ABSTRACT

Title of dissertation: DISTRIBUTED CONTINUOUS QUALITY ASSURANCE

Cemal Yilmaz, Doctor of Philosophy, 2005

Dissertation directed by: Associate Professor Adam Porter
Department of Computer Science

Quality assurance (QA) tasks, such as testing, profiling, and performance evaluation, have historically been done in-house on developer-generated workloads and regression suites. The shortcomings of in-house QA efforts are well-known and severe, including (1) increased QA cost and (2) misleading results when the test cases, input workloads, and software platforms at the developer's site differ from those in the field. Consequently, tools and processes have been developed to improve software quality by increasing user participation in the QA process. A limitation of these approaches is that they focus on isolated mechanisms, not on the coordination and control policies and tools needed to make the global QA process efficient, effective, and scalable. To address these issues, we have initiated the Skoll project, which is developing and validating novel software QA processes and tools that leverage the extensive computing resources of worldwide user communities in a distributed, continuous manner to significantly and rapidly improve software quality. We call this *distributed, continuous quality assurance* (DCQA). We envision a QA process conducted around-the-world, around-the-clock on a powerful computing grid pro-

vided by thousands of user machines during off-peak hours. Skoll processes are distributed, opportunistic, and adaptive. They are distributed: Given a QA task we divide it into several subtasks each of which can be performed on a single user machine. They are opportunistic: When a user machine becomes available we allocate one or more subtasks to it, collect the results when they are available, and fuse them together at central collection sites to complete the overall QA process. And finally they are adaptive: We use earlier subtask results to schedule and coordinate subtask allocation.

In this thesis, we build an infrastructure, algorithms and tools for developing and executing through, transparent, managed, and adaptive DCQA processes. We then develop several novel DCQA processes and empirically evaluate them, with a special focus on cost efficiency and applicability of these processes to real-life, highly-configurable software systems. Our results strongly suggest that these new processes are an effective and efficient way to conduct QA tasks such as evaluating performance characteristics and testing for functional correctness of evolving software systems.

DISTRIBUTED CONTINUOUS QUALITY ASSURANCE

by

Cemal Yilmaz

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
2005

Advisory Committee:

Associate Professor Adam Porter, Advisor
Assistant Professor Atif Memon
Associate Professor Bill Pugh
Assistant Professor Alan Sussman
Professor Yavuz Oruc

© Copyright by

Cemal Yilmaz

2005

To my son, Efe Kaan Yılmaz, and my wife, Esin Yılmaz

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincerest gratitude to my advisor, Dr. Adam Porter, for his precious guidance, support, optimism, and friendship. It has been a great pleasure to work with such an extraordinary individual and I certainly have learned a great deal from him. I would also like to thank Dr. Atif Memon for his extremely valuable comments and extensive assistance.

I am also thankful to all members of my preliminary and final examination committee, Dr. Adam Porter, Dr. Atif Memon, Dr. Bill Pugh, Dr. Alan Sussman, and Prof. Yavuz Oruç, for their time reviewing the manuscript and for giving me valuable feedback.

I would especially like to thank my wife, Esin Yılmaz. I would not be able to complete this journey without her unconditional love, continuous support, sincere encouragement, and endless belief in me. She has always been a great inspiration to me.

I would like to thank my mother, Dilek Yılmaz, my father, Süleyman Yılmaz, and my brother, Can Yılmaz, for their deep love and care. Special thanks go to my brother for making me laugh even at the most difficult times and for giving me the inspiration to carry on.

Finally, I would like to acknowledge the support and friendship from so many friends in College Park over the years in grad school: Mustafa Murat Tıkır, Tuna

Güven, Tolga Girici, Okan Kolak, Onur Kaya, Onur Ergin, Akın Aktürk, Fatih Demiray, Zafer Tuncalı, Murat Gözü, Oktay Demircan, Yeşim and Serdar Şahin, and many others that I forgot to mention here. Thank you all for making it a memorable journey.

TABLE OF CONTENTS

List of Tables viii

List of Figures ix

1 Introduction 1

2 Related Work 10
2.1 Current DCQA Approaches 10
2.2 Factor-Covering Designs for Interaction Testing 15
2.3 Fault Localization 20
2.4 Performance Evaluation and Optimization 23

3 The Skoll Project 29
3.1 Skoll Infrastructure 31
3.1.1 Configuration and Control Model 32
3.1.2 Intelligent Steering Agent (ISA) 34
3.1.3 Analysis Manager 39
3.1.4 Visualization Manager 40
3.2 Skoll Implementation 41
3.2.1 Configuring Skoll 41
3.2.2 Registering Skoll Clients 43
3.2.3 Requesting QA Jobs 43
3.2.4 Generating QA Jobs 44
3.2.5 Executing QA Jobs 47
3.2.6 Collecting QA Job Results 47
3.2.7 Analyzing and Visualizing QA Job Results 48

4 Initial Implementation and Evaluation 49
4.1 Background 51
4.1.1 Classification Trees 52
4.2 The Fault Characterization Process 53
4.3 Setting up the Skoll Infrastructure 55
4.4 Study 1: Clean Compilation 56
4.5 Study 2: Testing with Default Runtime Options 60
4.6 Study 3: Testing with Configurable Options 64
4.7 Discussion 67

5 A Sampling Strategy for Efficient Fault Characterization in Complex Configuration Spaces 70
5.1 Background 72
5.1.1 Covering Arrays 72
5.1.2 Variable Strength Covering Arrays 75
5.2 Revisiting the Fault Characterization Process 78

5.2.1 Evaluating Fault Characterizations 79
5.2.2 Reducing the Test Schedule Size 81
5.3 Experiments 81
5.3.1 Study 1: Revealing option-related failures with covering arrays 84
5.3.2 Study 2: Covering arrays with per test case characterization . 87
5.3.3 Study 3: Covering arrays with per test, failure case characterization 91
5.3.4 Study 4: Combined reduced schedules 96
5.4 Further Improving the Efficiency 99
5.4.1 Creating Variable Strength Covering Arrays 100
5.4.2 Evaluating Variable Strength Covering Arrays 101
5.4.3 Seeding Faults 103
5.5 Guidelines for Software Practitioners 105
5.6 Comparison with Random Schedules 109
5.7 Discussion 112

6 Main Effects Screening: A DCQA Process for Monitoring Performance Degradations in Evolving Software Systems 116
6.1 Performance-Oriented Regression Testing 119
6.1.1 The Main Effects Screening Process 119
6.1.2 Technical Foundations of Screening Designs 121
6.1.3 Screening Designs in Action 123
6.2 Feasibility Study 128
6.2.1 Experimental Design 128
6.2.2 The Full Data Set 130
6.2.3 Evaluating Screening Designs 131
6.2.4 Estimating Performance with Screening Suites 133
6.2.5 Screening Suites vs. Random Sampling 135
6.2.6 Dealing with Evolving Systems 137
6.3 Validating Basic Assumptions 139
6.3.1 D-optimal Designs 140
6.3.2 Breaking Aliases 144
6.3.3 Checking for Higher-Order Effects 146
6.4 Guidelines for Practitioners 149
6.5 Discussion 150

7 A Demonstration of the Generality of Skoll Infrastructure 153
7.1 Experiments 154
7.1.1 DCQA Scenario 155
7.1.2 Configuration and Control Model 155
7.1.3 Setting up the Skoll Infrastructure 156
7.2 Experimental Setup 159
7.3 Results and Discussions 160

8 Conclusions 162
8.1 Main Contributions 166
8.2 Limitations and Future Work 167

Bibliography 170

LIST OF TABLES

3.1 Some options and constraints. 34

4.1 An example exhaustive test schedule. 54

5.1 A covering array example: $CA(9; 2, 3, 3)$ 73

5.2 A mixed level covering array example, $MCA(12; 2, 3^2 4^1)$ 74

5.3 A VSCA example, $VSCA(10; 2, 3^3 2^3, CA(3, 3, 2))$ 77

5.4 An example exhaustive test schedule. 79

5.5 Size of test schedules for $2 \leq t \leq 6$. 83

5.6 Size of combined schedules. 96

5.7 Comparing fault characterization models obtained from VSCAs and CAs using F measures. 102

6.1 (a) 2^3 design and (b) 2_{IV}^{4-1} design 124

6.2 Some ACE+TAO options 129

6.3 Actual screening designs used in the experiments, calculated using the SAS statistical package 131

6.4 Partial aliasing structure for the important option in the 2_{IV}^{14-7}, 128-run design 143

7.1 Example configuration model. 155

LIST OF FIGURES

3.1 The Skoll architecture 30

3.2 Nearest neighbor strategy. 38

3.3 Server side application component interface. 42

3.4 Client side application component interface. 43

3.5 Interface between the ISA and navigation strategies. 44

3.6 An example QA job. 46

3.7 Interface between the ISA and adaptation strategies. 47

4.1 An example classification tree. 54

5.1 An example classification tree. 79

5.2 Error coverage statistics for 2-way covering arrays on Linux. 84

5.3 Error coverage statistics for 2-way covering arrays 86

5.4 Models for each test. 89

5.5 Models for each test and failure combination. 93

5.6 Fault characterizations for test #3, test #3 and error #2, and test #3 and error #17, respectively. 94

5.7 Fault characterizations for error #18 obtained from the exhaustive schedule, 2-way covering arrays, and 3-way covering arrays, respectively. 95

5.8 Models for combined schedules. 98

5.9 Comparing VSCAs with CAs at various frequency levels. 104

5.10 Scatter plots of F measures for 2-way and 4-way models. 107

5.11 Number of unique errors seen in random and t-way covering arrays. . 110

5.12 Fault characterization for test #2, ERR #18 obtained from the exhaustive schedule, a 2-way schedule, and a random schedule, respectively. 111

6.1 Option effects based on full data 132

6.2 Option effects based on screening designs 134

6.3 Q-Q plots for the top-2 and top-5 screening suites 136

6.4 Latency distribution from full, top-2, and random suites 137

6.5 Performance estimates across time 138

6.6 (a) Complete dealiasing experiment and (b) Up to, including 3rd-order effects dealiasing experiment 147

6.7 Looking for higher-order effects up, to including 3rd-order effects . . . 148

Chapter 1

Introduction

Software quality assurance (QA) tasks are typically performed in-house by developers, on developer platforms, using developer-generated input workloads. One benefit of in-house QA is that programs can be analyzed at a fine level of detail since QA teams have extensive knowledge of, and unrestricted access to, the software. The shortcomings of in-house QA efforts, however, are well-known and severe, including (1) increased QA cost and schedule and (2) misleading results when the test cases, input workloads, software versions and platforms at the developer's site differ from those in the field. These problems are magnified in performance-intensive software, such as that found in high-performance computing systems, distributed real-time and embedded systems, and the accompanying systems software (*e.g.*, operating systems, middleware, and language processing tools). This is because these software systems are increasingly subject to the following trends:

- **Increasing system complexity.** Performance-intensive software systems are getting complex in terms of their sizes, behavioral patterns, and interactions with both internal components and external environments. Consequently, it is getting increasingly hard to thoroughly test these systems.

- **Demand for user-specific customization.** Since performance-intensive software pushes the limits of technology, it must be optimized for particu-

lar run-time contexts and application requirements. One-size-fits-all software solutions often have unacceptable performance.

- **Severe cost and time-to-market pressures.** Global competition and market deregulation are shrinking budgets for the development and QA of software in-house. In particular, customers are often unwilling to pay for customized software. The result is that limited resources are available for the development and QA of highly customizable performance-intensive software.

- **Distributed and evolution-oriented development processes.** Today's development processes are distributed across geographical locations, time zones, and business organizations. This is done to reduce cycle time by having developers work simultaneously, with minimal direct inter-developer coordination. But it can also increase software churn rates, which in turn increases the need to detect, diagnose, and fix faulty changes quickly. The same is true for evolution-oriented processes, where many small increments are routinely added to the base system.

These three trends present new challenges to developers. One major new challenge is the explosion of the *software configuration space*. To support customizations demanded by users, performance-intensive softwares must run on many hardware and OS platforms and typically have many options to configure the system at compile- and/or run-time. For example, SQL Server 7.0 has 47 configuration options [54], Oracle 9 has 211 initialization parameters [59], and Apache HTTP Server Version 1.3 has 85 core configuration options [5]. Although the existence

of all these software parameters promotes flexibility and portability, it also means the software must be tested over an enormous number of different configurations. Consider the Oracle database management system with 211 options as an example. Even assuming that each of these options has two levels of settings only, the size of the configuration space reaches to 2^{211} configurations each of which may deserve extensive QA to validate. Although, in practice, often not all the configurations are valid due to inter-option constraints, the size of the configuration spaces still is a big concern.

When increasing configuration space is coupled with shrinking software development resources, it becomes infeasible to handle all QA in-house. For instance, developers may not have access to all the hardware, OS, and compiler platforms on which their software will run. In this environment developers are forced to release software with configurations that have not been subjected to extensive QA. Moreover, the combination of an enormous configuration space and tight development constraints cause developers to make design and optimization decisions without precise knowledge of the consequences in the field.

Solution approach: Distributed, continuous quality assurance (DCQA). To address these challenges, we have initiated the Skoll project. In this project, to significantly and rapidly improve software quality, we are developing and validating novel software QA processes and tools that leverage the extensive computing resources of potentially worldwide user communities in a distributed, continuous manner. We envision a QA process conducted around-the-world, around-the-clock

on a powerful, virtual computing grid provided by thousands of end-user machines during off-peak hours. Skoll processes are distributed, opportunistic, and adaptive. They are distributed: Given a QA task we divide it into several subtasks each of which can be performed on a single user machine. They are opportunistic: When a user machine becomes available we allocate one or more subtasks to it, collect the results when they are available, and fuse them together at central collection sites to complete the overall QA process. And finally they are adaptive: We use earlier subtask results to schedule and coordinate subtask allocation.

The Skoll project aims at providing developers with access to (1) wide range of platforms (*e.g.,* hardwares, operating systems, compilers, etc.), which may not be available in house, (2) more resources, which can allow large scale QA activities to be performed and make rare events (*e.g.,* intermittent failures) easier to see, (3) thorough, transparent, managed, and adaptive QA processes, and (4) diverse user profiles, which can further shape software development activities by revealing how systems are used in the field. This thesis mainly focuses on the first three bullets.

In this thesis, we build an infrastructure, algorithms and tools for developing and executing through, transparent, managed, and adaptive DCQA processes. We then develop several new DCQA processes and empirically evaluate them, with a special focus on cost efficiency and applicability of these processes to real-life, highly-configurable systems (e.g., application/web servers, middlewares, and database systems). Our results strongly suggest that these new processes are an effective and efficient way to conduct QA tasks such as evaluating performance characteristics and testing for functional correctness of large-scale software systems.

Creating and evaluating the DCQA vision presented challenges *e.g.*; how QA tasks will be decomposed into subtasks; in what order will subtasks be allocated; how will they be implemented so they run on a very wide set of client platforms; how will results be merged together and interpreted; if and how should the process adapt to incoming results; and how will the results of the overall process be summarized and communicated to software developers. We have created novel solutions to tackle these challenges.

Modeling the space of QA subtasks. A cornerstone of our approach is a formal model of the space of QA subtasks, called a *configuration and control model.* This model specifies how QA tasks are divided into smaller subtasks by modeling the aspects of the QA subtasks and the underlying software that will be varied under the control of the DCQA process. The configuration and control model is used in planning the global QA process, for adapting the process dynamically, and to aid in interpreting results.

Exploring the configuration space. The configuration spaces of DCQA processes can be quite large. Even with a large pool of user-supplied resources, brute-force QA approaches (*e.g.,* exhaustive testing) may be infeasible or simply undesirable (Chapter 4). Therefore efficient and effective *navigation strategies* are desirable. Given a DCQA process, a navigation strategy explores the configuration space and selects only a subset of the QA subtasks for testing in ways that do not compromise the accuracy and the goal of the DCQA process.

In this thesis, we introduce and evaluate several types of generic navigation strategies:

Exhaustive strategies. An exhaustive navigation strategy schedules the entire configuration space for testing. Although this type of strategy may not be feasible for "large" spaces, they are very effective for "small" ones in terms of the quality of the results they provide. In this work, we leverage exhaustive navigation strategies for small configuration spaces.

Sampling strategies. Given a DCQA process, the goal of a sampling strategy is to systematically sample the configuration space. Depending on the QA task, sampling can be based either on random selection or on some QA task-specific criteria, such as maximizing the "coverage" of program input combinations. In Chapter 5, we present various sampling strategies for functional testing of software systems and evaluate them against exhaustive and random selection strategies.

Observation-based space reduction strategies. Given a DCQA process, an observation-based space reduction strategy aims at reducing the whole configuration space to a manageable yet relevant subspace (in terms of the process goals). Observation-based space reduction strategies, unlike sampling strategies, first identify subspaces that affect the QA task results the most by conducting "economical" experiments across the entire configuration space. They then focus only on these important subspaces to perform further and more detailed exploration. In Chapter 6, we introduce an observation-based space reduction strategy for performance-oriented regression testing of software systems.

Adapting the DCQA process. As subtasks are scheduled and executed in parallel at several sites, subtask results are collected at a central collection site. As the results come in, we analyze them on the fly to (1) provide useful information to developers (*e.g.*, compact descriptions of failing subspaces) and (2) modify the DCQA process in ways that improve the process performance. The performance of DCQA processes can be improved by learning from earlier subtask results. For example, when some subspaces prove to be faulty why not refocus time and resources on other, unexplored parts of the configuration space. For such dynamic behavior, we introduce *adaptation strategies*. To make better use of time and resources, an adaptation strategy monitors the global process state, analyzes it, and modifies future subtask assignments accordingly. Chapter 3 introduces generic adaptation strategies for functional testing.

Evaluating the approach. With all the characteristics of the problem domain given so far in mind, we have designed and implemented the Skoll infrastructure. We have then developed and evaluated novel DCQA processes for both functional and performance testing of software systems. For each DCQA process, we have developed configuration and control models, navigation and adaptation strategies (as needed), and analysis tools. We have empirically evaluated them by conducting experiments on real, large-scale (*i.e.,* 2MLOC+) software systems.

In Chapter 4, we present the results of several initial feasibility studies where we applied Skoll for functional testing of a large, widely-deployed middleware system over its numerous configurations. One QA task implemented in these studies

was to determine which specific configuration options and their settings are highly correlated with the manifestation of failures. We call this *fault characterization*. We obtained fault characterizations by exhaustively testing the configuration space and feeding the results to a classification tree analysis. The output is a model describing the options and settings that best predict failures. In these studies, among other things, we observed that fault characterization models helped developers quickly pinpoint the root causes of failures, leading to a much quicker turn-around time for bug fixes. While we were pleased with these results, the fundamental downside of this approach was that we had to test the entire configuration space.

In Chapter 5, we examine an alternative navigation strategy for fault characterization. The idea is to systematically sample the configuration space test only the selected configurations, and compute fault characterizations on the resulting data. Our experiments suggest that this approach is as nearly accurate as that based on exhaustive testing, but is much cheaper; it provided a 50% to 99% reduction in the number of configurations to be tested.

In Chapter 6, we focus on performance-oriented regression testing and present an observation-based space reduction strategy. As highly-configurable performance-intensive software systems change, developers often run benchmarking regression tests across the entire configuration space to detect unintended side effects of these changes on system performance. On the other hand, time and resource constraints can severely limit the number of configurations that can be benchmarked and bad choices of these configurations can cause performance degradations to escape detection. We introduce a DCQA process called *main effects screening process*. The

main effects screening process aims at monitoring performance degradations across large configuration spaces as a result of system changes. In this process, we first identify configuration options that affect the system performance the most. We refer to these options as *important options*. From hereafter, whenever the underlying system changes, we systematically explore only the important options–effectively reducing the configuration space to just these few options–to obtain reliable estimates of the system performance across the entire configuration space. We then use the performance estimates to detect performance degradations. Our empirical evaluations suggest that the main effects screening process reliably and precisely detects key sources of performance degradations with significantly less effort compared to conventional techniques.

The remainder of this thesis is organized as follows: Chapter 2 discusses the related work; Chapter 3 introduces the DCQA vision and the Skoll project; Chapter 4 presents a set of initial feasibility studies as well as the fault characterization process; Chapter 5 revisits the fault characterization process and introduces sampling strategies to improve the efficiency of the process; Chapter 6 introduces the main effect screening process; Chapter 7 presents a study where we demonstrate the generality of the Skoll infrastructure and implementation; and Chapter 8 presents the concluding remarks.

Chapter 2

Related Work

This chapter discusses the literature relevant to our research.

2.1 Current DCQA Approaches

Distributed, continuous quality assurance is not a completely new idea. There are some DCQA approaches already in use. Here, we briefly discuss these approaches and describe their major limitations on which Skoll tries to improve.

Online crash reporting systems, such as Netscape Quality Feedback Agent [58] and Microsoft XP Error Reporting [55], gather system state whenever a system crashes. This simplifies user participation in QA by automating problem reporting. These approaches however has a very limited scope, performing only a very small fraction of typical QA activities (*e.g.*, they can be used only when software crashes.)

Many well-known projects, such as GNU GCC [36], CPAN [23], Mozilla [57], VTK (The Visualization Toolkit) [75], and ACE+TAO [73], distribute test suites that end-users run to evaluate installation success. Users can, but quite often don't, return the test results to the developers. One limitation of this approach is that the process is undocumented. Developers have no record of what was tested or how it was tested or what the results were. We believe that a great deal of useful information is lost this way.

Auto-build scoreboards are distributed testing tools that allow software to be built/tested at multiple internal/external sites on various platforms (*e.g.*, hardware, operating systems, and compilers). The Mozilla and ACE+TAO projects use these systems to track build results across various platforms. Bugs are reported via the Bugzilla [71] issue tracking system, which provides inter-bug dependency recording, advanced reporting capabilities, extensive configurability, and integration with automated software configuration management systems, such as CVS [24]. While these systems help with documenting the QA process, the decision of what to test is left to end users. Unless developers can control at least some aspects of the QA process, important gaps and inefficiencies creep in.

The VTK project uses an auto-build scoreboard system called Dart [27]. Dart supports a continuous build and test process that can start whenever repository checkins occur. Users install a Dart client on their platform and use this client to automatically check out software from a remote repository, build it, execute the tests, and submit the results to the Dart server. One major limitation of this system is that the underlying QA process is hardwired. Other QA processes or other implementations of the build and test process are not supported easily. Once started, the process doesn't change. It cannot exploit incoming results nor avoid newly discovered problems, which leads to wasted resources and lost improvement opportunities.

Pavlopoulou *et al.* introduce residual test coverage monitoring [62], aiming at monitoring the actual use of software in the field to validate and refine the models that developers have relied upon in their QA efforts. In particular, they argue that

monitoring the "residue" of test coverage criteria, *i.e.*, entities not covered by in-house testing, can be used to validate the thoroughness of testing. This approach monitors only the parts of the software which are not explored in in-house QA activities.

Orso *et al.* present the GAMMA system for remote monitoring of software systems [61]. Unlike the residual monitoring, the GAMMA system can be used to monitor any part of the software. Given a monitoring task (*e.g.,* monitoring statement coverage), the GAMMA system divides the task into several subtasks (*e.g.,* monitoring one statement) and distributes them across the fielded instances of the software. This is done to achieve monitoring in a low-impact and minimally-intrusive manner. The results are collected at a central site and then combined to construct the global monitoring information. Bowring *et al.* present case studies where they used the GAMMA system to monitor branch coverage information of a real but very small program (*i.e.,* 6,200 lines of code) [6]. The results of these studies suggest that accurate monitoring information can be obtained by distributing the monitoring task across the deployed instances of software.

Orso *et al.* then leverage monitoring information collected in the field to support and improve software maintenance activities for evolving systems [60]. In particular, they address impact analysis activities (*i.e.,* estimating the parts of the software that can be affected by proposed changes) and selective regression testing activities (*i.e.*, selecting test cases from a given set of regression tests, which are deemed to be tested to validate that changes are correct and do not adversely affect other portions of the software). Their feasibility studies suggest that monitoring

data obtained from the field can be used to improve the accuracy and effectiveness of impact analysis and selective regression testing compared to using in-house data. Moreover, they argue that in-house monitoring data may not reflect the actual use-case scenarios of software systems.

Ernst *et al.* introduce a technique and a system called Daikon to dynamically discover program invariants [32, 33]. Invariants are identified in a 3-step process: (1) the software under study is instrumented to trace program variables of interest (*e.g.,* u, x, y, z), (2) the instrumented program is executed over a representative set of test cases and values of the variables of interest are collected, and (3) invariants (*e.g.,* $x < y$, $z > 10$, $u = 1$) are inferred over both the traced variables and derived variables (*e.g.,* $y - x > 1$). They argue that program invariants can help developers identify program properties that must be preserved across software changes. In [34], Ernst *et al.* present several approaches to (1) make invariants more useful for developers by improving the relevance of reported invariants and (2) compute invariants faster. Ernst *et al.* use in-house resources to identify program invariants.

Partly inspired by Ernst's work, Liblit *et al.* explore the possibility of identifying root causes of failures by gathering a little bit information from every run of fielded instances of software. Liblit *et al.* sample the number of observations of each of a very large, but fixed set of predicates (*e.g.,* in a particular execution how many times the predicate $x > y$ was true). For Liblit, the execution profile of a run is a list of predicates, each of which is augmented with an integer number showing that how many times the predicate was true. Liblit's predicate representation and computation is similar to that of Ernst's invariant. However, Liblit uses them for

a different purpose (*i.e.,* identifying causes of failures). Execution profiles are collected from fielded software instances by sampling predicates in a fair and uniformly random manner. This sampling strategy allows Liblit to reduce performance degradations that end-users may experience without compromising the applicability of the approach. Then statistical analysis techniques are used to explore the differences between passing and failing execution profiles. The outcome is a set of predicates that are strongly correlated with the manifestation of failures. The case studies in [49] suggest that this approach can help developers pinpoint the root causes of failures. In [51], Liblit *et al.* claim that their approach is scalable and technically feasible.

Some software products such as Netscape Quality Feedback Agent and Microsoft XP Error Reporting Agent (as we described above) can recognize certain subsets of their own runtime failures and with the user's permission can report the failures via Internet to developers. These reports often include the execution profile of the software at the time of failure (*e.g.,* variable values, stack traces, current program counter, and register values, etc.). Podgurski *et al.* propose an automated support for classifying failure reports using supervised and unsupervised pattern classification and multivariate visualization techniques [63]. The idea here is to classify the failure reports into disjoint groups such that failures within a group are more likely to be caused by the same or similar bugs compared to failures across groups. They argue that the resulting classification can help developers (1) identify the root causes of failures, (2) reduce the number of reports that needs to be examined; diagnosing at least one failure report from a class can provide developers with

confidence that other reports in the same class are caused by the same or similar bugs, and (3) prioritize the failure reports in terms of their frequency of appearance and severity. The feasibility studies presented in [63] suggest that this approach can be quite effective.

Although existing distributed QA approaches help to improve the quality of software, they have significant limitations. Many of these systems have limited **scope** – e.g., they can be used only when software crashes. General QA support needs to be much broader. Another problem is that many of these systems fail to **document** the QA activities that have been performed. It is therefore usually impossible to determine the full extent of (or gaps in) the QA process. These systems give developers little or no **control** over the QA process. Typically users decide (often by default) what aspects of the system they will examine; so some configurations are evaluated multiple times, others not at all. Finally, these approaches do not automatically **adapt** to or learn from the QA task results obtained from other users. The result is an opaque, inefficient, and *ad hoc* QA processes. The Skoll project aims at improving these limitations.

2.2 Factor-Covering Designs for Interaction Testing

For a given software system, the set of all possible test scenarios is often too large to test exhaustively. Consider a simple graphical user interface (GUI) with 10 input fields, A through J, each of which has 4 possible input values, say 0 through 3. Exhaustively testing this GUI then requires $4^{10} = 1,048,576$ test cases, which may

be infeasible or simply undesirable under circumstances where the cost of executing a single test case is high and time and resources are limited. This is yet a very conservative example: Screens with 50 or more fields are not uncommon [29]. Note that, in these examples, the number of the possible test cases grows exponentially in the number of fields. Therefore, techniques for generating efficient and effective set of test cases that can be used as a means of verifying the correct operation of programs are of great importance to software practitioners.

One common solution approach to the problem given above is to test each input field one at a time by trying out all possible values of the current field while defaulting (or randomizing) other fields [29]. We refer to this approach as the one-factor-at-a-time approach. For our simple example, the one-factor-at-a-time approach requires us to test 31 cases. While 31 is a reasonable number of test cases, this testing strategy can miss some possibly important combinations of field values. For instance, assuming that the default values of A and B are 0, this strategy may not cover a test case in which $A = 1$ and $B = 1$.

Factor-covering designs [53, 9, 25, 11, 29, 48] are used to create a test plan that covers all pair-wise, triple, or t-way combinations of factor settings for user defined values of t. Here t is called the strength of the design. A "factor" denotes a system test parameter (*e.g.,* system's configuration option, user input, and workload parameter) that determines the system's test scenarios. Furthermore, each factor is assumed to have values from a discrete set of possible values. Back to our simple example, by using factor-covering designs, it is possible to create a test plan for 10 factors (*i.e.,* fields) each with 4 possible settings, which covers all pair-wise combi-

nations of factor settings (*i.e.,* field values) in only 25 test cases [21]. Compared to 31 test cases generated by the one-factor-at-a-time approach, this new test plan covers all 2-way combinations of field values and yet requires fewer test cases.

In the literature, two types of mathematical objects and their variations have been extensively used to create factor-covering designs. These objects are called *orthogonal arrays* and *covering arrays.*

For a given set of factors and their settings, a strength t orthogonal array is a test plan in which all t-way combinations of factor settings are covered *exactly once* (or in general exactly the same number of times) [38]. Mandl *et al.* use a variation of orthogonal arrays to test enumerated types in ADA compiler software [53]. Brownlie *et al.* later develop the orthogonal array testing system (OATS) [9]. Their empirical studies suggest that orthogonal arrays are effective in fault detection and provide good code coverage.

