ABSTRACT

Title of dissertation: LARGE SCALE DISTRIBUTED TESTING FOR FAULT CLASSIFICATION AND ISOLATION

Sandro M. Fouché, Doctor of Philosophy, 2010

Dissertation directed by: Professor Adam Porter
Department of Computer Science

Developing confidence in the quality of software is an increasingly difficult problem. As the complexity and integration of software systems increases, the tools and techniques used to perform quality assurance (QA) tasks must evolve with them. To date, several quality assurance tools have been developed to help ensure of quality in modern software, but there are still several limitations to be overcome. Among the challenges faced by current QA tools are (1) increased use of distributed software solutions, (2) limited test resources and constrained time schedules and (3) difficult to replicate and possibly rarely occurring failures. While existing distributed continuous quality assurance (DCQA) tools and techniques, including our own Skoll project, begin to address these issues, new and novel approaches are needed to address these challenges. This dissertation explores three strategies to do this.

First, I present an improved version of our Skoll distributed quality assurance system. Skoll provides a platform for executing sophisticated, long-running QA processes across a large number of distributed, heterogeneous computing nodes. This dissertation details changes to Skoll resulting in a more robust, configurable,

and user-friendly implementation for both the client and server components. Additionally, this dissertation details infrastructure development done to support the evaluation of DCQA processes using Skoll – specifically the design and deployment of a dedicated 120-node computing cluster for evaluating DCQA practices. The techniques and case studies presented in the latter parts of this work leveraged the improvements to Skoll as their testbed.

Second, I present techniques for automatically classifying test execution outcomes based on an adaptive-sampling classification technique along with a case study on the Java Architecture for Bytecode Analysis (JABA) system. One common need for these techniques is the ability to distinguish test execution outcomes (e.g., to collect only data corresponding to some behavior or to determine how often and under which conditions a specific behavior occurs). Most current approaches, however, do not perform any kind of classification of remote executions and either focus on easily observable behaviors (e.g., crashes) or assume that outcomes' classifications are externally provided (e.g., by the users). In this work, I present an empirical study on JABA where we automatically classified execution data into passing and failing behaviors using adaptive association trees.

Finally, I present a long-term case study of the highly-configurable MySQL open-source project. Exhaustive testing of real-world software systems can involve configuration spaces that are too large to test exhaustively, but that nonetheless contain subtle interactions that lead to failure-inducing system faults. In the literature covering arrays, in combination with classification techniques, have been used to effectively sample these large configuration spaces and to detect problem-

atic configuration dependencies. Applying this approach in practice, however, is tricky because testing time and resource availability are unpredictable. Therefore we developed and evaluated an alternative approach that incrementally builds covering array schedules. This approach begins at a low strength, and then iteratively increases strength as resources allow reusing previous test results to avoid duplicated effort. The results are test schedules that allow for successful classification with fewer test executions and that require less test-subject specific information to develop.

LARGE SCALE DISTRIBUTED TESTING FOR FAULT CLASSIFICATION AND ISOLATION

by

Sandro M. Fouché

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
2010

Foreword

Portions of this dissertation are derived from research and papers co-authored by the candidate and published elsewhere. Chapter 3 is based on *Techniques for Classifying Executions of Deployed Software to Support Software Engineering Tasks* [42] and *Low Overhead Classification of Deployed Software Executions* [38]. Chapter 4 includes text and diagrams from *Incremental Covering Arrays Failure Characterization: a detailed case study* [36].

Dedication

To John Gannon for setting the standard very high, and believing I could one day reach it. None of this would have been possible without him.

Acknowledgments

First and foremost I want to thank my advisor, Adam Porter who has given me considerable guidance and support throughout my time in graduate school. He has consistently believed in me and done everything he can to ensure my success. I also would like to thank my other committee members: Professor Michael Brin, Professor Bill Gasarch, Professor Atif Memon, and Professor James Purtilo for serving on my committee, providing mentorship and for their useful feedback.

Professor Myra Cohen at the University of Nebraska-Lincoln has been invaluable as a collaborator and for feedback on my research. Her research ideas and expertise have made a valuable contribution to the work presented in this dissertation. I also need to recognize and thank the other mentors and collaborators on my research: Michael Last, Professor Alessandro Orso, and Professor Doug Schmidt. I'd also like to thank the other members of my research group Charles Song, Lee Ellis, and Cemal Yilmaz for their assistance, technical skill and feedback as we worked together over the years.

I thank Fritz McCall, Joe Ridge and the rest of the UMIACS support staff for their technical assistance in designing, installing, operating, and maintaining the Skoll DCQA cluster and our other associated hardware and software.

I need to thank my many friends who have supported me throughout my time in graduate school and have made my stay in Maryland much more enjoyable, including (but not limited to): Soeren Bartels, Adam Bender, Darya Bobryakova, Laura Bright, Penelope Brooks, George Caragea, Derek Fisher, Shaun Gittens, Charles

Glover, Michelle Goodman, Debbie Heisler, Aram Khalili, Scott Lawrence, Louis Licamele, Anandhi Mani, Austin Parker, Brian Postow, Megan Rook-Koepsel, James Rose, Rob Sherwood, Amy Sliva, Jaime Spacco, Jef Thompson, Chadd Williams, and Peter Wood.

I would also like to acknowledge the help, guidance and support of Dr. Thelma Williams, Ritzie Coleman, Gwen and Allan Kaye as well as Nancy Garrison, without whom I might not have weathered the stormy bits of my time at Maryland as well.

Last but not least, I want to thank my wife Kimberly, our daughters Teresa and Alexandra and the rest of my family for their care, love, and understanding while I completed this work.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
CA	Covering Array
CBIT	Continuous Build, Integration and Test
CVS	Concurrent Versions System
CSV	Comma Separated Values
DBMS	Database Management System
DCQA	Distributed Continuous Quality Assurance
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
ID	Identifier
ISA	Intelligent Steering Agent
JABA	Java Architecture for Bytecode Analysis
MLOC	Million Lines of Code
NFS	Network File System
OOB	Out Of Bag
QA	Quality Assurance
QOS	Quality Of Service
RAMSS	Remote Analysis and Measurement of Software Systems
SOA	Service-Oriented-Architectures
SQL	Structured Query Language
XML	Extensible Markup Language

Chapter 1

Introduction

As software systems become increasingly complex, software quality assurance techniques must keep pace. Unfortunately, this is not the case. Software complexities that are not well-handled by current techniques include:

- operating system versions, dynamic libraries, third party components, and network dependencies cause distinct runtime execution profiles
- cross-platform requirements, security, and performance needs dictate a variety of compilation and runtime options.
- hardware differences effect the system as a whole
- market competition and customer needs force increased functionality (and shorter development cycles)

All these contribute to the increasing complexity and pace of software development; rendering software testing more important than ever. But quality assurance techniques have not kept pace with the increase in software complexity.

Many existing QA processes are static and do not make good use of testing resources; continuously testing a small number of fixed points in the configuration space, regardless of whether this testing uncovers faults. There is little point in continuously running tests that give little or no information about the system as it evolves; yet that is just what many QA processes do, repeating the same test schedules day after day. Static testing techniques that run the same tests over and over on a small subset of the possible system configurations aren't good uses of QA resources and time. To effectively test contemporary software, quality assurance tools and techniques have to be adaptive – to dynamically change according to the scope and schedule of the software system under development.

Consider for example, a typical web server. These systems can contain numerous compile- and run-time options for configuring OS dependent code: the maximum number of concurrent connections, which commands are enabled, and so on. Each specific configuration might bring with it unique features, flaws, performance profiles and constraints. The full *configuration space* of the web server is the exponentially large cross-product of all possible option settings of each component in the server system. Given n configuration options each with k_n settings, the *configuration space* is the N-dimensional set of possible configurations for the system –$configSpace(k_1, .., k_n)$. Any point in that space represents a single, testable, configuration of the software. Since any one of these configurations might be used in the field, software testing processes increasingly focuses on discovering subtle *interaction* failures; failures due to specific combinations of option settings. To completely test a software system, the test process would ideally consider each possible instance or setting of each component of the system – effectively testing every possible configuration that might occur. The combinatorial explosion of configuration options available presents an insurmountable obstacle for in-house QA efforts. Furthermore, market forces and cost-cutting measures focused on driving profit provide increasingly less time and physical resources for effective QA.

Given the large numbers of configurations possible and the pressure to release new versions of software, exhaustive testing is realistically impossible. Therefore, software developers usually limit their testing efforts to a relatively small portion of the configuration space. Historically, this has been done by: testing just a small set of default or popular configurations, randomly choosing some subset of configurations for testing, or choosing test configurations based on developer intuition and experience. While all of these methods can find flaws in a software system, they exhibit low or possibly poor *test diversity*; this is problematic for several reasons. First, they cover a limited – perhaps biased – sub-space of possible system configurations. Second, unless the test configurations are updated over time, testing will continue in already debugged configurations while leaving large sections of the configuration space unexplored. Finally, while some software failures are obvious and easily found, some bugs only occur under rare combinations of conditions or with very low frequency. Fixing these bugs can be difficult, if only because replicating them in a controlled manner is difficult. Quality assurance processes with poor test diversity may never isolate these faults, instead focussing on repeatedly searching for bugs in the same, well-tested, parts of a software system.

