the achieved coverage was covered by just the first configuration and the next few configurations added some more coverage. But the rest of the configurations in the set added very little additional coverage, in fact, the last 35 configurations added less than 10 lines each. This confirms that, MySQL's effective configuration space is also union of disjoint interactions. Its configuration options do not interact fully and many of the higher strength interactions subsumes the lower strength ones.

4.5 Estimating Configuration Interactions

So far we used the iTree discovery algorithm to efficiently discover high coverage configurations for software testing. Now we investigate whether iTree can be used to estimate a software system's configuration interactions. Knowing the in-

teractions of a system enables many applications such as configuration selection for regression testing and tool support for program understanding tasks.

4.5.1 Interaction Learning Approach

Even though the iTree algorithm already generates *proto-interactions* during it's discovery process, however, we cannot extract these proto-interactions directly from the interaction tree as full-fledged configuration interactions. As mentioned above, the proto-interactions do not always represent actual interactions because of two main reasons:

1. The proto-interactions are generated using only the sampled configurations available during each iteration of an iTree run.

2. The heuristic nature of iTree causes any inaccuracies to be propagated to the higher strength proto-interactions.

Instead, we developed an approach, we call *interaction learning*, that uses machine learning algorithms to "learn", from a full set of sampled configurations from an iTree run, the interactions required to execute any part of a system.

To illustrate the operations of this approach, we describe the process of learning the interactions of a system's line coverage. For each line of code, we create both positive and negative learning examples using the sampled configurations; each configuration option is an attribute and the class is either hit (line of code was executed under this configuration) or not hit (line of code was not executed under this configuration). In our implementation, we use decision trees to build models for classifying

whether a configuration can execute the line of code based on its option settings. The resulting classification models identify specific option settings that predict the coverage of the specified line of code.

There is an obvious optimization that can speed up this interaction learning process. As we have seen in the analysis data from Chapter 3, numerous lines of code can have the same configuration interactions. The lines with the same interactions would be executed by the same subset of sampled configurations as well, and this means their interactions can be classified by the same models. Therefore, we do not need to generate a classification model for each line, instead we can group together the lines executed by the same configurations and learn the interactions for each group.

The use of decision trees in the interaction learning approach could cause it to run into the *imbalance data problem* [38]. For example, lines of code with hard to cover, high strength interactions can have number of positive examples that are dwarfed by the number of negative examples. In such cases, the decision tree algorithms designed to meet a set error rate would simply generate classification models that only predict negative outcomes. The reverse can be true with overwhelming positive examples as well.

To deal with the imbalance data problem we used a technique called boosting [27, 60] to balance the examples by adding a weight value for each instance, the higher the weight the more an instance influences the decision tree model. In our implementation, the weight for each instance is calculated as follows. First, let $|negative|$ be the number of negative instances and $|positive|$ be the number of pos-

itive instances in the learning examples. Then the weight of each instance is given by:

$$weight_{negative} = \begin{cases} 1, & \text{if } |negative| >= |positive|\,; \\ \frac{|positive|}{|negative|}, & \text{if } |negative| < |positive|\,. \end{cases}$$

$$weight_{positive} = \begin{cases} 1, & \text{if } |negative| <= |positive|\,; \\ \frac{|negative|}{|positive|}, & \text{if } |negative| > |positive|\,. \end{cases}$$

This way we always ensure that the total weight of positive instances is roughly equal to the total weight of the negative instances.

4.5.1.1 Experimental Design

To evaluate the effectiveness of the interaction learning approach, we used vsftpd and ngIRCd since we can compare the estimated interactions against the configuration interactions calculated using symbolic evaluation runs. First, we wanted to see if the configuration set discovered by iTree using our adaptive heuristics can be used to produce accurate interaction estimations. We ran the interaction learning process on the 30 vote_adapt runs terminated by the default stopping criteria. These runs, which ran until there are no more proto-interactions in the interaction tree, on average analyzed 255 and 345 configurations for vsftpd and ngIRCd respectively. We experimented with using either C4.5 or CART decision trees to "learn" the interactions.

Next, we wanted to see if iTree runs using a slightly different set of heuristics

that generate bigger configuration sets would produce more accurate estimations. For these runs, we used the same aggregation classifier and default stopping criteria as the vote_adapt runs, but we generated a 3-way covering array for every iTree iteration. The resulting vote_t3 runs analyzed on average 375 and 672 configurations each for vsftpd and ngIRCd respectively; more configurations than the vote_adapt runs. For these experiments we also used either C4.5 or CART decision trees for the learning process.

We defined three metrics for evaluating the accuracy of the estimated configuration interactions:

- *Exact matches*, measures the number of lines which had estimated interactions that matched the actual interactions exactly. We note that for lines with multiple interactions, the estimations must match all interactions exactly.

- *Reachables*, measures the number of lines that did not have the exact estimated interactions, but the estimations still guaranteed the execution of these lines. For instance, at least one of the estimated interactions matched the actual interactions and/or the estimated interactions added additional option settings.

- *Unreachables*, measures the number of lines which had estimated interactions that cannot guarantee the execution of these lines. For instance, an estimated interaction that only included 4 out of the 6 specific option settings of the actual interaction.

	vote_adapt		vote_t3	
	C4.5	CART	C4.5	CART
Exact Matches	2426(95.17%)	2427(95.21%)	2425(95.14%)	2426(95.17%)
Reachables	118 (4.63%)	118 (4.63%)	118 (4.63%)	119 (4.67%)
Unreachables	4 (0.16%)	2 (0.08%)	4 (0.16%)	5 (0.20%)

(a) vsftpd

	vote_adapt		vote_t3	
	C4.5	CART	C4.5	CART
Exact Matches	3052(95.58%)	3007(94.17%)	3082(96.52%)	2980(93.33%)
Reachables	108 (3.38%)	152 (4.76%)	83 (2.60%)	180 (5.64%)
Unreachables	22 (0.69%)	22 (0.69%)	17 (0.53%)	22 (0.69%)

(b) ngIRCd

Figure 4.13: Accuracy of vsftpd and ngIRCd's estimated configuration interactions measured in three metrics.

4.5.1.2 Data and Analysis

We ran the interaction estimation experiments as described above. Figure 4.13 shows the three metrics, averaged across all 30 runs, for each experiment. The tables show the metrics, both the number of lines and its percentage out of the 2549 and 3193 achievable line coverage for vsftpd and ngIRCd respectively. First, we see that, for vsftpd and ngIRCd, the interaction learning approach was quite successful, regardless of the decision tree algorithm used. For these subject systems,

this approach generated the exact configuration interactions for about 95% of the source code. About 5% of the source code is still reachable using the estimated interactions and only less than 1% of the source code cannot be guaranteed to be executed using the interaction estimations.

Next, we see that the accuracy of the estimated interactions is almost the same using the different iTree heuristics. But we do notice a slight advantage for the vote_t3 runs; for vsftpd, a few more runs had 0 unreachable lines of code and for ngIRCd, on average fewer lines were unreachable using the estimated interactions. But all in all, the vote_adapt runs, despite having many fewer configurations in the configuration sets, estimated the configuration interactions with about the same level of accuracy as the the vote_t3 runs. This means that our adaptive heuristics is practical for both software testing and configuration space analysis tasks.

We examined the estimated interactions in more detail. We found out that, for the reachable lines, the extra option settings were mostly byproducts of the decision tree classification models. For instance, numerous lines of code from vsftpd had three 1-way interactions of tunable_local_enable, ¬tunable_anonymous_enable, and tunable_ssl_enable. But the estimated interactions included:

- tunable_local_enable
- ¬tunable_local_enable ∧ ¬tunable_anonymous_enable
- ¬tunable_local_enable ∧ tunable_anonymous_enable ∧ tunable_ssl_enable.

The extra option settings were used to branch on the different classifications in the decision tree models.

We also found out that some of the lines are unreachable because were not executed by the iTree runs. But for most of the unreachable lines, the estimated interactions included some but not all option settings of the actual interactions.For instance, a few of ngIRCd's lines required the 6-way interaction $\mathsf{Conf\_ListenIPv4} = 1 \wedge \mathsf{Conf\_MaxNickLength} = 8 \wedge \mathsf{Conf\_PredefChannelsOnly} = 1 \wedge \mathsf{Conf\_PongTimeout} = 20 \wedge \mathsf{Conf\_PingTimeout} = 3600 \wedge \mathsf{Conf\_MaxConnectionsIP} = 2$ and the estimated interaction was a 4-way interaction containing 4 of the 6 specific option settings.

Comparing to the techniques we developed in Chapter 3 to analyze the configuration interactions, the configuration interaction learning approach's speed improvement is incredible. For vsftpd and ngIRCd, it took several days of runtime on 40 machines to complete the *guaranteed coverage* analysis and a few hours on one machine to complete the *execution conditions* analysis. But the interaction learning process takes just a few seconds on one machine to analyze one iTree run, with close to perfect accuracy.