Cohen *et al.* argue that the coverage criterion of orthogonal arrays (*e.g.,* covering every pair of factor settings *exactly* once) are too stringent to perform factor-covering testing [18]. This criterion makes it hard to construct orthogonal arrays. In fact, for a given set of factors and settings it may not be possible to have an orthogonal array. For instance, no orthogonal array exists when there are 6 factors each with 7 levels of settings [18]. Cohen *et al.* then propose a new mathematical object called a *covering array.* A strength t covering array covers all t-way combinations of factor settings *at least once* [18] (See Section 5.1 for more details). Relaxing the coverage criterion can also reduce the size of the test plan required to cover t-way combinations. For example, for 126 two-level factors, a

covering array can cover all pair-wise combinations in only 10 test cases whereas an orthogonal array would require 128 test cases [29]. Dalal *et al.* show that for a fixed strength t, orthogonal arrays grow at least linearly with the number of factors while covering arrays grow only at a logarithmic rate [26].

Dalal *et al.* argue that testing all pairwise input combinations for a software system using covering arrays finds a large percentage of existing faults [25]. Dunietz *et al.* report on their experiences with using covering arrays for attaining code coverage. They argue that even low-strength covering arrays (*i.e.,* strength 2 or 3) can provide very high block coverage. However, attaining high path coverage may require higher strength covering arrays (*i.e.,* strength 4 or higher) [29]. In further work, Burr *et al.*, Clarke [17], and Kuhn *et al.* provide more empirical results on real software systems (ranging from real-time communication systems to graphical user interface applications), showing that factor-covering designs can be quite effective in software testing [11, 17, 29, 48]. For example, Burr *et al.* report an experiment conducted on Nortel's internal e-mail system, in which they were able to cover 97% of the branches with less than 100 test cases created by factor-covering designs, as opposed to 27 trillion exhaustive test cases [11].

Cohen *et al.* introduce the AETG system which generates covering arrays for a tester-specified set of factors, their settings, and constraints among them [19]. The AETG system uses an iterative greedy algorithm to construct covering arrays. Assuming that there are r already selected test cases, this algorithm first generates a set of candidate test cases for the $r + 1$ test case and then chooses the one that covers the most new pairs. This processes continues until a set of test cases which

satisfies the properties of covering arrays is found. Cohen, M. *et al.* argue that search-based algorithms (*e.g.,* hill climbing and simulated annealing) for creating covering arrays provide smaller sized arrays compared to greedy algorithms [21].

Cohen, M. *et al.* introduce a new factor-covering design called *variable strength covering arrays*, which allows one to vary the strength of the array across the factor space [22] (See Section 5.1 for more details). Note that regular covering arrays fix the strength t across the space. In [20, 22] Cohen, M. *et al.* present a discussion, providing scenarios of when variable strength arrays might be useful. They provide a model to define VSCAs and present a construction technique, however they have not provided any evidence of their effectiveness. We do not know of any studies to date (except the one presented in Chapter 5) that provide empirical results comparing variable strength arrays with their fixed level counterparts.

Covering arrays have been used most frequently to test input combinations of programs [25, 11, 29, 48]. These studies focus on finding unknown faults in already tested systems and equate covering arrays with code coverage metrics [19, 29]. In Chapter 5, we use covering arrays to systematically sample complex configuration spaces. Our approach is different in that we apply covering arrays to system configuration options and asses their effectiveness in both revealing failures and finding failure inducing options.

2.3 Fault Localization

Fault localization aims at helping developers efficiently and effectively identify the root causes of failures. This section discusses the literature on fault localization.

Software systems must often obey many rules for both correctness and performance. One example of such a rule is: A call to `lock(semaphore)` must be paired with a call to `unlock(semaphore)`. Engler *et al.* propose a static analysis technique to automatically check manually written system rules by using simple, system-specific compiler extensions [30]. They argue that simple rule checkers can be very effective in finding real bugs in real software systems, such as Linux and FreeBSD. Later, in [31], they show that how these rule instances can be derived automatically by inferring programmers' "belief" from the source code and cross-checking it for contradictions. Xie *et al.* introduce several rules to check redundancies in system code and argue that redundancies are often correlated with real bugs [80]. [45] proposes a technique to rank error reports emitted by static program analysis tools, which can help developers avoid false positives. Hovemeyer *et al.* leverage similar ideas together with various static program analysis techniques to find bugs in Java programs [41].

Dickinson *et al.* use machine learning techniques over execution profiles (*e.g.,* function call profiles) of programs to identify executions that are most likely to be faulty [28]. Faulty executions are further clustered such that similar execution profiles are placed in the same cluster while dissimilar profiles are placed in different clusters. This is done to reduce the number of executions that needs to be examined

for conformance to functional requirements. Since faulty executions in a given cluster present similar behaviors, several executions from each cluster, instead of all of them, can be selected and evaluated against requirements. They also provide several sampling strategies for selecting subset of executions from a cluster. In a sense, Dickinson *et al.* localize failures to execution profiles.

Brun *et al.*, like Dickinson *et al.*, use machine learning techniques over program executions [10]. However, they aim at identifying program properties (not program executions as discussed in [28]) that are highly correlated with the manifestation of failures. Program properties, such as "pointer `x` is never equal to `null`", are identified over both failing and passing runs of the program using Daikon [32] and a decision tree learning algorithm is used to identify failure inducing properties. Several feasibility studies in [10] suggest that identifying failure inducing properties can lead developers to the actual causes of failures.

Both Dickinson *et al.* and Brun *et al.* compute fault localization information via off-line analysis of execution profiles. Hangal *et al.* introduce the DIDUCE system, an on-line invariant detector and checker system for Java programs, to locate the root causes of failures by checking violations of dynamically discovered program invariants [37]. DIDUCE instruments a given program P and maintains invariants on the values of a set of tracked expressions. DIDUCE is trained by running P with a representative set of inputs, where for each tracked expression, an invariant hypothesis is constructed that satisfies all the values observed in the history of the execution so far. In the training phase, current hypotheses are relaxed to accommodate new values. Whereas, in the checking phase where DIDUCE is used

to identify the root causes of failures the violations of the hypotheses are reported as an indication of the root causes of failures. Feasibility studies presented in [37] suggest that this approach can be very effective. DIDUCE, unlike Daikon [10], does not operate on a fixed set of invariant specifications. It can learn new program invariants and/or relax already known invariants at run time as previously unknown cases are encountered.

Abramson *et al.*, instead of analyzing the whole execution profiles via machine learning techniques, propose to monitor and visually compare the program states (*e.g.*, contents of critical data objects) at some check-points (*e.g.,* before and after function calls) between the correct version of a program and its modified, faulty version [4, 3]. The feasibility studies conducted on scientific applications in [2] suggest that this approach can help developers locate bugs.

Zeller *et al.* introduce the *delta debugging algorithm* to isolate differences between passing and failing program inputs [81, 83]. Given a failing program input, the delta debugging algorithm algorithmically and iteratively simplifies the input using a binary search-like algorithm. At each iteration, a portion of the input is discarded according to the algorithm, and the program under study is fed the current input. Depending on whether the program fails or passes with this input, the algorithm determines which portions of the input should be examined in the next iteration. The algorithm stops when a minimal program input is found, in which removing any single input entry would cause the failure to disappear. [39] presents feasibility studies where delta debugging is successfully used to identify failure inducing HTML tags that causes Mozilla, a web browser, to crash.

In [82], Zeller *et al.* present an application of delta debugging algorithm to isolate cause-effects chains (*i.e.,* variables and values that cause failures) to help developers understand and locate the root causes of failures. The fundamental difference between delta debugging and other dynamic analysis-based approaches examined in this section is that the latter ones do not have any control over the fault localization process; they do not decide on the next QA task that needs to be performed. They assume that a large data set is available for analysis. On the other hand, delta debugging algorithmically decides what to test at each iteration based on the results of previous test results.

In Chapter 4 and 5, we introduce a fault characterization process to identify configuration options and their settings which are responsible for the manifestation of failures in complex configuration spaces. For a given configuration space, this process systematically samples the configuration space, tests only the selected configurations, and analyzes the resulting data to identify failure inducing option settings. In a sense, our approach is similar to the delta debugging approach; we decide what needs to be tested to reach our process goals. However, our approach is different in that we simultaneously analyze the causes of multiple failures whereas the delta debugging algorithm works with one failure at a time.

2.4 Performance Evaluation and Optimization

Performance-intensive software such as quality of service (QoS) enabled component middleware implementations, provides a range of configuration options that

can be used to customize and tune the QoS properties of the middleware. For example, the ACE+TAO system, a QoS enabled middleware system, provides ~500 configuration options that can be used to tune its behavior. This large number of configurations create problems for developers and users of performance intensive software to generate and validate configuration settings that maximize QoS.

Large-scale performance benchmarking testbeds. To evaluate key QoS characteristics of performance-intensive software, QA engineers today often handcraft individual QA tasks (*e.g.*, benchmarking experiments) by writing (1) interface definitions, *e.g.*, modeling the data exchange format between clients and servers, (2) component implementations, (3) test applications, *e.g.*, measuring key performance metrics, such as round-trip request latency, jitter, and throughput, and (4) scaffolding code, *e.g.*, scripts needed to startup daemons, initialize the software infrastructure and applications, run experiments, generate results, and tear down the experiment. Manually implementing these steps is tedious and error-prone since each step may be repeated many times for every QA experiment. Furthermore, in a handcrafted approach, QA engineers visualize experiments via application source code, which provides an excessively low level of abstraction.

BGML [46] is a model-driven benchmarking tool that allows component middleware QA engineers to (1) visually model interaction scenarios between configuration options and system components using domain-specific building blocks, *i.e.*, capture software variability in higher-level models rather than in lower-level source code, (2) automate benchmarking code generation and reuse QA task code across

configurations, (3) generate control scripts to distribute and execute the experiments across a grid of computers to monitor QoS performance behavior in a wide range of execution contexts, and (4) enable evaluation of multiple performance metrics, such as throughput, latency, jitter, and other QoS criteria.

EMULab [76] is a testbed that provides an environment for experimental evaluation of networked systems. EMULab provides tools that researchers can use to configure the topology of their experiments, *e.g.*, by modeling the underlying OS, hardware, and communication links. This topology is then mapped to ~250 physical nodes that can be accessed via the Internet [67]. The EMULab tools can generate script files that use the Network Simulator (NS) [74] syntax and semantics to run the experiment. The BGML tool can also generate NS scripts to integrate performance benchmarks with experiments in EMULab.

Feedback-driven performance optimization techniques. Traditional feedback-driven performance optimization techniques can be divided into the following categories: offline analysis techniques, online analysis techniques, and hybrid analysis techniques.

Offline analysis techniques leverage program analysis techniques to improve compiler-generated code which in turn improves system performance. Yotov *et al.* introduce the ATLAS numerical algebra system [43]. ATLAS uses an empirical optimization engine to decide the values of optimization parameters by generating different program versions that are run on various hardware/OS platforms. The output from these runs are used to select parameter values that provide the best

performance. Yotov *et al.* also compare their empirical optimization technique to a model-driven optimization technique. They argue that modeling complex systems (or even very simple ones) are difficult and error-prone and inaccurate models may reversely affect the optimization process. They claim that, in such cases, empirical optimization techniques may provide more reliable results.

The ATLAS system uses the one-factor-at-a-time approach (Section 2.2) to optimize system performance. The limitations of this approach are well-known and severe [78]: (1) optimal settings of factor settings can be missed, (2) interaction effects of factors can be overlooked, and (3) the conclusions reached from its analysis are not general (because of a lack of systematic search over factor settings).

Online analysis techniques leverage feedback control loop to dynamically adapt system performance at run time. Zhang *et al.* introduce the ControlWare middleware [84], which uses feedback control theory by analyzing the architecture and modeling it as a feedback control loop. Actuators and sensors then monitor the system and affect server resource allocation. Real-time scheduling based on feedback loops has also been applied to Real-time CORBA middleware [52] to automatically adjust the rate of remote operation invocation transparently to an application.

Hollingsworth *et al.* introduce the Active Harmony system which aims at improving systems' performance by reconfiguring them at runtime [40, 1, 72]. During the execution of a system, the system performance is monitored and the simplex algorithm [42] is used to find the system configuration that maximizes the system performance. Chung *et al.* apply the Active Harmony system to automate performance tuning of a wide range of software systems [14, 15]. They argue that

the Active Harmony system can adapt software systems to different workloads and improve their performance.

A fundamental downside of this approach is that since the optimal software configuration is found by varying and monitoring various configurations at runtime (*i.e.*, via the simplex algorithm), the system performance may be sacrificed until the algorithm converges to an optimal configuration. To address this issue, in [16], Chung *et al.* present a sensitivity analysis technique to speed up the convergence of the simplex algorithm by leveraging information from prior runs of the system under study.

Hybrid analysis techniques combine aspects of offline and online analysis techniques. Childers *et al.* introduce the continuous compilation strategy [13]. This strategy constantly monitors and improves application code using code optimization techniques. These optimizations are applied in four phases including (1) *static analysis*, in which information from training runs is used to estimate and predict optimization plans, (2) *dynamic optimization*, in which monitors apply code transformations at run-time to adapt program behavior, (3) *offline adaptation*, in which optimization plans are actually improved using actual execution, and (4) *recompilation*, where the optimization plans are regenerated.

Randy *et al.* present the Autopilot system [65]. The autopilot system provides a set of performance sensors, decision procedures, and policy actuators to perform adaptive tuning of software systems. The decision procedures (*i.e.*, when and how to reconfigure the system to improve its performance) are often realized as a pre-computed table. In this table, each row represents a rule consisting of two parts:

a condition clause and an action clause. The condition clause describes the system state in which the action in the action clause should be taken. The action clause is often realized as system reconfiguration parameters. The autopilot system, by using a scalable performance analysis environment [64], continuously monitors the system state (*e.g.*, system inputs and available resources), and (if needed) reconfigures it at runtime by conducting a simple table look-up operation. A limitation of this approach is that no information is provided on how to compute such decision tables.

In Chapter 6, we introduce a DCQA process for performance-oriented regression testing of highly-configurable software systems. This process aims at monitoring performance degradations across large configuration spaces as a result of system changes. As far as we are aware, no prior study has ever addressed the performance-oriented regression testing of software systems.

Chapter 3

The Skoll Project

To overcome the shortcomings of in-house QA activities, we have initiated the Skoll project. The Skoll project is developing and validating novel software QA processes and tools that leverage the extensive computing resources of potentially worldwide user communities in a distributed, continuous manner. The Skoll approach can also be usable over a company intranet.

The Skoll project aims at providing developers with access to (1) wide range of platforms (*e.g.,* hardwares, operating systems, compilers, etc.), which may not be available in house, (2) more resources, which can allow large scale QA activities to be performed and make rare events (*e.g.,* intermittent failures) easier to see, (3) thorough, transparent, managed, and adaptive QA processes, and (4) user profiles, which can further shape software development activities by revealing how systems are used in the field.

We define a distributed, continuous quality assurance (DCQA) process as one conducted around-the-world, around-the-clock on a powerful, virtual computing grid provided by thousands of end-user machines during off-peak hours.

The Skoll DCQA processes are distributed, opportunistic, and adaptive. They are distributed: Given a QA task we divide it into several subtasks each of which can be performed on a single user machine. They are opportunistic: When a user

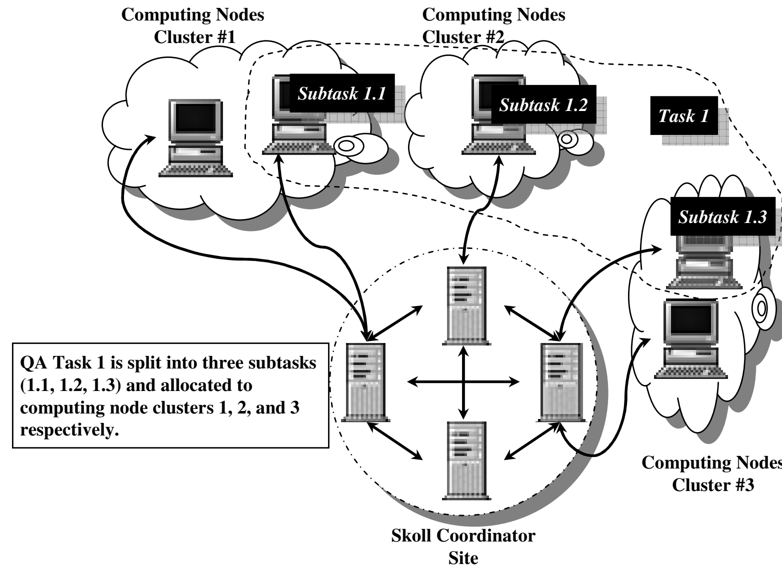


Figure 3.1: The Skoll architecture

machine becomes available we allocate one or more subtasks to it, collect the results when they are available, and fuse them together at central collection sites to complete the overall QA process. And finally they are adaptive: We use earlier subtask results to schedule and coordinate subtask allocation.

At a high level, Skoll processes resemble certain traditional distributed computations. General QA tasks are decomposed into many subtasks. As illustrated in Figure 3.1, these subtasks are then allocated to computing nodes, where they are executed. As subtasks run, control logic may dynamically steer the global computation for reasons of performance and correctness.

In Skoll, tasks are QA activities, such as testing, capturing usage patterns, and measuring system performance. They are broken down into subtasks, which perform part of the overall task. For example, a subtask might test a subset of system functions, monitor a subgroup of users, or measure performance under one

particular workload characterization. The subtasks in one of our feasibility studies in Chapter 4 do functional testing for a single, specific system configuration. The global process, by executing the right set of subtasks, does functional testing that "covers" the space of system configurations.

In Skoll, computing nodes are machines volunteered by end-users. These nodes request work from a server when they decide they are available. Ultimately, we envision Skoll processes involving geographically decentralized computing pools made up of thousands of machines provided by users, developers, and companies around the world. This environment will allow large amounts of QA to be performed at fielded sites using fielded resources, giving developers unprecedented access to user resources, environments, and usage patterns.

In the remainder of this chapter, we present the Skoll infrastructure and implementation.

3.1 Skoll Infrastructure

Skoll processes are based on a client/server model. Clients request QA jobs (QA subtask scripts) from a server that determines which subtask to allocate, bundles up all necessary scripts and artifacts, and sends them to the client. Realizing such a process presents challenges, *e.g.*; how tasks will be decomposed into subtasks; in what order will subtasks be allocated; how will they be implemented so they run on a very wide set of client platforms; how will results be merged together and interpreted; if and how should the process adapt to incoming results; and how

will the results of the overall process be summarized and communicated to software developers. We developed components and services to tackle these challenges.

3.1.1 Configuration and Control Model

A cornerstone of our approach is a formal model of the space of QA subtasks, called a *configuration and control model.* The configuration and control model defines how QA tasks are divided into several subtasks each of which can be executed at a single user machine. This information is used in planning the global QA process, for adapting the process dynamically, and to aid in interpreting results.

In the configuration model, subtasks are generic processes parameterized by *configuration options.* Currently, each option takes its value from a discrete number of settings. Configuration options capture the information (1) that will be varied under the control of the DCQA process or (2) that will be needed by the underlying software to build and execute properly. Such options are application specific and may not be restricted to traditional software configuration options only (*e.g.,* cache size, thread pool size, etc.) Any aspect of QA tasks and the underlying software system that will be varied under the control of the DCQA process can be represented as configuration options in the configuration and control model. Program inputs, workload parameters, test cases, and environmental conditions are among the examples of such aspects. This allows Skoll to be applicable to a wide range of software systems and QA tasks. In Chapter 7, we demonstrate the generality of the configuration and control model via an empirical study.

Defining a subtask then involves mapping each option to one of its allowable settings. We call this mapping a *configuration* and represent it as a set { (V_1, C_1), (V_2, C_2), ..., (V_N, C_N) }, where each V_i is a configuration option and C_i is its value, drawn from the allowable settings of V_i.

In practice not all configurations make sense (e.g., feature X not supported on operating system Y). We therefore allow *inter-option constraints* that limit the setting of one option based on the setting of another. We represent constraints as $(P_i \rightarrow P_j)$, meaning "if predicate P_i evaluates to $TRUE$, then predicate P_j must evaluate to $TRUE$." A predicate P_k can be of the form $false$, $true$, A, $\neg A$, $A \& B$, $A|B$, or simply $V_i = C_i$, where A, B are predicates, V_i is an option and C_i is one of its allowable values. A *valid configuration* is a configuration that violates no inter-option constraints.

Table 3.1 presents some sample options and constraints taken from one of our feasibility studies in Chapter 4. The sample options refer to things like the end user's compiler (COMPILER); whether to compile in certain features (AMI, CORBA_MSG, MINIMUM_CORBA); whether certain test cases are runnable in a given configuration (run(T)), and at what level to set a run-time optimization (ORBCollocation). One sample constraint shows that minimum CORBA specification does not support asynchronous method invocations (AMI). The other shows that a particular test can only run on a platform that uses a particular compiler, namely the SUN CC compiler version 5.1.

Option	**Settings**	**Interpretation**
COMPILER	{gcc2.96, SUNCC5_1}	compiler
AMI	{1 = Yes, 0 = No}	Enable Feature
CORBA_MSG	{1 = Yes, 0 = No}	Enable Feature
MINIMUM_CORBA	{1 = Yes, 0 = No}	Enable Feature
run(T)	{1 = True, 0 = False}	Test T runnable
ORBCollocation	{global, per-orb, NO}	runtime control
Constraints		
AMI = 1 → MINIMUM_CORBA = 0		
run(Multiple/run_test.pl) = 1 → (COMPILER = SUNCC5_1)		

Table 3.1: Some options and constraints.

3.1.2 Intelligent Steering Agent (ISA)

A distinguishing feature of Skoll is its use of an *intelligent steering agent* (ISA) to control the global QA process. The ISA controls the global process by deciding which valid configuration to allocate to each incoming Skoll client request. Once the valid configuration is chosen, the ISA packages the corresponding QA subtask implementation, consisting of the configuration parameters, build instructions, and QA-specific code (*e.g.*, regression/performance tests) associated with a software project. This package is called a *QA job*.

Skoll's formal configuration and control model lets us cast configuration selection and implementation as a planning problem. This problem requires automated constraint solving, scheduling, and learning. Consequently, we implemented the ISA using planning technology.

Given an *initial state*, a *goal state*, a set of *operators* (specified in terms of parameterized preconditions and effects on variables), and a set of *objects*, the ISA planner (currently Blackbox [44]) returns a set of actions (or commands) with ordering constraints that achieve the goal. In Skoll, the initial state is the base subtask

configuration. The base subtask configuration includes any option settings that the ISA must not modify (e.g., the client machine's OS). The goal state describes the desired configuration. The operators encode all the constraints, including those resulting from previously run subtasks. The output is a QA job.

For many planning problems, a single plan is sufficient. For Skoll, however, we need to generate many or even all acceptable plans (*i.e.*, subtask implementations). We therefore modified the Blackbox planner so that it can iteratively generate all acceptable plans. We also added a parameter to the ISA by which each acceptable plan is generated exactly once (*random selection without replacement*) or zero or more times (*random selection with replacement*).

Going back to our sample options in Table 3.1, if the process designer wants to ensure that all software configurations compile cleanly, he/she would use a configuration and control model without the test case- or runtime-specific options and would instruct the ISA to generate plans using the random selection without replacement strategy (each valid configuration is generated exactly once). If on the other hand the task were to capture performance measures on a wide variety of user machines, then the process designer might use all available options and have the ISA use the random selection with replacement strategy (which could generate specific valid configurations more than once).

The ISA's default behavior is to allocate subtasks upon request. No effort is made to optimize or adapt the global process. When more dynamic behavior is desired, process designers must develop application-specific navigation and adaptation strategies.

Navigation manager. The default behavior of the ISA, as explained above, is to exhaustively test the entire configuration space. On the other hand, the configuration space of a DCQA process can be quite large. Take a system with only 30 binary configuration options as an example. The configuration space of this system contains more than one billion valid configurations. Even with a QA task, spending only one second to test each configuration, it would take more than 30 years of machine time to test them all. As this example illustrates, for large configuration spaces, even with a large pool of user-supplied resources, brute-force QA approaches, such as exhaustive testing, may be infeasible or simply undesirable.

When exhaustive testing is undesirable, process designers must develop customized *navigation strategies*. Given a DCQA process, a navigation strategy explores the configuration space and schedules only a subset of the configurations for allocation in ways that do not compromise the accuracy and the goals of the DCQA process. For example, consider a DCQA process for functional testing of a software system across its numerous configurations. If $n+1$-way (and above) interactions of configuration options are considered negligible, then a navigation strategy that selects a minimal set of configurations in which all the n-way combinations of option settings are covered may be good enough to ensure the system correctness at a fraction of the cost compared to exhaustive testing.

The navigation manager registers navigation strategies. In the presence of a registered navigation strategy, for each incoming QA job request, the ISA requests the navigation strategy to pick a valid QA subtask for the requesting client. Once the subtask is selected, the ISA creates the corresponding QA job and sends it out

to the requesting client.

Adaptation manager. As QA subtasks are performed, their results are returned to the ISA. By default, the ISA ignores these results. Often, however, we want to learn from incoming results. For example, when some configurations prove to be faulty, why not refocus resources on other unexplored parts of the configuration space. When such dynamic behavior is desired, process designers must develop customized *adaptation strategies*. An adaptation strategy monitors the global process state, analyzes it, and modifies future subtask assignments in ways that improve process performance.

In Skoll, adaptation strategies are executed when subtask results arrive. As they must process subtask results, adaptation strategies may need to be tailored to each DCQA process. The adaptation manager registers adaptation strategies.

Next, we describe three generic adaptation strategies we used in the feasibility studies described in Chapter 4.

Nearest neighbor search: Suppose a test reports a configuration in which test cases are failing. Developers might want to quickly identify other similar configurations that pass or fail. The nearest neighbor strategy is designed to generate such configurations. For example, suppose that a test on a configuration space with three binary options fails in configuration $\{0, 0, 0\}$. The nearest neighbor search strategy marks that configuration as failed and records its failure information. It then schedules for immediate testing all valid configurations that differ from the failed one in the value of exactly one option: $\{1, 0, 0\}$, $\{0, 1, 0\}$ and $\{0, 0, 1\}$, *i.e.*, all

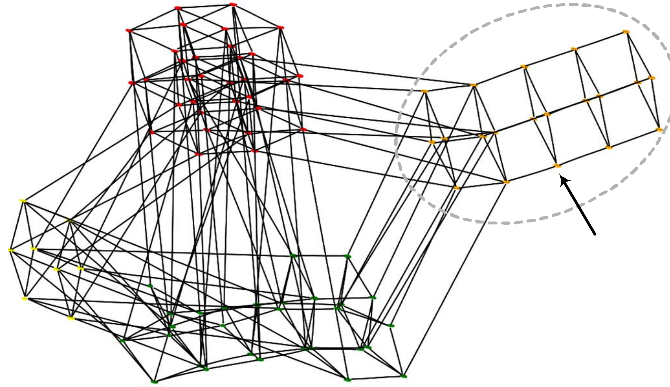


Figure 3.2: Nearest neighbor strategy.

distance one neighbors. This process continues recursively. Figure 3.2 depicts the nearest neighbor strategy on a configuration space taken from one of our feasibility studies. Nodes represent valid configurations; edges connect distance one neighbors. The dotted ellipse encircles configurations that failed for the same reason. The arrow indicates an initial failing node. Once it fails, its neighbors are tested; they fail so their neighbors are tested and so on. The nearest neighbor search strategy stops when nodes outside an ellipse are tested (since they will pass or fail for a different reason). The idea here is to quickly identify the shape of a failing subspace such as the one marked with the ellipse in Figure 3.2. As the shape of a failing subspace becomes statistically apparent, temporary constraints can be introduced to the configuration and control model to avoid exploring this subspace to save time and resources for untested configurations.

Temporary constraints: Suppose that a software fails to build whenever binary option AMI = 0 and binary option CORBA_MSG = 1. Note that the failing subspace for this case comprises up to 25% of the entire configuration space. In this

situation, developers would obviously want to prevent continued testing of these configurations and would prefer to use their resources to test other parts of the configuration space. To make this possible we created an adaptation strategy that inserts *temporary constraints*, such as $\text{AMI} = 0$ & $\text{CORBA_MSG} = 1 \rightarrow \textit{false}$ into the configuration and control model. This excludes configurations with the offending option settings from further exploration. Once the problem that prompted the temporary constraints has been fixed, the constraints are removed, thus allowing normal ISA execution. These constraints can in fact be used to spawn new Skoll subtasks that test patches on only the previously failing configurations.

Terminate/modify subtasks: Suppose a test program is run at many user sites, failing continuously. At some point, continuing to run that test program provides little new information. Time and resources might be better spent running some previously unexecuted test program. This adaptation strategy monitors for such situations and, depending on how it is implemented, can modify subtask characteristics or even terminate the global process.

3.1.3 Analysis Manager

As subtasks are scheduled and executed in parallel at several sites, subtask results are collected at the Skoll server. As the results come in, we analyze them on the fly to (1) provide useful information to developers (*e.g.*, compact descriptions of failing subspaces) and (2) modify the DCQA process in ways that improve the process performance.