My thesis is that we could improve test diversity by applying techniques that dynamically adapt based on existing test results. Specifically, my research has applied adaptation in three ways, at the tool level, in generating sampling models, and finally for test scheduling. My initial work focussed on revamping the existing Skoll test infrastructure to support adaptive strategies and adding scalability and robustness to the architecture. I then used Skoll as a testbed for applying adaptation to the generation of sampling models for execution data. Finally, a similar dynamic approach was used to generate test schedules for an automated QA process where a number of variables, including the complexity of configuration interactions and available QA resources, could not be predicted ahead of time *a priori* and might vary significantly over time.

1.0.1 Skoll DCQA System

Fundamentally, the challenge facing any QA effort is the scope of modern software; as systems grow in size and complexity the set of parameters affecting the correct functioning of the softwares grows exponentially. Each underlying parameter: system architecture, compilation options, run-time settings, host configuration, and user preferences multiplies the size of the testing problem. Conceptually, we can view any possible configuration of a software system with n parameters as a point within an n-dimensional space. Consequently, the QA process amounts to a search of the n-dimensional parameter space, hunting for those particular points (configurations) that contain bugs. Historically, that search was performed in an ad-hoc, brute force fashion, proceeding through that conceptual space in a, hopefully, organized manner. But modern systems are growing faster than in-house QA teams can successfully search the parameter space; new versions of programs ship with only a fraction of the potential parameter space searched, and commensurately few errors found. Indeed, many projects ship without clear knowledge of the amount of testing completed relative the scope of the possible QA effort. Distributed continuous quality assurance (DCQA) in general, and Skoll in particular, attempt to address the problem of scale in the QA process by taking a divide and conquer approach to searching the parameter space. These tools utilize the power of distributed computing to methodically partition the software parameter space and search for faults. But even still it is apparent a brute force search of an application's entire parameter space is to large for the time between (increasingly short) product cycles.

Yilmaz et al. [84] explored this general approach to support fault characterization of large-scale configurable systems. In that work, covering arrays were used to generate test schedules. Those schedules were executed in parallel across a grid of computers, results returned to central servers, and discovered failures automatically classified to help developers find their underlying causes. The overall process was managed by the Skoll system [61] – a distributed continuous quality assurance (DCQA) environment that allows for highly parallel execution of QA processes. The results suggested that the covering array test schedules produced better classification models than equivalently-sized random samples and that the process scaled reasonably well to large configuration spaces.

Chapter 2 details cross-cutting infrastructure development to support the QA strategies presented herein. The existing Skoll framework provided mechanisms for adaptive testing, but did not address scalability, flexibility or the ease of use requirements for deploying a DCQA process in large-scale, real-world environments. The changes made leverage external tools to provide low-level services, while refining Skoll's approach to QA tasks as a whole; increasing the capacity, capability, and flexibility of Skoll as a whole.

With a viable testbed for further research completed, our emphasis shifted, from providing more computational cycles for quality assurance ,to tools that attempt to reduce the size of the problem space through the use of adaptive techniques. Adaptive techniques reduce the size of the parameter space by identifying and focussing QA effort on portions of the space that illuminate failures in the system – parts of the parameter space that don't help identify bugs are given decreased importance.

1.0.2 Adaptive Sampling Association Trees

There is increasing interest in research related to the remote analysis and measurement of software systems (RAMSS) [12, 13, 34, 39, 54, 58, 62, 66, 68, 70, 72, 85]. In general, these approaches instrument numerous instances of a software system, each in possibly different ways, and distribute the instrumented instances to a large number of remote users. As the instances run, they collect execution data and send them to one or more collection sites. The data are then analyzed to better understand the system's in-the-field behavior. RAMSS techniques typically collect different kinds of data and use different analyses to achieve specific software engineering goals. One characteristic common to many of these techniques is a need to distinguish execution outcomes; there are many scenarios in which this information is useful. A first example is remote analyses that use information from the field to direct debugging effort and need to know whether that information comes from a successful or failing execution (e.g., [58, 67]). A second example is self-managing applications that reconfigure themselves when performance is degrading and so must be able to determine when the system has entered a problematic state. Knowing the outcomes of remote executions would also be useful in the nowadays common situation of software systems released with known problems (e.g., for the aforementioned time-to-market considerations). Being able to automatically identify specific problematic behaviors as soon as these systems are deployed allows for tasks such as measuring how often specific problems occur (e.g., to prioritize the debugging effort) and gathering detailed information about likely causes of problems in a selective way (e.g., by using the behavioral models to aid in diagnosing problem causes and location). Finally, automatic outcome classification can also be used in-house, in place of expensive test oracles, for newly generated test cases once enough have been (manually) classified.

Chapter 3 presents our approach to address this problem using statistical learning algorithms to adaptively model and predict execution outcomes based on *execution data*, that is, data collected at runtime. More specifically, the techniques build behavioral models by analyzing execution data collected from one or more program instances (e.g., by executing test cases in-house or in the field or by examining fielded program instances under user control). Developers then either analyze the models directly (e.g., for fault localization) or use the models to gather further information from other program instances. In the latter case, they lightly instrument numerous instances of the software (i.e., the instrumentation captures only the small subset of execution data referenced by the behavioral model); which are then sent to end users who run them. As the instances run, the collected execution data is fed to the previously built model to predict whether the current run is likely to be a passing or failing execution. We implemented and evaluated a technique called adaptive sampling association trees; this approach can build accurate classification models, while incurring substantially less overhead than existing techniques. With this approach individual program instances sparsely sample a set of potential measurements. Initially, the sampling weights are uniform, but over time they are adapted to favor useful measurements with the lowest overhead. Our goal, therefore, is to maximize the information content of the collected data while minimizing the performance penalty, dynamically selecting instrumentation to optimize our QA process.

1.0.3 Incremental Covering Arrays

Finally, Chapter 4 returns to the original work that motivated updates to the Skoll system and addresses the issues with that approach. One approach for testing large configuration spaces involves using a sampling strategy derived from mathematical objects called covering arrays [24, 28, 49]. This approach generates a test schedule that satisfies specific coverage metrics, that of testing all t-way combinations of the configuration options. Covering arrays, however, while promising, have several limitations. First, covering arrays depend on developer insight to select the key sampling parameters; in order to reliably classify faults that are caused by t configuration options, samples must be built that test all t-way combinations of these options. This means the tester must know *a priori* what *strength*—value of t—to use. If t is set too large, resources will be wasted executing long test schedules; while selecting t to be too small may result in poor classification results. Since systems often have multiple failures with different causes, either – or even both – of these situations is virtually guaranteed. Second, because it is not generally possible to use a portion of a t-way covering array to reliably classify faults caused by fewer than t options, developers must run the covering array as a unit, waiting until all tests have been run before classification can start. In this situation, there is no way to ensure that faults are found and classified as early as possible. Third, as testing continues, each covering array schedule is generated independently from all others. There is no mechanism to exploit configurations that have already been tested in previous covering array schedules. This approach often runs more tests than necessary, incorrectly correlates failures with configuration parameters, duplicates work, and suffers delays in reporting classification information.

To deal with this limitations we developed a new approach called *incremental covering arrays*. The key feature of our redesigned approach is that it efficiently creates multiple strength covering array test schedules. It begins by building a low strength covering array, testing the indicated configurations, doing automatic failure classification, and providing classification results to developers. It then uses incrementally stronger covering arrays as time and testing resources allow. A central tactic of this approach is to lower the cost of incremental execution by carefully reusing results from earlier test runs. It thereby dynamically adapts to provide the best classification data possible with the available time and QA resources. Together these tools and process form the basis for my on-going research in adaptive techniques for quality assurance.

Chapter 2

Skoll DCQA System

The bulk of the new code implementation in my research has focused on evolving the Skoll DCQA system into a reliable, easily configurable, and user-friendly QA system suitable for wide-spread deployment in real-world QA environments. To provide a context for the work already completed and it's relationship to prior work, this chapter starts with an overview of Skoll QA process and introduction to the previous version of the system. I then detail revisions to the Skoll system as well as the design and deployment of the dedicated Skoll QA cluster. Finally, I present information on several QA targets tested using the current system with an emphasis on our high-level research goals for the Skoll project, the practical design decisions that guided the development of the system, and lessons learned through the implementation of the QA process.

2.1 Overview

The Skoll project was designed to provide tools for the coordination and control of effective, quality assurance processes. Specifically, Skoll aims to intelligently direct Internet computer resources in a distributed and continuous manner to significantly and rapidly improve software quality. Designed as a quality assurance system to be deployed across geographical locations and business organizations, Skoll provides a set of tools, policies, and practices for distributed, continuous quality assurance.

Skoll is based on a client/server model meant to be distributed across a wide variety of heterogeneous platforms and host operating systems. Skoll servers are responsible for planning, coordination, and results processing for QA tasks. Meanwhile the client performs individual QA sub-tasks under the direction of Skoll servers, including downloading, building, and testing applications. The interaction between a Skoll client and server is best expressed via a brief example QA processing task.