We also note that the optimization we implemented by grouping together lines of code executed by the same configurations was quiet effective as well. Of the 2549 and 3193 lines of code for vsftpd and ngIRCd respectively, there were only 52 and 61 distinct groups on average that the approach needed to build classification models for.

Configuration Interactions	C4.5	CART
Strength 1	17	47
Strength 2	126	149
Strength 3	477	333
Strength 4	970	364
Strength 5	1294	369
Strength 6	1234	259
Strength 7	429	143
Strength 8	60	57
Strength 9	1	11
Strength 10	0	3
Total	4608	1735
LOC Covered	98759(100%)	97981(99.21%)

Figure 4.14: Number and strength of MySQL's estimated configuration interactions.

4.5.2 Analyzing Configuration Interactions of MySQL

To estimate the configuration interactions of MySQL, we applied the configuration interaction learning approach on one of the vote_adapt iTree runs that achieved the highest coverage. Of the 98759 lines of code this iTree run executed, only 1616 groups needed to be classified. Figure 4.14 summarizes the estimated configuration interactions using either C4.5 or CART decision trees.

First, we see that the number of interactions are quite different between these two classifiers. C4.5 estimated many more interactions than CART and these inter-

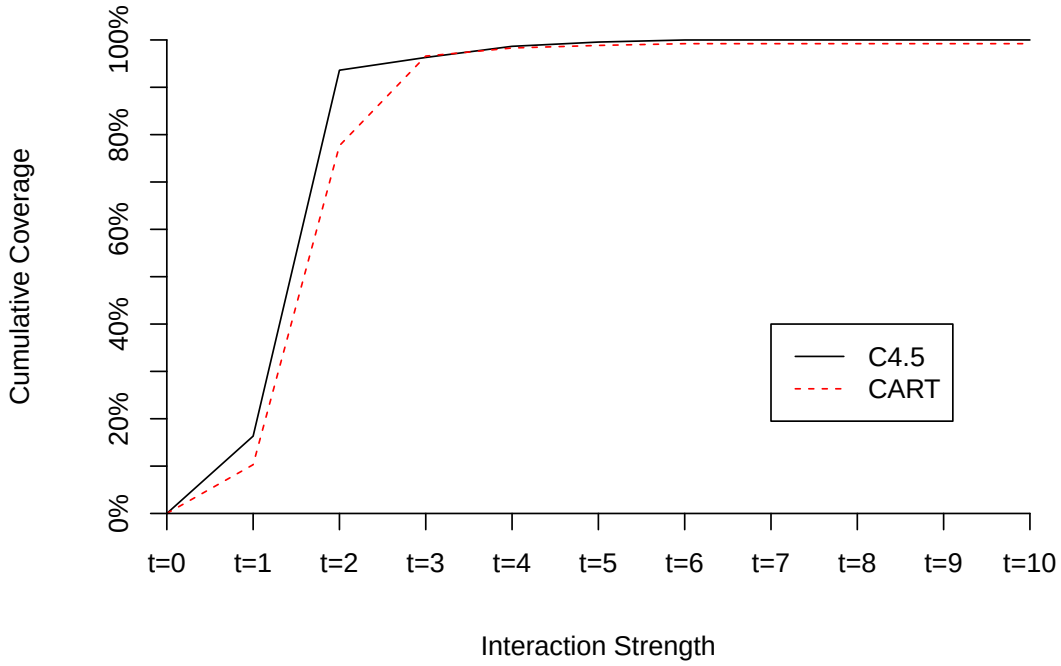


Figure 4.15: MySQL's cumulative guaranteed coverage at each interaction strength.

actions accounted for 100% of vote_adapt run's coverage. But, even though CART estimated $\sim 62\%$ fewer interactions, the coverage accounted for by these interactions is still over 99%. This shows that many of the missing interactions do not execute many lines of code. With further analysis, we determined that using the CART decision tree, there were 77 groups that could not be classified, while all groups were classified by C4.5. These results suggest that C4.5 decision tree produces more complete and accurate configuration interaction estimations for MySQL.

Next, we see that the number of interactions at each t strength, while more than vsftpd and ngIRCd, is still very small compare to MySQL's full configuration space and theoretically possible configuration interactions. This observation

is consistent with our conjecture that the effective configuration spaces are quite small.

In Figure 4.15, we plotted the cumulative coverage that can be achieved at each interaction strength. In terms of line coverage, plots for both classifiers looked similar. We clearly see the effects of *enabling options* at $t = 2$. Examining the data, we found that the interactions which contributed the most coverage at this strength involved various combinations of settings for two options: default_storage_engine and event_scheduler.

The configuration interaction learning approach handled this much larger subject system with ease. Even with ∼1M LOC, this approach took around 30 minutes to complete on a single machine. And we note that, a major portion of the analysis time was spent on extracting the line coverage information from the zipped log files created by the Skoll execution framework.

4.6 Summary

In this chapter, we developed a new sampling approach, the *interaction tree discovery algorithm* or iTree, that leverages the structures of the effective configuration spaces to discover highly effective configurations. This approach uses the *interaction tree* to divide up a software system's configuration space and iteratively searches for high coverage configurations. In this way, the configuration sets constructed as iTree executes have the potential to achieve high coverage during testing.

We conducted several experiments designed to evaluate the performance of

the iTree algorithm and to improve the heuristics that guide its iterative search process. Using the symbolic evaluation results for vsftpd and ngIRCd, we measured iTree's performance under various combinations of ML algorithms, covering array t strengths and other parameters. In each case, our experimental results showed that this new approach can efficiently achieve full coverage during testing. And we found that the best choice for iTree is to use a voting protocol to combine multiple ML classifiers, and to use an adaptive sampling approach that generates more configuration samples as needed by the classifiers during each iteration.

We then empirically evaluated iTree by comparing it against traditional CIT and similarly sized sets of randomly selected configurations. For vsftpd and ngIRCd, our results show that iTree is more likely to find high coverage configuration sets, and it does so more rapidly than the existing sampling approaches. In another 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 correspondingly sized configuration sets produced by either traditional CIT or random sampling.

We next used a greedy algorithm to select from an iTree run, a small configuration set, the *minimized iTree set*, that still achieves all of the coverage of the full iTree run. For all our subject systems, these sets are quite small and the first configuration can achieve most of the maximum coverage. This confirms that the effective configuration spaces looks more like a union of disjoint interactions rather than a monolithic cross-product of all configuration option settings.

Finally, We developed the *interaction learning* approach to estimate a software system's configuration interactions. This approach uses decision tree classifiers and iTree's execution results, to quickly and accurately estimate the interactions. For our subject systems, vsftpd and ngIRCd, it correctly estimated the interactions for over 95% of their source code. The estimated interactions of these systems can guarantee the execution of over 99% of their source code. This approach is also very light-weight. For vsftpd and ngIRCd, it took just seconds to analyze one iTree run. We then used the interaction learning approach to estimate MySQL's configuration interactions; sometime that's not possible using previous tools and techniques. Even with ∼1M LOC, our approach only took around 30 minutes to complete the analysis on one machine. The estimated configuration interactions of MySQL are consistent with our conjectures about the effective configuration spaces.

Chapter 5

Leveraging the Effective Configuration Space

In the Chapter 4, we took another major step toward our ultimate goal of leveraging effective configuration spaces to improve software engineering for configurable systems. We developed the iTree algorithm to efficiently discover highly effective configurations to test and analyze. In addition, we developed an *interaction learning* approach that can accurately estimate configuration interactions using the execution data of an iTree run. In this chapter, we look at our third research hypothesis: We can greatly improved numerous software engineering tasks by leveraging a system's effective configuration space. Specifically, we want to develop tools and techniques to improve the testing of highly configurable systems. We can formulate these objectives in the following two research questions:

1. Can the iTree algorithm work with today's development processes to effectively ensure quality of software systems?

2. Can the knowledge of effective configuration space be used to dramatically improve software testing tasks?

To address the first question, we developed an iTree-based automated distributed testing framework to address the needs of ever shorter development cycles. We explain how iTree is designed to ease the integration with existing quality assurance (QA) processes and infrastructures. To demonstrate, we integrated iTree with

Skoll [55], a distributed continuous quality assurance (DCQA) process and framework that enables the parallelization of configuration-aware software testing. We also discuss our implementation choices and lessons learned while developing this testing framework for MySQL, a modern open source database system.

To address the second question, we performed an extensive study on regression testing of configurable software. Using the study results, we then developed a technique that can select very small sets of highly productive configurations that target the modified code during regression testing. Our evaluation shows that, by leveraging the configuration interactions calculated or estimated from previous testing sessions, we greatly reduced the time and cost of the configuration-aware regression testing.

This chapter is organized as follows. Section 5.1 details the design and implementation of the iTree-based automated distributed testing and analysis framework. Section 5.2 presents our study on regression testing of configurable software and the technique we developed to improve regression testing.