DCQA processes may require different analysis techniques (*e.g.,* data mining and statistical analysis techniques). The analysis manager allows application-specific analysis tools to be easily integrated with Skoll. We have found that analysis tools that are robust to imperfect, missing and noisy data are desirable in Skoll. This is because, in Skoll, analysis tools operate on data collected at various configuration points in the configuration space. Observations made at a given configuration point often behave as a random variable. Consider multiple readings of a software performance metric (*e.g.,* throughput) at the same configuration. The values that will be obtained cannot be predicted with certainty; they often have some degree of randomness and noise. Another characteristic of the collected data is that they often have missing data points, *e.g.,* inter-option constraints may prevent certain configurations from being tested.

3.1.4 Visualization Manager

The visualization manager helps developers organize and visualize large amount of process results. Although data presentation techniques play an important role for comprehension of data, they are not among the main focuses of this work. We included them in the infrastructure for completeness.

The fundamental challenge we faced during the design of Skoll is to provide an infrastructure that is able to support a wide range of software systems and DCQA processes. Almost all of the Skoll components, such as navigation/adaptation strategies and analysis/visualization tools should be able to be tailored to application-

specific needs. Consequently, we provided well-defined interfaces to the Skoll components. Furthermore, we leveraged database management systems to store critical process data. This helped us reduce the communication burdens among the Skoll components by providing a unified, easy-to-access data storage.

In the next section, we present the high level implementation details of Skoll components.

3.2 Skoll Implementation

We implemented Skoll as a client/server system. To ensure cross-platform compatibility, the Skoll system is written entirely in Perl and all communications between the Skoll server and clients are done in XML via the HTTP protocol (*e.g.,* via GET and POST methods).

In the rest of this section, we present high level implementation details by following a logical sequence of events that happen during the execution of a DCQA process.

3.2.1 Configuring Skoll

Skoll is designed to support a wide range of software systems and QA processes. Consequently, it requires reconfiguration at an acceptable level of effort before it can be used with a given software system and a QA process. Process designers are responsible for implementing and setting up the application-specific Skoll components.

```
interface ServerSideApplicationComponent {
  boolean init()
  Instructions QA_job(Configuration c)
  boolean finalize()
}
```

Figure 3.3: Server side application component interface.

As a first step, process designers should choose a QA task that will be executed by Skoll. Then, they should create a configuration and control model, defining how this task is divided into several subtasks. The interpretation and execution details of QA subtasks are application specific. Process designers provide this information by implementing two interfaces, namely *ServerSideApplicationComponent* (Figure 3.3) and *ClientSideApplicationComponent* (Figure 3.4) [1]. The details of these interfaces are given later in this chapter. Here, we briefly summarize them.

ServerSideApplicationComponent component is used at the Skoll server to assist the ISA to interpret QA subtasks and to create actual QA jobs. The Skoll server calls the *init* and *finalize* methods right before starting a new DCQA process and just after finishing up with one, respectively. ClientSideApplicationComponent component is used at the Skoll client to execute the QA jobs sent by the ISA. The *init* and *finalize* methods of this component is called before and after executing a QA job, respectively.

[1]Note that, for clarity purposes, we simplified the interfaces given in this chapter; in the actual implementation, these interfaces may be somewhat more complex.

```
interface ClientSideApplicationComponent {
  boolean init(QAJob job)
  InstructionResult dispatch_instruction(Instruction i)
  boolean finalize()
}
```

Figure 3.4: Client side application component interface.

3.2.2 Registering Skoll Clients

Once the Skoll server is started up, end-users register with the server via a web-based registration form. In return, they receive a Skoll client kit. This kit consists of a cross-platform compatible client software and a configuration template. The configuration template contains any user-specific information that may not be modified by the ISA when generating QA jobs. The template can be modified by end-users who wish to restrict the QA jobs they accept from the Skoll server.

3.2.3 Requesting QA Jobs

Once installed, the Skoll client periodically or on-demand requests QA jobs from the Skoll server. At each request, the Skoll client automatically detects its operating system (*i.e.,* OS version, kernel version, vendor, etc.), compiler (*i.e.*, version, patches, etc.), and hardware specifications (*i.e.,* CPU details, number of CPUs, memory sizes, etc.), packages them together with the configuration template into a QA job request message (QAJobReqMsg), and sends it to the ISA.

```
interface NavigationStrategy {
  boolean init()
  Configuration schedule_config_for(QAJobReqMsg msg)
  boolean finalize()
}
```

Figure 3.5: Interface between the ISA and navigation strategies.

3.2.4 Generating QA Jobs

The ISA responds to each incoming request with a QA job which is customized in accordance with the characteristics of the client platform. The ISA does this via a sequence of steps. As a first step, it checks if there are already scheduled configurations for allocation. As explained in Section 3.1.2, navigation and adaptation strategies can schedule configurations for future allocation. If this is the case, the first valid configuration for the requesting client is selected for allocation. If no scheduled configuration is found and there is a registered navigation strategy, the ISA requests the navigation strategy to select a valid configuration. Otherwise, the ISA selects a configuration using its default navigation strategy.

The interface between the ISA and navigation strategies is given in Figure 3.5. Application-specific navigation strategies should implement this interface and register with the navigation manager before they can be used with Skoll. The *init* and *finalize* methods are called when DCQA processes start and finish, respectively. The ISA requests a configuration via the *schedule_config_for* method by passing the QA job request message as an argument. The result is a valid configuration to be allocated to the requesting client.

Once a configuration (*i.e.*, QA subtask) is selected, the ISA consults to the

ServerSideApplicationComponent component via the *QA_job* method by passing the selected configuration as an argument. This method returns a set of instructions which will assist the client to carry out the assigned QA subtask. The ISA then packages these instructions and the selected configuration into a QA job (QAJob). A unique ID (QAJobID) is assigned to each QA job and stored in the Skoll database along with the QA job information.

Figure 3.6 shows an example QA job generated by the ISA. It consists of two main sections: a configuration section and an instruction section. Upon receiving this QA job, the Skoll client would download ACE+TAO v5.2.3 from a CVS repository located at cvs.doc.wustl.edu, configure it using the AMI and CORBA_MSG options, build the ACE+TAO system as well as a test case, run the test, parse the results and then upload the results to the Skoll server.

The Skoll client kit provides implementations for a set of generic instructions e.g., setting environment variables, downloading a software from a remote CVS repository, starting/stopping a log, running system commands, uploading a log file, etc. Each instruction is implemented as a separate component, which is in compliance with a common interface. This ensures that Skoll's default instruction set can easily be expanded. Furthermore, instruction components are loaded dynamically at runtime as needed, which allows the Skoll client to be upgraded with a new set of instruction components even after deployment.

```
<QAJob>
  <configuration>
    <option name=AMI val=0 />
    <option name=CORBA_MSG val=1 />
  </configuration>

  <instructions>

    <start_log target=all-activity.log />

    <download>
      <cvs>cvs.doc.wustl.edu</cvs>
      <module>ACE+TAO</module>
      <version>v5.2.3</version>
    </download>

    <configure />

    <build target=ACE />
    <build target=TAO />
    <build target=tests/HelloWorld />

    <run-test target=tests/HelloWorld />

    <stop_log />

    <parse target=all_activity.log />

    <upload_results />

  </instructions>
</QAJob>
```

Figure 3.6: An example QA job.

```
interface AdaptationStrategy {
  boolean init()
  Configurations adapt_to(QAJobID id)
  boolean finalize()
}
```

Figure 3.7: Interface between the ISA and adaptation strategies.

3.2.5 Executing QA Jobs

The Skoll client executes the set of instructions one by one in the order they are received. Any instruction, which is not in the default set of instructions supported by the Skoll client, is considered application-specific and passed to the ClientSideApplicationComponent component via the *dispatch_instruction* method (Figure 3.4). The client side application component is responsible for executing the instruction.

All client activities are stored into a log file (*e.g.,* "all-activity.log" in Figure 3.6). The log file consists of multiple sections where each section corresponds to an instruction executed by the client (*e.g.,* "download" and "build"). Once the QA subtask is completed, the client is often asked to parse the log file into an XML document, summarizing the QA subtask results.

3.2.6 Collecting QA Job Results

QA job results are collected and stored in a database at the Skoll server. Once the database is populated, the ISA is notified about the incoming results. The ISA may use this information to modify future subtask allocation via adaptation strategies as explained in Section 3.1.2.

Figure 3.7 shows the interface between the ISA and adaptation strategies.

As in the case of navigation strategies, the *init* and *finalize* methods are called once DCQA processes start and finish, respectively. The ISA notifies the registered adaptation strategies via the *adapt_to* method by passing the QA job ID as an argument. The adaptation strategies can then analyze the current state of the process and schedule configurations for future allocation.

3.2.7 Analyzing and Visualizing QA Job Results

The analysis and visualization of QA job results are application-specific. Depending on the characteristics of the QA task at hand and the preferences of developers, some analysis/visualization tools can be preferable over others. Therefore, Skoll provides a web-based portal to various analysis and visualization tools.

Chapter 4

Initial Implementation and Evaluation

The Skoll project's goals are ambitious. To help achieve them, we conducted a large feasibility study where we implemented the Skoll infrastructure and developed a DCQA process for functional testing of ACE+TAO software systems across their numerous configurations.

ACE+TAO are large middleware projects for performance-intensive, distributed software applications. ACE [69] implements core concurrency and distribution services. TAO is a CORBA object request broker (ORB) built on top of ACE [70].

We chose these projects for several reasons. First, they share the key characteristics common to modern software systems. They have a 2MLOC+ source code base and substantial test code. ACE+TAO run on dozens of OS and compiler platforms and are highly configurable, with hundreds of options supporting a wide variety of program families. ACE+TAO are maintained by a geographically distributed core team of ~140 developers. Their code base is dynamically changing and growing with 400+ CVS repository commits per week on average.

The second reason is that, like many infrastructure systems, ACE+TAO developers cannot test all possible platform and OS combinations because there simply are not enough people, OS/compiler, platforms, CPU cycles, or disk space to run the hundreds of ACE+TAO regression tests in a timely manner. Moreover, since

ACE+TAO are designed for ease of subsetting, several hundred orthogonal features/options can be enabled/disabled for application-specific use-cases. Thus, the number of possible configurations is far beyond the resources of the core ACE+TAO development team.

Currently, the ACE+TAO developers run their regression tests on 100+ workstations and servers at a dozen sites around the world. These machines continuously test a fixed set of configurations (chosen in an *ad hoc* manner) against the latest code base. The interval between build/test runs ranges from 3 hours on quad-CPU Linux machines to 12-18 hours on less powerful machines. The test results are stored on daily basis and visualized via a web-based scoreboard system. No effort is made to control and adapt the process and document and analyze the results.

This chapter describes a feasibility study where we used Skoll for functional testing of ACE+TAO across its numerous configurations. We used three QA task scenarios applied to a specific version of ACE+TAO: (1) checking for clean compilation, (2) testing with default runtime options, and (3) testing with configurable runtime options. In addition, we provided ACE+TAO developers with concise characterizations of failing subspaces in terms of classification tree models. As we will see later in this chapter, fault characterization models helped the developers quickly pinpoint the potential causes of failures.

In each study, we developed a configuration and control model and implemented the necessary Skoll components (*i.e.,* we implemented generic navigation and adaption strategies and analysis tools). Each configuration and control model defined a valid software configuration space for ACE+TAO. The global QA task

was to exhaustively test the functionality of ACE+TAO across the configuration space. A QA subtask was then defined as testing a single valid configuration from this space.

We conjecture that the Skoll infrastructure is easy to use and can implement a variety of QA processes. We also conjecture that our prototype Skoll QA process is superior to ACE+TAO's *ad hoc* QA processes as it (1) automatically manages and coordinates the QA process, (2) detects problems more quickly on the average, and (3) automatically characterizes subtask results, directing developers to potential causes of failures. As we identified problems with the ACE+TAO, we time-stamped them and recorded pertinent information. This allowed us to qualitatively compare Skoll's performance to that of ACE+TAO's *ad hoc* process.

The remainder of this section is organized as follows: Section 4.1 briefly explains the classification tree analysis; Section 4.2 introduces the fault characterization process; Section 4.3 describes how we set up the Skoll infrastructure for our feasibility studies; Section 4.4, 4.5, and 4.6 describe the studies we conducted; and Section 4.7 presents concluding remarks.

4.1 Background

In our studies, as subtask results return, we use classification tree analysis to characterize failing configuration subspaces. This section provides some background information on classification trees.

4.1.1 Classification Trees

Classification tree analysis (CTA) uses a recursive partitioning approach to build models that predict a configuration's class (e.g., passing or failing) based on the settings of the options that define a configuration. This model is tree-structured (See Figure 4.1). Each node denotes an option, each edge represents a possible option setting, and each leaf represents a class or set of classes (if there are more than 2 classes).

Classification trees are constructed using data called the *training set.* A training set consists of configurations, each with the same set of options, but with potentially different option settings together with known class information. Based on the training set, models are built as follows:

1. For each option, partition the training set based on the settings of that option.
2. Evaluate each result based on how well the partition separates configurations of one class from those of other classes. This evaluation is often realized as an entropy measure [8].
3. Select the option that creates the best partition and make it the root of the tree.
4. Add one edge to the root for every possible option setting.
5. For each new edge (*i.e.*, each subset of the partition), repeat the process. The process stops when no further split is possible (or desirable).

To evaluate the model, we use it to predict the class of previously unseen configurations. We call these configurations the *test set*. For each configuration, we begin with the option at the root of the tree and follow the edge corresponding to the option setting found in the new configuration. This process continues until a leaf is encountered. The class label found at the leaf is interpreted as the predicted class for the new configuration. By comparing the predicted class to the actual class, we estimate the accuracy of the model.

In this research, we use the classification trees to extract failure-inducing option setting patterns. That is we extract a set of options and their settings from the tree that characterize failing configurations. We use the Weka [77] implementation of the J48 classification tree algorithm with the default confidence factor of 0.25 to build classification tree models.

4.2 The Fault Characterization Process

The ultimate goal of the fault characterization process is to identify specific options and their settings which are highly correlated with the manifestation of failures. We call this *fault characterization*. Fault characterization is done by exhaustively testing the entire configuration space and feeding the results to a classification tree analysis. The output is a model describing the options and settings that best predict failures. This section discusses the fault characterization process over an example.

Table 4.1 depicts the results of exhaustively testing a system that has three configuration options ($o1$, $o2$, and $o3$), each of which has three possible settings (0,

Config			Result	Config			Result
o1	o2	o3		o1	o2	o3	
0	0	0	PASS	1	1	2	ERR #1
0	0	1	PASS	1	2	0	ERR #1
0	0	2	ERR #3	1	2	1	ERR #1
0	1	0	PASS	1	2	2	ERR #1
0	1	1	PASS	2	0	0	ERR #2
0	1	2	PASS	2	0	1	ERR #2
0	2	0	PASS	2	0	2	ERR #2
0	2	1	PASS	2	1	0	ERR #2
0	2	2	PASS	2	1	1	ERR #2
1	0	0	ERR #1	2	1	2	ERR #2
1	0	1	ERR #1	2	2	0	ERR #3
1	0	2	ERR #1	2	2	1	ERR #2
1	1	0	ERR #3	2	2	2	ERR #2
1	1	1	ERR #1				

Table 4.1: An example exhaustive test schedule.

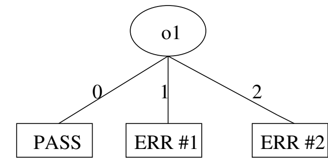


Figure 4.1: An example classification tree.

1, and 2). The system has no inter-option constraints, therefore, there are 27 valid configurations. In this example, four outcomes were observed – test PASSed, test failed with ERR #1, test failed with ERR #2 and test failed with ERR #3.

Feeding this data to a classification tree algorithm yields the classification tree model shown in Figure 4.1. This simple model tells us that the setting of option $o1$ is strongly correlated with two failure outcomes: ERR #1 and ERR #2. That is, configurations with $o1 == 1$ fail with ERR #1 and those with $o1 == 2$ fail with ERR #2.

Note that the fault characterization model given in Figure 4.1 is not perfectly accurate; ERR #3 occurs in configurations having all settings of $o1$ and $o2$ and 2 of the 3 settings of $o3$. We discuss how to evaluate such models when we revisit the fault characterization process in Section 5.2.

4.3 Setting up the Skoll Infrastructure

We implemented the Skoll components described in Chapter 3.

Configuration and control model: We developed different configuration and control models for each scenario. The Skoll system automatically translated the models into the ISA's planning language.

Navigation strategy: We configured the ISA as a stand-alone process running the Blackbox planner and instructed it to navigate the configuration and control space using random sampling without replacement.

Adaptation strategy: We implemented the nearest neighbor, temporary constraints, and terminate/modify subtasks adaptation strategies (Section 3.1.2). We used temporary constraints and terminate/modify subtasks adaptation in each scenario, but used nearest neighbor adaptation only when the configuration space was considered large. In practice, process designers determine the criteria for deciding when a configuration space is large or small.

Analysis technique: We provided ACE+TAO developers with concise characterizations of failing subspaces in terms of classification tree models.

We installed Linux and Win32 Skoll clients and one Skoll server across 25 (11 Linux and 14 Win32) workstations distributed throughout computer science labs at the University of Maryland. All Linux Skoll clients ran on Linux 2.4.9-3 stations and used gcc v2.96 as their compiler; the Win32 clients ran on Windows XP stations with Microsoft's Visual C++ v6.0 compiler. On both platforms, we used TAO v1.2.3 with ACE v5.2.3 as the subject software.

4.4 Study 1: Clean Compilation

ACE+TAO allow many features to be compiled in or out of the system. Features are often left out, for example, to reduce memory footprint in embedded systems. The QA task for this study was to determine whether each ACE+TAO feature combination compiled without error. This is important for systems distributed in source code form, since any valid feature combination should compile. Unexpected build failures not only frustrate users, but also waste a lot of time. For example, compiling the 2MLOC+ took us roughly 4 hours on a 933 MHz Pentium III with 400 Mbytes of RAM, running Linux.

Configuration and control model. The feature interaction model for ACE+-TAO is undocumented, so we built our initial configuration and control model bottom-up. First, we analyzed the source and interviewed several core ACE+TAO developers. We selected 18 options; one of these options was the OS; the remaining 17 were binary-valued compile-time options that control build time inclusion of various CORBA features. We also identified 35 inter-option constraints. For example, one constraint is (AMI = 1 $\rightarrow$ MINIMUM_CORBA = 0). This means that asynchronous method invocation (AMI) is not supported by the minimum CORBA specification. This configuration and control space has over 164,000 valid configurations. Since none of the constraints were related to the OS option, the space was divided equally by OS, *i.e.*, 82,000 valid configurations per OS.

Study execution. Because the configuration space was large, we used the nearest neighbor adaptation strategy. We also configured the ISA to use random sampling without replacement since we felt that one observation per valid configuration was sufficient.

After testing ∼500 configurations, the terminate/modify adaptation strategy signaled that every configuration had failed to compile. We terminated the process and discussed the results with ACE+TAO developers. We discovered that the problem lay in 7 options providing fine-grained control over CORBA messaging policies. It turned out that the code had been modified and moved to another library and developers (and users) failed to establish if these options still worked.

Based on this feedback ACE+TAO developers chose to control these policies at link-time, not at compile time. We therefore refined our configuration model by removing the options and corresponding constraints. Since these options appeared in many constraints – and because the remaining constraints are tightly coupled (*e.g.*, were of the form ($A{=}1 \rightarrow B{=}1$) and ($B{=}1 \rightarrow C{=}1$)) – removing them simplified the configuration model considerably. As a result, the configuration model contained 11 options (one being OS) and 7 constraints, yielding only 178 valid configurations. Note that, in this chapter, we investigate only a small subset of ACE+TAO's entire configuration space; the actual space is obviously much larger.

We then continued the study using the new configuration and control model and removing the nearest neighbor adaptation strategy (since now we could easily build all valid configurations). Of the 178 valid configurations only 58 compiled without errors. For the 178-58=120 configurations that did not build, automatic

fault characterizations helped to clarify the conditions in which they failed.

Results and observations. Beyond identifying failures, in several cases, automatic fault characterizations provided concise, statistically significant descriptions of the failing subspaces. Below we describe the failures, present the automatically generated characterizations, and discuss the actions taken by ACE+TAO developers.

The ACE+TAO build failed at line 630 in `userorbconf.h` (64 configurations - 32 per OS) whenever AMI = 1 and CORBA_MSG = 0. ACE+TAO developers determined that the constraint AMI = 1 → CORBA_MSG = 1 was missing from the configuration and control model. Therefore, we refined the model by adding this constraint.

The ACE+TAO build also failed at line 38 (at line 37 for Win32[1]) in `Asynch_Reply_Dispatcher.h` (16 configurations) whenever CALLBACK = 0 and POLLER = 1. Since this combination of option settings should be legal, this was determined to be a previously undiscovered bug. Until the bug could be fixed, we temporarily added a new constraint CALLBACK = 0 & POLLER = 1 → *false* to the configuration and control model.

The ACE+TAO build failed at line 137 in `RT_ORBInitializer.cpp` (40 configurations) whenever CORBA_MSG = 0. The error was reported on line 665 in file `RT_Policy_i.cpp` when the system was compiled under Win32. Again, we attribute

[1]We noted that the compilers (gcc and MSVC++) reported different line numbers for the same error, requiring manual examination and matching of error messages.

this difference to the compiler and not to an ACE+TAO platform-specific problem. The problem was due to a `#include` statement, missing because it was conditionally included (via a `#define` block) only when CORBA_MSG = 1.

Lessons learned. We found that even ACE+TAO developers do not completely understand the configuration and control model for their very complex system. In fact, they provided us with both erroneous and missing model constraints. We discovered that model building is an iterative process. Using Skoll we quickly identified coding errors (some previously undiscovered) that prevented the software from compiling in certain configurations. We learned that the temporary constraints and terminate/modify subtasks adaptation strategies performed well, directing the global process towards useful activities, rather than wasting effort on configurations that would surely fail without providing any new information.

ACE+TAO developers also told us that automatic characterization was useful to them because it greatly narrowed down the issues they had to examine in tracking down the root cause of the failure. We also learned that as fixes to problems were proposed, we could easily test them by spawning a new Skoll process based on the previously inserted temporary constraints. That is, the new Skoll process would test the patched software only for those configuration that had failed previously.

In this study, we did not find any platform-specific compilation problems, other than differences in compiler-generated error messages and their locations. Before moving on to the next study we worked around constantly failing configurations by leaving the appropriate temporary constraints in the second study's configuration

and control model.

4.5 Study 2: Testing with Default Runtime Options

The QA task for the second study was to determine whether each compile-time configuration would run the ACE+TAO regression tests without error with the system's default runtime options. This activity is important for systems that distribute tests to run at installation time because it is intended to give the user confidence that he or she has correctly installed the system. To perform this task, users compile ACE+TAO, compile the tests, and execute the tests. For example, on our Linux machines this took around 8 hours: about 4 hours to compile ACE+TAO, about 3.5 hours to compile all tests, and 30 minutes to execute them.

Configuration and control model. In this study we used 96 ACE+TAO tests, each containing its own test oracle and reporting success or failure on exit. These tests are often intended to run in limited situations, so we extended the configuration space, adding test-specific options.

The new test-specific options contain one option per test. They indicate whether that test is runnable in the configuration represented by the compile time options. For convenience, we named these options $run(T_i)$. We also defined constraints over these options. For example, some tests should run only on configurations with more than the minimum CORBA features. So for all such tests, T_i, we added a constraint $run(T_i) = 1 \rightarrow$ MINIMUM_CORBA $= 0$. This prevents us from running tests that are bound to fail. By default, we assume that all tests are

runnable unless constrained to be otherwise.

Study execution. After making these changes, the model had 11 compile-time options with 10 constraints and 96 test-specific options with an additional 120 constraints. We again configured the ISA for random sampling without replacement. We do not use the nearest neighbor adaptation strategy since we only tested the 58 configurations that built in Study 1. In this study, automatic characterization is done separately for each test and error message combination, but is based only on the settings of the compile time-options.

Results and observations. Overall, we compiled 4,154 individual tests. Of these 196 did not compile, 3,958 did. Of these, 304 failed, while 3,654 passed. This process took ~52 hours of computer time. As in the first study we now describe some of the interesting failures we uncovered, the automatically-generated fault characterizations, and the action taken by ACE+TAO developers.

In several cases, multiple tests failed for the same reason on the same configurations on both Linux and Win32. For example, test compilation failed at line 596 of `ami_testC.h` for 7 tests, each when (CORBA_MSG = 1 and POLLER = 0 and CALLBACK = 0). This was a previously undiscovered bug. It turned out that certain files within TAO implementing CORBA messaging incorrectly assumed that at least one of the POLLER or CALLBACK options would always be set to 1. ACE+TAO developers also noticed that the failure manifested itself no matter what the setting of the AMI was. This was also a previously undiscovered problem

because these tests should not have been runnable when AMI = 0. Consequently, there was a missing testing constraint, which we then included in the test constraint set.

A group of three tests failed only when (OS = Win32). These tests failed because the ACE server failed to start on Win32 platforms. This failure was caused by incorrect coding (Linux vs. Win32) of server-invocation scripts in the tests. Increasing test diversity (*i.e.*, increasing the number of platforms on which tests run) helped to find this problem.

A group of two tests failed in 17 configurations when (OS = Win32 and TAO_HAS_AMI_POLLER = 0). These tests failed because clients did not get correct (or any) response from the server. These tests should actually have failed on Linux too. However, Linux (and some flavors of Unix) tolerates some amount of invalid memory scribbling without killing the process, thereby allowing the test to pass, even though it should have failed. The failure was revealed only on Win32 because it is more rigorous in its memory management.

A group of three tests failed in 6 configurations when (OS = Linux and TAO_HAS_AMI_POLLER = 0 and TAO_HAS_INTERCEPTORS = 0 and TAO_HAS_NAMED_RT_MUTEXES = 1). The same three tests failed in 10 configurations when (OS = Linux and TAO_HAS_AMI_POLLER = 0 and TAO_HAS_INTERCEPTORS = 1). It turned out that this failure was a side-effect of the order in which the test cases happened to execute. This problem was specific only to the Linux platform. It had nothing to do with the specific test cases themselves or the options (except OS = Linux). The tests failed when Linux ran out of the shared memory

segments (SHM) available to the OS. We discovered that TAO leaks SHM segments on Linux. If enough tests were run on a particular Linux machine, the machine ran out of the SHM segments, causing all subsequent tests to fail. If these particular tests had been run earlier, they would not have failed. In effect, we inadvertently conducted a load test on some machines.

One test `MT_Timeout/run_test.pl` failed in 28 of 58 configurations on both platforms with an error message indicating response timeout. No statistically significant fault characterization model could be found. This suggests that the error report might be covering multiple underlying failures, that the failure(s) manifests themselves intermittently, or that some other factor, not related to configuration options, is causing the problem. It appears that particular problem appears intermittently and is related to inconsistent timer behavior on certain OS/hardware platform combinations.

A group of three tests failed in all configurations regardless of the OS. Even though the underlying problem that led to the failures was not configuration-specific, the overall Skoll automation process helped to uncover it. The failures were caused by memory corruption due to erroneous command-line processing. Whenever the test script used a particular command-line option, namely `ORBSkipServiceConfigOpen`, the tests failed. The usage of the above mentioned option is not mandatory for the scripts but Skoll used it during the model-building and stepwise refinement of command line options, identifying this previously undiscovered problem.

Lessons learned. We easily extended and refined the initial configuration model to create more complex QA processes. We again were able to carry out a sophisticated QA process across networked Skoll clients on a continuous basis. In this case, we exhaustively explored the configuration space in less than a day and quickly flagged numerous real problems with ACE+TAO. Some of these problems had not been found with ACE+TAO's *ad hoc* QA processes. We learned that increasing test diversity, *e.g.*, increasing the number of platforms and configurations on which tests run, helps to reveal previously undiscovered failures.

We also learned several things about automatic fault characterizations. In particular, the generated models can be unreliable. We use notions of statistical significance to help indicate weak models (See Section 5.2 for more details). Also, the tree models we use may not be reliable when failures are non-deterministic and the ISA has been configured to generate only a single observation per valid configuration. In the presence of potentially non-deterministic failures, therefore, it may be desirable to configure the ISA for random selection with replacement.

4.6 Study 3: Testing with Configurable Options

The QA task for the third study was to determine whether each configuration would run the ACE+TAO regression tests without error over all settings of the system's runtime options. This is important for building confidence in the system's correctness. To do this users compile ACE+TAO, compile the tests, set the appropriate runtime options, and execute the tests. For us, each task would have taken

about 8 hours on our Linux machines.

Configuration and control model. To examine ACE+TAO's behavior under differing runtime conditions, we modified the configuration and control model to reflect 6 multi-valued (non-binary) runtime configuration options. These options set up to 648 different combinations of CORBA runtime policies: when to flush cached connections, what concurrency strategies the ORB should support, etc. Since these runtime options are independent, we did not add any new constraints.

After making these changes, the compile-time option space had 11 options and 10 constraints, there were 96 test-specific options, and there were 6 runtime options with no new constraints.

Study execution. The configuration and control space for this study had 37,584 valid configurations. At roughly 30 minutes per test suite, the entire process involved around 18,800 hours of computer time. Given the large number of configurations, we used the nearest neighbor adaptation strategy.

Results and observations. The total number of test executions was 3,608,064. Of these, 689,603 test failed, with 458 unique error messages. As in the first two studies, we now describe some of the interesting failures we uncovered.

One observation is that several tests failed in this study even though they had not failed in Study 2 (when running tests with default runtime options). Some even failed on every single configuration (including the default configuration tested earlier), despite not failing in Study 2! In the latter case, the problems were often

caused by bugs in runtime option setting and processing code. In the former case, the problems were often in feature-specific code. ACE+TAO developers were intrigued by these findings because they rely heavily on testing of the default configuration by users at installation time, not just to verify proper installation, but to provide feedback on system correctness.