First, a Skoll client sends a *QA job request* to the server. A Skoll server then selects an appropriate project and test configuration based on client capability, task availability, and QA scheduling requirements. Additionally, the specific QA configuration within the server provides a set of constraints on the configuration options to be applied to limit the selection of QA task to valid configurations. Once a particular configuration has been selected, the resulting QA job is sent to the client for execution. The Skoll client then applies any locally-defined constraints (e.g. time, permission, security, etc.) to the QA job sent by the server; resulting in a final set of QA sub-tasks that the client executes. Once execution is completed, the client sends any job results and logs back to the server for analysis and storage.

While this basic operational overview of Skoll has remained consistent between the previous and current implementations of Skoll several important improvements to the system have been made to increase the flexibility of the system. A brief introduction to the previous work is warranted to serve as background to the changes implemented.

2.2 Previous Work

The initial implementation of Skoll was primarily devised as a feasibility study for a new DCQA process and, as such, was designed and implemented in a fairly rigid manner. The system existed primarily as a single server process that dispatched commands to a QA-target specific client application. The result was that, while Skoll worked within the confines of a small research environment, deploying it across large numbers external clients would have been problematic. Additionally, each new QA target required a substantially different, customized Skoll client which limited the installed base of client hosts.

2.2.1 Server

The initial server was implemented as several hundred lines of Perl organized in a modular fashion, but comprising a single runtime process. Specifically, the server contained modules for: handling parsing and generating XML messages, managing client registration and generation, management of QA task configuration information, an intelligent steering agent (ISA), and results processing and visualization. While all the functionality is necessary, the overall monolithic nature of the resulting server code was sub-optimal. The server was limited because:

- it only supported a single thread of execution
- changes to the QA configuration required a restart of all the server subsystems
- a separate server instance (and network configuration) were required for each

QA subject

- it constituted a single point of failure for the entire backend system
- the Skoll server implemented no security for the data or QA processes

One issue of specific concern in our use of Skoll for adaptive testing (a feature that the intelligent steering agent was meant to facilitate) was the need to restart the server after configuration changes. In our initial field tests using Skoll it became evident that the ability to modify the test configuration was critical to our ability to implement some adaptive policies. While in practice hard resets of the server are not a problem, the distributed and continuous nature of our testing meant that each restart wiped out all state information about any test client actively running QA processes. While the data lost by this process would have been on (as of now) previous versions of the QA subject, that data might still prove useful for debugging purposes. Allowing for frequent shifts in configuration as well as QA process, without resetting the server process each time, would be the single biggest improvement to enable adaptive testing using Skoll. While it might have been possible to retrofit fixes for the restart problem, as well as the majority of the other issues, into the existing codebase, starting with a new system architected from the ground up to support a larger variety of projects seemed to be the best course of action.

2.2.2 Client

The Skoll client followed a similar design to the server. Created using a couple hundred lines of Perl, well organized into modules that supported XML parsing and generation, task execution, and network communication. But, the primary drawback of the old Skoll client was that it was customized for both the QA subject and the specific QA host system. Developers interested in using Skoll for their project would first have to create XML configuration information detailing the QA parameters of their application. Next, they were required to modify the prototype Skoll client to meet the specific needs of their test project. Any application-specific functionality they required would have to be implemented in Perl using the Skoll API. This presents an immediate problem as a prerequisite for implementing new QA processes in Skoll is that QA engineers are forced to become familiar with the Skoll API and codebase, and must possibly also learn to program in Perl.

Subsequently, after the server configuration and application-specific version of the Skoll client were completed, each QA user was required to register each QA test host with the Skoll server. The server then generated a unique client instance for that specific host which had to be downloaded and manually installed on each machine. Client code generated for one host was not intended to run on others; failure to run the client installation process or significantly upgrading the QA host computer would result in faults during QA testing. This made it time-consuming to add large numbers of test clients and difficult to leverage network disk resources or automated system integration tools.

Finally, the primary test subject for the initial implementation of Skoll, by it's nature, made the prototype server configuration and test client cumbersome to understand and modify. The initial QA subject, ACE+TAO (a Corba implementation), [31] is a large system (2 MLOC+) with a large configuration space. The resulting sample Skoll configuration files are comprehensive; however, they require significant time and effort to comprehend. Additionally, testing a Skoll installation by attempting to execute the included ACE+TAO test process requires up 12-18 hours to complete just a single test execution. Using the original Skoll package a starting point for new QA processes was a daunting challenge to newcomers to Skoll specifically, and DCQA in general.

2.2.3 Limitations of Skoll

The previous implementation of Skoll had several limitations that restricted its accessibility and limited our ability to perform certain research tasks. Succinctly, Skoll:

- lacked simple and convenient configuration. Configuration via XML server files was cumbersome and overly centralized. A mechanism that allowed for remote configuration updates as well as automated manipulation of the system configuration was required.
- had limited flexibility in installation and operation. Since several hours (if not days) of development had to go into deploying a new QA process on Skoll, there was a significant barrier to entry. Additionally, the highly customized nature of each deployment made leveraging existing work difficult.
- inability to perform QA simultaneously on multiple test subjects. As initially implemented Skoll had no facilities for deploying or managing multiple QA subjects. If clients attempted to connect to one Skoll server, and it was inactive

or had no QA tasks to process, the clients would not be assigned work. Even if there was another QA process with pending work to distribute.

- required a hard reset of the server to support any configuration change. The need to restart the server after each configuration change made using adaptive testing techniques cumbersome if not impossible.

Addressing these limitations was the first task in my research. Guided by working with the initial Skoll implementation and the need to adapt Skoll to a variety of new projects I proposed and implemented a new Skoll infrastructure to support future research using adaptive techniques.

2.3 Current Implementation

Skoll's initial implementation endeavored to show it's viability as a research platform for DCQA processes, but our goals had expanded and our current and future work dictated several changes to the Skoll system. Specifically, Skoll had to address our needs for:

- a large installed user-base to provide massive amounts of data for use in probabilistic data analysis techniques.

- a variety of QA targets to show that our QA techniques apply generically to several problem domains.

- increased understanding of real-world computing conditions to drive future DCQA research.

- the flexibility to handle dynamic test processes

We have implemented several changes to the existing Skoll system to meet our ongoing research needs including changes to enhance client usability, the ability to target multiple QA projects simultaneously, and the development of an in-house QA test cluster. Throughout it's initial implementation, Skoll had evolved to track changes to its primary QA subject: ACE+TAO, but was not easily adaptable to accommodate other projects. This section documents the new Skoll architecture targeted at: adoption of adaptive QA strategies, enhancing usability in Enterprise environments, increased overall functionality and more convenient use. The basic theory of operation for the new version of Skoll does not differ significantly for the previous version, but behind the scenes radical changes were made to the operation of the system. The next subsection provides an overview of the new server process. Subsection 2.3.2 details changes to the client program, and finally subsection 2.3.3 outlines the client-server interactions of the new system.

2.3.1 Server

Conceptually, the server is a major departure from the previous incarnation. The priority with the server was to implement much of the common DCQA functionality in the server process itself, and then to add application-specific or esoteric capabilities using external tools and processes. To accomplish this, a great deal of the server functionality in the new Skoll system has been implemented in a DBMS system – specifically MySQL 5.1, but any relational database would suffice – and ancillary functions are handled with small scripts written in Perl. Figure 2.1 shows the structure of the DB Schema used for all the current Skoll projects. The tables that make up the server schema for a single QA project are:

- *parameters* - a list of named options $(V_1, .., V_n)$ that represent configurable settings and decision points affecting the QA subject and define the possible configuration space for any QA tasks.
- *parameter values* - a list of sets of named values $(C_1^1, .., C_1^j, .., C_n^1, .., C_n^k)$ that map onto parameters defining the range for each parameter.
- *configurations* - the run queue of QA jobs (scheduled, running and completed). Each configuration represents a single point in the QA configuration space under test.
- *configuration values* - a list of sets of option-value pairs $(V_1, C_1^x), ..(V_n, C_n^y)$ that map to a configuration
- *result values* - a list of 3-tuples (named tests, outcomes, and error messages) mapped to results

  $((T_1, O_1, E_1), .., (T_n, O_n, E_n))$
- *results* - a list of completed QA jobs that maps result-values to a configuration.
- *variables* - a list of supersede-able QA environment variables $(E_1, .., E_s)$ that are sent to every QA client

- *tasks* - an ordered list of system commands $[F_1, .., F_r]$ that when combined with a configuration and variables assignments constitutes a single QA task.

Each test subject gets an independent database (named skoll_$project-name$) on the DB server and needs to have the basic schema instantiated and populated. The Skoll distribution includes a MySQL script that creates an empty QA project database and a basic Perl script that populates the parameter and configuration tables from simple text files. One side benefit of this implementation is that any SQL-compliant DBMS system and compatible design tools can be used to instantiate and manage a QA project. There are a wealth of command-line, GUI, and platform specific tools available for managing SQL databases that are available to assist in working with Skoll DB entities.

In addition to the table elements in figure 2.1, each Skoll project database has a small number of stored procedures. There routines are written in SQL (specific to the underlying DBMS) and provide QA developers with an abstract interface to the database structures while allowing for data transformation, normalization, and policy enforcement. Most QA engineers should never need to modify these procedures. However, since they are stored on a per project basis, if it becomes necessary to customize their behavior it can be done without impacting other QA projects running in simulataneously.