5.1 iTree-based Automated Distributed Framework

Today's software development processes are moving towards more iterative and incremental cycles to encourage rapid responses to changes and uncertainties [3, 66]. This new trend presents many challenges to developers, including the challenge of adequately performing QA tasks with increasingly shorter development cycles. In addition, QA processes themselves require ever more sophisticated and flexible

control mechanisms to meet the QA goals of today's complex and rapidly changing systems. These challenges are especially great for configurable software systems because of the explosion of their QA task spaces; these systems often run on multiple hardware and OS platforms and have many options to configure the systems at compile- and run-time.

iTree was designed to tackle the challenges of testing configurable systems; it reduces the otherwise infeasible configuration spaces by selecting only the effective configurations to test. However, even with iTree's promising performance, the QA task spaces are still magnified by the numerous configured instances that the existing QA processes must handle. Fortunately, with ever cheaper commodity computing resources and the advent of cloud computing [2], there should be plenty of cost-effective CPU cycles that can be directed towards configuration-aware QA activities. But, coordinating these CPU cycles, which might be distributed across different physical locations and operating environments, brings about a new set of challenges. For example, a configuration-aware QA space must be well understood and modeled in order to be divided to run in parallel.

Encouraged by iTree's performance, we decided to create an iTree-based, automated distributed testing framework to tackle these challenges. To avoid reinventing the wheel, iTree was designed to be loosely coupled with the underlying QA processes and can easily plug into existing test frameworks. To demonstrate, we integrated iTree with Skoll [55], a distributed continuous quality assurance (DCQA) process and framework developed and housed at University of Maryland. Figure 5.1 show the architecture diagram of this integrated framework. In this framework, iTree in-

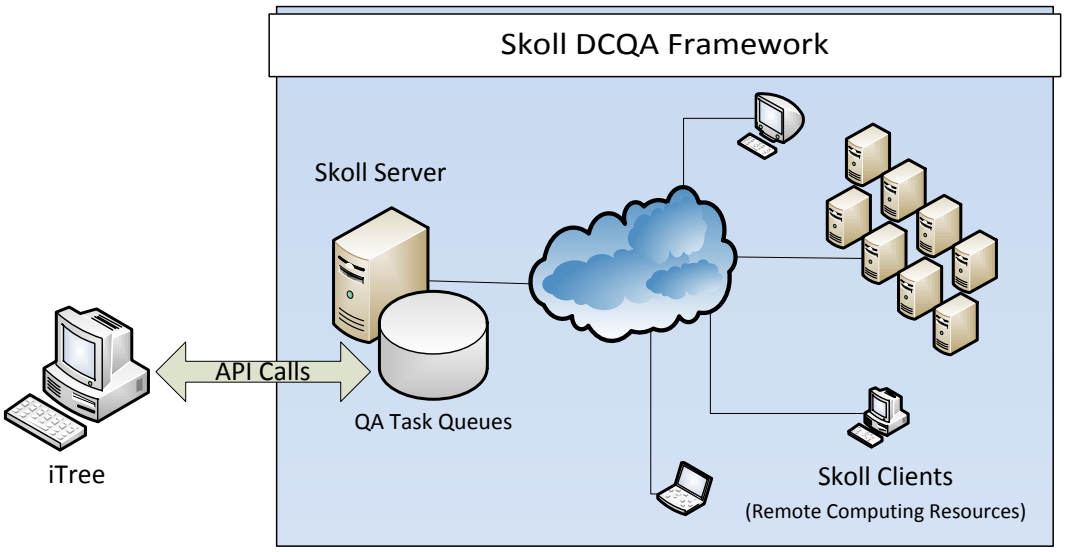


Figure 5.1: The automated distributed testing framework created by integrating iTree with the Skoll DCQA framework.

telligently assigns the configurations that need to be analysis as Skoll QA tasks, and the Skoll framework distributes these QA tasks to available computing resources for massive parallel execution.

5.1.1 Skoll Overview

The Skoll DCQA process was developed to coordinate and control feedback-driven testing for configurable software systems. Specifically, it was designed to leverage distributed computing resources in a continuous manner to significantly and rapidly verify configurable software quality. The Skoll framework is implemented using a client/server architecture, in which clients request QA tasks from a server that is responsible for planning and coordinating the testing process.

The operation of the Skoll DCQA process involves numerous decisions such as, what are the sub-tasks needed to complete the QA tasks (e.g., compiling the source code and running the test suite), which artifacts are required by the QA sub-tasks (e.g., a specific version of source tree), which available clients have the required environments to test configuration specified by the QA tasks (e.g., Linux environment with GCC compiler), and how the test results will be collected and interpreted (e.g., collect gcov coverage data and test results). To help developers implement these decisions, Skoll provides flexible models for the system under test and its QA processes, including a configuration space model that supports both compile-time and run-time configuration options and settings.

The Skoll server is where the developers implement the configuration space and process control models, and it is the central controller of the entire DCQA process. To do this, the server has an Intelligent Steering Agent (ISA) that maintains the progress of the QA process (already analyzed configurations and test results). Based on the QA progress, the ISA uses various *adaptation strategies*, such as nearest neighbor or adaptive sampling, to selectively choose which configurations to test next. The server then generates the QA tasks for the selected configurations and bundles up all necessary artifacts and scripts according to the developer-specified models.

The Skoll clients are remote computing resources that elected to participate in a virtual computing grid dedicated to the Skoll DCQA processes. When a client decide it is available to perform QA activities, it requests for QA tasks from the server. At each request, the client sends to the server information that describes

```
1   interface executeConfigSet {
2       boolean configResultExists(Configuration id)
3       boolean isConfigScheduled(Configuration id)
4       boolean scheduleConfig(Configuration id)
5       ExecutionData getConfigResult(Configuration id)
6   }
```

Figure 5.2: iTree's API for implementing adapters for different execution frameworks.

its environment, including its OS, hardware specifications, compiler versions, etc. The server then uses this information to assign a QA task that the client has the appropriate environment for its successful execution. Once the execution of a QA task is completed, the client then sends the test results back to the server where the ISA analyzes the data and updates the progress of the QA process.

5.1.2 iTree Integration With Skoll

iTree's executeConfigSet() is actually an Application Programing Interface (API). We designed this API to allow developers to provide adapter implementations that can tap into various execution frameworks.

Figure 5.2 shows the interface needed to integrate iTree with different execution frameworks. The API is designed to control the execution of individual configurations. The configResultExists() and isConfigScheduled() methods check if the specified configuration has been executed or is already scheduled for execution by the underlying framework, respectively. The scheduleConfig() method schedules the specified configuration for execution, and the getConfigResult() method retrieves the execution results of the specified configuration.

In our iTree-based testing framework, the iTree algorithm essentially replaces one of Skoll's existing adaptation strategies implemented in the ISA. ISA offers an API to add new strategies for configuration selection. However, to make iTree decoupled from Skoll, we chose to interface with Skoll in a more generic fashion.

During our previous efforts to improve the flexibility of the Skoll DCQA framework, we made several modifications that proved to have simplified the iTree integration process. These improvements include:

1. An API that allows direct manipulation of Skoll server's QA task queues (scheduled and running tasks). These queues are implemented as MySQL tables and the API is implemented using MySQL stored procedures.

2. A standalone tool and a Java library that simplifies the scheduling of Skoll QA tasks via the API. This tool checks the QA tasks against the modeled configuration space and QA process for potential errors.

These improvements allowed us to implement the first three methods of executeConfigSet() using the adapter design pattern.

Since Skoll is designed to support a wide range of software systems and QA processes, it requires customization before it can be used with a new software system and QA process. Given a QA task, the first step for configuring Skoll is to create configuration and QA process models that specify how the QA tasks are divided into several sub-tasks. These models are also implemented as MySQL tables.

iTree's configuration space model, which is an one-to-many mapping of configuration options to their settings, can directly map onto the Skoll configuration

space model. The QA process model, on the other hand, is system specific. For MySQL, we designed the Skoll QA task to download a specified version of source code from MySQL's Bazaar version control system; compile an instance according to the compile-time option settings for the configuration selected for testing; instrument the instance with the gcov [31] profiling tool; run the regression test suite provided in by the MySQL source tree; and collect the execution data and upload it to a location where iTree can access. The last step allowed us to implement the getConfigResult() method in the iTree's API.

Once integrated, this iTree-based Skoll framework can easily take advantage of a research cluster composed of over 120 CPUs dedicated entirely to our research work. The Skoll's client/server architecture also allows us to easily scale up this execution framework with various computing resources (e.g., Amazon EC2), as long as the environment allows the Skoll Client software (implemented in Java and Perl) to be installed.

5.1.3 Discussion

The goal of this iTree-based framework is to parallelize the testing and analysis of configurations. However, there are practical limitations on how much parallelization can be achieved using the iTree algorithm.

The iTree approach uses an iterative search algorithm, in which the test results from the previous iteration must analyzed entirely before the next iteration can begin. In our current implementation, a Skoll QA task involves compiling and

testing a single configuration. This means that adding more CPUs than the number of configurations sampled by iTree during an iteration will not speed up the testing process; each iteration completes only after the configuration that takes the longest time to test and analyze completes.