A group of three tests had particularly interesting failure patterns. These tests failed between 2,500 and 4,400 times on each platform. In each case automatic fault characterization showed that the failures occurred when `ORBCollocation` = NO. No other option influenced failure manifestation. In fact, it turned out that this setting was in effect over 99% of the time when these tests failed.

TAO's `ORBCollocation` option controls the conditions under which the ORB should treat objects as being co-located. The `NO` setting means that objects are never co-located. When objects are not co-located they talk to each other via the network. When they are co-located, they can communicate directly. The fact that these tests worked when objects communicated directly, but failed when they talked over the network clearly suggested a problem related to message passing. In fact, the source of the problem was a bug in the routines for marshalling/unmarshalling object references.

Lessons learned. We learned several things from Study 3. First, we confirmed that our general approach could scale well to larger configuration spaces. We also reconfirmed one of our key conjectures: that data from the distributed QA process can be analyzed and automatically characterized to provide useful information to

developers. We also saw how the Skoll process gives better coverage of the configuration space than does the process used by ACE+TAO (and, by inference, many other projects).

We also note that our nearest neighbor adaptation strategy explores configurations until it finds no more failing configurations. In cases where a large subspace is failing a lot of work will be done (*e.g.*, as described above in roughly 5,000 out of a total 20,000 configurations, `ORBCollocation` = NO and the test failed). Looking back at the data it's clear that we could have stopped the search much earlier as the shape of the failing subspace becomes statistically significant and still correctly identified the problem. We intend to explore this issue in the future.

4.7 Discussion

In this chapter, we conducted an initial set of large feasibility studies where we implemented the Skoll infrastructure and used it for functional testing of ACE+TAO software system across its numerous configurations. We learned several things from these studies:

- We validated the Skoll implementation.
- We discovered that the Skoll approach and infrastructure are promising. Using Skoll, we iteratively modeled complex configuration spaces, developed large scale, sophisticated QA processes, and carry out them across networked Skoll clients on a continuous basis. We observed that increasing test diversity helped to reveal many previously unknown bugs.

- The Skoll's configuration and control model was flexible and easy to extend. We leveraged the configuration and control model in planning the global QA process, for adapting the process (*e.g.*, adding temporary constraints to avoid constantly failing subspaces), and to aid interpreting the results. Moreover, we observed that even the core ACE+TAO developers do not completely understand the complex configuration space for their system. In fact, we discovered both erroneous and missing model constraints. As a result, we discovered that model building is an iterative process and having a formally defined configuration and control model is valuable.

- The temporary constraints and terminate/modify subtasks adaptation strategies performed well, directing the global process towards useful activities, rather than wasting effort on configurations that would surely fail without providing any new information.

- Skoll covered ACE+TAO's configuration space better than the ACE+TAO's *ad hoc* approach. We quickly discovered real bugs, some of which had not been identified previously.

- We confirmed one of our key conjectures: that data from a DCQA process can be analyzed and automatically characterized to provide useful information to developers. ACE+TAO developers reported that automatic fault characterizations greatly simplified tracking down the root causes of certain failures.

- We also observed that the configuration spaces of DCQA processes can be

quite large. Even with a large pool of user-supplied resources, brute-force QA approaches (*e.g.,* exhaustive testing) may be infeasible or simply undesirable.

In the next chapter, we introduce a sampling strategy for efficient fault characterization in complex configuration spaces.

Chapter 5

A Sampling Strategy for Efficient Fault Characterization in Complex Configuration Spaces

Many modern software systems must be customized to specific run-time contexts and application requirements. To support such customization, these systems provide numerous user-configurable options. For example, some web servers (*e.g.*, Apache), object request brokers (*e.g.*, TAO), and databases (*e.g.*, Oracle) can have dozens, even hundreds, of options. While this flexibility promotes customization, it creates many potential system configurations, each of which may need extensive QA to validate. We call this problem *software configuration space explosion.*

The software configuration space explosion coupled with distributed development environments and/or third-party components increases the need to quickly detect and fix faulty changes. In our previous work (Chapter 4), we introduced the fault characterization process which aims at identifying failure-inducing configuration options and their settings. Our experiments suggested that fault characterization models can help developers pinpoint the root causes of failures.

While we were pleased with this outcome, the fundamental downside of this approach was that we have to test the entire configuration space. In our previous work, for instance, that means that nearly 19,000 times, remote clients downloaded, configured and compiled the 2M+ lines of system code, and then executed a battery

of tests. For each client this took about 6–8 hours. Furthermore, this was only a small subset of the system's entire configuration space. The actual space is much bigger. Clearly, some more efficient process will be necessary in general.

In this chapter, we propose and evaluate an alternative strategy. The idea is to systematically sample the configuration space, test only the selected configurations, and conduct fault characterization on the resulting data. The sampling approach we use are based on calculating two different kinds of mathematical objects, called a *covering array* and a *variable strength covering array.* Covering arrays (described in Section 5.1.1) induce a test schedule ensuring that all t-way interactions between options are observed at least once. Variable strength covering arrays (described in Section 5.1.2) provide finer control over covering array construction. We empirically assess the effect of sampling on the resulting fault characterizations and provide guidelines to practitioners for their use. Our results strongly suggest that sampling via covering arrays and variable strength covering arrays is nearly as accurate as that based on exhaustive data, but is much cheaper (it provided a 50% to 99% reduction in the number of configurations to be tested).

The remainder of this chapter is organized as follows: Section 5.1 briefly explains the mathematical objects we used; Section 5.2 revisits the fault characterization process; Sections 5.3 and 5.4 describe the studies we conducted; Section 5.5 provides practical advice to the users of this approach; Section 5.6 compares covering arrays to random sampling; and Section 5.7 presents concluding remarks.

5.1 Background

In this chapter we propose a 3-step process for characterizing faults. First we systematically sample a system's entire configuration space using two types of mathematical objects called a covering array and a variable strength covering array as opposed to using the entire configuration space as we did in Chapter 4. Next we use Skoll to test individual configurations at remote user sites which relay the results to a central server. Finally, we classify the test results and provide the resulting models to the system's developers.

In this section we provide some background information on the mathematical objects we used for sampling.

5.1.1 Covering Arrays

The software systems and QA processes we consider in this research have *options*, each of which takes its value from a set of valid *settings*. Our main goal is to identify and characterize failures that are caused by specific combinations of option settings. Therefore, it is important that we maximize the "coverage" of option setting combinations. However, we also want to do this at some reasonably low cost. Consequently, we also want to minimize the total number of configurations tested. The set of configurations chosen for testing is called the *test schedule*.

Our approach to doing this is to compute t-way coverage using a combinatorial object called a *covering array*. A covering array, $CA(N; t, k, v)$, is an $N \times k$ array on v symbols with the property that any $N \times t$ sub-array contains all ordered t-sets

Configuration No	Option A	Option B	Option C
1	0	0	0
2	0	1	1
3	0	2	2
4	1	0	1
5	1	1	2
6	1	2	0
7	2	0	1
8	2	1	0
9	2	2	1

Table 5.1: A covering array example: $CA(9; 2, 3, 3)$

of size v at least once [12]. The *strength* of the array is denoted by t. For instance, given a covering array of strength $t = 2$ we can arbitrarily select any 2 columns from the covering array to form a new sub-array. We are guaranteed that any ordered pair from the v values will be found in at least one row of this sub-array. When using our Skoll system, each of the configuration options is a column of the covering array. Each option setting is mapped to one of the v values for that column. This gives us a covering array derived test schedule or *CA test schedule*. A CA test schedule for a configuration space is a set of N test configurations in which all t-way combinations of option settings appear at least once.

Suppose we have the following example system. It has three options, A, B and C. Each of these options has three possible settings, represented by the numbers 0, 1 and 2. This system has 27 possible configurations. A $CA(9; 2, 3, 3)$ for this system is shown in Table 5.1. In this subset of 9 configurations, all possible pairs of combinations for 0, 1 and 2 are found in any two arbitrarily selected columns.

Since many software systems do not have the same number of option settings for each option, we use a *mixed level* covering array to model such systems. An

Configuration No	Option A	Option B	Option C
1	0	1	1
2	0	2	0
3	2	1	0
4	0	0	2
5	1	1	2
6	2	2	2
7	2	2	3
8	2	0	1
9	1	0	0
10	1	2	1
11	1	1	3
12	0	0	3

Table 5.2: A mixed level covering array example, $MCA(12; 2, 3^2 4^1)$

$MCA(N; t, k, (v_1, v_2, ...v_k))$, is an $N \times k$ array on s symbols, where $s = \sum_{i=1}^{k} v_i$. In this array each column $i \quad (1 \leq i \leq k)$ contains elements from a set S_i with $|S_i| = v_i$. The rows of every $N \times t$ sub-array cover all t-tuples of values from the t columns at least once. A shorthand notation is used to describe a covering array by combining v_i's that are the same and representing this number as a superscript. For example if we have 4 v's each with 3 values, this can be written as 3^4. In this manner an $MCA(N; t, k, (v_1 v_2 ... v_k))$ can also be written as an $MCA(N; t, (s_1^{p_1} s_2^{p_2} ... s_r^{p_r}))$ where $k = \sum_{i=1}^{r} p_i$.

Returning to the previous example, suppose option C now has 4 possible settings instead of 3. We can create a mixed level covering array using 12 configurations (shown in Table 5.2). In this example all possible pairs of the 4 settings for option C are combined with the 3 settings for options A and B. The combinations of all 3 settings from options A and B are all accounted for as well. This is an $MCA(12, 2, 3^2 4^1)$.

In this chapter, we restrict ourselves to mixed level covering arrays. Therefore we will use the general term *covering array* to refer to these from now on.

Covering arrays have the property that each t-tuple is used *at least* once, which means they can be arbitrarily large, therefore one of the goals, in building these, is to minimize N. A variety of computational methods exist that can be used to find covering arrays with a small N for a given set of parameters. In [20] several greedy algorithms are compared with heuristic search such as simulated annealing and hill climbing. Simulated annealing gives a consistently small N when $t = 2$ or $t = 3$. Therefore, we chose this as our construction method. Simulated annealing is a standard combinatorial optimization technique (see [20] for a more thorough discussion of this algorithm). In our implementation of the simulated annealing method, the cost function is the number of uncovered t-sets remaining, *i.e.*, a covering array has a cost of 0. We begin with an unknown N for a particular set of parameters, repeating the annealing process many times, using a binary search strategy to find the smallest N which gives us a solution [20].

5.1.2 Variable Strength Covering Arrays

A covering array defines a "fixed" t across all of the k columns. In [22, 20] an aggregate object called a *variable strength covering array* is defined. A *variable strength covering array* is a covering array of strength t with subsets of columns of strength $\geq t+1$. It is denoted as a $VSCA(N; t, (v_1, v_2, .., v_k), C)$. More formally, it is an $N \times k$ mixed level covering array, of strength t containing C, a vector of

covering arrays each of strength $> t$ and defined on a subset of the k columns.

This structure provides the ability to tune a test schedule so that certain sets of options are tested more strongly (i.e. we use higher strength for certain option groups) without losing the base property of t-way coverage across the whole system. This capability can be leveraged when it is too expensive to use a higher t for *all* options. When there is information prescribing that a subset of option combinations is more likely to contain faults, or that a subset of options has greater consequences if a failure is induced from that region, the variable strength covering array defines a higher strength coverage for those subsets. Sometimes, a variable strength test schedule can be created that is the same size as a covering array test schedule. This occurs when there is a large imbalance in the numbers of option settings across the system. We take advantage of this situation in some of the studies in this chapter.

Suppose we add 3 options, each with 2 possible settings (0 or 1) to our example system. The original options, A, B and C still have 3 settings each. The new options D, E and F are known to have a lot of interrelated functionality and are therefore more likely to interact. We would like to test the setting combinations between these options exhaustively. To do this requires at least 8 configurations. If we want to create a three way covering array for the whole system, however, it would require at least 27 configurations since the first three options, contain 3 settings each. It is possible to create two individual covering arrays for each subsystem and put them side by side, but that does not guarantee that all combinations between option A and E or B and E are tested for example. Instead, a VSCA can be constructed (see Table 5.3). This subset of 10 configurations, includes all possible combinations

Configuration No	Option A	Option B	Option C	Option D	Option E	Option F
1	2	2	1	1	0	1
2	0	2	2	0	0	1
3	2	1	2	0	1	0
4	1	0	2	1	1	1
5	0	2	0	1	1	0
6	0	1	1	1	1	1
7	0	0	1	0	1	1
8	2	0	0	0	0	0
9	1	1	0	0	0	1
10	1	2	1	1	0	0

Table 5.3: A VSCA example, $VSCA(10; 2, 3^3 2^3, CA(3,3,2))$

between options D, E, and F, as well as including all possible 2-way combinations between *any* of the six options. It is a $VSCA(10; 2, 3^3 2^3, CA(3; 3, 2))$.

Like fixed strength covering arrays, variable strength covering arrays can be constructed with simulated annealing (described in [22]). The cost function is changed to equal the missing t-sets added to the sum of the missing tuples for all covering arrays in the vector C.

As we have discussed in Section 2.2, covering arrays have been used frequently to test input combinations of programs. Mandl [53] first used orthogonal arrays, a special type of covering array in which all t-sets occur *exactly* once, to test enumerated types in ADA compiler software. This idea was extended by Brownlie *et al.* [9] who developed the orthogonal array testing system (OATS). They provided empirical results to suggest that the use of orthogonal arrays is effective in fault detection and provides good code coverage. Dalal *et al.* [25] argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults. In further work, Burr *et al.*, Dunietz *et al.* and Kuhn *et al.* provide

more empirical results to show that this type of test coverage is effective [11, 29, 48]. These studies focus on finding unknown faults in already tested systems and equate covering arrays with code coverage metrics [19, 29].

Our approach is different in that we apply covering arrays to system configuration options and we assess their effectiveness in revealing option-related failures and identifying failure inducing options.

A structure similar to the variable strength covering array is first suggested in [19] termed "hierarchical test suites", but no empirical evidence of their success is given. In [20, 22] Cohen, *et al.* present a discussion, providing scenarios of when variable strength arrays might be useful. They provide a model to define VSCAs and present a construction technique, however they have not provided any evidence of their effectiveness. We do not know of any studies to date that provide empirical results comparing variable strength arrays with their fixed level counterparts.

5.2 Revisiting the Fault Characterization Process

In this section, we revisit the fault characterization process introduced in Section 4.2 to provide techniques for (1) evaluating fault characterizations and (2) reducing the cost of the process without compromising its accuracy.

In Section 4.2 we illustrated the fault characterization process using an example configuration space. Table 5.4 depicts the same configuration space used in that section. We obtained the fault characterization by feeding the exhaustive testing results of this space to a classification tree algorithm. The resulting fault character-

Config			Result	Config			Result
o1	o2	o3		o1	o2	o3	
0	0	0	PASS	1	1	2	ERR #1
0	0	1	PASS	1	2	0	ERR #1
0	0	2	ERR #3	1	2	1	ERR #1
0	1	0	PASS	1	2	2	ERR #1
0	1	1	PASS	2	0	0	ERR #2
0	1	2	PASS	2	0	1	ERR #2
0	2	0	PASS	2	0	2	ERR #2
0	2	1	PASS	2	1	0	ERR #2
0	2	2	PASS	2	1	1	ERR #2
1	0	0	ERR #1	2	1	2	ERR #2
1	0	1	ERR #1	2	2	0	ERR #3
1	0	2	ERR #1	2	2	1	ERR #2
1	1	0	ERR #3	2	2	2	ERR #2
1	1	1	ERR #1				

Table 5.4: An example exhaustive test schedule.

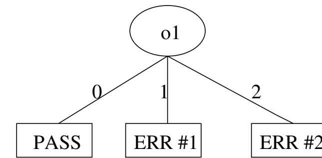


Figure 5.1: An example classification tree.

ization model is given in Figure 5.1. This model told us that the setting of option $o1$ is strongly correlated with two failure outcomes: ERR #1 and ERR #2. That is, configurations with $o1 == 1$ fail with ERR #1 and those with $o1 == 2$ fail with ERR #2.

5.2.1 Evaluating Fault Characterizations

In practice, classification trees may not be perfectly accurate. Some reasons for this might include:

1. the failure is unrelated to the option settings. For example, in our earlier example, ERR #3 occurs in configurations having all settings of $o1$ and $o2$ and 2 of the 3 settings of $o3$; or

2. the model building approach identifies spurious, but non-causal patterns.

This research focuses on option-related failures. Therefore, we attempt to remove the non-option-related failures from our analysis. Since we can't do this automatically, we simply removed any failure from consideration that occurred in less than 3% of the test runs. Our rationale was that deterministic failures involving up to 5 binary options should manifest at least this many times as should also non-deterministic failures involving fewer options, but appearing with a reasonable frequency (e.g., failures involving 3 options with the failure manifesting 1/4 of the time).

To evaluate the accuracy of classification tree models we use several standard metrics. Precision (P) and recall (R) are two widely used accuracy metrics. For a given failure class E, they are defined as follows:

$$recall = \frac{\#\ of\ correctly\ predicted\ instances\ of\ E\ by\ the\ model}{total\ \#\ of\ instances\ of\ E}$$

$$precision = \frac{\#\ of\ correctly\ predicted\ instances\ of\ E\ by\ the\ model}{total\ \#\ of\ predicted\ instances\ of\ E\ by\ the\ model}$$

Recall measures how well the model predicts configurations that in fact experience failure E. Precision, on the other hand, measures how many configurations are falsely identified as experiencing failure E. In general, both measures are important. We want high recall because otherwise the models may miss relevant characteristics or add irrelevant ones. And we want high precision to minimize wasting resources while investigating false alarms.

Since neither measure predominates our evaluation, we combine the measures using the F metric [66]. This is defined as:

$$F = \frac{(b^2+1)PR}{b^2P+R}$$

Here, b controls the weight of importance to be given to precision and recall: $F = P$ when $b = 0$ and $F = R$ when $b = \infty$. Throughout this chapter, we compute F with $b = 1$, which gives precision and recall equal importance.

5.2.2 Reducing the Test Schedule Size

While the model in Figure 5.1 explains the observed failures reasonably well, it does so at the cost of exhaustively testing the configuration space. Since this obviously won't scale, we wanted to find a way to build fault characterization models based on data taken from only a subset of the entire configuration space.

Interestingly, we would have derived the same tree model using data from only 1/3 of the configuration space (these configurations are boxed in Table 5.4). This sample was not chosen at random. Instead, the selected configurations constitute a 2-way covering array of the configuration space (This is the same covering array depicted in Table 5.1). That is, all pairwise combinations of the option settings appear in the boxed configurations. If these results hold in practice, it would greatly reduce the cost of fault characterization, without compromising its accuracy. Thus, we evaluate this conjecture throughout the rest of this chapter.

5.3 Experiments

This section presents several initial studies of our modified fault characterization approach using only a subset of the configuration space. Our goal is to compare the costs and benefits of the modified approach to those of the original approach

which requires testing the entire configuration space. We used ACE+TAO as our subject software in these studies.

In the study described in Section 4.6, we modeled and studied a small subset of the system's entire configuration space. This model consisted of 10 compile-time and 6 runtime options. Each compile-time option was binary-valued, while the runtime options had differing numbers of settings: four options with three levels, one option with four levels, and one option with two levels. All told, this configuration space had 18,792 valid configurations.

Compile-time options allow features, such as asynchronized method invocation (AMI) and CORBA messaging, to be compiled in or out of the system. Runtime options provide more fine-grained control over the runtime behavior of the system, such as object collocation strategies and connection purging strategies.

We tested each configuration using 96 developer-supplied regression tests on both Red Hat Linux 2.4.9-3 and Windows XP Professional. Each test was designed to emit an error message in the case of failure. We captured and recorded the results of each test. In this chapter, we adopt the results of these tests and refer them as *exhaustive results*. As an example, it took us one machine year to run the experiment on the Linux platform.

To evaluate the use of covering arrays, we created five different t-way covering arrays for this configuration space. We allowed t to range between 2 and 6. Specifically, we computed an $MCA(N; t, 29^1 4^1 3^4 2^1)$ for each value of t. Note that because of the numerous compile time errors we uncovered earlier in Chapter 4, we chose to group the 10 compile time options into a single option with 29 settings, thus the

CA Strength (t)	No. of Configurations (N)
2	116
3	348
4	1229 - 1236
5	3369 - 3372
6	9433 - 9453

Table 5.5: Size of test schedules for $2 \leq t \leq 6$.

model has seven configuration options. The first corresponds to the 29 successfully compiled static configurations, and the rest correspond to the 6 runtime options.

We reran the regression tests for each of these t-way test schedules on both platforms and used classification trees to automatically characterize the test results. We then compared the fault characterizations obtained from t-way schedules to the ones obtained from exhaustive testing. Our goal was to see how well we could detect runtime errors and the failure-inducing options that lead to them using only this subset of configurations.

Table 5.5 gives the covering array size N for each value of t. When $t \leq 3$ all five arrays were the same size N. For these we were able to construct covering arrays with the smallest mathematically possible number of rows. When $t \geq 4$, the problem of building a minimally-sized covering array becomes harder so we obtained a range of sizes.

In the remainder of this section, we present the results of four studies, each examining a different aspect of the covering array enhanced fault characterization process. The first study examines how well CA test schedules reveal option-related failures. The second study uses CA test schedules and builds one characterization model for each test (pass vs. fail). The third study uses CA test schedules, but builds

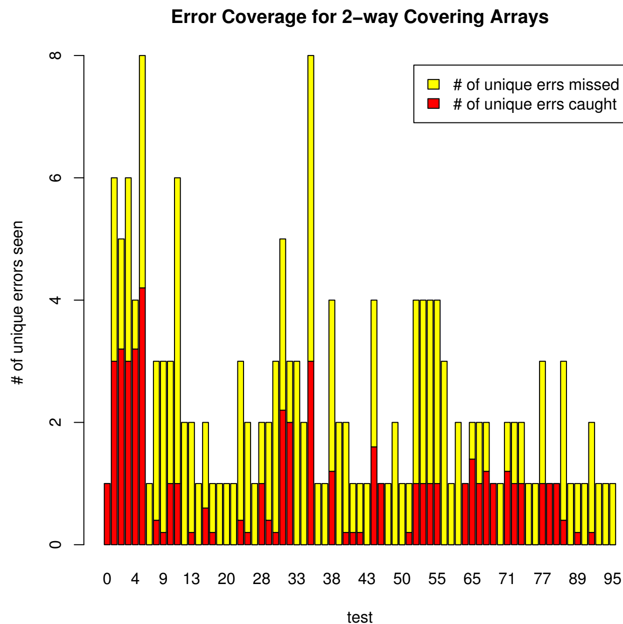


Figure 5.2: Error coverage statistics for 2-way covering arrays on Linux.

one characterization model for each observed failure on each test (pass vs. failure-1 vs. failure-2, etc.). Finally, the fourth study repeats the third, but compares using the combination of several lower strength covering arrays to using one more expensive to obtain higher strength covering array.

5.3.1 Study 1: Revealing option-related failures with covering arrays

The first question we examined was whether testing only the configurations in the CA test schedule negatively affected fault detection. If it does, then characterization based on CA schedules will obviously suffer.

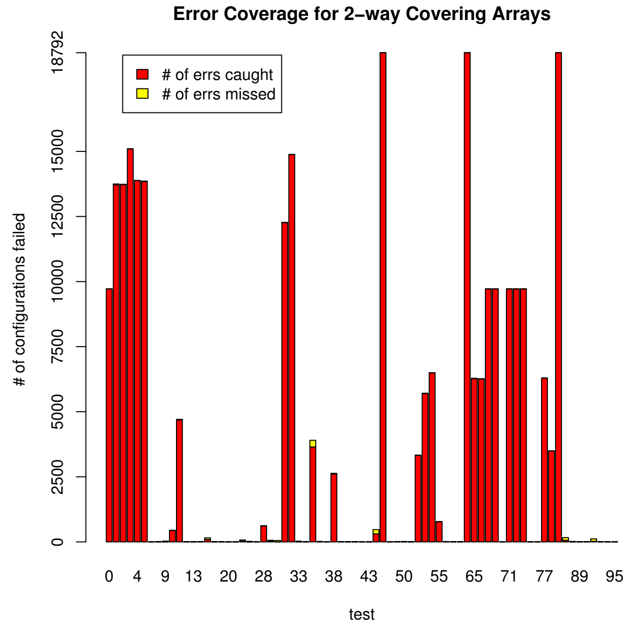
Figure 5.2 plots error coverage statistics for 2-way covering arrays on the Linux platform. We show data only for 2-way covering arrays as they are the smallest. In this figure, each bar represents one test case. Tests that never failed are omitted.

The height of a bar represents the number of unique error messages observed with the exhaustive test schedule. The lower part of a bar (darker color) shows the average number of unique errors observed by the five 2-way schedules. For example, using the exhaustive schedule we observed eight unique error messages while running test #35. Using the 2-way schedules, however, we only observed three of them on average. The Windows platform showed similar results.

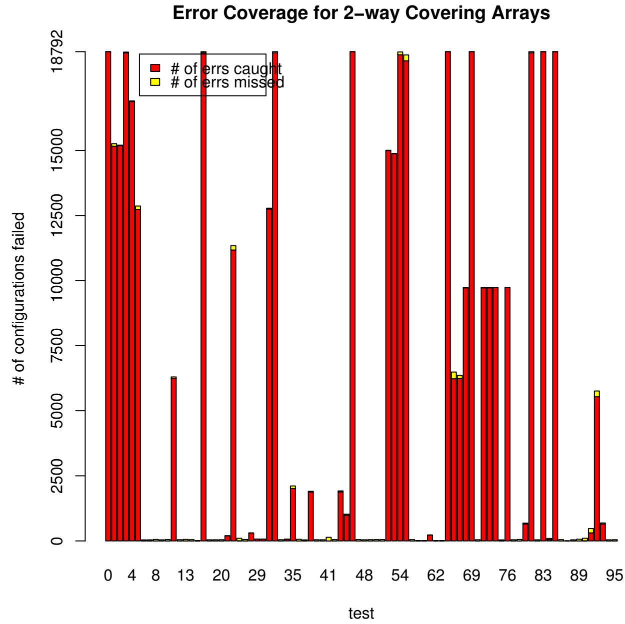
Figure 5.3 provides another view of this data. Instead of the number of unique error messages, it depicts the number of configurations in which each test failed. The lower part of each bar shows the average number of failing configurations whose error message was detected at least once by the 2-way schedules. The upper part indicates the average number of failing configurations whose error messages was not detected by the 2-way schedules. As suggested by the figure, failures not detected by the 2-way schedules occur with very low frequency. Alternatively, the CA schedules were able to detect all faults that appeared with a reasonably large frequency.

Since we are interested in using CA schedules to characterize option-related failures, we aren't overly concerned with rarely-occurring failures. This is because rarely-occurring failures are likely to be either (1) not related to option settings; if they were they would appear more frequently, or (2) are unlikely to be accurately characterized even with exhaustive testing; *e.g.*, a failure that occurs exactly once in 20,000 configurations does not allow for much statistical generalization.

To minimize problems due to rarely-occurring failures, for the rest of this chapter we therefore consider a failure to be potentially option-related only if it appears in more than 3% of the configurations. See Section 5.2.1 for a more detailed



(a) Linux



(b) Windows

Figure 5.3: Error coverage statistics for 2-way covering arrays

explanation. This gave us 40 "potentially" option-related failures on the Linux platform and 49 on the Windows platform. From now on these are referred to simply as *option-related failures.*

We then checked the effectiveness of covering arrays in revealing option-related failures. It turned out that each and every t-way schedule revealed all option-related failures on both platforms.

5.3.2 Study 2: Covering arrays with per test case characterization

The previous study suggested that testing with CA schedules revealed potentially option-related failures as well as exhaustive testing did. Given this assurance, we now want to compare fault characterization based on CA schedules to that based on exhaustive testing.

In this study, for each test case, we build one characterization model for all failures observed in any of the scheduled configurations.

Creating classification tree models

For each configuration in the exhaustive schedule (*i.e.*, in the entire configuration space), we ran all the regression tests and recorded their pass/failure information. For each test case, this results in a set of passing configurations and f sets of failing configurations, one for each unique observed failure. For each test case, we then built one model that characterizes all $f + 1$ possible outcomes and tested it on the exhaustive data set. This tells us how well, in the best case, the models

characterize the faults.

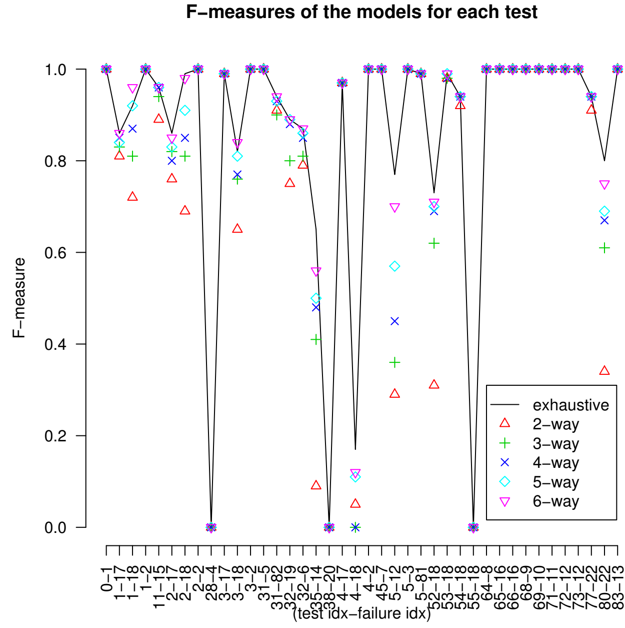
We then repeated the process above for only the scheduled configurations (*i.e.*, those selected by the covering array). We then built classification tree models using this data. We tested the models, however, on the exhaustive data set. This tells us how well the models, built using only a subset of the data, characterize the faults.