To implement multiple QA projects within the same DBMS server, Skoll maintains a separate table that lists all the known *projects*. This table allows Skoll to dispatch clients that are not allocated to a specific project to work on any project

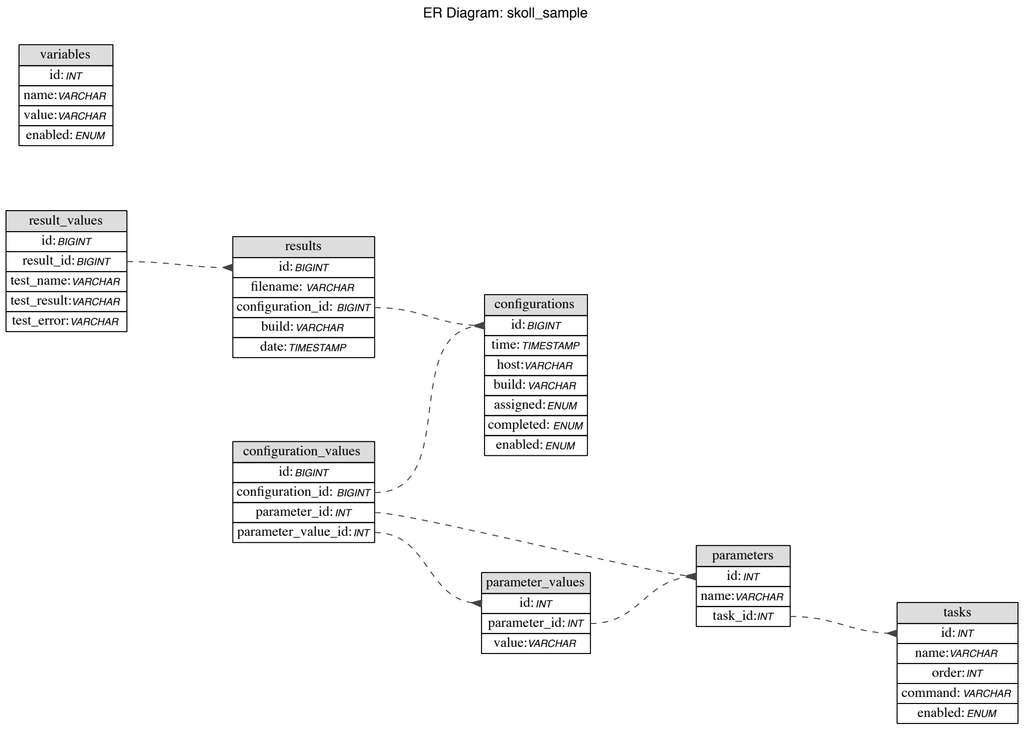

Figure 2.1: Structure of the Database Underlying the Current Skoll Server

that currently has QA jobs pending. Each record in the projects table contains the name of the project, and a bit indicating whether or not the project is actively accepting new clients. While not currently implemented, it would be a small matter to add a $DB - host$ field to each project record to support server redirection in a future version of Skoll.

Finally, the current Skoll implementation includes a results processing subsystem. The current results system is broken up into three components: data acquisition, extraction and processing, and delivery. Data acquisition is handled by a small Perl script, that retrieves the results from an upload directory and inserts them into the Skoll database. Another script that runs queries against the QA project database and formats the output as a CSV file is responsible for extraction. That same script also initiates data processing by invoking an instance of the statistical package R [7]. Finally, the data is moved to our results web server via a simple file transfer script. Since results parsing and data processing are highly specific to the QA project and the needs of the development team it makes sense to decouple this subsystem from the rest of the server. The script used for data extraction from the database often needs to be customized. Nevertheless, the Skoll distribution includes a basic script that will dump data from any project for those who only need simple functionality. This script aslo serves and an example and starting point for users to create a more custom script. Note that while we preferred Perl for text processing and database access and R for data analysis, QA engineers are free to use any tools that suit their needs for these tasks – so long as they have the ability to transact with an SQL-compliant DBMS.

Security in Skoll is provided by database user/password security implemented by the backend DBMS. This prevents malicious manipulation of QA data on the Skoll server. While the communication channel used between a Skoll client and server doesn't use a secure protocol, the communication channel itself often must be securely tunneled to meet datacenter firewall requirements. Nonetheless, a secure client/server protocol would be beneficial for wide-spread adoption of Skoll outside of research environments. One advantage of using an off-the-shelf DBMS implementation as the basis for the Skoll server is that any and all existing security polices and products are immediately compatible with the Skoll server. This decision has also had other beneficial consequences. DBMS systems are generally designed for high transaction volumes and support multi-threaded operation allowing for many hundreds (or possibly thousands) of simultaneous client connections transacting on several projects in parallel. Most modern DBMS systems (including later versions of MySQL) support: clustering, distributed operation, replication, data partitioning, and RPC; these features allow for scalable, reliable, and flexible operation of Skoll servers. Relational databases have been available for several years and there is a wealth of documentation, knowledge and trained personnel that are available as resources for Skoll users.

2.3.2 Client

Critical to our goal of having a large installed user-base for Skoll is the ease of joining and participating in Skoll QA projects. While obtaining and deploying a single instance of Skoll on a client machine has been a relatively simple process, optimizing the installation and use of the Skoll client is important to its future success. Previously, a user needed to:

1. visit our website [86]
2. fill out a web-form with details of your client configuration
3. download a custom configured client
4. run our installation script
5. start the client

These five steps are straight-forward and not terribly difficult, and for a single client instance this would be acceptable. Unfortunately, these steps have traditionally been ill-documented, required domain specific knowledge, and presented a significant amount of labor to perform on large numbers of machines. Additionally, Skoll imposed several constraints on the run-time configuration of the client, and provided no mechanism for circumventing those constraints.

One of our highest priorities in embarking on revisions to Skoll was to improve the end-user experience by allowing users of all skill levels to participate in Skoll QA projects. The revised Skoll client now attempts to make intelligent decisions on behalf of the user, and allows the user to override our assumptions by modifying a client configuration file (config/ClientConfig.xml).

Most significantly the changes implemented in the Skoll client now make it simple to use Skoll on a large number of clients installed in an enterprise setting.

On our own Skoll cluster, a single download of the client to the file server can be used to run on every machine in our cluster. The steps to acquire, install, and run the Skoll client are now:

1. download the current Skoll client
2. start the client

This procedure is the same for a single home user or an enterprise administrator with thousands of systems.

There are currently two implementations of the Skoll client–one implemented in Perl and another in Java. Fundamentally, they have the same structure and capabilities, but future development will probably focus on the Java implementation since Java provides more advanced functionality and a smaller installation footprint. Essentially the client is composed of a small amount of code to execute QA jobs, a dynamic variable store, and support libraries for database communication and HTTP transmission. The client functions as a remote execution engine gathering QA environment information and QA tasks from a Skoll server, executing those tasks locally, and then returning the results to the server. The goal in the client design was a minimal footprint with low-overhead. Some sample scripts are included with the client providing examples for invoking the client from cron and for guaranteeing exclusive execution. QA developers are free to augment client functionality with tools installed on the client system and executed from within the QA task list (ex. results processing, log parsing, etc).

2.3.3 Theory of Operation

Once a Skoll QA project has been configured and instantiated within a DBMS server, and a Skoll client has been installed and started on QA host, the following client-server transactions occur within the system:

1. client contacts the server to indicate availability to participate in *project* (or null if not configured for a specific project)
2. server looks up *project* in the projects table and indicates to the client if the project is active
3. if active, the client requests the QA environment variables
4. the server returns the list of variables associated with the QA project
5. the client allocates a dynamic variable buffer and overrides any defined variables with configured local values.
6. the client requests a QA configuration and task list
7. the server assigns a QA configuration to the client and returns the data to the client
8. the client and server disconnect while client processes the QA job
9. the client transmits the QA results back to the server
10. the server updates the QA configuration queue and stores results

Because the environment variables are stored in a dynamic buffer their values can be updated by the client during the QA job. We have used this in conjunction with client-side system commands in the task list to implement conditional execution of tasks depending on the return status of previous tasks. A sample QA task list is included in Table 2.1.

One major improvement in the new design is that the Skoll server process is almost entirely stateless. A crash during QA activities only affects clients actively transacting with the server, and only causes data loss in the event of a failure

name	order	command	enabled
checkout	45	cd $workDir; sh -c "test -d sample || svn co sample "	true
update	50	cd $workDir/sample; svn update	true
clean	75	cd $workDir/src; make clean	true
config	100	cd $workDir/src; ./configure	true
build	110	cd $workDir/src; make	true
test	120	cd $workDir/src/mysql-test; ./mysql-test-run.pl –force	true
log	130	$binDir/log_result -p $project $logFile	false

Table 2.1: Example Skoll QA Task List

during the actual transmission of results. Because we're careful to retain results on the client until the server confirms receipt, this eventuality is also handled. All other clients continue operation without awareness of, or effect from, the server crash. The small amount of state the server has, namely that a client has been issued a QA configuration and it's start time, are used to timeout and re-dispatch configurations assuming that the client has failed to complete it's QA job.