For MySQL, using similar hardware and software environments, the difference in runtime between the fastest and slowest configurations can be more than an hour. This is because the the slowest configurations usually have failing test cases that cause the MySQL server to restart during the testing phase. However, this limitation can be mitigated by designing the Skoll QA tasks to be more divisible. For instance, we can allow individual test cases to be executed on different CPUs for better parallelization.

There are ways to conserve CPU cycles as well. For instance, our current implementation treats different combinations of compile-time and run-time option settings as unique configurations. But some of these unique configurations can share the same compiled instances for the testing phase. If we design the Skoll QA tasks to share compiled instances, then savings can be gained. For MySQL, we analyzed the configuration sets discovered by the vote_adapt iTree runs. On average there are 131 unique compiled instances in a set of 190 configurations, which is about 31% savings in compile time.

Overall these optimizations depend on the underlying QA processes and the execution frameworks, not the iTree approach itself. Thus, we will leave the analysis and development of these optimization techniques as future work.

5.2 Configuration-Aware Regression Testing

Thus far our efforts have been focusing on testing an entire system when there is no prior knowledge about its configuration space. However, as software development moves toward more iterative cycles, performing this type of testing every time the system is modified is unnecessary and can be prohibitively expensive. In this section, we look to address regression testing for configurable systems.

Regression testing is one of the necessary but expensive maintenance activities – the process of validating modified software systems to provide confidence that the modifications are correct and to detect whether any errors were introduced into the previously tested code. Regression testing, which should be performed each time a system is modified, accounts for as much as one half of the cost of software maintenance [44], which is itself a major portion of the overall cost of software production. The impact of configurability can significantly magnify the cost of this task.

To date, most regression testing research has primarily focused on techniques for regression test case selection or test case prioritization [63, 73, 48, 56]. Much less attention, however, has specifically been paid to regression testing for highly configurable systems. In the one example we are aware of, Qu et al. [57] studied whether CIT could effectively support regression testing on a single configurable system, the vim text editor. In that work, Qu et al. used 2-way covering arrays to regression test the modified system and measured how block coverage and fault detection effectiveness varied across system configurations for a given test suite and a set of program

changes. Their basic findings were that individual program changes affected different configurations differently, and that, therefore, systematically covering system configurations was an effective heuristic for regression testing.

Intrigued by Qu et al.'s work, we decided to conduct a series of follow-on empirical investigations on configuration-aware regression testing. We believe that the knowledge of effective configuration spaces may provide valuable information for selecting the configuration / test case pairs that should be executed during regression testing.

5.2.1 Regression Testing Analysis

Qu et al. examined 4 general research questions:

1. Do program changes affect different configurations differently?
2. Does test case selection depend on configuration?
3. Is covering array sampling more cost-effective than random sampling?
4. Can the historical behavior of individual configurations be used to prioritize configurations to find faults more quickly?

To address the first two questions, Qu et al. measured how block coverage and fault detection effectiveness varied across system configurations for a given test suite and set of program changes. They observed considerable variation. For example, some faults were detectable in every configuration while other faults were detectable only in some configurations. Similarly, individual test cases had differing

fault detection abilities under different configurations. These findings suggest the potential to improve regression testing cost-benefits because many configuration / test case pairs cannot detect faults and therefore need not be executed.

To address the third question they examined whether CIT approaches improved cost-effectiveness compared to testing with randomly sampled configurations. Specifically, they compared the fault detection ability of a 2-way covering array to that of an equal number of randomly-selected configurations. Here they found only a small difference between the two approaches, with CIT samples showing higher average fault detection effectiveness. We note, however, that CIT approaches provide a constructive method for choosing the size of the configuration sample, whereas random selection, in practice, would depend on developer intuition.

Finally, to address the fourth question they examined whether CIT techniques could be used to effectively prioritize the order in which configurations are tested, thus finding more failures with limited resources or detecting bugs earlier in the testing process. They used two CIT-based techniques: pure prioritization and regeneration. For both techniques, they used a covering array to test an initial version of the system and then calculated an *interaction benefit* for each configuration option setting, based on the coverage or the fault detection ability observed in the initial testing. They then used these interaction benefits to guide the testing of a subsequent system version. With the pure prioritization approach, they assigned an execution order to the covering array configurations based on the interaction benefits of each configuration's option settings. With the regeneration approach, they constructed a new covering array for the current testing session by selecting

configurations that constituted a covering array, but that also had configuration option settings with the highest interaction benefits. Their results showed that both techniques detected faults earlier than the unordered configurations, and the regeneration technique gave the best results. That is, historical information could be used to learn which configurations had higher fault detection ability over time and that executing higher fault detection configurations before lower fault detection ones led to earlier fault detection.

5.2.1.1 Replicating Qu et al.

Our first set of studies partially replicates Qu et al.'s earlier studies on configuration-aware regression testing using covering arrays. For these studies we focused on our subject system vsftpd. We acquired two consecutive versions of vsftpd (2.0.6 and 2.0.7), and determined that 12 lines of code were both modified between the two versions and reachable by our test suites. We treat these 12 lines of code as the complete set of modifications that need to be regression tested in the later version of vsftpd.

We generated multiple samples of the configuration space, which define the concrete configurations that are executed during a regression testing session. Again, we used two sampling approaches: covering arrays and random sampling. For covering array sampling, we used the CASA [30] to generate 30 sets of 2- and 3-way covering arrays at each t strength, based on our configuration space model defined for vsftpd. The covering arrays we generated had 10–12 configurations for the 2-way

samples and 32–35 configurations for the 3-way samples. For each covering array we also generated an equally sized randomly sampled set of configurations.

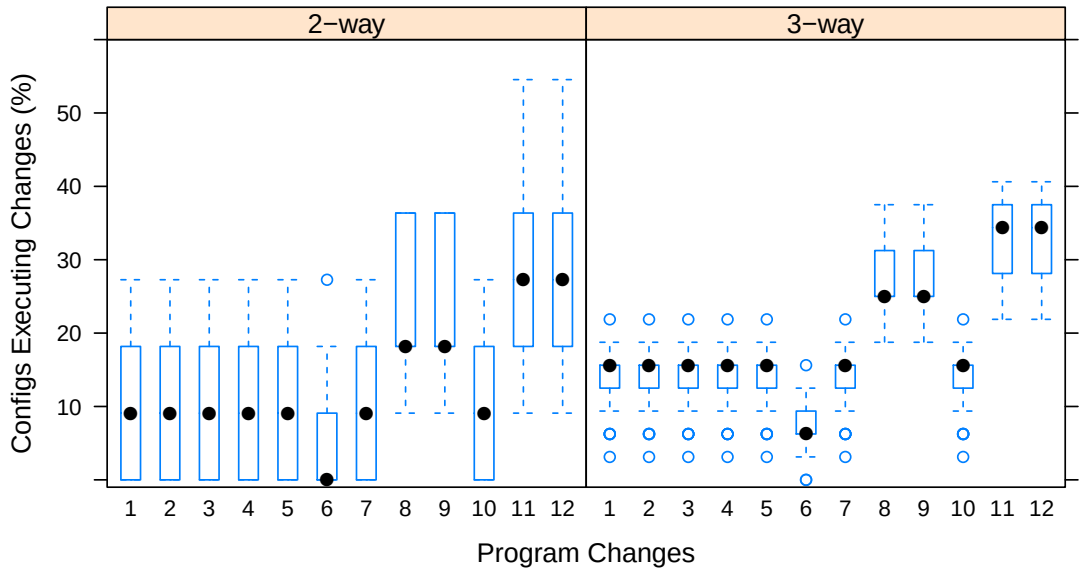
Using vsftpd, its test suite, and the 2- and 3-way covering arrays we generated, we then performed several new studies, aimed at revisiting Qu et al.'s four research questions.

RQ1: Effect of Configuration on Regression Testing To address this first question, we measured the number of configurations in each covering array that executed each change. To do this we executed vsftpd on each of the 30 sets of the 2-way and the 30 sets of the 3-way covering arrays. Figure 5.3(a) depicts some of our results.