In the rest of the chapter, we'll refer to the models obtained from the covering arrays and the exhaustive suite as *reduced models* and *exhaustive models*, respectively.

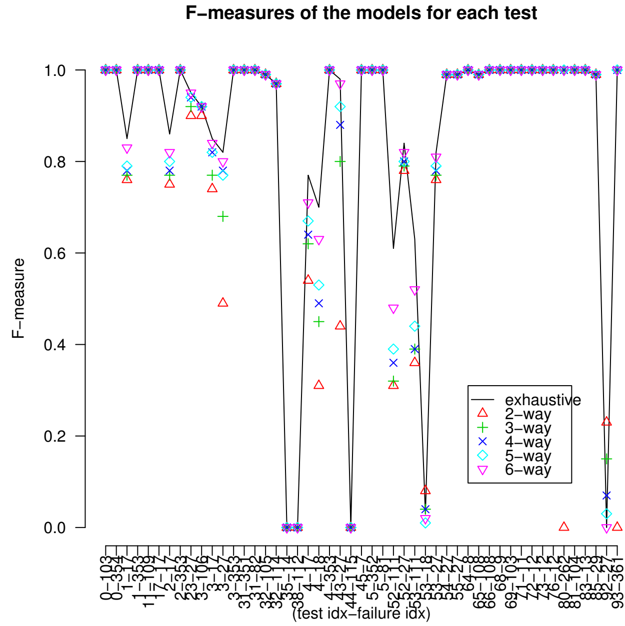
Evaluation

Figure 5.4 shows the F measures for the reduced models and for the exhaustive models for the 40 (49) option-related failures on Linux (Windows) platform. The vertical axis denotes the F measure, and the horizontal axis denotes the test and error index. For example, the first tick on the horizontal axis, which is 0-1, represents the first error observed in test case 0.

The figure suggests that the F measures for the reduced models are almost always near those of the exhaustive models. That is, if the exhaustive models characterize the failure well, then so do the reduced models. If they don't, then neither do the reduced models. This is true no matter what the strength of the covering array (the level of t) is. For example, on Linux, 78% of the models obtained from the 2-way schedules gave F measures within 0.1 of the exhaustive models; 88% of them were within 0.2. The higher the strength of the covering arrays, the closer



(a) Linux



(b) Windows

Figure 5.4: Models for each test.

the F measures were. Another interesting observation is that the 2-way covering arrays achieve this performance while reducing the number of configurations to be tested by 99.4%. To give a sense of the savings, we note that it took us 8 hours to compile ACE+TAO, compile the test cases, and execute them for each and every configuration. Using 2-way schedules would have saved us almost a year of machine time, without substantially lowering the accuracy of the fault characterizations.

Our analysis also suggests that the higher the F measure, the more similar the exhaustive and reduced models were in terms of the model rules (specific options and settings captured within the models). To do this analysis we first paired the exhaustive and reduced models for each test case. We then divided the pairs of models into four categories based on the *strength* of the F measures of the exhaustive models: very high ($F = 1$), high ($0.8 < F < 1$), moderate ($0 < F \leq 0.8$), and low ($F = 0$).

For the very high F-measure group the paired models were exactly the same (except for the 2-way models for failures 80-262 and 93-361 on Windows). That is, the exhaustive and reduced models contained the same rules to describe the failures. The two exceptions happened on failures that manifested in relatively few configurations (*i.e.,* in both cases, the number of failing configurations in the entire space was right at the edge of 3% threshold value). Consequently, the 2-way schedules observed the failures in very few configurations (*i.e.*, in 4 configurations on average), which negatively affected the resulting fault characterization models. The similarity between paired models decreased steadily as we moved down to the high and moderate F-measure groups. In the moderate F-measure group, the rules captured

by the reduced models (especially the 2-way models) tended to differ substantially from those captured by the exhaustive models – See failures 52-18, 80-22, and 35-14 in Figure 5.4a. In these cases we saw that using higher strength covering arrays boosted performance.

The low-F measure group comprises models that failed to find any accurate pattern to the failures. Since no accurate pattern is found, the reduced and the exhaustive models may find different, but equally inaccurate patterns.

These results suggest that covering array schedules can generate data that is capable of accurately characterizing the options and option settings in which specific option-related failures manifest. Moreover, as we will show in Section 5.5, the concept of pattern strength gives us a way to determine whether the classification tree model is likely to be reliable, and, therefore, likely to help developers find an actual failure cause.

5.3.3 Study 3: Covering arrays with per test, failure case characterization

Building classification models with several classes can lead to situations where there is too little data from which to conclude class assignment or to situations where global model building choices lead to suboptimal models for individual classes.

In this study we attempt to circumvent this problem by building one characterization model for each test and failure combination.

Creating classification tree models

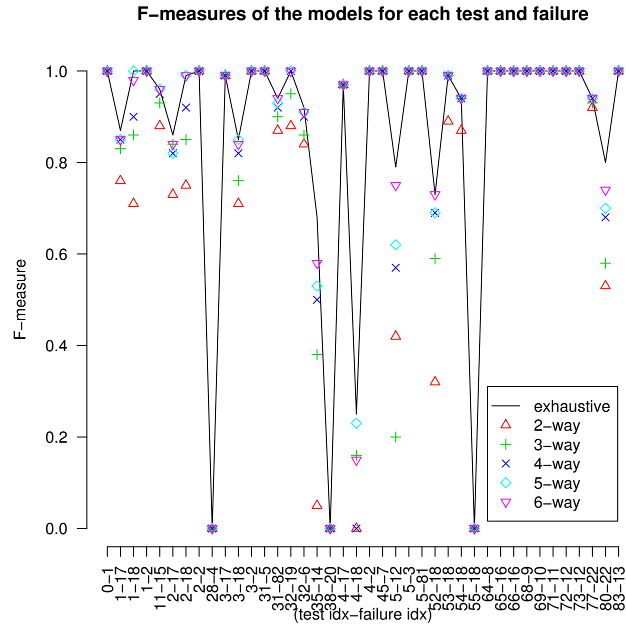
Just as in Study 2, we ran all test cases on every configuration in the configuration space and recorded their pass/failure information. For each test and failure f we created a training data set. Here we recoded the test outcomes into two classes: those failing with failure f and those passing. We repeated the process with the CA schedules and compared the results.

Evaluation

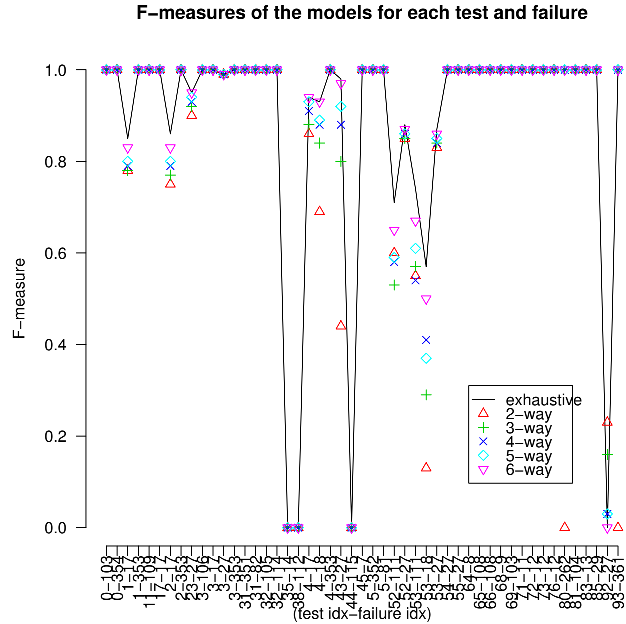
Figure 5.5 shows the F measures for the models. At a first glance this performance is indistinguishable from that of Study 2. Consequently, our findings in Study 2 also apply to this study.

One important way in which these two approaches differ however is in the readability of the resulting models. When we build one model for multiple failures, as we did in Study 2, extraneous information can creep into the patterns that describe the different failures.

Figure 5.6(a), (b) and (c) illustrate this situation. Figure 5.6(a) shows the characterization for two failures that occurred during the execution of test #3 on Linux (we've excluded other errors to simplify the discussion). This model says that error #2 occurs when CALLBACK==0 and that error #17 occurs when CALLBACK==1 and ORBCollocation==NO. Although this seems like a reasonable classification, it is slightly inaccurate. The *actual* causes of these errors are known to us from previous studies. We describe them next.



(a) Linux



(b) Windows

Figure 5.5: Models for each test and failure combination.

```
CALLBACK=0:ERR #2
CALLBACK=1
| ORBCollocation=glb:PASS
| ORBCollocation=orb:PASS
| ORBCollocation=NO:ERR #17
```

(a)

```
CALLBACK=0:ERR #2
CALLBACK=1:PASS
```

(b)

```
ORBCollocation=glb:PASS
ORBCollocation=orb:PASS
ORBCollocation=NO:ERR #17
```

(c)

Figure 5.6: Fault characterizations for test #3, test #3 and error #2, and test #3 and error #17, respectively.

Error #2 occurs during the compilation of the test case. Certain files within TAO implementing CORBA messaging incorrectly assume that CALLBACK option would always be set to 1. Consequently, when CALLBACK==0 certain definitions are unset.

Error #17 occurs when the ORBCollocation optimization is turned off. ACE+-TAO's ORBCollocation option controls the conditions under which the ORB should treat objects as being collocated. Turning it off means that objects should never be treated as being collocated. When objects are not collocated they call each other's methods by sending messages across the network. When they are collocated, they can communicate directly, saving networking overhead. The fact that these tests work when objects communicate directly, but fail when they talk over the network clearly suggests a problem related to message passing. In fact, the source of the problem was a bug in the routines for marshaling/unmarshalling object references.

Returning to Figure 5.6(a), we know that error #2 occurs when CALLBACK-==0 and that error #17 occurs when ORBCollocation==NO. That is, the setting of CALLBACK has no effect on the manifestation of error #17. The appearance of the CALLBACK option in the pattern for error #17 is an artifact of the modeling process when there are multiple classes being modeled together. When we remove

```
POLLER=0
| DIOP=0
| | INTERCEPTOR=0
| | | MUTEX=0:PASS
| | | MUTEX=1:ERR #18
| | INTERCEPTOR=1:ERR #18
| DIOP=1
| | INTERCEPTOR=0:ERR #18
| | INTERCEPTOR=1
| | | MUTEX=0:ERR #18
| | | MUTEX=1:PASS
POLLER=1:PASS
```

(a)

```
POLLER=0:ERR #18
POLLER=1:PASS
```

(b)

```
POLLER=0
| MUTEX=0
| | INTERCEPTOR=0:PASS
| | INTERCEPTOR=1:ERR #18
| MUTEX=1:ERR #18
POLLER=1:PASS
```

(c)

Figure 5.7: Fault characterizations for error #18 obtained from the exhaustive schedule, 2-way covering arrays, and 3-way covering arrays, respectively.

this coupling and build a separate model for each test and failure combination, this problem doesn't appear. In fact, the fault characterizations, shown in Figures 5.6(b) and (c), are exact and are the actual causes of the failures.

Using this per test, per failure characterization, we see that as the strength of the covering arrays increases, fault characterizations move closer to the ones obtained from the exhaustive schedule. We illustrate the differences among the characterizations obtained from different strength covering arrays in Figure 5.7.

Figure 5.7(a), (b), and (c) show the fault characterizations obtained from the exhaustive schedule, 2-way covering arrays, and 3-way covering arrays, respectively for error #18 which occurred during the execution of test #3 on Linux platform.

The exhaustive model correlates the failure with four options and gives an F measure of 0.849. The 2-way model is able to link the failure to only one option. This results in an F measure of 0.747. On the other hand, the 3-way model associates the failure with three options and resulted in a better F measure, (0.795), than the 2-way model.

Schedule	Size
2-way-combined	344.20
3-way-combined	1357.60
4-way-combined	3450.60
5-way-combined	8422.00

Table 5.6: Size of combined schedules.

5.3.4 Study 4: Combined reduced schedules

As shown in Table 5.5, the size of the CA test schedules grows rapidly as t increases. The cost to create them does as well (the cost is exponential in t). In this study we examined how combined lower strength schedules compare to single higher strength covering arrays (e.g., 3, 2-way covering arrays vs. 1, 3-way covering array).

Specifically, we combined schedules in such a way that the size of the combined t-way schedules is close to the size of a single $(t+1)$ schedule. We then compared the combined schedules to the uncombined ones. This is interesting because the cost of creating $(t+1)$-way schedules can be significantly higher than the cost of obtaining t-way schedules. If t-way-combined and $(t+1)$-way schedules have comparable performance measures then using the combined schedules can be cost-effective.

Creating classification tree models

We created combined t-way schedules by merging randomly selected uncombined t-way schedules. No duplicate test configurations were allowed. We created 5 combined schedules for t from 2 to 5. We didn't combine 6-way schedules because the average size of the 6-way schedules was almost half that of the exhaustive

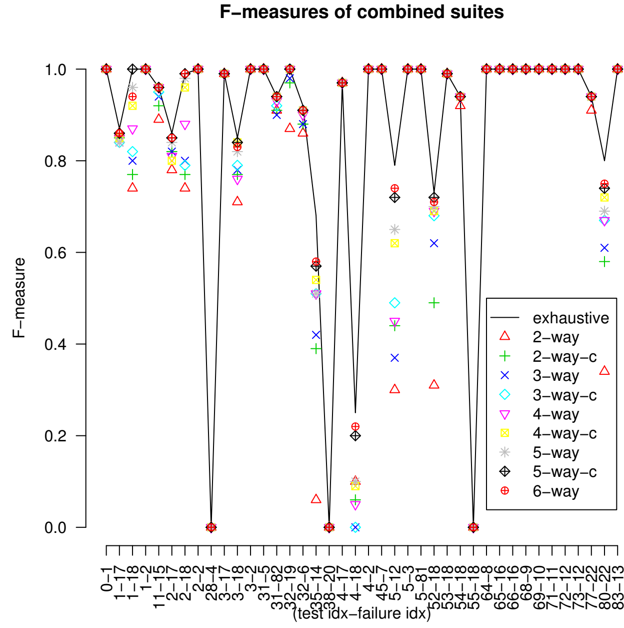
schedule. The average sizes of the t-way-combined schedules are given in Table 5.6. Classification models were built as in Study 3.

Evaluation

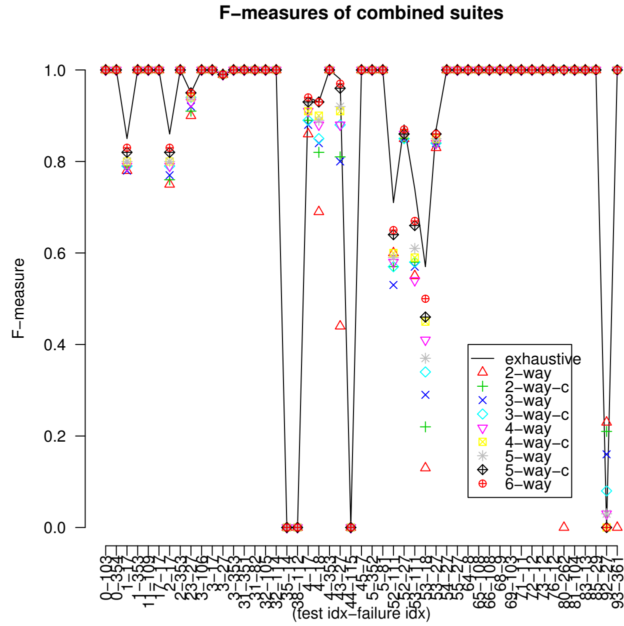
Figure 5.8 plots the F measures for t-way and t-way-combined schedules. t-way-combined schedules result in better fault characterizations than the t-way ones, but do not do quite as well as the $t+1$-way ones. In particular, the combined schedules boost the characterizations of faults while single lower strength schedules give low F measures (*i.e.*, less than 0.5).

For example, consider the 2-way, 2-way-combined (2-way-c) and 3-way models for test #35, error #14 shown in Figure 5.8a. The F measures for these models are 0.06, 0.39 and 0.42 respectively. The combined schedule gives an F measure that is much closer to that of the 3-way schedule. On the other hand, when the F measures of single schedules are already high (say greater than 0.5), the combined schedules don't improve performance to a great degree.

One possible explanation for the closeness in results between the $t+1$-way and combined schedules is that the combined schedules cover 82-89% of the $t+1$ tuples. Thus, they provide many of the data points seen in the $t+1$ covering arrays, but at a lower construction cost.



(a) Linux



(b) Windows

Figure 5.8: Models for combined schedules.

5.4 Further Improving the Efficiency

Our current sampling strategy is based on computing a t-way covering array that covers all t-way combinations of option settings. This sampling strategy, by fixing t, the strength of the array, across the entire configuration space, treats configuration spaces as flat spaces; each t-way combination of option settings are considered equally likely to cause failures. However, our experience suggests that configuration spaces are often composed of several subspaces each with, potentially, a different level of risk of causing failures. For example, our experiment data on ACE+TAO shows that faults tended to be concentrated all in static options or all in runtime options, not generally a mix of both. Testing higher-level interactions in high-risk subspaces while keeping a relatively low-level interaction coverage in the overall space can improve the efficiency of the fault characterization process.

To apply this new approach, we need a new sampling strategy which enables us to vary the coverage requirements across the space. Consequently, we decided to use variable strength covering arrays (VSCA) as our new sampling strategy (See Section 5.1.2 for more details).

We hypothesize that VSCAs can improve the efficiency of the fault characterization process in two ways: (1) they can reduce the cost of the process without compromising its accuracy by only testing the required set of high-level interactions or (2) for the same cost, they can improve the accuracy of the process by testing more interactions. We evaluate this hypothesis in the rest of this section.

5.4.1 Creating Variable Strength Covering Arrays

We have created several VSCAs for our configuration model given in Section 5.3. Our strategy was to use higher strength coverage between only the runtime options. The reason behind this strategy is two-fold. First, we observed that a significant fraction of the failures we saw involved runtime options. Therefore, testing higher level interactions between runtime options can improve the characterization models for these faults without compromising the others. Secondly, the overriding factor in the size of our covering array is the single static option; in the covering array model we used, the 10 compile time options were grouped into a single option with 29 settings whereas the runtime options have at most 4 option settings. Leveraging this fact, by manipulating the configuration space of the runtime options independently, allows us to create VSCAs with 2 levels of strength (*i.e.,* the highest level of strength is assigned to the runtime options) and with overall sizes very close to that of our fixed strength covering arrays. This provides a way to reliably evaluate the performance boost due to VSCAs by comparing them to similar sized CAs.

We created our VSCAs with the highest level of strength that could be obtained for all of the runtime options, that would create a VSCA very close to the size of one of our covering arrays with fixed strength. The first two VSCAs created have a base strength of 2. In the first one, a $VSCA(N; 2, 29^1 4^1 3^4 2^1, MCA(N; 4, 4^1 3^4 2^1))$, the 6 runtime options have strength 4. The size of this VSCA is 116 which is exactly the same as the fixed strength

covering array with $t = 2$. We call this array the "2-way-overall-4-way-runtime" (abbreviated as the "2c4r" array). The second VSCA created has $t = 5$ for all of the runtime options, $(VSCA(N; 2, 29^1 4^1 3^4 2^1, MCA(N; 5, 4^1 3^4 2^1))$. This VSCA has 324 configurations which is slightly less than the 3-way fixed strength array (348 configurations). This array is called the "2-way-overall-5-way-runtime" array (abbreviated as "2c5r"). The third VSCA, $((VSCA(N; 3, 29^1 4^1 3^4 2^1, MCA(N; 5, 4^1 3^4 2^1)\ )$, has a base strength of $t = 3$, while the 6 runtime options are of strength $t = 5$. This is called the "3-way-overall-5-way-runtime", (abbreviated "3c5r"). This array has 367-368 configurations which is comparable with the size of the 3-way arrays.

By creating 2-way-overall-4-way-runtime, 2-way-overall-5-way-runtime, and 3-way-overall-5-way-runtime test schedules, we expect to improve the efficiency of the process in characterizing faults which are caused by interaction of 4 or more runtime options.

5.4.2 Evaluating Variable Strength Covering Arrays

As in Section 5.3, we computed five different schedules for each of 2-way-overall-4-way-runtime, 2-way-overall-5-way-runtime and 3-way-overall-5-way-runtime array. We ran all test cases on every configuration selected by these VSCAs and recorded their pass/failure information. We created fault characterization models for each test and failure as described in Section 5.3.3 and then compared the resulting models to those of fixed strength covering arrays where $t = 2, 3, 4$.

Table 5.7 compares the F measures of characterization models obtained from

Failure	OS	2-way	2c4r	2c5r	3-way	3c5r	4-way
2-17	Linux	0.78	0.81	0.83	0.81	0.83	0.81
80-22	Linux	0.34	0.51	0.65	0.61	0.65	0.67
4-18	Windows	0.69	0.79	0.83	0.84	0.85	0.88

Table 5.7: Comparing fault characterization models obtained from VSCAs and CAs using F measures.

VSCA and CA schedules for some failures caused by runtime option settings. In this table, we observe that (1) the 2c4r schedules improve the fault characterizations over the same sized 2-way schedules, (2) the 2c5r schedules, compared to the 3-way schedules, result in comparable–in most cases better–characterizations while providing a 6% reduction in the number of configurations to be tested, and (3) the fault characterization models obtained from 3c5r schedules are always better than those of 3-way schedules.

Although these results are encouraging, they are by no means conclusive. The fundamental reason behind this is that we had a very limited number of cases in which we could have observed the performance improvement due to VSCAs. The fixed strength schedules, even the low strength ones (e.g., 2-way and 3-way schedules), almost always resulted in perfect models (*i.e.,* F$\approx$1) for faults caused by interactions of runtime options. This may suggest that there are a limited number of faults involving four or more runtime options in our experimental data, which would prevent us from evaluating VSCAs properly.

To further evaluate VSCAs, we decided to run a clean room experiment where we controlled faults, the frequencies of failures, and the options which are responsible for the manifestation of the failures. Consequently, we chose to seed faults into our

configuration space.

5.4.3 Seeding Faults

In order to evaluate the performance boost due to the VSCAs, we decided to seed 4-way faults into our configuration space (the same configuration space that we used in our earlier experiments), which are caused by simultaneous interactions of 4 runtime options.

We randomly selected 4 runtime options from our configuration space (one option with two levels of setting and three options with three levels of setting each). We then seeded a unique fault for each combination of these runtime option settings. This gave us 54 unique 4-way faults. For each fault, we then created a separate test case, failing deterministically only on configurations in which the right combination of option settings is met. We repeated the same processes to seed faults with various occurrence frequencies (*i.e.,* 80%, 60%, 40%, and 20%). At an x% occurrence frequency, failures manifest themselves only at x percent of the configurations in which the failure inducing conditions are met.

For each occurrence frequency, we ran all 54 hypothetical test cases on the fixed strength and variable strength schedules, and recorded their pass/failure information. We computed the fault characterization models using the Weka ld3 algorithm [77][1] for each test and failure and then compared the resulting models.

Figure 5.9, grouped by the occurrence frequency, compares the distributions of the F measures obtained from the fixed and variable strength schedules. We

[1] The ld3 algorithm performs better than the J48 on small training data sets

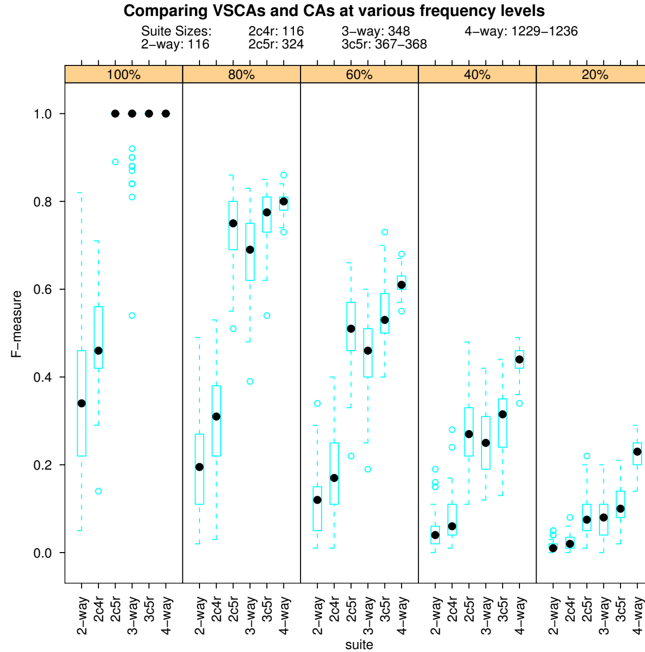


Figure 5.9: Comparing VSCAs with CAs at various frequency levels.

observed that the VSCAs resulted in better characterization models when compared with their fixed cost case counterparts (size-wise), *i.e.*, 2-way vs. 2c4r, 3-way vs. 2c5r and 3-way vs. 3c5r. The differences were more pronounced at the 80% level. For instance, at the 100% occurrence frequency, the 2c4r provides a better classification, but once a larger subset of configurations are included, all of the characterization models do equally well. At the 80% level all of the VSCAs show improvement over their CA counterparts. For instance the 2c5r improves over the 3-way CA even though the 2c5r contains slightly fewer configurations. The performance differences gradually begin to diminish, however, as the level of occurrence frequencies drop below 80%.

5.5 Guidelines for Software Practitioners

We have evaluated our fault characterization process by comparing it to the results of exhaustive testing. In practice, developers will not have access to this information. Therefore, in this section, we provide preliminary guidelines on how to use this approach in practice. In particular, we examine how to interpret reduced models, how to estimate whether the reduced models are reliable, how to select the appropriate strength level for the covering arrays, how to vary the strength across the configuration spaces and how to work with a set of models.

We begin with describing an analysis method that we applied to our experiment results and then give guidelines for fixed strength covering arrays based on this analysis. We then provide guidelines for variable strength covering arrays based on our experience.

Classification tree models can be partially evaluated without a traditional test set. Typically this is done using a k-fold stratified cross-validation strategy [77]. Assuming that $k = 10$, for example, the training data is randomly divided into ten parts. Within each part the classes should be represented in approximately the same proportions as in the original data set. Next, for each of the 10 parts, a model is built using the remaining nine-tenths of the data and tested to see how well it predicts that part. Finally, the ten error estimates are averaged to obtain an overall error rate. A high error rate indicates that the models are highly sensitive to the subset of the data with which they are constructed. This suggests that the models may be "overfit" and shouldn't be trusted.

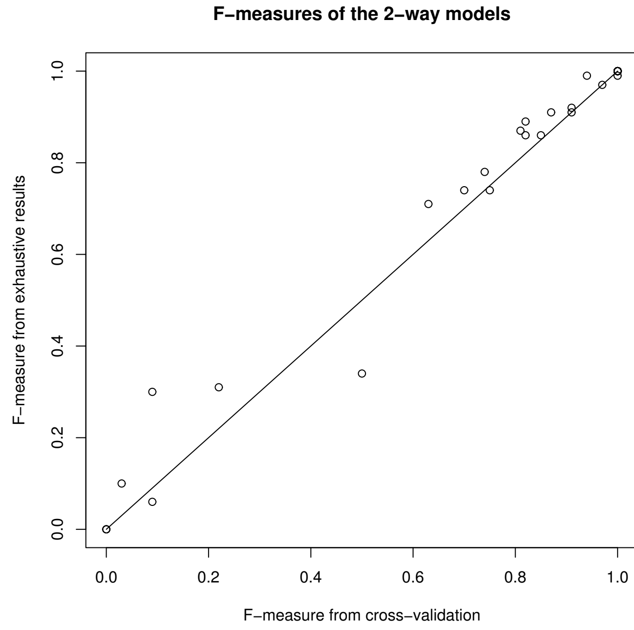
We performed stratified ten-fold cross-validation on our reduced models from Study 3. We only present the analysis results obtained from the Linux experiments. The results for the Windows experiments are similar. We found that whenever the reduced model's cross-validation F measures were 0, the failure was either very rare (not considered option-related) or was an option-related failure for which even the exhaustive model couldn't find a fault characterization (i.e., $F = 0$). These failures were, namely 28-4, 38-20, and 55-18. This suggests that models with 0 F measures are unlikely to signal option-related failures.

As a next step, we investigated the relation between the cross-validation F measures and the F measures of the exhaustive models. Figures 5.10(a) and (b) depict scatter plots of these two F measures for the 2-way and the 4-way models, respectively. We show only two figures due to space limitations. The trends of the other models are similar. We see the two F measures are very similar (they lie near the $x = y$ line). The higher the strength of the arrays, the closer the F measures are.

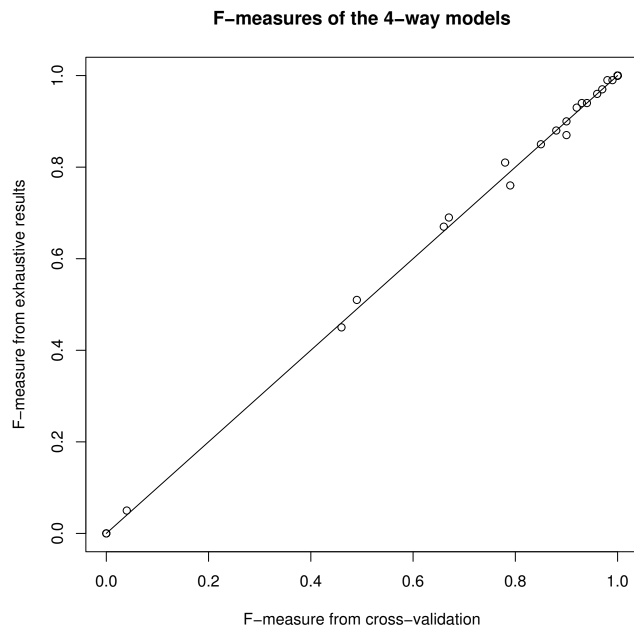
This suggests that F measures from the cross-validation of reduced models can help estimate the performance of the models when they are applied to the exhaustive results.