2.4 Skoll DCQA Evaluation Cluster

As part of the new implementation of Skoll and our ongoing DCQA testing we designed and built a dedicated quality assurance computing cluster. Composed of 176 Intel platform computers deployed between the University of Maryland, College Park and Vanderbilt University, the Skoll cluster is a large-scale testbed for quality assurance analysis. Designed to reflect common enterprise architecture, the cluster takes a novel approach for academic distributed systems. Typical academic computing systems are optimized for solving individual, large-scale problems using parallel computing techniques. Computers in these academic installations are commonly used in concert to tackle a large problem often with centralized command and control facilities. Distributed continuous quality assurance, on the other hand can be thought of as repetitively solving many, small, discrete problems. In this light, the Skoll cluster eschews common high performance computing wisdom, instead using commodity systems and technologies. Had the Skoll cluster utilized the specialized components and software often deployed in academic computing environments, the likelihood our results would reflect, or our software would find adoption in, environments outside of academic circles would be smaller. This section describes the specific design goals and implementation issues we addressed during the creation of the cluster.

Design goals The Skoll cluster was designed to test our distributed continuous quality assurance implementation in a simulated enterprise environment. Our design had several unique goals:

- the use of commonly deployed hardware
- diverse systems
- network topologies typical of those deployed around the world

Each of these goals drove the design decisions below.

Commodity Systems The Skoll cluster is implemented using commodity Dell computer systems; these systems are similar to the those deployed throughout businesses world-wide. While it is more common for academic computing clusters to use specialized equipment to meet high performance computing needs, we have specifically chosen hardware systems that have widespread adoption in mainstream computing. These systems, while designed for desktop use, represent a balance between cost and performance, and are common throughout office environments the world over. Typical of desktop computers, our commodity systems are memory constrained, have limited hard disk space, and only utilize off-the-shelf networking components. The cluster also eschews the typical split-power distribution, redundant components and other fail-safe systems; each node in the system therefore reflects real-world capacity for failure and outage. It is important to note that the system is designed around modeling distributed behavior and not overall computational power, unlike most advanced computing clusters.

Heterogeneous Environment While the cluster is currently implemented with a limited number of hardware configurations (the configurations used at Maryland differ from those at Vanderbilt), the intention is to provide a heterogeneous computing environment, especially as hardware is added during future expansion. Seldom in real-world systems are companies afforded the luxury of a single hardware platform deployed throughout the enterprise. Consequently, the unique challenges and opportunities that arise from a diverse computing environment are critical to our research. Here again, we forego typical research computing practice in favor of improving our QA model. Most large scale-cluster environments prefer a homogenous environment to simplify management and operations of the cluster. As implemented today the Skoll cluster uses two basic hardware configurations, but several different operating systems–including varieties of Linux, FreeBSD, Solaris and Microsoft Windows. The cluster is deployed also with support for re-configuring OS installations to support our research in heterogeneous environments.

Loosely Coupled Systems The biggest departure from typical research computing environments is the cluster's network implementation. Deployed using only off-the-shelf components, our systems are not designed for parallel or tightly coupled computing tasks. Utilizing only common 100-megabit and gigabit Ethernet, the Skoll cluster looks little like typical research computing installations. No effort has been made to provide synchronization or coordination facilities between individual computers used in the Skoll cluster, in marked contrast to typical distributed computing environments. Additionally, the Skoll cluster only uses standard Internet connectivity between the two sites, forgoing the largely academic (but high-performance) Internet2.

Implementation Details Initially deployed for use in the Fall of 2005 and fully operational in the summer of 2006, the University of Maryland component of the Skoll DCQA Evaluation Cluster features:

- 120x Dell PowerEdge servers
  - 2.8 GHz Intel Pentium 4 CPU
  - 1-unit rack mount cases
  - 1 gigabyte of system memory
  - 40 gigabytes local hard drive storage

  - 2x gigabit network interfaces

- Sun Sunfire V240 file server
  - Dual 1.5 GHz UltraSparc IIIi CPUs
  - 2 gigabytes of system memory
  - 146 gigabytes of local hard drive storage
  - 4x gigabit network interfaces

- Sun StorEdge Fibre-Channel Disk Array
  - 3 terabytes of storage
  - 12x 250 gigabyte Serial-ATA hard drives

- 3x Dell PowerConnect 3448 switches

Nodes in the cluster are mostly configured with a mix of Redhat Enterprise Linux Advanced Server 4 and Windows XP. Short term work has begun to expand the number of operating systems running on the cluster; as well as increase network performance to the file server. Running on Sun Solaris 10 and the Veritas File System, the cluster's file server provides the majority of filesystem spaces to the nodes via NFS.

2.5 DCQA Test Subjects

Revisions to Skoll, as well as development of the evaluation cluster, initially focussed on our ongoing effort to provide continuous QA of our previous test subject: the ACE+TAO Corba implementation [31]. In addition to updating Skoll to increase its capabilities and flexibility, we tracked development changes to the ACE+TAO build and QA processes. Under continuous (sometimes heavy) development, weighing in at more than two million lines of code, and with approximately 40 participating programmers from a variety of institutions, ACE+TAO is an ideal test subject for any automated DCQA process. ACE+TAO has evolved into a system with the flexibility and scope beyond its developers ability to test completely; making it a perfect candidate for continued research with Skoll.

But to increase our flexibility and understanding of Skoll's performance independent of ACE+TAO, most of our subsequent work has focussed on other QA projects–primarily MySQL during their transition from MySQL 4 to MySQL 5 and subsequent minor releases. MySQL [64] is an open-source SQL-compliant DBMS that has been under development for 15 years. MySQL has over 2 MLOC, works on 20+ computer platforms, has over 450 functional tests, and has been downloaded more than 10 million times. In short, it provides ample opportunity to test Skoll against real-world situations. The majority of our testing has focussed on a subset of the MySQL configuration space that is limited to only 72 million configurations, but was much larger than the ad-hoc testing the developers had done to date. Details on the empirical study that came out of our DCQA process can be found in section 4.5.

In all, the current Skoll DCQA system has successfully completed over 375,000 QA jobs, totaling approximately 1.5 million computing hours, with little human intervention (except in dealing with running out of filesystem space).

2.6 Related Work

Several research projects have explored remote analysis and monitoring in software systems in general and distributed continuous quality assurance in particular. Several open-source projects include regression test suites that end-users can use to evaluate installation, performance, and/or functional issues. Some examples include GCC [3], MySQL [64], and the Linux Test Project [5]. While users have the ability to return test results to the program developers they often do not. Even when users return data, their testing is not systematic, controlled, or well documented – often crucial configuration information is not captured as part of the test process. The results are therefore rarely useable for precision fault isolation and provide more of a coarse feedback mechanism.

Recognizing there is often inadequate time to complete the QA process, and that systems are deployed without complete testing, residual test coverage monitoring, Pavlopoulou et al. [71], augments traditional static analysis methods (statement coverage, etc.) with remote monitoring and analysis for the portions, or residual, of QA subjects that were incompletely evaluated prior to deployment. While their work utilizes remote monitoring as the basis for lighter re-instrumentation of future clients, their technique particularly focuses on RAMSS as an ancillary process to in-house testing and does not represent a viable system for long-term deployment.

A direct ancestor to our own Skoll system–software tomography [14] and the Gamma system [69], presented by Orso, Bowring, and colleagues, decomposes the task of performing QA into sub-tasks that are distributed across many, lightly-instrumented program instances. Gamma addresses the same fundamental issues addressed in previous work with Skoll [60], namely to provide a complete quality assurance process while burdening end-users with only minimal impact. By subdividing the QA process into many smaller tasks, distributing the QA effort across many test clients, and integrating the results to formulate a complete view of the QA subject, the Gamma system was able to provide a picture of QA data typically thought of as highly-intrusive (e.g. statement and method coverage) in a minimally-intrusive manner. Furthermore, case studies with Gamma showed that it performed successfully with a medium-sized system: JABA [10] (approx. 40K LOC).

Liblit et. al. [55, 56, 57] focuses on bug isolation using software tomography. Liblit is primarily concerned with applying sampling methods across deployed instances of programs to track bugs that may not occur during in-house testing. Also, the team introduces statistical debugging methods in RAMSS and leverages it to perform crash prediction using collected data. Case studies with their Cooperative Bug Isolation techniques were able to identify previously unknown bugs [59] in deployed software including, Rhythmbox and EXIF.

Other DCQA processes do exist – for instance Dart [2] and CruiseControl [1] – but these systems are largely ad-hoc and have limited scope and flexibility in implementing the QA process. Most existing approaches limit the QA process to detecting failed builds, crashes, or narrowly-scoped software failures. They don't provide mechanisms for continuous collection, analysis, and tracking of CBIT data. Moreover, existing systems often inadequately document their QA process, making it hard to use the resulting data for scientific measurement and analysis.

2.7 Future Work

Skoll continues to evolve to meet our research needs. Moving forward, the following limitations in Skoll will need to be addressed:

- intelligent steering agent
- automatic client architecture detection
- extending QA parameters
- coordinated groups of clients
- network modeling

Details of future improvements to Skoll are outlined as part of this section.

2.7.1 Intelligent Steering Agent

One by-product of the new implementation of Skoll is that the intelligent steering agent was never re-implemented within the new server infrastructure. Today the QA job scheduler is implemented as a priority run queue, with no provision to re-order jobs based upon test results. The best mechanism to re-introduce the ISA would be add triggers or pre-/post-event stored procedures to the existing DBMS. These events could then be used to invoke an external ISA to make any necessary changes to the job order/scheduling priorities. This strategy has the benefit of being modular, extensible, and stateless.