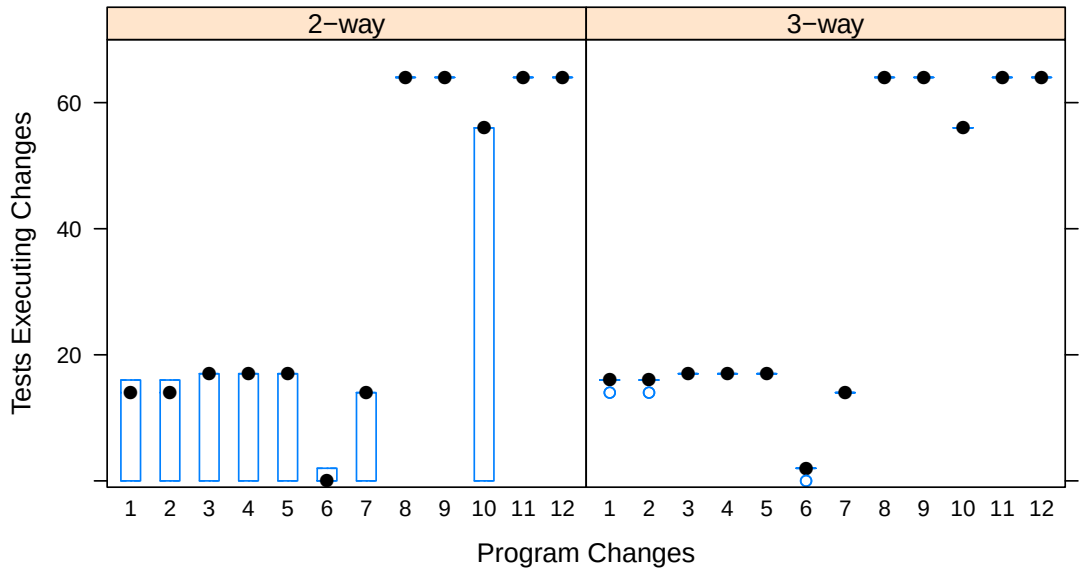
First we see that across all changes, most were executed by well fewer than 40% of the covering array configurations. That is, for each change, many of the selected configurations were unnecessary as they could not exercise the change.

Next, we observed that across all changes, different changes were executed in different numbers of configurations. For example, with both the 2-way and 3-way covering arrays, change 6 was rarely executed, while changes 8, 9, 11 and 12 were executed in roughly one third of the configurations. These results suggest that some modifications are reachable in more configurations than are other modifications.

Finally, we observed that, across individual changes, the change execution rates for the 2- and 3-way covering arrays were essentially identical (remember that the resolution of the measurements is quite coarse). This suggests that coverage of the change may be correlated with the characteristics of the underlying configura-



(a) Configurations that executed each program change.



(b) Test cases that executed each program change.

Figure 5.3: Results of regression testing of vsftpd using 2-way and 3-way covering arrays.

tions. For 2- and 3-way covering arrays those characteristics are likely to be the specific pairs or triples of option settings found in a given configuration.

RQ2: Test Selection for Configurable Systems Different configurations execute different program changes. Similarly, different test cases may execute changes in some configurations, but not in others. To better understand this issue we measured the number of test cases (out of the 64 in total) that executed each change when using the covering arrays. This information is presented in Figure 5.3(b). First, we observed striking similarities between the 2- and 3-way covering arrays, although the variation was greater for the 2-way covering arrays because more of them failed to execute all of the changes. Across all changes, we observed that some were executed by the entire test suite and some were executed by only a small fraction of it. For example, change 6 was always executed by very few test cases, while changes 8, 9, 11 and 12 were always executed by all test cases under all of the covering arrays.

Overall we see that even when a program change can be executed in a specific configuration, the number of specific test cases in which that happens can vary considerably. Additionally, putting this data together with that of RQ1, we see that better test selection techniques based on both configuration and test cases might greatly improve cost-effectiveness. For example change 6 is executed by very few configurations, but even in those configurations very few test cases exercise this change. Therefore, for change 6 many configuration / test case pairs can be safely skipped during regression testing.

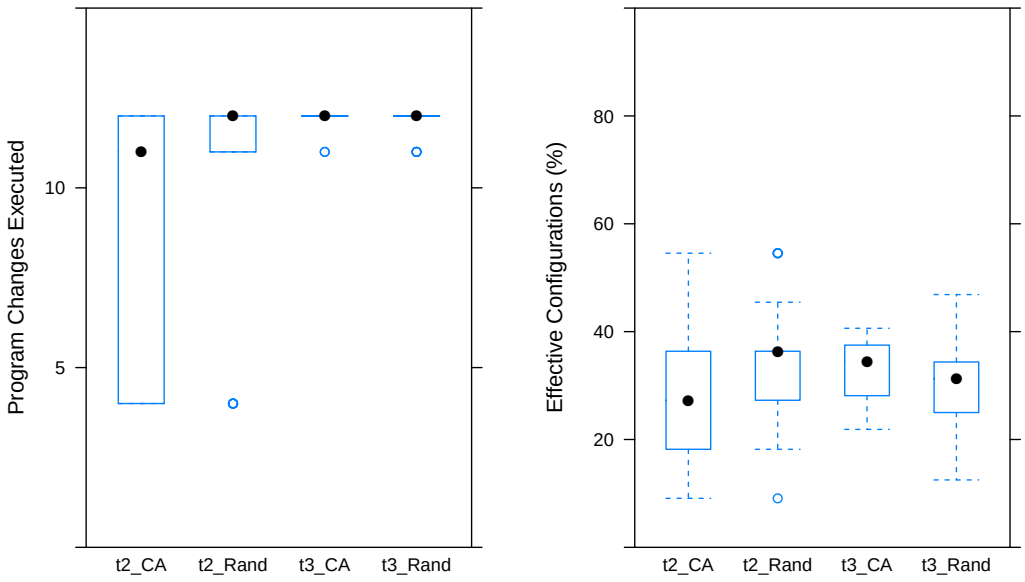


Figure 5.4: Comparing regression testing effectiveness of covering arrays and random samples.

RQ3: Covering Arrays vs. Random Sampling To better understand covering array effectiveness, we next compared the performance of the 2- and 3-way covering arrays to that of 60 equally sized, but randomly-sampled sets of configurations. For each set we measured two outcomes:

1. Total coverage – the number of program changes executed in at least one configuration.

2. Effective configuration set size – the number of configurations in which at least one program change was executed.

The data appears in Figure 5.4. First, we see that both sampling approaches were capable of exercising all 12 program changes. Next, we see that the performances of 3-way covering arrays were essentially identical to the equally-sized ran-

dom samples. In contrast, 2-way covering arrays had more low-performing samples than did random sampling.

Figure 5.4 also shows the size of the effective configuration sets. Here we observed that, for both sampling approaches, few configurations exercise the changes. Overall, the results for both approaches were similar, however, the number of effective configurations for the 3-way sized random sets had somewhat more variability.

Overall, we see that covering arrays themselves added little over random sampling. However, we note again that covering arrays are constructive but random sampling has no built-in way to determine the sample size and thus may be less reliable in practice.

RQ4: Improvements with Prioritization and Regeneration To partially replicate Qu et al.'s studies on the effectiveness of prioritization, we first measured the line coverage achieved by each configuration in every 2- and 3-way covering array. For each covering array, we then created an execution order using a greedy algorithm. Starting with no lines covered, the algorithm iteratively chooses the next configuration that covers the most currently uncovered lines (ties broken randomly). This continues until all changed lines executed by the covering array have been covered. All remaining configurations are then added at the end in random order.

We then measured, for both ordered and unordered samples, the number of configurations needed to cover the maximum amount of program changes executed. We found that, for these changes, line coverage correlates well with regression testing performance. For the unordered samples, some needed as many as the entire set

of configurations. For the ordered samples, however, the 2-way covering arrays need no more than 3 configurations and the 3-way needed no more than 2 configurations. The 3-way covering arrays executed maximum amount of program changes sooner because of the larger sample size increased the number of high coverage configurations in each covering array.

We noticed that, for both 2- and 3-way covering arrays, a small portion of configurations had exceptionally high coverage compared to the rest. With further investigation, we found that these high coverage configurations all included vsftpd's *enabling options* while the others did not. We believe that regenerating configurations using important interactions such as the enabling options, instead of Qu et al.'s regeneration with important individual option settings, could yield configurations with even better regression testing effectiveness.

Study Results: When using covering arrays as a sampling mechanism, our findings were generally consistent with Qu et al.'s. We observed that regression testing performance varied by configurations. That is, some changes were unreachable in some configurations, but reachable in others. Next, we observed that the effect of configurability is intertwined with the behavior of individual test cases (and the location of the changed code). Some test cases execute a change in some configurations; others in all configurations.

Next, we observed that covering arrays and equally sized random samples had comparable performance, and both approaches create sampling configuration sets that are inefficient during regression testing. Finally, we observed that prioritiza-

tion / regeneration could lead to full coverage of the changes earlier than if no prioritization / regeneration were done.

On the other hand, we also observed variations disappeared when using 3-way instead of 2-way covering arrays. Therefore, we suspect that Qu et al.'s findings may be somewhat specific to their use of 2-way covering arrays as the configuration sampling approach. Thus, their results might or might not extend to other sampling approaches or to configurable systems in general.

5.2.1.2 Further Analysis with Interactions

To deepen our understanding of regression testing for configurable systems, we conduct further analysis, this time taking advantage of the configuration interactions we calculated in the Chapter 3. The main goal of these analyses was to examine how configuration relates to regression testing performance – independent of any particular sampling approach or change location.

The configuration interaction data can help us determine the exact set of configuration / test case pairs that execute each line of code in the subject systems, for the given test suites. With this information we can compute which test cases need to be run, in which configurations, should a given set of changes occur. Now we examine Qu et al.'s research questions using the configuration interaction data.

RQ1: Effect of Configuration on Regression Testing With the configuration interaction data, we can better explain the effect of configuration on regression testing.