Based on the findings above, we give the following guidelines to users of fixed strength covering arrays:

1. Use the F measures obtained from cross-validations of reduced models to flag unreliable models.



(a)



(b)

Figure 5.10: Scatter plots of F measures for 2-way and 4-way models.

2. Higher F values are more likely to signal accurate fault characterizations, which in turn can help pinpoint the causes of failures quickly and accurately. Investigate the models with the highest F-measures first.

3. Consider using higher strength covering arrays or combined ones for the failures whose F values are low (i.e., less then 0.5).

The users of the variable strength covering arrays, in addition to the guidelines above, need to know how to vary t across the entire configuration space. As we described in Section 5.1.2, VSCAs are desirable when it is too expensive to use a higher t for all options. Based on our experience with highly-configurable systems and VSCAs, we present the following guidelines to users:

1. Leverage apriori knowledge of the system under test, if it is available. Information that leads to high-risk subspaces are valuable, *e.g.,* information recommending that a subset of option combinations is more likely to cause failures, or that recent changes in the code base affect a certain set of option interactions, etc. Consider assigning higher level strengths to high-risk subspaces.

2. Leverage fixed strength covering arrays to pinpoint high-risk subspaces, if no or limited reliable apriori information is available. Start with a fixed strength covering array and analyze the resulting fault characterization models to identify subsets of options that are highly correlated with the manifestation of failures. Consider assigning higher level strengths to these subsets.

3. Leverage the fact that there may be some configuration options that dictate the size of the covering arrays. These options are the ones which have the largest number of settings. For example, in our experiments, the overriding option in the size of the covering arrays was the one static option with 29 settings. Consider manipulating the configuration space of non-overriding options independently via assigning higher strengths. Depending on the configuration space, this strategy may provide higher strength coverage at no or reasonable cost (See Section 5.4.1 for more details).

5.6 Comparison with Random Schedules

In this section, we compare the effectiveness of t-way and randomly selected schedules using our experiment data on ACE+TAO. For this, we created 100 random schedules for each value of t where the size of each random schedule is the same as the corresponding t-way schedule. Since the CAs and VSCAs we created in this research are comparable in size, the results obtained from this section are also applicable to VSCAs unless otherwise stated.

Our first concern was to see how well the random schedules revealed failures. Figure 5.11 contains boxplots for the number of failures observed by the random and t-way schedules conditioned on t. In general we see that the higher the value of t (and thus the larger its size), the greater the number of failures observed. The t-way schedules tend to reveal slightly more failures than the corresponding random schedules with less variance.

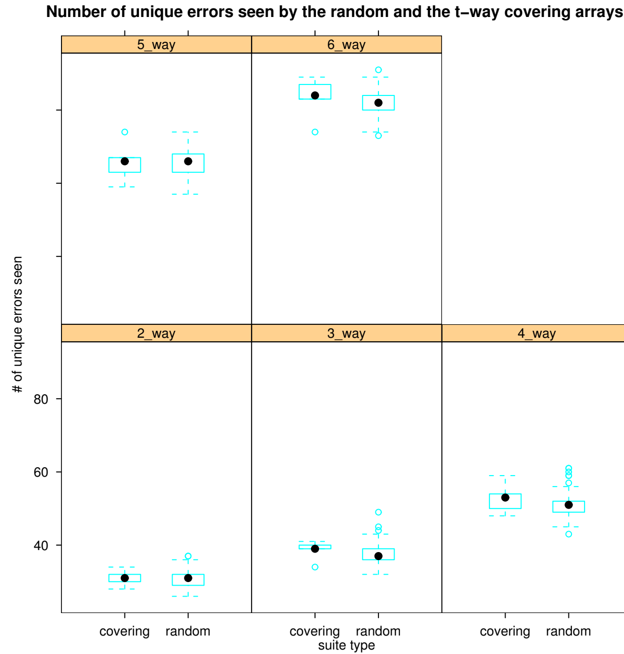


Figure 5.11: Number of unique errors seen in random and t-way covering arrays.

Next we evaluated the two scheduling approaches in terms of their fault characterizations. For this, we randomly chose 15 randomly selected schedules for each value of t and created the classification tree models for option-related failures. In general, we observed that random and t-way schedules yielded comparable fault characterization models.

Random schedules, however, sometimes completely missed option-related failures or resulted in unbalanced sampling of the failing subspaces. In the first situation, obviously, the models ignored the failure because it had not been observed when running the random schedule. The second situation occurs when some parts of the configuration space are tested much more frequently than others. This often led to spurious options to be included in the models.

Figure 5.12 illustrates this situation by contrasting the fault characterizations

```
POLLER=0
| MUTEX=0
| | INTERCEPTOR=0:PASS
| | INTERCEPTOR=1:ERR #18
| MUTEX=1
| | INTERCEPTOR=0:ERR #18
| | INTERCEPTOR=1:PASS
POLLER=1:PASS
```

(a)

```
POLLER=0:ERR #18
POLLER=1:PASS
```

(b)

```
POLLER=0
| ConnectStrategy=0:PASS
| ConnectStrategy=1:ERR #18
| ConnectStrategy=2:PASS
POLLER=1:PASS
```

(c)

Figure 5.12: Fault characterization for test #2, ERR #18 obtained from the exhaustive schedule, a 2-way schedule, and a random schedule, respectively.

for test #2, ERR #18 obtained from the exhaustive schedule, a 2-way schedule, and a random schedule. The F measures for the models are 0.993, 0.774, and 0.436, respectively. The exhaustive schedule gave the model shown in Figure 5.12(a). Compare this to the 2-way schedule appearing in Figure 5.12(b). The latter is simpler and thus incorrect in some cases because it doesn't recognize the importance of the MUTEX option. Still, it doesn't include any unrelated options that would distract a developer trying to find the cause of the failure.

The model created from the random schedule however (Figure5.12(c)) includes a node for the ConnectionStrategy option right under the node for the POLLER option. Our analysis shows that this option is unrelated to the underlying failure. This happened because, with the random schedule, when POLLER == 0, 86% of the configurations with ConnectionStrategy == 1 fail with ERR #18. Thus, to the model building algorithm ConnectionStrategy == 1 appears to be important in explaining the underlying failure. In contrast, in the exhaustive and 2-way schedules only 21% and 33% of the configurations with ConnectionStrategy == 1 fail. This difference is simply due to an "unlucky" random selection that produced an

unbalanced sampling of the underlying configuration space.

In summary, we observed that random and t-way schedules gave comparable fault characterizations on the average, but that the random schedules sometimes created unreliable models. Moreover, in practice, the covering array approach automatically determines the size of the schedule, whereas there's no way to predetermine the correct size of a randomly selected schedule.

5.7 Discussion

Fault characterization in configuration spaces can help developers quickly pinpoint the causes of failures, hopefully leading to much quicker turn-around time for bug fixes. Therefore, automated techniques, which can effectively, quickly, and accurately perform fault characterization, can save a great deal of time and money throughout the industry. This is especially true where system configuration spaces are large, the software changes frequently, and resources are limited.

To make the process more efficient, we first recast the problem of selecting test schedules (determining which configurations to test) as a problem of calculating a fixed strength, t-way covering array over the system configuration space. Using this schedule, we ran tests and fed the results to a classification tree algorithm to localize the observed faults. We then compared the fault characterizations obtained from exhaustive testing to those obtained via the covering array-derived schedule.

- We observed that building fault characterizations for each observed fault rather than building a single one for all observed faults led to more reliable models.

- We observed that even low strength covering arrays, which provided up to 99% reduction in the number of configurations to be tested, often had fault characterizations that were as reliable as those created through exhaustive testing.

- Higher strength covering arrays performed better than lower strength ones and yielded more precise fault characterizations, but were more costly.

- We showed that we can improve the fault characterization accuracy at a low construction cost by combining lower strength covering arrays rather than increasing the covering array strength.

We were also able to develop some diagnostic tools to support software practitioners who want to use fixed strength covering arrays in fault characterizations. In particular we found that:

- Low F measures in the exhaustive models tended to be associated with overfit models or non-option-related failures. These models are not likely to help developers identify option-related failures.

- We found that the F measures taken from 10-fold cross-validation were highly correlated and nearly identical with those taken from exhaustive models. This suggests that that cross-validation measures, which can be taken without having already done exhaustive testing, might be a useful surrogate for the exhaustive model F measures.

To further improve the fault characterization process, we introduced and evaluated the use of a new kind of covering array, called a variable strength covering array, as a sampling strategy. Variable strength covering arrays, unlike their fixed strength counterparts, allow us to test higher level interactions only in subspaces where they are needed (*i.e.,* in high-risk subspaces), while keeping a low level of coverage across the entire space. We computed several variable strength arrays to focus testing on the runtime options. Their sizes were close to those of the fixed strength arrays, allowing us to make comparisons. To gain a better insight into the usefulness of these arrays we conducted a simulation where we seeded 4-way interaction faults into our configuration space. We observed that variable strength arrays slightly improved the efficiency of the fault localization process in two ways:

- They reduced the cost of the process without compromising its accuracy.
- For the same cost, they improved the accuracy of the process.

We also provided users of variable strength covering arrays with guidelines on how to vary t across the configuration space.

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice. One potential threat is the representativeness of the ACE+TAO subject applications, which though large are still just one suite of software systems. A related issue is that we have focused on a relatively simple and small subset of the entire configuration space of ACE+TAO; the actual configuration space is much

larger. While these issues pose no theoretical problems we need to apply our approach to larger, more realistic configuration spaces in future work to understand how well it scales.

Chapter 6

Main Effects Screening: A DCQA Process for Monitoring Performance Degradations in Evolving Software Systems

The quality of performance-intensive software systems, such as high-performance scientific computing systems and distributed real-time and embedded (DRE) systems, depend heavily on their infrastructure platforms, such as the hardware, operating system, middleware, and language processing tools. Developers of such systems often need to tune the infrastructure and their software applications to accommodate the (often changing) platform environments and performance requirements. This tuning is commonly done by (re)adjusting a large set (10's-100's) of compile- and run-time configuration options that record and control variable software parameters, such as different operating systems, resource management strategies, middleware and application feature sets; compiler flags; and/or run-time optimization settings.

Although these software parameters promote flexibility and portability, they also require that the software be tested in an enormous number of configurations. This creates serious challenges for developers who must ensure that their decisions, additions, and modifications work across this large (and often changing) configuration space:

- Settings that maximize performance for a particular platform/context may

not be suitable for different ones and certain groups of option settings may be semantically invalid due to subtle dependencies between options.

- Limited QA budgets and rapidly changing code bases mean that developers' QA efforts are often limited to just a few software configurations, forcing them to extrapolate their findings to the entire configuration space.

- The configurations that are tested are often selected in an *ad hoc* manner, so quality is not evaluated systematically and many quality problems escape detection until systems are fielded.

Since exhaustively testing all configurations is infeasible under the circumstances listed above, developers need a quick way to estimate how their changes and decisions affect software performance across the entire configuration space.

Developers of highly configurable performance-intensive software systems often use a type of performance-oriented "regression testing" to ensure that their modifications have not adversely affected their software's performance across its large configuration space. Unfortunately, time and resource constraints often limit developers to benchmarking of a small number of configurations and unreliable extrapolation from these results to the entire configuration space. This causes many performance degradation to escape detection until systems are fielded.

In this chapter, to improve performance assessment of evolving systems across large configuration spaces, we introduce and evaluate a DCQA process called *main effects screening*. We also introduce and evaluate an accompanying observation-based space reduction strategy to reduce the configuration space to a manageable

size, allowing more targeted QA to be performed. This strategy executes formally designed experiments across the configuration space to identify a subset of "important" performance-related configuration options. Whenever the software changes thereafter, the main effects screening process exhaustively explores all configurations of the important options to get a reliable estimate of the overall system performance across the entire configuration space. These estimates are then used to monitor performance degradations. This approach is feasible because the new configuration space is much smaller than the original, and hence more tractable using limited resources.

We present an evaluation of our new DCQA process via several feasibility studies on several large, widely-used performance-intensive software systems. Our results indicate that (1) the main effects screening process can correctly identify the subset of options that are important to system performance, (2) monitoring only these selected options helps to quickly detect key sources of performance degradation at an acceptable level of effort, and (3) alternative techniques with equivalent effort give less reliable results.

The remainder of this chapter is organized as follows: Section 6.1 introduces the main effects screening process; Section 6.2 presents the feasibility studies conducted; Section 6.3 validates the basic assumptions made in the these studies; Section 6.4 presents practical guidelines for practitioners; and Section 6.5 presents concluding remarks.

6.1 Performance-Oriented Regression Testing

As software systems change, developers often run regression tests to detect unintended functional side effects. Developers of performance-intensive systems must also be wary of unintended side effects on their system performance. To detect such performance problems, developers often run benchmarking regression tests periodically. However, time and resource constraints (and often high change frequencies) severely limit the number of configurations that can be examined. For example, our experience with the ACE+TAO systems suggest that only a small number of default configurations are benchmarked routinely by the core development team, who thus get a *very* limited view of their middleware's performance. Problems not readily seen in these default configurations therefore often escape detection until systems based on ACE+TAO are fielded by end-users.

This section describes how we address this problem by using Skoll to develop and implement a new DCQA process called *main effects screening*. We also describe the formal foundations of our approach, which is based on *design of experiments theory* [78], and give an example that illustrates key aspects of our approach.

6.1.1 The Main Effects Screening Process

Main effects screening is a technique for rapidly detecting performance degradation across a large configuration space as a result of system changes. Our approach relies on a class of experimental designs called *screening designs* [78], which are highly economical and can reveal important *low order effects* (such as individual

option settings and option pairs/triples) that strongly affect performance. We call these most influential option settings "main effects."

At a high level, main effects screening involves the following steps: (1) *compute* a formal experimental design based on the system's configuration model, (2) *execute* that experimental design across the Skoll DCQA grid by running and measuring benchmarks on specific configurations dictated by the experimental design devised in step 1, (3) *collect, analyze and display* the data so that developers can identify the main effects, (4) *estimate* overall performance whenever the software changes by evaluating all combinations of the main effects (while defaulting or randomizing all other options), and (5) *recalibrate* the main effects options by restarting the process periodically since the main effects can change over time, depending on how fast the system changes.

The assumption behind this five step process is that since main effects options are the ones that affect performance the most, evaluating all combinations of these option settings (which we call the "screening suite") can reasonably estimate performance across the entire configuration space. If this assumption is true, testing the screening suite should provide much the same information as testing the entire configuration space, but at a fraction of the time and effort since it is much smaller than the entire configuration space.

6.1.2 Technical Foundations of Screening Designs

For main effects screening to work we need to identify the main effects, *i.e.*, the subset of options whose settings account for a large portion of performance variation across the system's configuration space. One obvious approach is to test every configuration exhaustively. Since exhaustive testing is infeasible for large-scale, highly configurable performance-intensive software systems, developers often do some kind of random or *ad hoc* sampling based on their knowledge of the system. Since our experience indicates that these approaches can be unreliable, we need an approach that samples the configuration space, yet produces reasonably precise and reliable estimates of overall performance.

The approach we chose for this research uses formally-designed experiments, called *screening designs*. Screening designs are highly economical experimental designs whose primary purpose is to identify important low-order effects, *i.e.*, first-, second-, or third-order effects. In general, an nth-order effect is an effect caused by the simultaneous interaction of n factors. For instance, for certain web server applications, a 1st-order effect might be that performance slows considerably when logging is turned on and another might be that it also slows when few server threads are used. A 2nd order effect involves the interaction of two options, *e.g.*, web server performance may slow down when caching is turned off *and* the server performs blocking reads.

There are many ways to compute screening designs. The one we use is based on traditional factorial designs. Consider a *full* factorial design involving k binary

factors. Such a design exhaustively tests all combinations of the factors. Therefore, the design's run size (number of experimental observations) is 2^k. Although this quickly becomes expensive, it does allow one to compute all effects.

To reduce costs statisticians have developed *fractional* factorial designs [78]. These designs use only a carefully selected fraction (such as 1/2 or 1/4) of a full factorial design. This saves money, but does so by giving up the ability to measure some higher-order effects. This is because the way observations are chosen aliases the effects of some lower-order interactions with some higher-order ones. That is, it lumps together certain high- and low-order effects on the assumption that the high-order effects are negligible.

Screening designs push this tradeoff to an extreme. Roughly speaking, at their smallest, screening designs require only as many observations as the number of effects one wishes to calculate (*i.e.*, k observations to compute k 1st-order effects). Of course, experimenters will often use more than the minimum number of observations to improve precision or to deal with noisy processes. As before, this is only possible because the design aliases some low-order with some high-order effects.

While this aliasing may seem problematic, screening designs have been used extensively to understand and improve products and processes developed in manufacturing, engineering, and physical sciences. Their success stems largely from the ability to use them in an iterative, "quick and dirty" fashion, *i.e.*, to focus on major problem sources, a few at a time, rather than trying to understand and fix all problems simultaneously. Since our objective with main effects screening is also to produce a rough – but reliable – estimate of overall performance, we conjecture that

screening designs provide the appropriate foundation for our DCQA process.

6.1.3 Screening Designs in Action

To show how screening designs are computed, we now present a hypothetical example of a software system with 4 binary configuration options, A through D, with no inter-option constraints. To compute a specific screening design, developers must (a) decide how many observations they can afford, (b) determine which effects they want to analyze, and (c) select an aliasing strategy consistent with these constraints. Note that in practice screening designs are usually computed by automated tools.

The configuration space for our sample system has $2^4 = 16$ configurations. Therefore, the corresponding full factorial design involves 16 observations. Let's assume that our developers, however, can afford to run only 8 observations. Obviously, the full factorial design would therefore be unacceptable. Let's also assume that our developers are mostly interested in capturing the 4, 1st-order effects (*i.e.*, the effect of each option by itself.) Given this goal the developers decide to use a screening design.

The first step in computing the screening design is to create a 2^3 full factorial design over 3 (arbitrarily selected) options, in this case A, B, and C. This design is shown in Table 6.1(a) with the binary option settings encoded as (-) or (+). This is the starting point because $2^3 = 8$ is the maximum number of observations the developers can afford to run.

This initial design would be fine for 3 options. But it can't handle our devel-

A	B	C
-	-	-
+	-	-
-	+	-
+	+	-
-	-	+
+	-	+
-	+	+
+	+	+

(a)

A	B	C	D
-	-	-	-
+	-	-	+
-	+	-	+
+	+	-	-
-	-	+	+
+	-	+	-
-	+	+	-
+	+	+	+

(b)

Table 6.1: (a) 2^3 design and (b) 2^{4-1}_{IV} design

oper's 4-option system since they've already decided that the full factorial design would be too expensive. We need a way to stretch the current 3-option design to cover the 4th option.

In an experiment, each effect is estimated by using at least one degree of freedom. Our current design has $2^3 - 1 = 7$ degrees of freedom. We use 3 degrees of freedom to estimate the effects of A, B, and C. The remaining degrees of freedom would normally be used to estimate higher-order effects, but since we are only interested in the 1st order effects, we can instead use them to estimate the effect of option D, *i.e.*, we can extend the design and estimate the effect of option D without going to a 2^4 full factorial design. As mentioned previously, this done by aliasing some effects.

Here the developers must specify which effects can be safely aliased. They do this by choosing a desired *resolution* for the screening design. In resolution R designs, no effects involving i factors are aliased with effects involving less than $R-i$ factors. Since our developers are only interested in computing 1st-order effects, they choose a resolution IV design. With this design, 1st-order effects will be aliased

with 3rd-order or higher effects and 2nd-order effects will be aliased with 2nd-order or higher effects. Our developers, therefore, are assuming that 3rd-order or higher effects are negligible and that they are not interested in computing the 2nd-order effects. If the developers are unwilling to make these assumptions, then screening designs may be inappropriate for them (or they may need to use the screening designs in an iterative, exploratory fashion).

The final step in the process to compute the specific values of option D consistent with the developers desire for a resolution IV design. To do this a function called a *design generator* must be determined. This function computes the setting of option D based on the values of options A, B and C. In our hypothetical example, automated tools chose $D = ABC$ as the design generator. This means that for each observation, D's setting is computed by multiplying the settings for options A, B, and C (think of + as 1 and - as -1). In general there is no closed-form solution for choosing a design generator. In fact, in some situations, none may exist. For this chapter we identified design generators using the factex function of the SAS/QC package [68]. This search-based function essentially looks through numerous possible design generators until an acceptable one is found (or until the algorithm gives up, if none can be found).

Table 6.1(b) gives the final design, which is identified uniquely as a 2^{4-1}_{IV} design with the design generator $D = ABC$. The 2^{4-1}_{IV} designation means that the total number of options is 4, that we will examine a 1/2 ($2^{-1} = 1/2$) fraction of the full factorial design, and that the design is a resolution IV screening design.

Design generators introduce aliases to the design. For example, our design

generator $D = ABC$ aliases the effect of option D with the interaction effect of option A, B, and C; the estimate of the effect of option D includes the influence of 3rd-order effect ABC. That is, at the end of the experiment, the collected data cannot distinguish the effect of D from the effect of ABC. $D = ABC$ is not the only aliasing structure for our example design.

Notationally, the aliasing relation of the design is also denoted by $\mathbf{I} = ABCD$. $\mathbf{I} = ABCD$ is called the *defining relation* of this design where **I** denotes the column of all +'s (*i.e.,* the product of the column A, B, C, and D is all +'s). The aliases in a design can be revealed by multiplying the effects by the defining relation. For example, in our resolution IV design, to find the alias for option A, we multiply both sides of $\mathbf{I} = ABCD$ by A, that is, $A\mathbf{I} = AABCD$. Since the product of the column A and **I** gives the column A itself (*i.e.,* -+=- and ++=+) and the product of the column A by itself (AA) gives **I** (*i.e.,* --=+ and ++=+), we find out that the effect of option A is aliased with the interaction effect of option B, C, and D ($A = BCD$). By following the same procedure, we can find all the aliases for the 1st-order effects, which are $A = BCD$, $B = ACD$, $C = ABD$, and $D = ABC$. Note that although all the 1st-order effects are aliased, they are all aliased with 3rd-order effects, which is consistent both with our developers' assumption that 3rd-order effects are negligible and with the definition of resolution IV designs.

After defining the screening design, developers can execute it across the Skoll grid. For our process, each observation involves measuring a developer-supplied benchmarking regression test while the system runs in a particular configuration. Once the data is collected we would analyze it to calculate the effects. Since frac-

tional factorial designs are balanced and orthogonal designs [7] (*i.e.,* the observations are un-biased), the effect calculations are simple. For binary options (with settings - or +), the main effect of option A, ME(A), is

$$ME(A) = z(A-) - z(A+) \tag{6.1}$$

where z(A-) and z(A+) are the mean values of the observed data over all runs where option A is (-) and where option A is (+), respectively.

If appropriate, 2nd-order effects can be calculated in a similar way. The interaction effect of option A and B, INT(A, B) is:

$$INT(A, B) = 1/2\{ME(B|A+) - ME(B|A-)\} \tag{6.2}$$

$$= 1/2\{ME(A|B+) - ME(A|B-)\} \tag{6.3}$$

Here $ME(B|A+)$ is called the conditional main effect of B at the + level of A. The effect of one factor (*e.g.*, B) therefore depends on the level of the other factor (*e.g.*, A). Similar equations exist for higher-order effects [78].

Once the effects are computed developers will want to determine which of these effects are important and which are not. There are several ways to determine this, including using standard hypothesis testing approaches. For this chapter we opted not to use formal hypothesis tests primarily because they require strong assumptions about the standard deviation of the experimental samples. In future work we will avoid this problem by simply replicating observations in the experimental design. For this work however, we display the effects graphically and ask developers to use their expert judgment to decide which effects they consider important.

6.2 Feasibility Study

This section describes a feasibility study that assesses the implementation cost and the effectiveness of the main effects screening process described in Section 6.1 on a suite of large, performance-intensive software systems.

6.2.1 Experimental Design

Hypotheses. Our feasibility study explores the following hypotheses: (1) our Skoll environment cost-effectively supports the definition, implementation and execution of our main effects screening process, (2) the screening design used in main effects screening correctly identifies a small subset of options whose effect on performance is important, and (3) exhaustively examining just the options identified by the screening design gives performance data that (a) is representative of the system's performance across the entire configuration space, but less costly to obtain and (b) is more representative than a similarly-sized random sample.

Subject applications. The experimental subject applications for this study were based on a suite of performance-intensive software: ACE v5.4 + TAO v1.4. ACE and TAO are ideal subjects for our feasibility study since they share many characteristics with other highly configurable performance-intensive software systems. For example, they collectively contain over 2M+ lines of source code, functional regression tests, and performance benchmarks contained in ~4,500 files that average over 400 CVS commits per week.

Option Index	Option Name	Option Settings
A	ReactorThreadQueue	{FIFO, LIFO}
B	ClientConnectionHandler	{RW, MT}
C	ReactorMaskSignals	{0, 1}
D	ConnectionPurgingStrategy	{LRU, LFU}
E	ConnectionCachePurgePercent	{10, 40}
F	ConnectionCacheLock	{thread, null}
G	CorbaObjectLock	{thread, null}
H	ObjectKeyTableLock	{thread, null}
I	InputCDRAllocator	{thread, null}
J	Concurrency	{reactive, thread}
K	ActiveObjectMapSize	{32, 128}
L	UseridPolicyDemuxStrategy	{linear, dynamic}
M	SystemPolicyDemuxStrategy	{linear, dynamic}
N	UnqPolicyRevDemuxStrategy	{linear, dynamic}

Table 6.2: Some ACE+TAO options

Application scenario. Due to recent changes made to the message queuing strategy, the developers of ACE+TAO were concerned with measuring two performance criteria: (1) the latency for each request and (2) total message throughput (events-/second) between the ACE+TAO client and server. For this version of ACE+TAO, the developers identified 14 binary run-time options they felt affected latency and throughput (see Table 6.2). Thus, the entire configuration space has $2^{14} = 16,384$ different configurations.

Experimental process. Our experimental process used Skoll to implement the main effects screening process and evaluate our three hypotheses. To do this, we executed the main effects screening process across a prototype Skoll grid of dual processor Xeon machines running Red Hat 2.4.21 with 1GB of memory in the real-time scheduling class. The experimental task involved running a benchmark application in a particular configuration, which evaluated the application scenario outlined

above by creating an ACE+TAO client and server. For each task we measured message latency and overall throughput between the client and the server. The client sends 300K requests to the server, where after each request it waits for a response from the server and records the latency measure. At the end of 300K requests, the client computes the throughput achieved in terms of number of requests served per second. We finally analyzed the resulting data to evaluate our hypotheses. Section 6.5 describes the limitations with our current experimental process.

6.2.2 The Full Data Set

To evaluate our approach, we generated performance data for all 16,000+ valid configurations, which we refer to as the "full suite" and the performance data as the "full data set." We then examined the effect of each option and judged whether they had important effects on performance using a graphical method called *half-normal probability plots*, which show each option's effect against their corresponding coordinates on the half-normal probability scale. If $|\theta|_1 \leq |\theta|_2 \leq ... \leq |\theta|_I$ are the ordered set of effect estimations, the half-normal plot then consists of the points

$$(\Phi^{-1}(0.5 + 0.5[i - 0.5]/I), |\theta|_i) \; for \; i = 1, ..., I \tag{6.4}$$

where Φ is the cumulative distribution function of a standard normal random variable.

The rationale behind half-normal plots is that unimportant options will have effects whose distribution is normal and centered near 0. Important effects will also be normally distributed with means different that 0. If no effects are important, the

Design Index	Design	Design Generators
Scr_{32}	2_{IV}^{14-9}	$F = ABC$, $G = ABD$, $H = ACD$, $I = BCD$, $J = ABE$, $K = ACE$, $L = BCE$, $M = ADE$, $N = BDE$
Scr_{64}	2_{IV}^{14-8}	$G = ABC$, $H = ABD$, $I = ABE$, $J = ACDE$, $K = ABF$, $L = ACDF$, $M = ACEF$, $N = ADEF$
Scr_{128}	2_{IV}^{14-7}	$H = ABC$, $I = ABDE$, $J = ABDF$, $K = ACEF$, $L = ACDG$, $M = ABEFG$, $N = BCDEFG$

Table 6.3: Actual screening designs used in the experiments, calculated using the SAS statistical package

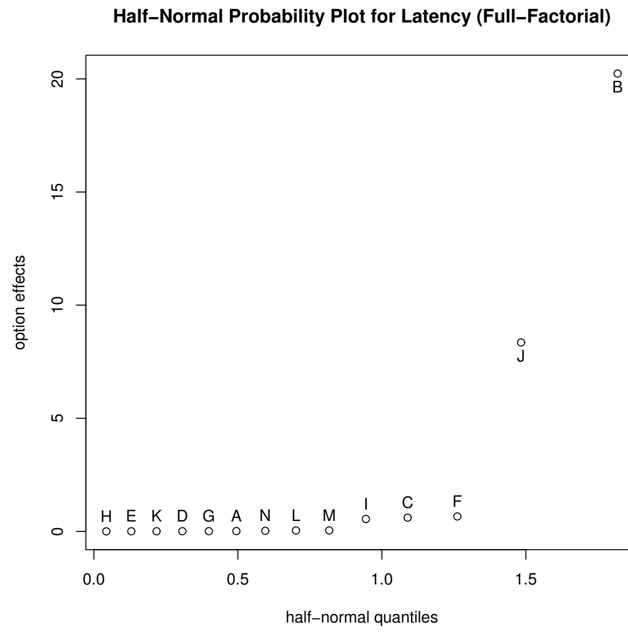
resulting plot will show a set of points on a rough line near $y = 0$. Options whose effects deviate substantially from 0 are considered important.