2.7.2 Automatic Client Architecture Detection

One of the goals of Skoll is to provide the ability to execute tests across a large, heterogeneous cross-section of client hosts, therefore tools to support that effort are necessarily dependent on knowing the context of client execution. The Skoll client currently depends on the underlying QA project to detect the client platform or architecture. Future client versions need to integrate architecture detection to enable a wider variety of host platforms and to vet clients for participation in QA projects before a QA task is assigned. Our previous attempts to implement this type of detection proved to be more complicated than we initially anticipated. If possible integrating an existing library or external tool would be the most viable option.

2.7.3 Extending QA Parameters

Currently, Skoll considers transmission and compilation of QA subjects, as well as the QA test cases themselves, distinct from other configuration parameters. Future versions of Skoll need to integrate these QA task parameters into a larger, more comprehensive view of the parameter space. The same criteria, metrics, visualizations, and adaptive strategies would therefore apply to transmission, compilation and test cases. The end-to-end deployment of an application, and the state of the system over time, would be part of the larger quality assurance process. Fundamentally this is not a large change to Skoll, but rather reflects relatively small changes to the handling of project configuration, client configuration, and test cases within the Skoll system.

2.7.4 Coordinated Clients

Current Skoll clients operate as independent entities. As each client starts up, it requests a QA job from the server; it is then free to complete the QA task when and as it sees fit. The Skoll server makes no attempt to coordinate separate clients to perform a combined test, but that limitation constrains the systems ability to perform QA tasks on distributed software applications. With the ubiquity of client-server and peer to peer applications, the ability to test systems that run on separate computers appears to be critical.

While it is possible to fake Skoll control of a multi-part system by wrapping the execution all of the parts into a single QA task, Skoll support for coordinated execution has several advantages. First, since the location and exact nature of the clients participating in a QA task would be out of the target applications control, the test would represent accurate, real-world computational conditions. Second, the exact workload generated could vary dynamically, since the actual clients available and their behavior would be subject to availability and configuration. Finally, success and failure reporting would be under finer-grained control of the system, which would lead to more flexibility in automated planning of further test executions.

The changes to the system required to implement coordinated groups of clients is significant and may exceed the scope of the research proposed here. Continued evaluation of the feasibility and requirement for this feature is anticipated as part of our on-going research.

2.7.5 Network Modeling

As Skoll adapts to target more network-centric computing, the current network configuration would have to evolve to model a large variety of network topologies and conditions. Typical conditions on the Internet provide varying quality of service (QOS) by location and time. Efforts to understand the behavior of QA subjects that depend on the network for functionality and performance will require Skoll to identify, track, and manage network QOS as a parameter under QA control.

Future implementations of the Skoll DCQA Evaluation Cluster may include network traffic shaping components to simulate more erratic conditions on the Internet in a controlled manner. On a longer time-scale further research partnerships, as well as external QA participants, will bring more diverse, if less controlled, network conditions into the space of capabilities offered by the evaluation system.

Chapter 3

Low Overhead Fielded Instrumentation using Adaptive Sampling Association Trees

3.1 Overview

Several research efforts are focusing on tools and techniques to support the remote analysis and measurement of software systems (RAMSS) [12, 13, 34, 39, 45, 54, 58, 62, 66, 68, 70, 72, 85]. In general, these approaches instrument numerous instances of a software system, each in possibly different ways, and distribute the instrumented instances to a large number of remote users. As the instances run, they collect execution data and send them to one or more collection sites. The data are then analyzed to better understand the system's in-the-field behavior.

RAMSS techniques typically collect different kinds of data and use different analyses to achieve specific software engineering goals. One characteristic common to many of these techniques is a need to distinguish execution outcomes. There are many scenarios in which this information is useful. A first example is remote analyses that use information from the field to direct debugging effort and need to know whether that information comes from a successful or failing execution (e.g., [58, 67]). A second example is self-managing applications that reconfigure themselves when performance is degrading and so must be able to determine when the system has entered a problematic state. Knowing the outcomes of remote executions would also be useful in the now common situation of software systems released with known problems (e.g., for time-to-market considerations). Being able to automatically identify specific problematic behaviors as soon as these systems are deployed would allow for tasks such as measuring how often specific problems occur (e.g., to prioritize the debugging effort) and gathering detailed information about likely causes of problems in a selective way (e.g., by using the behavioral models to aid in diagnosing problem causes and location). Finally, automatic outcome classification could also be used in-house, in place of expensive test oracles, for newly generated test cases once enough have been (manually) classified. These are just some examples of cases in which such data would be useful.

Despite many recent advances, existing techniques suffer from numerous problems. First, they often make oversimplifying assumptions (e.g., they equate failing behaviors with system crashes) or assume that the classification of the outcome is provided by an external source (e.g., the users). These assumptions severely limit the kinds of program behaviors that can be analyzed and the applicability of the techniques. Second, these techniques often require collecting massive amounts of data – imposing significant overheads on every participating program instance. Example overheads include: code bloat due to code rewriting, bandwidth occupation due to remote data collection, and slowdown and perturbed performance due to code instrumentation. Finally, these techniques are not designed to adapt gracefully over time as systems, environments, and usage patterns change.

This chapter proposes and evaluates four new techniques for automatically classifying execution data collected from deployed applications by execution outcomes. To be able to perform controlled experimentation and suitably validate our results, in this chapter we focus our empirical investigation on binary outcomes only: "pass," which corresponds to executions that produce the right results, and "fail," which corresponds to incorrect executions. Conceptually, however, the techniques should be equally applicable to any discrete set of outcomes.

Our techniques use statistical learning algorithms to model and predict execution outcomes based on *execution data*, that is, data collected at runtime for these executions. More specifically, the techniques build behavioral models by analyzing execution data collected from one or more program instances (e.g., by executing test cases in-house, in the field, or by examining fielded program instances under user control). Developers can then either analyze the models directly (e.g., for fault localization) or use the models to gather further information from other program instances. In the latter case, they lightly instrument numerous instances of the software (i.e., the instrumentation captures only the small subset of execution data referenced by the behavioral model). These lightly instrumented instances are then distributed to users who run them in the field. As the instances run, the collected execution data is fed to the previously built model to predict whether the current run is likely to be a passing or failing execution.

To help us achieve this high-level vision, we initially defined and developed an instantiation of the technique and performed a four-part feasibility study [41]. Our initial approach is effective and efficient in cases where we can fully train the models in-house, using accurate and reliable oracles. However, the approach is not applicable if part (or all) of the training must be performed on deployed instances because it imposes too much time and space overhead during the training phase. To address this issue, we extend our initial work by developing and evaluating three improved classification techniques that can build models with substantially less data than that required by our initial technique, while maintaining the same accuracy. The first new technique can build reliable models while observing less than 10% of the complete execution data. Each instance collects a different subset of the execution data chosen via uniform random sampling. The second technique is also able to reliably classify executions based on a small fraction of execution data, but it adds the ability to adapt the sampling over time to maximize the amount of information in the data. Finally we modify the second approach with a cost utility function that allows us to maintain high accuracy while significantly minimizing the overhead caused by our instrumentation. The chapter presents all four techniques and discusses their strengths and weaknesses.

We also present several empirical studies in which we applied these techniques to multiple versions of a medium-sized software subject and studied their performance. Our goal was to evaluate the techniques, better understand several issues crucial to the techniques' success, and thereby refine the techniques and improve their ultimate implementations. The first set of studies looks at whether it is possible to reliably classify program executions based on readily-available execution data, explores the interplay between the type of execution data collected and the accuracy of the resulting classification models, and investigates how much data is actually necessary for building good classification models. The second and the third studies examine the extent to which our two newly defined classification techniques allow for minimizing data collection (and the data collection overheads), while maintaining the accuracy of the classification.

The main contributions of this work are:

- A high level vision for an approach that automatically and accurately classifies execution data as coming from runs with specific execution outcomes, collected with low overhead from fielded programs.
- Four instantiations of the approach, three of which are based on newly-defined classification techniques, that can classify execution data according to a binary outcome: pass versus fail.
- An empirical evaluation of several key issues underlying these (and similar) techniques as well as an evaluation of the instantiated techniques themselves.

In the rest of the chapter, we first provide background information on classification techniques and present several example scenarios that describe how classification techniques might be used to support the remote measurement and analysis of software systems (Section 3.2). Section 3.3 describes the experimental subject and data with which we evaluate our four techniques. Then, we introduce our initial approach and present the results of multiple empirical studies aimed at understanding the approach's performance (Section 3.4). Based on these results, we invent, describe, and empirically evaluate a new classification approach called association trees (Section 3.5). In Section 3.6 we develop and evaluate an improved association tree algorithm, called adaptive sampling association trees. An enhanced version of adaptive sampling association trees that include weighting to minimize overhead is presented in section 3.7. In Section 3.8 we discuss related work and, finally, in Section 3.9 we provide some general conclusions and outline future-work directions.

3.2 Background and Motivation

In this section, we first provide background information on techniques for classifying program executions and then motivate our research by presenting, in the form of scenarios, four applications of classification techniques to support software engineering tasks.