Across the 12 lines of code changed between the 2 versions of vsftpd, we found that there are only 3 unique configuration interactions; one involving 2 option settings, one involving 3, and one involving 4. These interactions were only present in a small number of configurations, which explains why the effective configuration set sizes were so small. In particular, the interaction for change 6 involves 4 specific option settings. Since these option settings will only appear by chance in a 2- or 3-way covering array, therefore, that change was rarely executed.

It is clear that source code's interaction strengths play an important role in determining their coverage by the sampled configuration sets. Using the analysis results from Chapter 3, we know that 77.56% for vsftpd and 83.35% for ngIRCd, can be executed with interactions of strength 3 or less. But to thoroughly regression test lines of code with higher strength interactions would require much larger covering array samples (up to 7-way for vsftpd and 6-way for ngIRCd). Therefore, the regression testing effectiveness of a set of configuration is different for lines of code at different locations in the system.

RQ2: Test Selection for Configurable Systems The configuration interaction data also shows that some test cases will not execute certain lines of code in any configuration. This is in some ways analogous to previous findings on regression test selection for single configuration systems. That is, some test cases can be safely skipped because the test case is not affected by the changed code. With the interaction data, we have a more nuanced picture. Here we see that the same line of code can have different configuration interactions for different test cases. We found

28% and 35% such lines of code for vsftpd and ngIRCd respectively.

Changing these lines of code implies executing different configuration / test case pairs during regression testing. This means a configuration that executed a change in one test case might not be able to execute the same change in a different test case, thus unsafely skipped necessary testing due to configuration. Therefore, the notion of test suite level coverage or fault detection might actually be incomplete. Without the configuration interaction data to pin point these configuration / test case pairs, bugs can potentially lie undetected.

RQ3: Covering Arrays vs. Random Sampling It is obvious that the effectiveness of regression testing using a sampling approach depends on the degree to which the sampled configurations include the interactions for the changed code. Using the comparison results from Chapter 3, we know that the random samples and the covering array samples covered similar percentages of the interactions at any given sample size.

However, the analysis results from Chapter 3 also concluded that both covering arrays and random sampling are inefficient at covering actual interactions of a software system. For example, most of our configuration samples did not include the enabling options of our subject systems, thus these configurations cannot regression test majority of the source code.

RQ4: Improvements with Prioritization and Regeneration In our attempt to replicate Qu et al., we greedily prioritized configurations according to the number

of lines they covered. We observed that this scheme achieved maximum line coverage after only 2 or 3 configurations. Looking at the configuration interaction data, we observe that this approach achieves maximum coverage quickly because each high priority configuration simultaneously satisfies multiple configuration interactions, i.e., they pack multiple interactions into a single configuration. Since there were relatively few total configuration interactions, maximum line coverage is achieved quickly. And since maximum line coverage implies coverage of the program changes, even a simple prioritization scheme can improve fault detection time.

5.2.2 Targeted Regression Set

Our findings so far suggest that configuration interactions may provide valuable information for selecting the configurations / test case pairs that should be executed during the regression testing of highly configurable systems. Specifically, for a test case and a set of program changes, we can determine the interactions under which each test case executes any program change. Using this information we can then generate, one for each test case, a hopefully small set of configurations that covers all such interactions. We call this set of configurations a *targeted regression configuration set (TRCS)*.

We implemented a prototype system for generating the TRCSs. At a high level, the algorithm works as follows. First, we use the configuration interaction data to determine all the unique interactions for the set of program changes. We then use a divide and conquer algorithm we developed to compute a small set of

configurations that covers *all* of these configuration interactions. It is important to note that TRCS aims to guarantee execution of each program change using all of the interactions that exercise this change, which is a much stronger coverage criterion than the one used by the *minimal covering set* or the *minimized iTree set*.

The divide and conquer algorithm works as follows. First it partitions the set of configuration interactions into multiple small groups. For these studies we restricted each group to having no more than 10 interactions, but other sizes might be more cost-effective. Next, within each partition, the algorithm then attempts to merge multiple non-conflicting interactions into larger compound interactions. Two configuration interactions are non-conflicting if they are simultaneously satisfiable. The algorithm uses *dynamic programming* to determine the merging operations leading to the fewest possible compound interactions. If in any of the partitions at least one merge occurred, the algorithm repeats itself on the current set of compound interactions; otherwise it terminates. Next, we compute, the TRCS – a set of concrete configurations that satisfies the compound interactions.

We generated a TRCS for the 12 changes made to vsftpd, and the number of configurations in the set is 3. Not only did the total coverage of the TRCS include all 12 changes, every configuration in the TRCS was effective as well. This preliminary experiment suggest that configuration interactions does provide valuable information for selecting the configurations / test case pairs for regression testing of configurable systems.

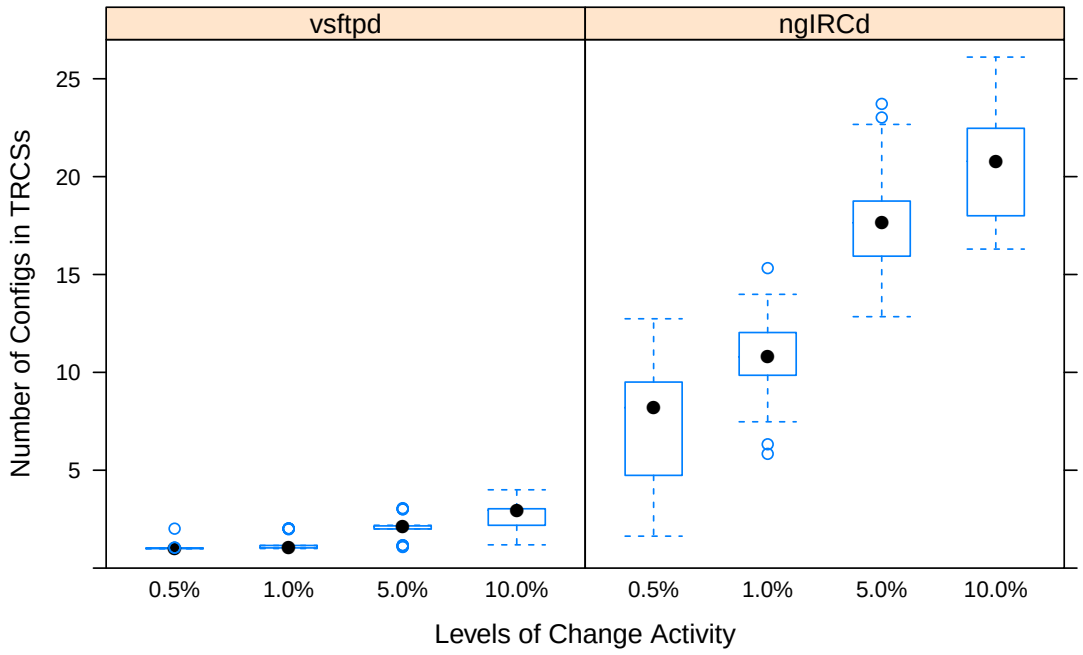


Figure 5.5: Size of TRCSs generated for vsftpd and ngIRCd during regression testing simulations.

5.2.2.1 Data and Analysis

To further evaluate our prototype system, we ran a series of regression testing simulations on both vsftpd and ngIRCd. In these simulations, we randomly selected 0.5%, 1%, 5% or 10% of the overall source code to be changed between the previous and current versions of the systems. Using each subject system's configuration interaction data, we then generated TRCSs for the program changes under every test case. For each level of change activity, we ran the simulation 30 times for every test case. Figure 5.5 shows the average size of the TRCSs generated for each test case.

Not surprisingly, as the number of program changes increases, the size of these

TRCSs increased as well. However, even at 10% of source code changes, the size of the TRCSs remained relatively small for both subject systems. On average, each test case's TRCS contained no more than 5 and 27 configurations, for vsftpd and ngIRCd, respectively, at all levels of change. This means regression testing using TRCSs would execute fewer configuration / test case pairs than those executed when testing with 2-way covering arrays. Using 2-way covering arrays, each test case would need to be executed under 11 and 32 configurations, for vsftpd and ngIRCd respectively. Moreover, the TRCSs guarantee the execution of all changes under all configuration interactions. For our subject systems, this guarantee would require 7-way covering arrays with order of magnitude more configurations to satisfy.

These simulation results showed that our TRCS technique can indeed generate efficient sampling sets that provide full coverage for program changes under all affected configuration interactions.

5.3 Summary

In this chapter, we developed tools and techniques that can leverage the effective configuration spaces to improve the testing and analysis of configurable software systems, and can do so practically during today's increasingly iterative and incremental development cycles.

We first showed that our iTree algorithm can be integrated with existing quality assurance processes and infrastructures. We discussed the design and implementation choices that we made to easy the integration process. We then presented

an iTree-based automated distributed testing framework that was built on top of Skoll, a distributed continuous quality assurance (DCQA) process and framework. Using the Skoll framework, the execution of highly effective configurations selected by iTree can be parallelized on multiple distributed computing resources.