Note that "importance" is not defined formally and differs in spirit from the traditional notion of statistical significance. In particular, developers must decide for themselves how large effects must be to warrant their attention. While this has some downsides (see Section 6.5), even with traditional statistical tests that measure statistical significance developers still must make judgments as to the magnitude of effects.

Figure 6.1 plots the effect across the full data set of each of the 14 ACE+-TAO options on latency and throughput. We see that options B and J are clearly important, whereas options I, C and F are arguably important, and the remaining options are not important.

6.2.3 Evaluating Screening Designs

We now evaluate whether the remotely executed screening designs can correctly identify the same important options discovered in the full data set. To do



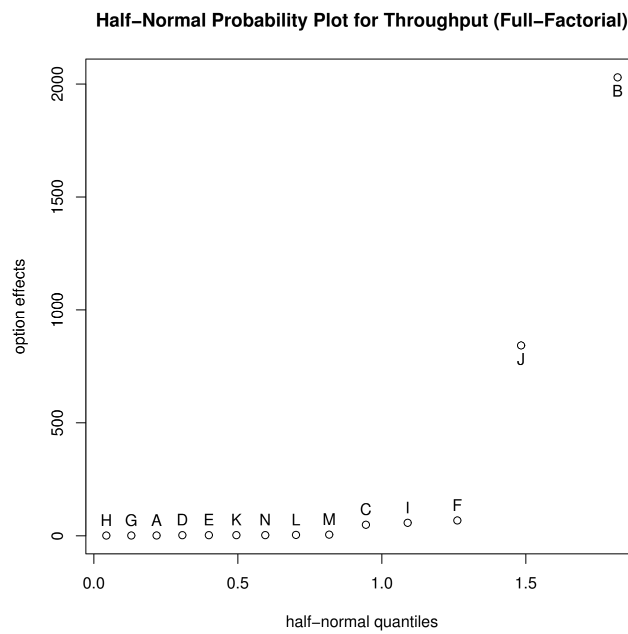


Figure 6.1: Option effects based on full data

this, we calculated and executed three different screening designs, whose specifications appear in Table 6.3. These designs examined all 14 options using increasingly larger run sizes (32, 64, or 128 observations). We refer to the screening designs as Scr_{32}, Scr_{64} and Scr_{128}, respectively.

Figure 6.2 shows the half-normal probability plots obtained from our screening designs. The figures show that all screening designs correctly identify options B and J as being important (as is the case in full-factorial experiment). Scr_{128} also identifies the possibly important effect of options C, I, and F. In this chapter, we only present data on latency. Throughput analysis shows identical results unless otherwise stated.

These results suggest that (1) screening designs can detect important options at a small fraction of the cost of exhaustive testing, (2) the smaller the effect, the larger the run size needed to identify it, and (3) developers should be cautious when dealing with options that appear to have an important, but relatively small effect, as they may actually be seeing normal variation (Scr_{32} and Scr_{64} both have examples of this).

6.2.4 Estimating Performance with Screening Suites

We now evaluate whether screening all the combinations of the most important options can be used to estimate performance quickly across the entire configuration space we are studying. The estimates are generated by examining all combinations of the most important options, while defaulting the settings of the unimportant

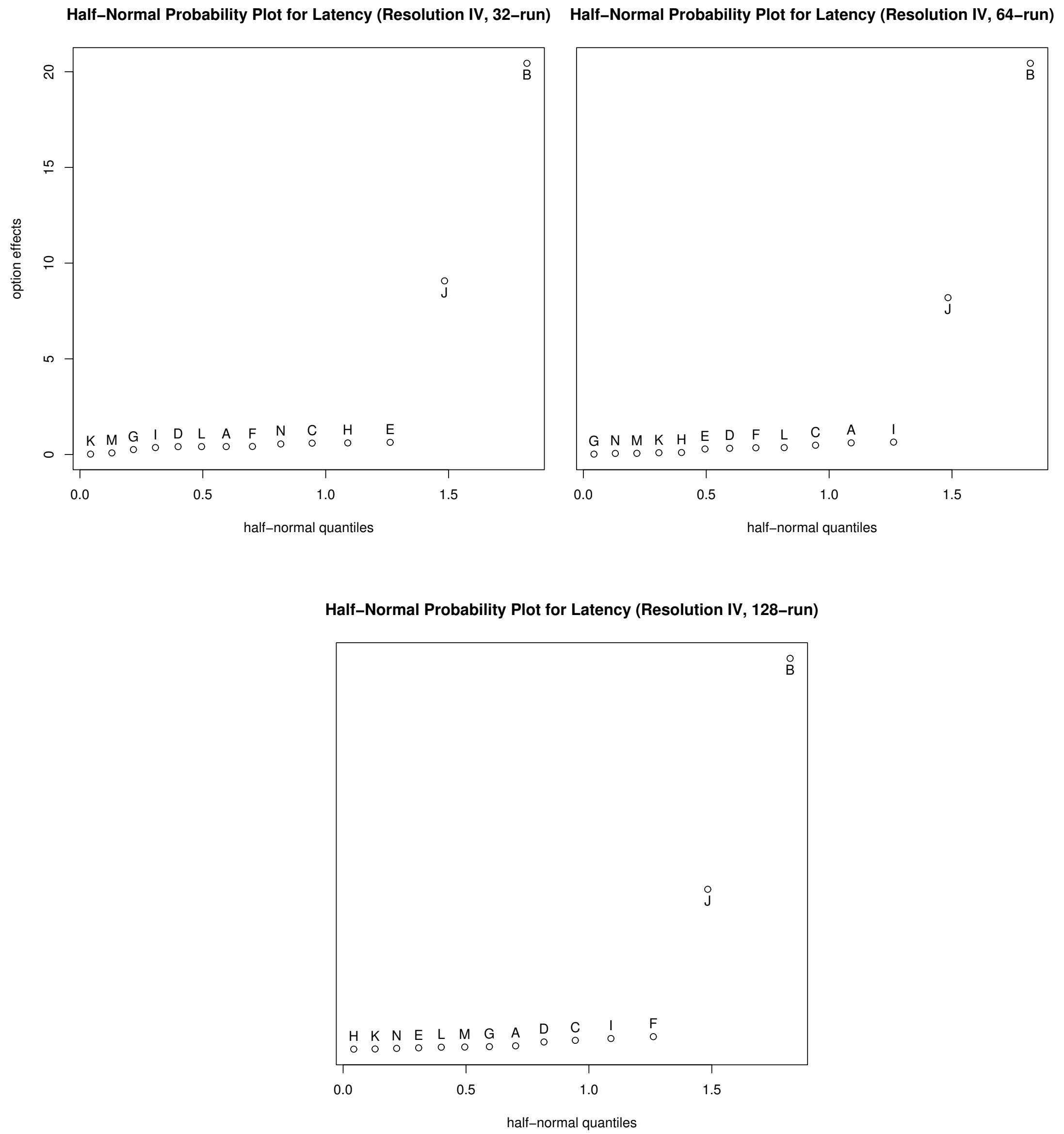


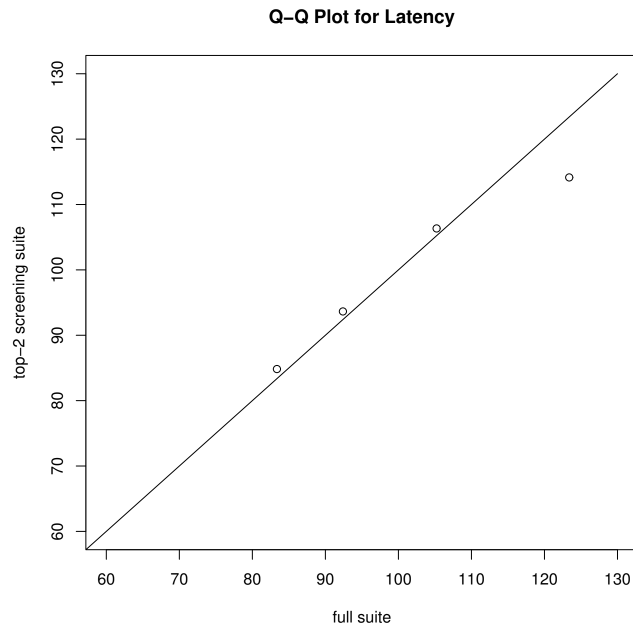
Figure 6.2: Option effects based on screening designs

options. In the previous section, we determined that options B and J were clearly important and that options C, I, and F were arguably important. Developers will therefore make the estimates based on benchmarking either 4 (all combinations of options B and J) or 32 (all combinations of options B, J, C, I, and F) configurations. We will refer to the set of 4 configurations as the top-2 screening suite and the set of 32 configurations as the top-5 screening suite.

Figure 6.3 shows the distributions of latency for the full suite vs. the top-5 screening suite and for the full suite vs. the top-2 screening suite. From the figure, we see that the distributions of the top-5 and top-2 screening suites closely track the overall performance data. Such plots, called quantile-quantile (Q-Q) plots, are used to see how well two data distributions correlate. To do this they plot the quantiles of the first data set against the quantiles of the second data set. If the two sets share the same distribution, the points should fall approximately on $x = y$ line. This data suggests that the screening suites computed at step 4 of the main effects screening process (Section 6.1) can be used to estimate overall system performance across the entire configuration space at extremely low time/effort, *i.e.*, running 4 benchmarks takes 40 seconds, running 32 takes 5 minutes, running 16,000+ takes 2 days.

6.2.5 Screening Suites vs. Random Sampling

Another question is whether our main effects screening process was any better than other low-cost estimation processes. In particular, we compared the latency distributions of several random samples of 4 configurations to that of the top-2



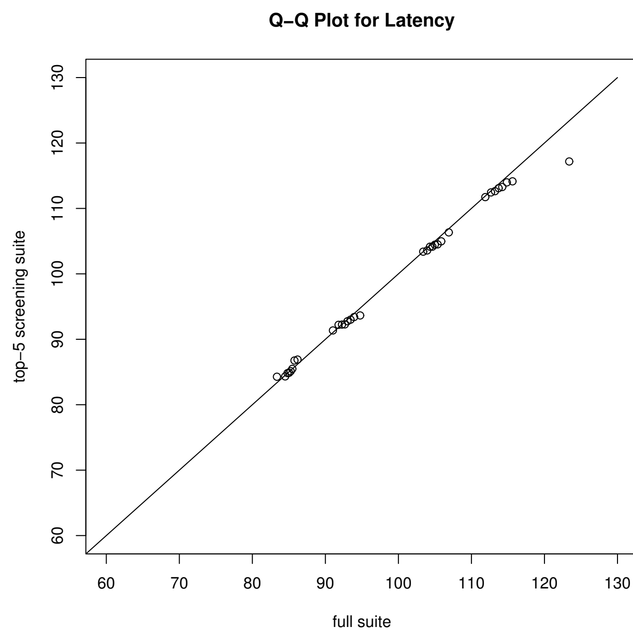


Figure 6.3: Q-Q plots for the top-2 and top-5 screening suites

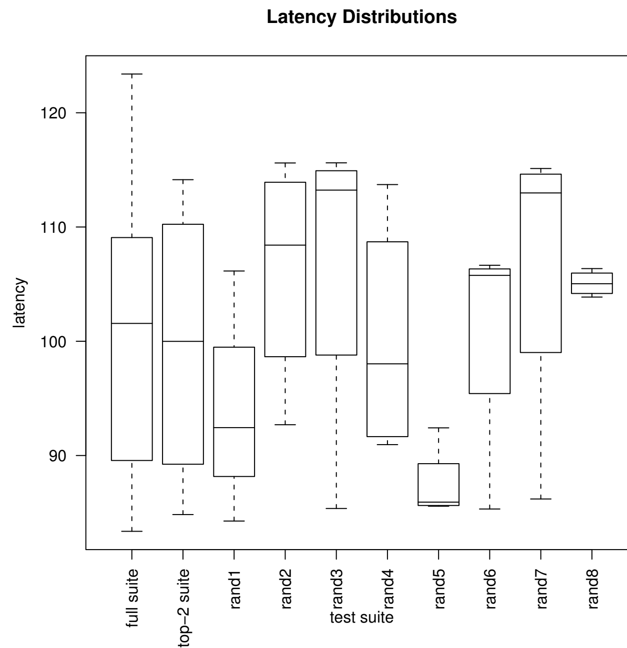


Figure 6.4: Latency distribution from full, top-2, and random suites

screening suite found by our process. The results of this test are summarized in Figure 6.4. These box plots show the distributions of latency metric obtained from exhaustive testing, top-2 screening suite testing, and random testing. These graphs suggest the obvious weakness of random sampling, *i.e.*, while sampling distributions tend toward the overall distribution as the sample size grows, individual small samples may show wildly different distributions.

6.2.6 Dealing with Evolving Systems

A primary goal of main effects screening is to detect performance degradations in evolving systems quickly. So far we have not addressed whether – or for how long – screening suites remain useful as a system evolves. To better understand this issue, we measured latency on the top-2 screening suite, once a day, using CVS snapshots of

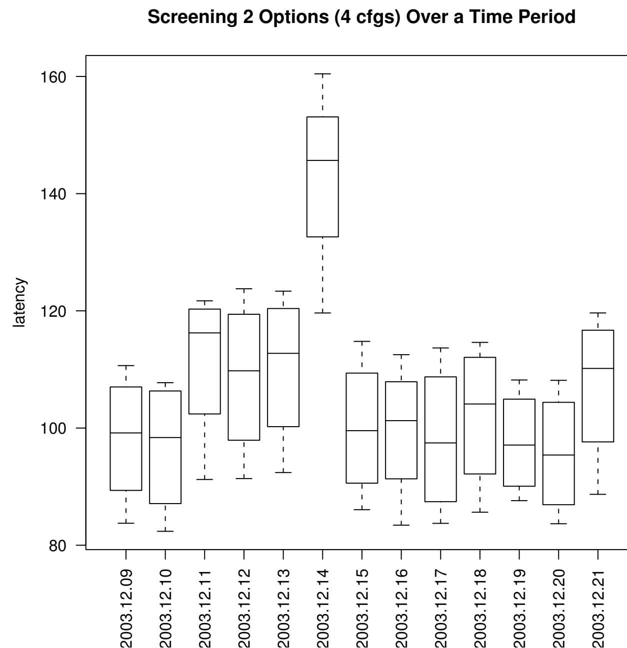


Figure 6.5: Performance estimates across time

ACE+TAO. We used historical snapshots for two reasons: (1) the versions are from the time period for which we already calculated the main effects and (2) developer testing and in-the-field usage data have already been collected and analyzed for this time period (see `www.dre.vanderbilt.edu/Stats/`), allowing us to assess the system's performance without having to exhaustively test all configurations for each system change.

Figure 6.5 depicts the data distributions for the top-2 screening suites broken down by date (higher latency measures are worse). We see that the distributions were stable the first two days, crept up somewhat for 3 days and then shot up the 4th day (12/14/03). They were brought back under control for several more days, but then moved up again on the last day. Developer records and problem reports indicate that problems were noticed on 12/14/03, but not before then.

Another interesting finding was that the limited testing done by ACE+TAO developers measured a performance drop of only around 5% on 12/14/03. In contrast, our screening process showed a much more dramatic drop – closer to 50%. Further analysis by system developers indicated that their unsystematic testing failed to evaluate configurations where the degradation was much more pronounced.

6.3 Validating Basic Assumptions

In our experiments in Section 6.2, we were given a highly-configurable system, a configuration space, and a benchmarking regression test. Our goal was to detect performance degradations across the configuration space as a result of system changes. First, we identified important options by leveraging fractional factorial screening experiments. We then empirically showed that (1) systematically monitoring only these important options can reliably estimate the overall performance across the entire configuration space and (2) these estimates are effective in detecting performance degradations.

An integral part of this process is to identify the important options. Having the right resolution for the screening experiment is the key to that. For example, in our experiments, we used resolution IV designs. That, is we aliased some 1st-order effects (*i.e.,* main effects) with some 3rd-order or higher effects and some 2nd-order effects with 2nd-order or higher effects. Therefore, by using resolution IV designs, we implicitly made the following assumptions:

1. The 1st-order effects we identified as important are really the important ones

rather than the higher-order effects they are aliased to.

2. Monitoring only 1st-order effects is sufficient for our subject system.

Validating these assumptions is crucial for the reliability of the results. In the presence of exhaustive benchmarking results, these assumptions can be validated by computing all the factorial effects and checking them against the assumptions. However, in real-life scenarios, it is not likely to have access to exhaustive results.

In this section, we validate these assumptions by conducting follow-up experiments without having access to exhaustive benchmarking results. In Section 6.4, we will use the results of this study to provide practical guidelines to the practitioners of the main effects screening process.

Throughout this section, once we identify the important options by conducting a screening experiment, we augment the current experiment with additional configurations to break the aliases for important options and look for higher-order effects. We then use Skoll to benchmark only the additional configurations, collect the results, and merge them together with the results of the original screening experiment. Our approach relies on another class of experimental designs called a *D-optimal design*.

6.3.1 D-optimal Designs

Experimental data can often be modeled by the general linear model, also called the multiple regression model [78]. In this research, since our focus is effect screening, we are interested in linear models only. In general, higher-order models

such as quadratic models are more desirable for response optimization. Suppose that the response y is related to p treatments (*e.g.,* configuration options), $x_1, x_2, \ldots, x_p$, and that N observation is collected in an experiment. Then the general linear model takes the form

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \epsilon_i, \qquad i = 1, \ldots, N, \tag{6.5}$$

where y_i is the ith value of the response, $x_{i1}, x_{i2}, \ldots, x_{ip}$ are the corresponding values of the treatments, and ϵ_i is the error in the observation.

These N equations can be written in matrix notation as follows:

$$\boldsymbol{y} = X\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{6.6}$$

where $\boldsymbol{y} = (y_1, \ldots, y_N)^T$ is the $N \times 1$ vector of responses, $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_p)^T$ is the $(p+1) \times 1$ vector of regression coefficients, $\boldsymbol{\epsilon} = (\epsilon_1, \ldots, \epsilon_N)^T$ is the $N \times 1$ vector of errors, and $\boldsymbol{X}$ is the $N \times (p+1)$ matrix given as

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{Np} \end{bmatrix}. \tag{6.7}$$

The purpose for collecting data in an experiment is to estimate and make inferences about the regression coefficients $\boldsymbol{\beta}$ and the error variance. Which combinations of treatment variables (*i.e.,* specific combinations of option settings) should be tested to estimate the model parameters is the main focus of the D-optimal designs.

For a given model, D-optimal designs [56] select an "optimal" set of specific option setting combinations from the entire configuration space to reliably and effi-

ciently estimate the model parameters. The optimality criterion used in generating D-optimal designs is one of maximizing $|\boldsymbol{X}^T\boldsymbol{X}|$, the determinant of the Fisher's information matrix $\boldsymbol{X}^T\boldsymbol{X}$ [35, 47]. The form of the $\boldsymbol{X}$ matrix is given in (6.7).

D-optimal designs, where 'D' stands for determinant, are computer-aided designs. Given a configuration space and a model the experimenter wishes to fit, search-based computer algorithms (*e.g.,* hill climbing and simulated annealing) are used to select a set of configurations which maximizes the optimality criterion. Therefore, unlike fractional factorial designs, the D-optimal designs may not be a perfect fraction of full factorial designs. D-optimal designs are preferable over standard classical designs such as full factorial and fractional factorial designs when (1) the standard designs require too many configurations to be tested for the amount of resources and time and (2) the configuration space is heavily constrained (*i.e.,* when not all the configurations are valid).

In this research, we leverage D-optimal designs to validate our assumptions in identifying important options by augmenting our screening designs. First, for each assumption, we create a linear model (such as the one given in 6.5) which will help us validate the assumption. We then feed this model and the original screening designs as "seeds" to the search-based algorithm computing the D-optimal designs. The result is a list of additional configurations to be benchmarked to reliably validate the assumption. Throughout the experiments, we used the `optex` function of the SAS/QC package to compute the actual D-optimal designs [68].

D-optimal designs are generally non-orthogonal designs (as is the case for the D-optimal designs we created in this chapter). Therefore, the effect calculations

Partial Aliasing Structure
B=ACH=CGKM=CKLN=DGHL=DHMN=EFHK=HIJK=ADEI=ADFJ
J=EFI=ACIK=BHIK=CDFH=DEGM=DELN=FGKN=FKLM=ABDF
C=ABH=ADMN=AIJK=BGKM=BKLN=DEHI=DFHJ=AEFK=ADGL
F=EIJ=BEHK=CDHJ=DGIM=DILN=GJKN=JKLM=ABDJ=ACEK
I=EFJ=ACJK=BHJK=CDEH=DFGM=DFLN=EGKN=EKLM=ABDE

Table 6.4: Partial aliasing structure for the important option in the 2_{IV}^{14-7}, 128-run design

given in formulas 6.1 and 6.3 cannot be used in the analysis of such designs. In this research, we applied a linear regression analysis to our designs using the `glm` procedure of the SAS package. The `glm` procedure uses the method of least squares to fit general linear models onto the observations. In particular, we leveraged the `glm` procedure for the analysis of variance for factorial effects. Specifically, we used the Type III sum of squares to identify important effects. The Type III sum of squares for a particular effect is the amount of variation in the response due to that effect. The Type III sums of squares are generally preferred for non-orthogonal designs [68].

In the remainder of this section, we use our 2_{IV}^{14-7}, 128-run experiment as the basis of our analysis. We specifically chose this experiment because it was the only experiment where we identified five important options rather than two. Note that the analysis presented in this section is readily applicable to any screening experiment.

6.3.2 Breaking Aliases

In the 2_{IV}^{14-7}, 128-run experiment, we identified options B and J as being clearly important and options C, F, and I as being arguably important. In this experiment, some 1st-order effects were aliased with some 3rd-order or higher effects. Table 6.4 depicts the aliasing structure for the important options up to 4th-order effects. Note that the complete aliasing structure for the important options contains a total of 640 aliases. Due to space limitations, we could not list them all here, but they can be computed using the defining relations of our 128-run screening experiment as discussed in Section 6.1.3.

Consider the alias $B = ACH$ for the important option B; the effect of option B is aliased with the interaction effect of option A, C, and H. Since the experimental data cannot distinguish B from ACH, there is an ambiguity if B or ACH is important. If ACH, not B, is the important one then our performance predictions would obviously suffer. The same is true for the rest of the aliases. Therefore, if suspected, the ambiguity of aliased effects for the important options should be resolved to ensure the reliability of the performance predictions.

To resolve the ambiguities, we first formally modeled the important effects and their 640 aliases by the following linear formula:

$$\begin{aligned} y &= \beta_0 + \beta_B x_B + \beta_J x_J + \beta_C x_C + \beta_F x_F + \beta_I x_I \\ &\quad + \beta_{ACH} x_A x_C x_H + \beta_i x_i, \quad \forall i \in S - \{ACH\} \end{aligned} \tag{6.8}$$

where S is the set of all 640 aliased effects, β_0 is the intercept term, $x_B = -1, 1$ according to the level of option B (the definitions for the other x's are similar), and

β's are the model coefficients.

Since the 1st-order effects other than B, J, C, F, and I are negligible (Section 6.2.3), we excluded them from the model. Note that the performance of the D-optimal designs often depends on the validity of the model which drives the D-optimality search. Since we created our model based on the results of a formal screening experiment, our model is objective.

We then augmented our 128-run screening experiment using the D-optimality criterion for our model. We were able to manage this with 2328 additional configurations. We benchmarked only the additional configurations, collected the results, and analyzed them together with the results of the original screening experiment.

Figure 6.6(a) plots the Type III sum of squares for the factorial effects. In this figure, the effects are ordered in descending order. Note that, due to space limitations, only the top 10 effects are given. The higher the sum of squares, the more important the effects are. As can be seen from this figure, options B, J, C, F, and I are the important ones not the higher-order interactions they are aliased to. Further statistical analysis also showed that these options are statistically significant at %99.99 confidence level or better.

In cases where time and resource constraints prevent complete dealiasing of important options, one can consider (1) dealiasing them up to a certain level of order (*e.g.,* up to 3rd-order or 4th-order effects) or (2) dealiasing only the suspected aliases.

To see the reduction in the number of configurations to be tested, we augmented our original design to dealias the important options up to and including

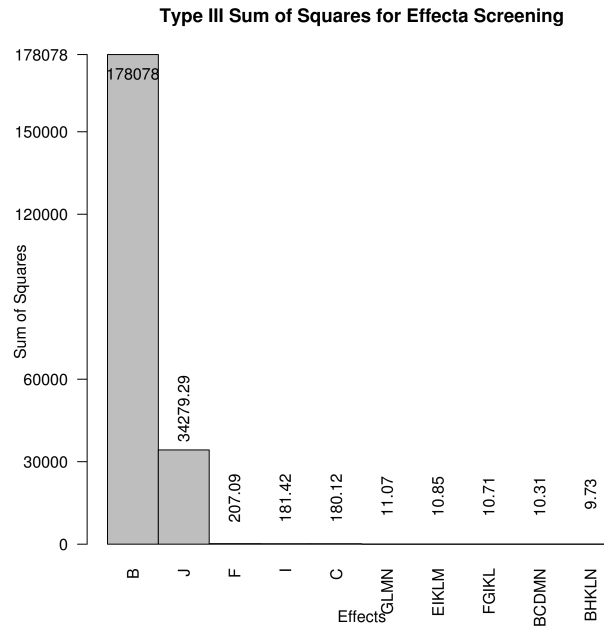
3rd-order effects. It took us only 64 additional configurations to augment our screening experiment for this scenario. As can be seen in Figure 6.6(b), we were able to reach the same conclusion as the complete dealiasing experiment. In general, the cost reduction techniques given above are obviously not as reliable as the complete dealiasing approach. Therefore, care must be taken in the analysis and interpretation of such experiments.

6.3.3 Checking for Higher-Order Effects

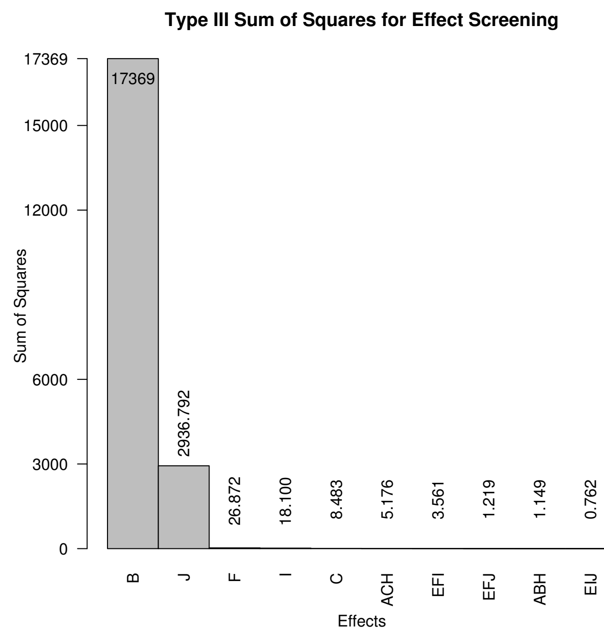
Our second assumption was that monitoring only 1st-order effects would be sufficient for our subject system. In this section, we investigate the validity of this assumption by examining higher-order effects.

As a general principle of empirical analysis, we need at least one degree of freedom for estimating each effect and each configuration benchmarked gives us one degree of freedom. Therefore, as the level of order of effects that needs to be examined approaches the number of options in an experiment, the design for the experiment converges to the full factorial design. For example, in our configuration space, only the full factorial design can be used to analyze all the effects up to and including 14th-order effects. Therefore, unless we can afford exhaustive benchmarking, we need to bound the highest level of order up to which we look for important effects.

In this section, without losing the generality of the approach presented, we decided to look for important effects up to and including 3rd-order effects. Just as



(a)



(b)

Figure 6.6: (a) Complete dealiasing experiment and (b) Up to, including 3rd-order effects dealiasing experiment

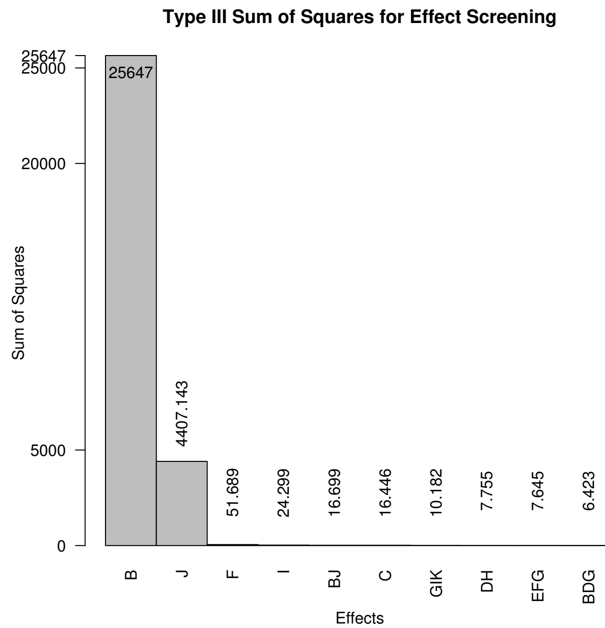


Figure 6.7: Looking for higher-order effects up, to including 3rd-order effects

in Section 6.3.2, we augmented our 2_{IV}^{14-7}, 128-run screening experiment using the D-optimality criterion. Only this time our generalized linear model was consisted of all the 1st-, 2nd-, and 3rd-order effects. We were able to augment the original experiment with additional 381 configurations.

Figure 6.7 shows the results of this study. We identified the top 6 effects, B, J, F, I, BJ, and C, as statistically significant at %99.9 confidence level or better. Among these effects, we have only one interaction effect, BJ. This effect involves only options that are already considered important by themselves, which supports the assumption that monitoring only the first-order effects was sufficient for our subject system.