3.2.1 Classification of program executions

Machine learning techniques are concerned with the discovery of patterns, information and knowledge from data. They often work by analyzing a training set of data objects, each described by a set of measurable *features* (also called *predictors*[1]), and by concisely modeling how the features' values relate to known or inferred higher-level characterizations. These models are often used to predict the characterizations of other data objects whose characterizations are unknown. Supervised learning is a class of machine learning techniques in which the characterization of each training set object is known at model building time. When the possible characterizations come from a discrete set of categorical values (e.g., "good," "average,"

[1]In the rest of the chapter, we use the terms feature and predictor interchangeably.

and “bad”), the supervised learning problem is known as *classification*. In this work we focus on classification techniques.

Classifying program executions means using readily-available execution data to model, analyze, and predict (more difficult to determine) program behaviors. A typical classification process has two phases: training and classification. Figure 3.1 shows a high-level view of the classification approach that we use in this research. In the training phase, we instrument program instances to collect execution data at runtime and, in the case of collecting data in the field, attach a built-in oracle to the deployed instances. Then, as the instances run, we collect the resulting data. The figure shows two extremes of training phase data collection. In one case, data collection is performed in-house on instances running existing test cases. In the other case, it is performed in the field on instances running under user control. In the former, a traditional oracle can be used to label each run based on its outcome (e.g., “high throughput”, “average throughput”, and “low throughput” if the behavior of interest is performance; “pass” or “fail” if the behavior of interest is functional correctness). In the latter case, the execution data would be collected while the instances run on the users’ platforms against actual user input and each run would be labeled by a built-in oracle attached to the deployed programs.

Note that in these two cases there is a clear trade-off between strength of the oracle and the completeness of the data. In-house, we can often rely on accurate oracles because we typically have complete control over the execution and the (computational and space) overhead of the data collection is usually not an issue. However, in-house we can observe only a limited number of behaviors (the

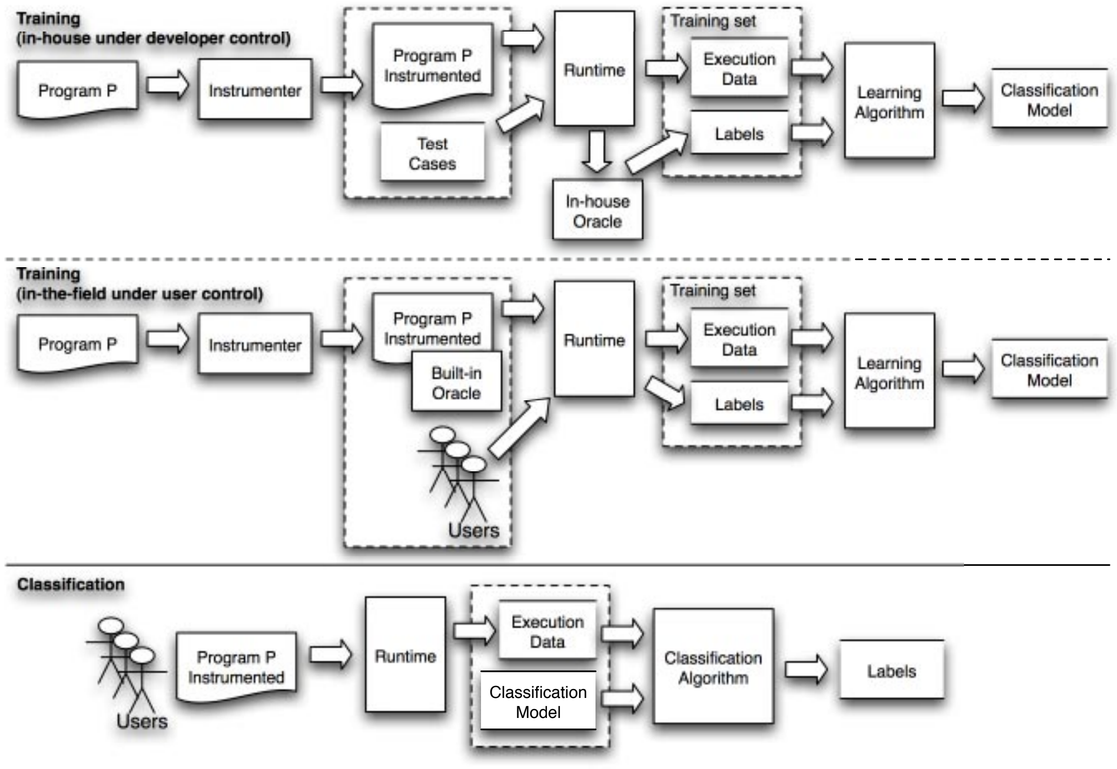

Figure 3.1: Overview of the technique.

ones exercised by the test inputs available and occurring in the hardware and software configurations available), and the models constructed from these observations may not be representative of real executions. In the field, we must typically rely on limited oracles because we must limit the overhead of the data collection (and of the oracle). On the other hand, in-the-field executions are typically much more numerous, varied, and representative than in-house test runs. Which approach is more appropriate depends on the task at hand, as discussed in Section 3.2.2.

Once the labeled training-phase data has been collected, it is fed to a learning algorithm, which analyzes it and produces a classification model of program executions.

After the classification phase, we re-instrument the code to capture only the data needed by the models. When the instances are later run in the field by actual users, appropriate execution data are collected and fed to the previously-built classification model to predict the behavior (i.e., the label) associated with the current execution.

To carry out these processes developers must consider several issues:

- *Developers must determine which* **specific behaviors** *they want to classify.* For instance, one might want to identify previously-seen behaviors, such as open bugs identified during regression testing, or previously-unseen behaviors, such as potential performance problems on less popular hardware platforms (that might not be available in-house). The specific application considered drives several process choices, including:
  - *The outcomes that must be detected.* Developers may want to classify executions in which the system crashes, exceptions are thrown, incorrect responses are given, transaction times are too long, responses are too slow, and so on. There may be only two outcomes (e.g., "pass" or "fail") or multiple outcomes. Here developers must create oracles and measurement instruments that make these outcomes observable.
  - *The environments in which the system must run.* Developers may want to observe system execution on different operating systems or with the system in different operational configurations.
  - *The range of inputs over which system execution will be monitored.* In some cases, developers may be interested in a small set of behaviors

captured by a specific test suite. In others, they may want to see the system execute under actual end-user workloads.

- *Developers must also determine the* **execution data** *on which the classification will be based.* Classification can be based on many different types of execution data, such as execution counts, branch frequencies, or value spectra. Developers must create appropriate measurement instruments to capture these different kinds of data. Also, developers must be aware of the amount of data they will capture. The more data captured, the higher the runtime overhead and the more resources needed to analyze the resulting data.

- *Developers must decide the* **location** *where training-phase data collection occurs.* This data collection can be done in-house or in-the-field. In general, as stated above, we assume that fielded data collection can be done on many program instances while in-house data collection is limited to few instances. We also assume that individual fielded instances cannot tolerate as much data collection volume as in-house instances.

- *Finally, developers must decide which* **classification technique** *to use to create the classification models.* There is a vast array of established classification techniques, ranging from: classical statistical methods (such as linear and logistic regression) to neural network and tree-based techniques (e.g., [32, 44]) to the more recent Support Vector Machines [76]. As we explain in the following sections, most of these techniques have strengths and weaknesses when applied to the classification of remote executions. Which technique to use ultimately depends on the specific task or scenario considered.

Figure 3.2 shows, in an intuitive way, how the various process decisions described above are intimately intertwined. For example, developers who want to understand possible performance problems experienced by end users may want to monitor different execution environments and observe actual usage patterns. As a result, they are forced towards in-the-field data collection, which in turn tends to limit the volume of data collected by each instance and necessitates observing many instances. Alternatively, developers may be interested in collecting very precise information about execution paths corresponding to failing runs of an existing regression test suite to study and understand specific open bugs. In this case, the developers may opt for using sophisticated, heavyweight test oracles and for collecting substantial amounts of data in-house on a small number of instances. In either case, the learning technique chosen must be suitable for the type of data collected.

3.2.2 Scenarios

As mentioned in the overview, this chapter presents four techniques for classifying executions of remotely operating programs. To provide context for their evaluation, we discuss three possible target scenarios in which they might be used: automated identification of known failures in deployed programs, remote failure modeling, and performance modeling of fielded executions.