We next turned our attention to improving the regression testing of configurable software systems. We conducted an extensive study on how program changes affect different configuration / test case combinations. Our analysis results showed that only a few carefully selected configurations are needed to exercise the changes made to most parts of a system. We then developed an algorithm that uses the configuration interaction data to generate a *targeted regression set*, a small set of configurations that, for a given set of program changes, can execute every change under every interaction that exercise the change. Our experiments showed that the targeted regression sets for our subject systems are indeed very small for most system changes.

Chapter 6

Conclusions and Future Work

In this dissertation, we challenged the widely accepted assumptions used to deal with the *software configuration space explosion* problem and took several major steps toward improving the testing and analysis of highly configurable software systems. In this chapter, we conclude this dissertation by summarizing its contributions and discuss future work.

6.1 Contributions

This dissertation has yield several contributions to the area of software engineering. We identified and provided support for three primary research hypotheses:

1. For many practical tasks, a system's effective configuration space is a small subset of its full configuration space.

2. We can efficiently discover or approximate the effective configuration space of a software system.

3. We can greatly improved numerous software engineering tasks by leveraging a system's effective configuration space.

The following sections discuss how we addressed these research hypotheses.

6.1.1 Scientific Understanding of Configuration Spaces

To date, as far as we know, no study has been done to scientifically understand the effects of configuration on software systems' behavior using a white-box approach. We developed new analytical techniques to perform empirical studies that uncovered new knowledge of software configuration previously not possible to obtain.

In Chapter 3, we first used symbolic evaluation to calculate the effects of configuration on the behavior of software systems. The evaluation results suggest that the effective configuration spaces are much smaller than what developers previously assumed. We next developed techniques to abstract the interactions between configuration option settings. Keeping existing threats to validity in mind, we were able to draw several conclusions. First, we found that configuration interactions were quite rare in the systems we studied; only a handful of specific options setting combinations are needed to exercise all of the system behavior. Second, most of the interactions needed to achieve good structural coverage were low strength, but higher strength interactions are needed to achieve maximum coverage. Finally, the higher strength interactions were usually just lower strength interactions with additional constraints.

Using the configuration interaction data, we were able to evaluate the existing configuration space sampling approaches. We found out that existing approaches, such as CIT or random sampling, are quite ineffective. For example, CIT techniques does too much work trying to cover all interactions of a certain t strength, but at

the same time, they often missed higher strength interactions needed to achieve maximum coverage. Based on our evaluation results, we concluded that a more effective sampling approach should focus on the coverage of effective configuration interactions. We experimented with packing the interactions into configurations using a greedy algorithm and found that it took only five to ten configurations to achieve the maximal coverage for our subject systems.

6.1.2 The iTree Algorithm

The insights from our studies of configuration spaces and existing sampling approaches led us to create a new heuristic-based dynamic approach called iTree, an interaction tree discovery algorithm. This scalable approach combines low-strength covering arrays, runtime instrumentation, and machine learning techniques to efficiently navigate through enormous configuration spaces in search of highly effective configurations. The evaluation results strongly suggest that iTree, running with the optimizations and heuristics we developed, can achieve higher coverage at lower cost than any existing sampling techniques. The results of our scalability evaluation, which applied iTree to the ∼1M-LOC MySQL database, suggest that iTree is a promising technique that can scale to practical industrial systems.

We next developed the interaction learning approach to estimate, using the execution results of iTree runs, the configuration interactions of software systems. Analysis results showed that this approach is not only light-weight, but also produces very accurate interaction estimations, making leveraging knowledge of effective con-

figuration spaces practical for many software engineering tasks. Using these new techniques, we were able to approximate MySQL's effective configuration space at very low cost, something that is infeasible using any existing technique.

6.1.3 Practical Applications of the Effective Configuration Space

Encouraged by the performance of our new effective configuration space discovery approaches, we built an automated framework for the testing of highly configurable software systems. We integrated the iTree algorithm with the Skoll [55] continuous distributed quality assurance process and framework. This integrated testing framework allows iTree to scale up to practical industrial software systems by parallelizing its configuration space search process across multiple distributed computing resources. This is a first step towards building a practical testing environment for configurable systems.

Finally, we analyzed the affects of program changes have on configuration / test case selection for regression testing. We developed an algorithm that selects a small, targeted set of configurations, using the configuration interaction data, to execute all of the program changes under only the affected configuration / test case combinations. This way the cost of configuration-aware regression testing is much lower than testing with CIT.

6.2 Future Work

In this section, we present a research vision that extends beyond this dissertation. We have also identified shortcomings of our approaches that may be the subject of future research. We envision an integrated suite of tools that together provide a complete environment for designing, developing and testing configurable software systems. Specific ideas for the future work are as follows.

6.2.1 Extending Studies

All of the conclusions from the empirical studies of this dissertation are specific to our subject systems, test suites, and configuration spaces; further work is clearly needed to establish more general trends.

First, we plan to extend our studies to include more subject systems. In this work we used 3 subject systems, vsftpd, ngIRCd and MySQL. Each is widely used server software, but not representative to all industrial applications. We plan to include other types of systems ranging from desktop applications to operating systems.

Next, we plan to use larger and more complex configuration spaces for our subject systems to get more complete information. In order to keep our analyses tractable, we focused on sets of configuration options that we determined to be important. The size of these sets was substantial, but did not include every possible configuration option. And we will improve and optimize our tools and techniques to better handle the analysis efforts required for these studies.

In addition, we plan to extend our analysis to other types of program behaviors. The program behaviors we studied in this dissertation included different structural coverage criteria. Other program behaviors such as fault detection or data flows might lead to different results.

Finally, we would like to augment the test suites both in quantity and quality. The individual test cases we used tend to be focused on specific functionality, rather than combining multiple activities in a single test case. Taken together, they have reasonable coverage, but they were not designed for extensive testing of the systems; these test suites are more like typical regression suites than customer acceptance or functional test suites.

6.2.2 Improving iTree with Static Analysis

Our iTree approach, in its current implementation, uses pure dynamic analysis. We believe that combining iTree's dynamic analysis with even simple static analysis can greatly improve its performance.

Aside from some obvious optimizations and general heuristics, iTree essentially treated every program entity independent of each other. However, as we have seen in the example program from Section 4.1, the structure of a system's source code greatly influences how configuration interactions relate to one another. For instance, the higher strength interactions of a system tend to be refinements of its lower strength ones with additional option setting constraints. By adding some light-weight static analysis of the source code, we can provide iTree with more accurate

estimation of each proto-interaction's priority score during execution and direct the discovery process towards interactions that can execute paths with more potential to find previously uncovered code.

We can also use control-flow analysis to discover the relationships amongst the program entities. Using the relationship data, we can improve the interaction learning approach to generate more accurate configuration interaction estimations and/or reduced the sample size needed for the interaction learning process.

Currently, iTree requires a configuration space model to be provided before the testing process can start. This can be tricky, especially determining the values for the integer configuration options. For vsftpd and ngIRCd, we used symbolic evaluation to determine some of the option settings in our configuration space models. We can implement other light-weight static analysis to extract, from a system's source code, good option settings to include in its configuration space model.

6.2.3 Configuration Documentation

Configurability of software systems is not always well documented, and it is rarely linked to the software artifacts. Therefore, details of configuration such as implementation choices tend to be lost after the development stage. In the previous chapters we spent tremendous effort to recover these lost details, but we argue that information about a system's configuration must be carefully documented and maintained just like other artifacts such as the system's source code. Therefore, we propose to create the *configurability annotations*, a tool to facilitate the documen-

tation of a software system's configurability.

These configurability annotations augment a system's source code, much like the JavaDoc [41] tool, to include configuration information. There are two main advantages to maintaining configuration information along with the source code. First, for most tasks, the source code is the most natural medium for the developers to interact with the software systems. Second, modifications made to the configurability of the systems are either implemented in the source code or reflected in the execution of the source code.

Since developers know best how the configurability of a system is implemented, annotations made by the developers are the most effective way to gather information about the configuration space. This developer input is also a valuable resource that can be leveraged to speed up our iTree discovery process. However, it might be impractical to expect the developers to provide all of the configuration details, and the information provided by developers could be incomplete or inaccurate. Therefore, we plan to supplement the developer annotations with the estimated configuration interaction data. Combining both the developer inputs and interaction data forms a powerful framework for configurable software tool support. We envision Integrated Development Environment (IDE) integration of configurability annotations at different granularities of the source code such as files, methods or lines.

The following two scenarios demonstrate how this tool could aid program understanding tasks for configurable systems:

- *Scenario 1*: When the developer annotates the source code, the IDE can use

the developer input to pre-populate the interaction tree to quickly verify the annotated information with actual execution data. The results from the verification process can either improve developer's confidence of their understanding of the system or provide corrections to any wrong assumptions the developer had about the system's configuration space.