6.4 Guidelines for Practitioners

In this section, we provide practical guidelines on how to use the main effects screening approach in practice. In particular, we examine how to select an appropriate resolution and size for the screening designs, how to identify the important options, and how to validate the basic assumptions in identifying the important options.

Based on our experience, we give the following guidelines to the users of the main effects screening process:

1. **Choose the resolution.** Leverage apriori knowledge of the system under test, if it is available, to decide which high-order effects are considered negligible (*i.e.,* the resolution of the screening experiment). If no or limited apriori information is available, use screening experiments in an iterative manner to obtain this information.

2. **Choose the size.** Depending on the availability of resources and how fast the underlying system changes, figure out the maximum number of configurations that can be afforded in the screening experiment. Note that the resolution of the experiment chosen in step (1) may dictate the minimum size of the experiment. If this size is not feasible, consider lowering the resolution of the experiment by carefully choosing the aliasing structure so that no potentially important higher-order effects are aliased with lower-order ones.

3. **Identify the important options.** Once the screening design is computed,

conducted and then analyzed, use the half-normal probability plots to identify important options. Note that if no effects are important, these plots will show a set of points on a rough line near $y = 0$. Substantial deviations from this line indicate important effects. Depending on the desired precision of the performance estimates, decide how large effects must be to warrant the attention.

4. **Validate the basic assumptions.** If needed, validate the assumptions imposed by the choice of the resolution. Consider using D-optimal designs to augment the screening experiment to (1) dealias the important options and (2) look for higher-order effects.

6.5 Discussion

This chapter presents a new distributed continuous quality assurance (DCQA) process called *main effects screening* that is designed to detect performance degradations in evolving performance-intensive software systems across large configuration spaces. To evaluate this process, we conducted a formally-designed experiment across a grid of computers in the Skoll environment. The results of this experiment helped in estimating performance across the large configuration space of ACE+TAO software systems.

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial

practice. One potential threat is that several steps in our process require human decision making and input, *e.g.*, developers must provide reasonable benchmarking applications and must also decide when they consider effects to be important.

Another possible threat to external validity concerns the representativeness of the ACE+TAO subject applications, which though large are still just one suite of software systems. A related issue is that we have focused on a relatively simple and small subset of the entire configuration space of ACE+TAO that only has binary options and has no inter-option constraints. While these issues pose no theoretical problems (screening designs can be created for much more complex situations), we need to apply our approach to larger, more realistic configuration spaces in future work to understand how well it scales.

Another potential threat is that for the time period we studied, the ACE+TAO subject application was in a fairly stable phase, *i.e.*, changes were made mostly to fix bugs and reduce memory footprint, but the system's functionality was relatively stable. For situations where a system's basic functionality is in greater flux, it may be harder to distinguish significant performance degradation from normal variation.

Despite these limitations, we believe our study supports our basic hypotheses. We reached this conclusion by noting that our studies showed that: (1) screening designs can correctly identify important options, (2) these options can be used to quickly produce reliable estimates of performance across the entire configuration space at a fraction of the cost of exhaustive testing, (3) the alternative approach of random or *ad hoc* sampling can give highly unreliable results, (4) the main effects process detected performance degradation on a large and evolving software system,

and (5) the screening suite estimates were more precise than the ad hoc process currently used by the developers of the subject system. Main effects screening can also be a defect detection aid, *e.g.*, if the screened options change unexpectedly, developers can reexamine the software to identify possible problems with software updates.

Chapter 7

A Demonstration of the Generality of Skoll Infrastructure

We have so far used Skoll on the ACE+TAO software system with specific QA tasks. These QA tasks involved testing for functional correctness and evaluating performance characteristics of ACE+TAO across its numerous configurations. Our configuration and control models in these studies were mainly consisted of traditional system configuration options (*e.g.*, options that allow features, such as asynchronized method invocation (AMI) and CORBA messaging, to be compiled in or out of the system and options that provide more fine-grained control over the runtime behavior of the system, such as object collocation strategies and connection purging strategies).

In this chapter, our ultimate goal is to demonstrate that the Skoll system and the DCQA ideas are not limited only to the ACE+TAO software system or, in general, to highly-configurable systems. To do this, we have picked a different system, constructed a configuration and control model with various types of options (*e.g.*, user actions, HTML tags, and system configuration options), and developed a simple yet interesting DCQA process. We have then implemented the necessary Skoll components (*e.g.*, application specific server- and client-side components, navigation and adaptation strategies, and analysis tools) and executed the new DCQA process using Skoll.

We conjecture that (1) our configuration and control model is flexible, easy to extend, and not restricted to traditional software configuration options only and (2) the Skoll infrastructure and implementation are flexible enough to carry out a variety of QA tasks. This chapter presents a feasibility study that addresses these conjectures.

7.1 Experiments

Our DCQA scenario in this research is mainly inspired by a well-known software failure documented in the software engineering literature [82, 83, 39, 81]: A certain version of Mozilla [57], an open source Web browser, crashes when printing HTML documents which contain the HTML tag `select`. The `select` tag creates a drop-down list and allows users to choose from a fixed set of values; one or several at once.

This is an interesting failure to base our DCQA scenario on, because the failure manifests itself only when certain combinations of input cases (*i.e.*, an HTML document containing a `select` tag) and user actions (*i.e.*, printing an HTML document) come together. Furthermore, identifying failure inducing combinations is of great importance to the developers of the system since it can reduce the turn-around time for bug fixes and increase the utilization of testing resources. For example, once printing an HTML document containing a `select` tag proves to be faulty, time and resources can be refocused on other unexplored parts of the testing space.

Option	**Type**	**Settings**	**Interpretation**
select	HTML tag	$\{1 = \text{Yes}, 0 = \text{No}\}$	Select tag
table	HTML tag	$\{1 = \text{Yes}, 0 = \text{No}\}$	Select tag
print	User action	$\{1 = \text{Yes}, 0 = \text{No}\}$	Select user action
bookmark	User action	$\{1 = \text{Yes}, 0 = \text{No}\}$	Select user action
safe-mode	Run-time option	$\{1 = \text{Yes}, 0 = \text{No}\}$	Enable feature
sync	Run-time option	$\{1 = \text{Yes}, 0 = \text{No}\}$	Enable feature

Table 7.1: Example configuration model.

7.1.1 DCQA Scenario

We defined our DCQA scenario as follows: Due to recent changes made to the Mozilla's code base, the developers of Mozilla are concerned with testing their system with various combinations of HTML tags and user actions across its numerous run-time configurations. In particular, they are interested in system crashes. When a crash occurs, they want to identify HTML tags which led the Mozilla to the crash. Since time and resources are limited, once a failure inducing tag is identified, they want to avoid testing the tag any longer to save time and resources for untested cases.

In our actual implementation of this scenario, we were forced to simulate the Mozilla crashes (See Section 7.2 for more details).

7.1.2 Configuration and Control Model

We developed a configuration and control model for our scenario. Table 7.1 depicts a subset of the actual model used our experiments. In this model, we have 3 types of configuration options: HTML tag options, user action options, and run-time configuration options. The HTML tag options, user action options, and run-

time options define a configuration subspace for combinations of HTML tags, user actions, and Mozilla's run-time configurations, respectively. Together they define a configuration space for testing.

Our global QA task was to exhaustively test Mozilla across the configuration space defined by the model. A QA subtask was then defined as testing a single configuration from this space. The Skoll system automatically translated the model into the ISA's planning language.

7.1.3 Setting up the Skoll Infrastructure

We implemented the necessary Skoll components for our DCQA scenario, including the application specific server- and client-side components and an analysis and adaptation strategy.

Application specific server-side component. For each incoming QA job request, once the ISA selected a valid configuration for the requesting client, this component mapped the selected configuration to a set of instructions and test scripts which will help the client execute the assigned subtask. The ISA then packaged these artifacts and sent it to the requesting Skoll client.

In particular, this component mapped (1) the HTML tag options to a valid HTML document which contains only the selected tags, (2) the selected user action options to a set of graphical user interface (GUI) events which automatically mimics end-users' actions, and (3) the run-time options to a script which configures Mozilla at run-time. Consider that the following configuration is selected by the ISA for

testing:

`{select=1,table=0,print=1,bookmark=0,safe-mode=0,sync=1}`

For this configuration, the application specific server-side component would generate a valid HTML document which contains only the `select` tag (but not the `table` tag), a sequence of GUI events to print the generated document (but not the GUI events for bookmarking it), and a script which configures Mozilla to run in the normal user mode (not in the safe mode) and make synchronous GUI calls.

Application specific client-side component. We implemented an application specific client-side component to execute the assigned subtasks at the Skoll client. For each assigned subtask, this component (1) downloaded the Mozilla software from a remote repository, (2) configured it, (3) opened up the generated HTML document, (4) executed the sequence of user actions (in a pre-specified order) using a GUI test automation tool [79], and then (5) sent the results (*i.e.*, Mozilla crashed or not crashed) to the Skoll server. Each of these steps was realized as an individual instruction in the QA jobs sent by the Skoll server (See Figure 3.6 for an example QA job). We implemented only the application specific instructions for step (3) and (4). For the rest, we used the Skoll's default set of instructions.

Navigation strategy. We instructed ISA to navigate the configuration and control space using random sampling without replacement (*i.e.*, each valid configuration was scheduled exactly once).

Analysis and adaptation strategy. We implemented an online analysis and adaptation strategy for our scenario. As the QA subtask results came in, this

strategy analyzed the results on the fly to identify failure inducing HTML tags. Once such a tag was identified, this adaptation strategy automatically introduced temporary constraints (*e.g.*, $select = 1 \rightarrow false$) to the current configuration and control model (see Section 3.1.2 for more details on temporary constraints). By doing this, we avoided testing configurations from constantly failing subspaces. Note that, since each HTML tag has a binary setting and there are no inter-option constraints, even identifying one such failure inducing tag would cut the testing space into a half.

We based our analysis and adaptation strategy on the well-known *delta debugging algorithm* [81, 83]. The delta debugging algorithm iteratively and algorithmically isolates differences between passing and failing program inputs [81, 83]. Given a failing HTML document (*i.e.*, one that causes Mozilla to crash), the delta debugging algorithm iteratively simplifies the document using a binary search-like algorithm. At each iteration, an algorithmically chosen portion of the HTML document is discarded and the remaining portion is tested. Depending on the test result (*i.e.*, fail or pass) the algorithm determines which portions of the input should be tested in the next iteration. It stops when a minimal program input is found, in which removing any single input entry would cause the failure to disappear. Feasibility studies described in [81, 83] suggest that delta debugging algorithm is effective in identifying failure inducing HTML tags.

In this chapter, since our goal is to demonstrate the generality of the Skoll system and the DCQA ideas, we are not concerned with introducing a novel approach to identify failure inducing tags. Instead, we implemented the delta debugging algo-

rithm and adopt it as our analysis and adaptation strategy. In fact, we implemented a parallel version of the delta debugging algorithm. The original algorithm assumes a serial execution of test cases.

For a failing configuration, our analysis and adaptation strategy generated new QA subtasks by changing only the failing HTML document according to the delta debugging algorithm. The rest of the artifacts for the failing configuration (*e.g.*, the run-time configuration and the generated user actions) remained the same. The generated QA subtasks were then scheduled for future allocation. Our analysis and adaptation strategy allocated QA subtasks iteratively until the delta debugging algorithm converged to an answer, finding the failure inducing tag(s). Then, for each failure inducing tag, we introduced temporary constraint(s) to the current configuration and control model.

Note that we could have used the delta debugging algorithm to identify combinations of HTML tags and user actions which cause Mozilla to crash [81, 83]. However, we opted out this possibility to keep the implementation of the delta debugging algorithm simple.

7.2 Experimental Setup

We installed 10 Skoll clients and one Skoll server across workstations distributed throughout computer science labs at the University of Maryland. All Skoll clients ran on Linux 2.4.9-3 platform. In the configuration and control model, we had 26 HTML tag options, 6 run-time configuration options, and 3 user action options.

All told, the size of the configuration space was 2^{35}.

We used Mozilla v1.7 as our subject software. However, although the Mozilla failure that inspired us for this study was real and reported by an end-user (see bug report 69634 in `bugzilla.mozilla.org`), we were not able to find (or build) the version that crashes as described. Instead, we chose a Mozilla version and simulated the crashes only when the necessary conditions are met (*i.e.*, printing an HTML document containing a `select` tag). Otherwise, we executed the QA subtasks as described in this chapter. Note that this simulation suffices to reach our experiment goals.

7.3 Results and Discussions

After 30 hours of work, we were able to implement our DCQA scenario using the Skoll infrastructure and tools and execute it successfully as described. We did not observe Mozilla crashes other than the ones we simulated.

Out of 30 hours, we spent 10 hours to fix numerous bugs in the Skoll implementation. Some of the bug fixes removed things that limited the generality of Skoll, (*i.e.*, the exercise was valuable to improving Skoll's generality). We spent another 10 hours to gain domain knowledge on Mozilla and other tools we used, including implementing the delta debugging algorithm, figuring out what configuration options are available for configuring Mozilla, and integrating a GUI test automation tool [79] to mimic user actions.

By conducting this study:

- We validated the Skoll implementation one more time.
- We demonstrated that our configuration and control model is not restricted to traditional software configuration options only. By using the model, we were able to define a test space consisting of subspaces for input cases (*i.e.*, HTML tag options), user actions (*i.e.*, user action options), and traditional software configuration options (*i.e.*, Mozilla's run-time configuration options).
- We demonstrated the generality of the Skoll system and DCQA ideas by developing and executing a simple yet interesting DCQA process at a reasonable level of effort. We were able to integrate an existing test automation tool (*i.e.*, a GUI test automation tool [79]), an analysis technique (*i.e.*, delta debugging algorithm), and an adaptation strategy (*i.e.*, temporary constraints) in to the Skoll infrastructure.

Chapter 8

Conclusions

In this dissertation, to overcome the shortcomings of in-house QA efforts, we have defined a novel notion of distributed, continuous quality assurance (DCQA). We have built an infrastructure, algorithms and tools for developing and executing through, transparent, managed, and adaptive DCQA processes. We have then developed several novel DCQA processes and empirically evaluated them, with a special focus on cost efficiency and applicability of these processes to real-life, highly-configurable software systems (e.g., application/web servers, middlewares, and database systems). Our results strongly suggest that these new processes are an effective and efficient way to conduct QA tasks such as evaluating performance characteristics and testing for functional correctness of evolving, large-scale software systems.

We envision a distributed, continuous quality assurance process (DCQA) conducted around-the-world, around-the-clock on a powerful, virtual computing grid provided by thousands of end-user machines during off-peak hours. We call this DCQA processes are distributed, opportunistic, and adaptive. They are distributed: Given a QA task we divide it into several subtasks each of which can be performed on a single user machine. They are opportunistic: When a user machine becomes available we allocate one or more subtasks to it, collect the results when they are

available, and fuse them together at central collection sites to complete the overall QA process. And finally they are adaptive: We use earlier subtask results to schedule and coordinate subtask allocation.

To evaluate the DCQA ideas, we have first built a generic infrastructure. A cornerstone of this infrastructure is a formal model of the space of QA subtasks, called a configuration and control model. In general, a configuration and control model specifies how QA tasks are divided into smaller subtasks. We have used this model in planning the global QA process, for adapting the process dynamically, and to aid in interpreting results. We have then developed and evaluated several novel DCQA processes using our infrastructure and empirically evaluated them by conducting large-scale experiments on real software systems.

In Chapter 4, we have presented the results of several initial feasibility studies where we applied the DCQA ideas for functional testing of a large, widely-deployed middleware system over its numerous configurations. We have also introduced and evaluated the fault characterization process. The fault characterization process aims at identifying configuration options and their settings which are highly correlated with the manifestation of failures. We obtained the fault characterizations by exhaustively testing the entire configuration space and feeding the results to a classification tree analysis. Our experiments suggest that automatic fault characterizations can help developers quickly pinpoint the root causes of failures, leading to a much quicker turn-around time for bug fixes.

In these initial set of feasibility studies, we have also learned that the QA subtask spaces of DCQA processes can be quite large. Even with a large pool of user-

supplied resources, brute-force QA approaches (*e.g.*, exhaustive testing) may become infeasible or simply undesirable. Consequently, we have then focused on efficient and effective strategies to reduce the space of QA subtasks into a manageable size in ways that do not compromise the accuracy and the goal of the DCQA processes.

In Chapter 5, we have introduced and evaluated sampling strategies for fault characterizations in complex configuration spaces. The idea was to systematically sample the configuration space, test only the selected configurations, and compute fault characterizations on the resulting data. We have based our sampling strategy on computing two different kinds of mathematical objects, called a covering array and a variable strength covering array. Our experiments suggest that this approach is as nearly accurate as that based on exhaustive testing, but is much cheaper; it provided a 50% to 99% reduction in the number of configurations to be tested.

In the literature, covering arrays have been frequently used to test input combinations of programs [53, 9, 25, 11, 29, 48]. These studies focus on finding unknown faults in already tested systems and equate covering arrays with code coverage metrics. Our research is different in that we have applied covering arrays to system configuration options and we have assessed their effectiveness in revealing option-related failures and identifying failure inducing options.

Variable strength covering arrays have been discussed in [19, 20, 22]. These studies provide scenarios of when variable strength arrays might be useful. As far as we are aware, no prior study has provided any evidence of their effectiveness and compared them with their fixed level counterparts as we have done in this research.

In Chapter 6, we have focused on performance-oriented regression testing and

introduced and evaluated the main effects screening process. The main effects screening process aims at detecting performance degradations across large configuration spaces as a result of system changes. We have introduced an observation-based space reduction strategy for this process where the idea was to focus limited resources and time on the configuration options that matter most (*i.e.*, wasting less resources on noncritical options). For that, we have leveraged the design of experiments theory. In particular, we have leveraged fractional-factorial screening designs and d-optimal designs. Our empirical evaluations suggest that the main effects screening process can reliably and precisely detect key sources of performance degradations with significantly less effort compared to conventional techniques.

For performance-intensive software systems, although performance-oriented regression testing is as important as functional regression testing, we are not aware of any prior work addressing the performance-oriented regression testing of software systems. Moreover, as far as we are aware, no prior study in the computer science literature has provided any empirical results on the effectiveness of screening designs in evaluating system performance characteristics.

In the research presented in Chapter 5 and 6, we were given a space for testing (defined by the configuration and control model) as well as a DCQA process goal (*i.e.*, identifying failure inducing options or detecting performance degradations). Our solution approaches were consisted of two dimensions. The first dimension was to decide what to test in this space. This dimension is important because it helps us to keep the cost of the process under control. The second dimension was to analyze the results to reach the process goals. However, the second dimension

makes the first dimension even more difficult to realize; the test cases should be carefully chosen from the testing space so that the results can reliably be analyzed to reach the process goals. For instance, the selected test cases should not introduce bias to the analysis.

Having these two dimensions is another distinguishing feature of our research. Prior studies often realize only one of these dimensions. For example, the studies presented in [19, 20, 22, 53, 9, 25, 11, 29, 48] realize only the first dimension whereas the studies presented in [51, 50, 49, 28, 63, 10] realize the second dimension only.

8.1 Main Contributions

The main contributions of this research can be summarized as follows:

1. We define a novel notion of distributed, continuous quality assurance (DCQA).
2. We present a generic infrastructure, tools, and algorithms for developing and executing managed, thorough, transparent, and adaptive DCQA processes.
3. We introduce novel DCQA processes for both functional testing and performance-oriented regression testing of software systems with a special focus on cost efficiency and applicability of these processes to real-life, highly-configurable systems.
4. We demonstrate the broader applicability of the design of experiments theory to model and improve software quality.
5. We present initial large-scale empirical evaluation of the DCQA ideas using

real-life software systems in real-life scenarios. Our results strongly suggest that DCQA processes are an effective and efficient way to conduct QA tasks such as evaluating performance characteristics and testing for functional correctness of large-scale software systems.

8.2 Limitations and Future Work

We have so far evaluated the DCQA ideas by conducting empirical studies on the ACE+TAO software systems. ACE+TAO are a family of large (*i.e.*, 2MLOC+), widely-deployed, performance-intensive middlware systems.

All empirical studies suffer from threats to their internal and external validity. For this thesis, we are primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice.

One potential threat to external validity concerns the representativeness of the ACE+TAO subject applications, which though large are still just one suite of software systems. A related issue is that, in our experiments, we have focused on a relatively simple and small subset of the entire configuration space of ACE+TAO.

Another potential threat is that for the time period we studied the ACE+TAO system, the system was in a fairly stable phase, *i.e.*, changes were made mostly to fix bugs and reduce memory footprint, but the system's functionality was relatively stable. For situations where a system's basic functionality is in greater flux, it may be harder to analyze the results of the DCQA processes presented in this thesis. Take the main effects screening process as an example, when the system's

basic functionality is in flux, it may be harder to distinguish significant performance degradation from normal variation.

To have a better assessment of the current limitations and benefits of the DCQA ideas, as a future work, we plan to apply our approach to a wide range of software systems with larger, more realistic configuration spaces. The following list summarizes potential future research directions:

- Our prototype DCQA system is a research system. Thus, there is a lot of room for improving its implementation (*e.g.*, in terms of scalability, fault tolerancy, and portability) and making it an out-of-the-box solution.

- A large-scale DCQA testbed can be created to develop and evaluate DCQA processes in real software development cycles of real software systems.

- The configuration and control model can be enriched to handle hierarchical models (not just the flat option spaces currently supported). Priorities can be incorporated in the model so that different parts of the configuration space can be explored with different frequencies. Real-valued option settings can also be incorporated into the model.

- The ISA can be enhanced to allow planning based on cost models and probabilistic information, *e.g.*, if historical data suggests that users with certain platforms send requests at certain rates, the ISA can take this information in account when allocating QA jobs.

- We have so far treated software systems as black-boxes. The connection be-

tween the DCQA ideas and white-box QA techniques can be explored.

- The DCQA ideas can be used to collect diverse user profiles in the field. The effect of these profiles on software development activities can be explored.

- Other DCQA scenarios (not just the ones concerning highly-configurable systems) can be explored.

- The connection between the design of experiments theory and the quality assurance of software systems can further be explored.

BIBLIOGRAPHY

[1] Exposing application alternatives. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 384, Washington, DC, USA, 1999. IEEE Computer Society.

[2] D. Abramson, , and R. Sosic. Relative debugging using multiple program versions. In *Key note address, 8th int. symp. on languages for intensional programming*, 1995.

[3] D. Abramson, , and R. Sosic. Guard: A relative debugger. In *Software practice and experience*, pages 185–206, 1997.

[4] D. Abramson, I. Foster, J. Michalakes, and R. Sosic. Relative debugging: A new paradigm for debugging scientific applications. In *The communications of the association for computing machinery*, pages 67–77, 1996.

[5] Apache HTTP Server. `http://www.apache.org`.

[6] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. *SIGSOFT Softw. Eng. Notes*, 28(1):2–9, 2003.

[7] G. Box and R. Meyer. An analysis for unreplicated fractional factorials. In *Technometrics*, pages 11–18, 1986.

[8] L. Breiman, J. Freidman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth, Monterey, CA, 1984.

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

[10] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, pages 480–490. IEEE Computer Society, 2004.

[11] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review*, 1998.

[12] M. Chateauneuf and D. Kreher. On the state of strength-three covering arrays. *Journal of Combinatorial Designs*, 10(4):217–238, 2002.

[13] B. Childers, J. Davidson, and M. Soffa. Continuous Compilation: A New Approach to Aggressive and Adaptive Code Transformation. In *Proceedings of the International Parallel and Distributed Proces sing Symposium*, Apr. 2003.

[14] I.-H. Chung and J. K. Hollingsworth. Runtime selection among different api implementations. *Parallel Processing Letters*, 13(2), 2003.

[15] I.-H. Chung and J. K. Hollingsworth. Automated cluster-based web service performance tuning. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, pages 36–44, Washington, DC, USA, 2004. IEEE Computer Society.

[16] I.-H. Chung and J. K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *SC '04: Proceedings of the 2004*

ACM/IEEE conference on Supercomputing, page 30, Washington, DC, USA, 2004. IEEE Computer Society.

[17] J. M. Clarke. Auotamated test generation from a behavioral model. In *Proceedings of the eleventh international software quality week*, 1998.

[18] D. Cohen, S. Dalal, A. Kajla, and G. Patton. The automatic efficient test generator (aetg) system. In *Proceedings of the Fifth international Symposium on Software Reliability Engineering*, pages 303–309, 1994.

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

[20] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE '03)*, pages 38–44, 2003.

[21] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th international conference on Software engineering*, pages 38–48, 2003.

[22] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, and J. S. Collefello. Variable strength interaction testing of components. In *Proceedings of the 27th international computer software and applications conference*, 2003.

[23] Comprehensive Perl Archive Network (CPAN). `http://www.cpan.org`.

[24] CVS: Concurrent Versions System. `http://www.nongnu.org/cvs`.

[25] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE)*, pages 285–294, 1999.

[26] S. R. Dalal and C. L. Mallows. Factor-covering designs for testing software. *Technometrics*, 40(3):234–243, 1998.

[27] Dart: Tests, Reports, and Dashboards. `http://public.kitware.com/Dart`.

[28] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd international conference on Software engineering*, pages 339–348. IEEE Computer Society, 2001.

[29] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE '97)*, pages 205–215, 1997.

[30] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-speci c, programmer-written compiler extensions. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 1–16, 2000.

[31] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.

[32] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, pages 213–224. IEEE Computer Society Press, 1999.

[33] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *IEEE transactions on software engineering*, volume 27, pages 99–123, 2001.

[34] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, pages 449–458. ACM Press, 2000.

[35] L. S. Finn and D. F. Chernoff. Observing binary inspiral in gravitational radiation: One interferometer. *Phys. Rev. D*, 47(6):2198–2219, 1993.

[36] GNU GCC. `http://gcc.gnu.org`.

[37] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM Press, 2002.

[38] A. Hedayat, N. Sloane, and J. Stufken. *Orthogonal arrays.* Springer-Verlag, New York, 1999.

[39] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 135–145. ACM Press, 2000.

[40] J. K. Hollingsworth and P. J. Keleher. Prediction and adaptation in active harmony. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 180, Washington, DC, USA, 1998. IEEE Computer Society.

[41] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *In Companion of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[42] J.Nelder and R.Mead. A simplex method for function minimization. *The Computing Journal*, 7(4):308–313, 1965.

[43] Kamen Yotov and Xiaoming Li and Gan Ren et.al. A Comparison of Empirical and Model-driven Optimization. In *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2003.

[44] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[45] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *In 10th Annual International Static Analysis Symposium*, 2003.

[46] A. S. Krishna, N. Wang, B. Natarajan, A. Gokhale, D. C. Schmidt, and G. Thaker. CCMPerf: A Benchmarking Tool for CORBA Component Model

Implementations. In *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04)*, Toronto, CA, May 2004. IEEE.

[47] A. Krolak, J. A. Lobo, and B. J. Meers. Estimation of the parameters of the gravitational-wave signal of a coalescing binary system. *Phys. Rev. D*, 48(8):3451–3462, 1993.

[48] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002.

[49] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154. ACM Press, 2003.

[50] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Sampling user executions for bug isolation. In *First international workshop on remote analysis and measurement of software systems*, 2003.

[51] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. public deployment of cooperative bug isolation. In *Second international workshop on remote analysis and measurement of software systems*, 2004.

[52] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms. *Real-Time Systems Journal*, 23(1/2):85–126, 2002.

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

[54] Microsoft SQL Server. http://www.microsoft.com.

[55] Microsoft XP Error Reporting. http://support.microsoft.com.

[56] T. Mitchell. An algorithm for the construction of the 'd-optimal' experimental designs. *Technometrics*, 16(2):203–210, 1974.

[57] Mozilla Web Browser. http://www.mozilla.org.

[58] Netscape Quality Feedback System. http://www.netscape.com.

[59] Oracle 9. http://www.oracle.com.

[60] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 128–137. ACM Press, 2003.

[61] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. *SIGSOFT Softw. Eng. Notes*, 27(4):65–69, 2002.

[62] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st international conference on Software engineering*, pages 277–284. IEEE Computer Society Press, 1999.

[63] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th international conference on Software engineering*, pages 465–475. IEEE Computer Society, 2003.

[64] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, 1993.

[65] R. L. Ribler, H. Simitci, and D. A. Reed. The autopilot performance-directed adaptive control system. *Future Gener. Comput. Syst.*, 18(1):175–187, 2001.

[66] C. V. Rijsbergen. *Information Retrieval.* Butterworths, London, UK, 1979.

[67] Robert Ricci and Chris Alfred and Jay Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review*, 33(2):30–44, Apr. 2003.

[68] SAS. `www.sas.com`.

[69] D. Schmidt and S. Huston. *C++ Network Programming: Resolving Complexity with ACE and Patterns.* Addison-Wesley, 2001.

[70] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, Apr. 1998.

[71] B. B. T. System. http://www.bugzilla.org.

[72] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *IEEE/ACM Supercomputing*, 2002.

[73] The ACE ORB. http://www.cs.wustl.edu/~schmidt/TAO.html.

[74] The Network Simulator. http://www.isi.edu/nsnam/ns.

[75] The Visualization Toolkit. http://public.kitware.com/VTK.

[76] B. White and J. L. et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[77] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[78] C. F. J. Wu and M. Hamada. *Experiments: Planning, Analysis, and Parameter Design Optimization*. Wiley, 2000.

[79] X11-GUITest-0.20. search.cpan.org/~ctrondlp/X11-GUITest-0.20.

[80] Y. Xie and D. Engler. Using redundancies to find errors. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60. ACM Press, 2002.

[81] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European software engineering conference held jointly with*

the 7th ACM SIGSOFT international symposium on Foundations of software engineering, pages 253–267. Springer-Verlag, 1999.

[82] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM Press, 2002.

[83] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[84] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the International Conference on Distributed Systems 2002*, 2002.