- *Scenario 2*: If no detailed configuration information is available to the developer, then the estimated configuration interactions available from previous iTree runs (or configurations executed on-demand) can help the developer understand the system and aid with development and maintenance tasks. Since iTree is heuristic in nature, we cannot guarantee 100% accuracy, and the IDE could display support values for the configuration interactions (e.g., in the mouse-over pop-up balloons) and show the configuration execution results that provided the interactions.

6.2.4 Recommendation Systems

Finally, we plan to explore tool support for configurable software's development and maintenance tasks. Specifically, we plan to develop tools such as a recommendation system to guide the developers through the complex structures of a configurable software system.

There has been work on numerous recommendation systems for software engineering [84, 83, 36]. These systems support developers in their decision making while performing development and maintenance tasks that involve information seek-

ing. They often use data mining to extract predictive information from large data sets, such as source code, version history or bug reports, to find relationships between software artifacts the developers are working with. We believe that a recommendation system using the configuration interaction data can improve modification, debugging or refactoring tasks for configurable systems.

For instance, making changes to configurable systems is difficult due to the complications brought about by configuration. Understanding where to modify software features related to a particular option or group of options without affecting others can be a difficult task. Locating these features and their dependencies in a system's source code is not always trivial [22]; the implementation of a feature might be scattered throughout the source code or might be intertwined with the implementation of other features. Instead of relying on often poorly maintained documentation or manually inspecting the source code, we propose a recommendation system that can aid this program understanding task. Weiser et al. [77] suggested that experienced developers reduce programs to minimal forms that still produce specified behavior. This recommendation system would use the configuration interaction data to extract configuration slices that contain the interested features.

Configuration-aware recommendation systems can also help with preventing accidental bugs by detecting configuration option settings that are associated with a set of program changes. Such a recommendation system identifies potential omissions of other necessary changes that are related in terms of configuration but are structurally distant in the source code. Another recommendation system can iden-

tify potential architectural improvements for the system and facilitate refactoring tasks. For example, this recommendation system can identify candidate aspects in the source code that implementation cross-cutting features.

While this dissertation presented significant steps toward understanding, discovering and leveraging the *effective configuration spaces* of software systems, we hope it paves the way for many more initiatives to explore opportunities for improving software engineering tasks for highly configurable software systems.

Bibliography

[1] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Jpf-se: a symbolic execution extension to java pathfinder. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 134–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, February 2009.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[4] G. Bockle, P. Clements, J.D. McGregor, D. Muthig, and K. Schmid. Calculating roi for software product lines. *Software, IEEE*, 21(3):23 – 31, May-June 2004.

[5] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. *SIGSOFT Software Engineering Notes*, 29:195–205, July 2004.

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

[7] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, California, 1984.

[8] 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.

[9] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 480–490, Washington, DC, USA, 2004. IEEE Computer Society.

[10] 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.

[11] Paulo Marcos Siqueira Bueno and Mario Jino. Identification of potentially infeasible program paths by monitoring the search for test data. *Automated Software Engineering, International Conference on*, 0:209, 2000.

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

[13] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *Information and System Scurity, ACM Transactions on*, 12:10:1–10:38, December 2008.

[14] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[15] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–44, 1997.

[16] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ROSATEA '06, pages 53–63, New York, NY, USA, 2006. ACM.

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

[18] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 339–348, Washington, DC, USA, 2001. IEEE Computer Society.

[19] 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.

[20] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438 –444, July 1984.

[21] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in jpl's mission data system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 7, pages 259 –268 vol.7, 2000.

[22] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29:210–224, 2003.

[23] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen Mccamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2006.

[24] Sandro Fouché, Myra B. Cohen, and Adam Porter. Towards incremental adaptive covering arrays. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The*

foundations of software engineering, ESEC-FSE '07, pages 557–560, New York, NY, USA, 2007. ACM.

[25] Sandro Fouché, Myra B. Cohen, and Adam Porter. Incremental covering array failure characterization in large configuration spaces. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188, 2009.

[26] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying software failures. *Software Reliability Engineering, International Symposium on*, 0:451–462, 2004.

[27] Yoav Freund and Robert E. Schapire. Experiments with a New Boosting Algorithm. In *International Conference on Machine Learning*, pages 148–156, 1996.

[28] Critina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, SSR '01, pages 109–117, New York, NY, USA, 2001. ACM.

[29] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, July 2007.

[30] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An improved meta-heuristic search for constrained interaction testing. *Search Based Software Engineering, International Symposium on*, 0:13–22, 2009.

[31] GNU GCC. Gcov – a test coverage program, 2009.

[32] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43:206–215, June 2008.

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[34] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.

[35] M. Haran, A. Karr, M. Last, A. Orso, A.A. Porter, A. Sanil, and S. Fouche. Techniques for classifying executions of deployed software to support software engineering tasks. *Software Engineering, IEEE Transactions on*, 33(5):287 – 304, May 2007.

[36] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *Software Engineering, IEEE Transactions on*, 32:952–970, December 2006.

[37] Ying Hu, Ettore Merlo, Michel Dagenais, and Bruno Lag252;e. C/c++ conditional compilation analysis using symbolic execution. *Software Maintenance, IEEE International Conference on*, 0:196, 2000.

[38] Nathalie Japkowicz and Shaju Stephen. The class imbalance problem: A systematic study. *Intelligent Data Analysis*, 6:429–449, October 2002.

[39] David J. Kasik and Harry G. George. Toward automatic generation of novice user test scripts. In *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground*, CHI '96, pages 244–251, New York, NY, USA, 1996. ACM.

[40] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[41] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, SIGDOC '99, pages 147–153, New York, NY, USA, 1999. ACM.

[42] 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.

[43] D.R. Kuhn, D.R. Wallace, and Jr. Gallo, A.M. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, 30(6):418 – 421, June 2004.

[44] H.K.N. Leung and L. White. Insights into regression testing. In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60–69, Oct 1989.

[45] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

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

[47] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.

[48] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 241–251, New York, NY, USA, 2004. ACM.

[49] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *SPLC*, pages 196–210, 2010.

[50] David L. Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, 2:1–9, 1976.

[51] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automated and scalable t-wise test case generation strategies for software product lines. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:459–468, 2010.

[52] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[53] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. *Software Engineering, International Conference on*, 0:465, 2003.

[54] Andy Podgurski, Wassim Masri, Yolanda McCleese, Francis G. Wolff, and Charles Yang. Estimation of software reliability by stratified sampling. *Software Engineering Methodology, ACM Transactions on*, 8:263–283, July 1999.

[55] Adam Porter, Cemal Yilmaz, Atif M. Memon, Douglas C. Schmidt, and Bala Natarajan. Skoll: A process and infrastructure for distributed continuous quality assurance. *Software Engineering, IEEE Transactions on*, 33(8):510–525, August 2007.

[56] Xiao Qu, M.B. Cohen, and K.M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 255 –264, Oct. 2007.

[57] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86, New York, NY, USA, 2008. ACM.

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

[59] J. R. Quinlan. C4.5*: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[60] J. R. Quinlan. Bagging, boosting, and c4.5. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1*, AAAI'96, pages 725–730. AAAI Press, 1996.

[61] J.R. Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27(3):221–234, 1987.

[62] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S. Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 445–454, New York, NY, USA, 2010. ACM.

[63] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *Software Engineering Methodology, ACM Transactions on*, 6:173–210, April 1997.

[64] Richard L. Rudell. Multiple-valued logic minimization for pla synthesis. Technical Report UCB/ERL M86-65, UC Berkeley, 1986.

[65] S.R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(3):660 –674, May/June 1991.

[66] Raghvinder Sangwan, Neel Mullick, and Matthew Bass. *Global Software Development Handbook*. CRC Press, 2006.

[67] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21(4):294 – 324, 1998.

[68] Koushik Sen. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007. ACM.

[69] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE-13*, pages 263–272, 2005.

[70] Rajendra Singh Sisodia and Vijaykumar Channakeshava. Combinatorial approach for automated platform diversity testing. In *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, ICSEA '09, pages 134–139, Washington, DC, USA, 2009. IEEE Computer Society.

[71] Ben Smith and Martin S. Feather. Challenges and methods in testing the remote agent planner. In *In Proc. 5th Int.nl Conf. on Artificial Intelligence Planning and Scheduling (AIPS*, pages 254–263, 2000.

[72] Robert R. Sokal and F. James Rohlf. The comparison of dendrograms by objective methods. *Taxon*, 11(2):pp. 33–40, 1962.

[73] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 97–106, New York, NY, USA, 2002. ACM.

[74] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[75] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. *Software Product Line Conference, International*, 0:233–242, 2007.

[76] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[77] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25:446–452, July 1982.

[78] David Wheeler. Sloccount, 2009.

[79] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pages 359–368, June-July 2009.

[80] Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA*, pages 27–38, 2008.

[81] Yiling Yang, Xudong Guan, and Jinyuan You. Clope: a fast and effective clustering algorithm for transactional data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, pages 682–687, New York, NY, USA, 2002. ACM.

[82] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 31(1):20–34, 2006.

[83] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, 30:574–586, September 2004.

[84] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.