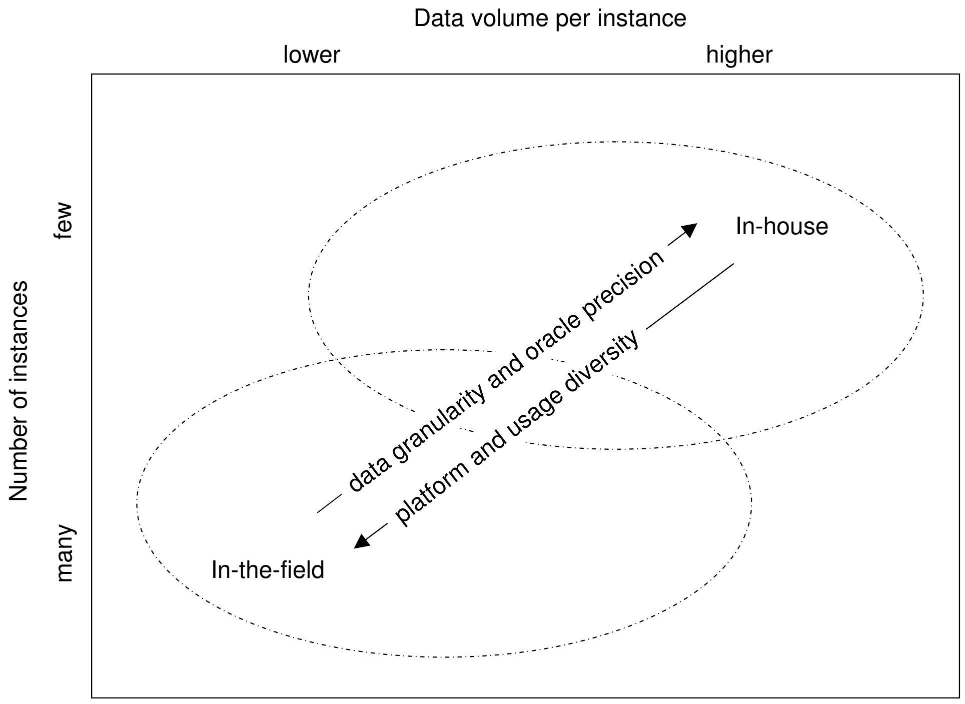

Figure 3.2: Building a classification process.

3.2.2.1 Automated Identification of Known Failures in Deployed Programs

In this scenario, developers want to automatically distinguish fielded program executions that succeed or fail in one of several known modes. Such information could be used in many ways; for instance, it could be used to automatically report data about a failing execution or to trigger data collection and reporting when the program fails. The information could also be used to measure the manifestation frequencies of different failures or to identify system configurations in which specific failures occur. If failure can be predicted early, then this information could also be used to trigger failure avoidance measures. One way to implement this scenario might be to test a program in-house with a known test suite, collect execution data and build classification models for each known failure, and attach the resulting models to fielded instances to predict whether and how a current run fails (or will fail).

Looking back at Figure 3.2 we see that this scenario sits in the upper right corner. Because developers will train the classification technique in-house for known failures, they are free to collect a substantial amount of data per instance and can use heavyweight oracles. On the other hand, they will be limited to instrumenting relatively few instances and observe a more narrow range of platforms and usage profiles than they could if they instrumented fielded instances. In Section 3.4 we describe how our first technique, based on random forests classifiers, can support this scenario.

3.2.2.2 Remote Failure Modeling

In this scenario, developers still want to model passing and failing executions. However, they want to collect the training data from fielded instances running under user control, rather than from in-house instances running existing test suites. The goal of the data collection is in this case to perform some failure analysis, such as debugging.

Developers might implement this scenario by instrumenting fielded instances and then collecting labeled execution data from them. The labels would indicate various outcomes, such as “successful”, “crashed”, “not responding”, or “exception X thrown at line Y”. The specific labels used, and the degree of confidence in the labeling, would depend on the oracles installed in the fielded instances. The collected data is then classified–to relate execution data patterns with specific outcomes. At this point, developers might simply examine the models directly and look for clues to the outcome’s cause. (An example of this approach is Liblit and colleagues’ statistical bug isolation work [54].) Alternatively, developers might execute a large number of test cases in-house and use the models to flag exemplars of a given outcome, that is, to find test cases whose behavior in-house is similar to that of fielded runs with the same outcome. This approach might be especially useful when multiple root causes lead to the same outcome (e.g., several different failures leading to a system crash).

Looking again at Figure 3.2, this scenario lies down and to left of the previous one. In this case, developers collect data in the field to model previously unseen failures. To avoid affecting users, developers need to limit their data collection per instance and must use lighter-weight oracles than they could have used if operating entirely in-house. On the other hand, they can instrument many instances and can observe a wide range of platforms and usage profiles. Section 3.5 describes how our second technique, based on association tree classifiers, can be used to support this scenario.

3.2.2.3 Performance Modeling of Field Executions

In this third scenario, developers want to investigate the causes of a performance problem believed to be due to system aging (e.g., memory leaks or improperly managed locks). Because these kinds of problems occur infrequently and may take a long time to develop, developers instrument a large number of instances in batches—with each batch collecting data over increasingly longer time periods—hoping to increase the chances of observing the problem early. They can achieve this goal by lightly instrumenting deployed program instances running under user control. This collected data is then modeled to predict incidences of the performance problem. The models are then linked back into programs deployed to beta test sites, where the models will trigger deeper data collection when impending performance troubles are predicted. (Note that this scenario assumed that the predictions can be made before the execution ends.)

This scenario lies at the bottom left of Figure 3.2. In this case, developers must be particularly careful to limit data collection overhead to avoid overly perturbing the instance's performance. They must likewise use very simple oracles, but can choose to instrument large numbers of instances and are likely to observe a wide range of platforms and usage profiles. One new aspect here is that, because each run can take a long time to complete, some incremental analysis might help limit overall costs. Section 3.6 describes our third technique, based on adaptive sampling association tree classifiers, which can be used to support this scenario.

3.3 Experimental Subject and Data

The goal of this work is to define and evaluate a set of classification techniques that can support some of the applications described in Section 3.2. To achieve this goal, we used an empirical approach—we designed and conducted several empirical studies that guided the development of the techniques, allowed for their early evaluation, and helped us improve our understanding of the issues underlying the proposed approach to further refine it.

To be able to compare the different techniques and perform controlled experiments, we used the same subject for all studies and targeted a version of the general classification problem involving a behavior that we can measure using an accurate oracle: passing and failing execution outcomes. Therefore, our executions have one of two possible labels: "pass" or "fail." Before presenting our techniques, in this section we introduce our experimental subject and data.

3.3.1 Experimental Subject

As a subject program for our studies, we used JABA (Java Architecture for Bytecode Analysis) [4], a framework for analyzing Java programs. JABA consists of about 60,000 lines of code, 400 classes, and 3,000 methods. JABA takes Java bytecode as input and then performs complex control-flow and data-flow analyses on it. For instance, JABA performs stack simulation (Java is stack based) to track the types of expressions manipulated by the program, computes definitions and uses of variables and their relations, and analyzes the interprocedural flow of exceptions.

We selected JABA as a subject because it is a good representative of real, complex software that can contain subtle faults. In particular, for the study, we considered 19 real faults taken from JABA's CVS repository. The 19 faults were selected by a student in a different research group. The student inspected all CVS commits starting from January 2005, identified the first 19 bug fixes reported in the CVS logs, and distilled the associated faults in the form of source-code differences. We then selected the latest version of JABA as our golden copy of the software and generated 19 different versions by inserting one fault into the golden copy. In this way, we can use the golden copy as an accurate oracle. We also created nine versions of JABA containing multiple faults.

3.3.2 Execution Data and Labels

To build a training set for the versions of JABA considered, we used a set of executions consisting of all the test cases in JABA's regression test suite. These test cases were created and used over the last several years of the system's evolution. Because JABA is an analysis library, each test case consists of a driver that uses JABA to perform one or more analyses on an input program. There are 7 such drivers and 101 input programs—divided into real programs (provided by users) and ad-hoc programs (developed to exercise a specific functionality). Thus, overall, there are 707 test cases (101 times 7).

For each of the versions, we ran the complete regression test suite and collected information about passing and failing test cases as well as various types of execution data. Because performance was not an issue in this case, we were able to collect multiple types of execution data at once. In particular, we collected statement counts, branch counts, call-edge counts, throw and catch counts, method counts, and various kinds of value spectra (e.g., relations between variable values at methods' entry and exit or maximum values of variables).

Considering all versions, we ran about 20,000 test cases. The outcome of each version v and test case t was stored in binary form: "1" if the execution of t on v terminated and produced the correct output; "0" otherwise. As mentioned above, we used the golden version of JABA as an oracle for the faulty versions. Because the test drivers output, at the end of each test-case execution, an XML version of the graphs they build, we were able to identify failures of t for v by simply comparing the golden output and the output produced by t when run on v. (Note that we canonicalize the XML representation to eliminate spurious differences. For example, in the case of multiple outgoing edges from one node, which may appear in different orders in different serializations, we order the edges alphabetically based on their labels.) In addition, we labeled each failing execution, as either a *fatal failure*, for executions that terminated because of an uncaught exception (analogous to a system crash for a C program), or a *non-fatal failure*, for executions that terminated normally but produced the wrong output.

Table 3.1 summarizes the distribution of failures and failure types across the different versions. Each row in the table shows the version number (Version), the list of faults included in each version (Faults included), the total number of failures (Total failures), both in absolute terms and in percentage, the number of non-fatal failures (Non-fatal failures), and the number of fatal failures (Fatal failures). Versions one to 19 are single-fault versions. Because we numbered these versions based on the ID of the fault they contain, the single fault ID associated with each version is the same as the version number. Versions 20 to 28 contain multiple faults, and column "Faults included" lists the IDs of the faults for each version. For example, version 28 contains 6 faults: 5, 9, 11, 12, 13, and 19. For both single- and multiple-fault versions, numbers of versions with a failure rate greater than 8% are highlighted in boldface. As the table shows, six of the 19 single-fault versions and eight of the nine multiple-fault versions made the 8% cutoff. Seven of the single-fault versions and eight of the multiple-fault versions produced fatal failures.

3.4 Classification Using In-House Data Collection

Our goal in this part of the work is to define a technique that can classify executions of deployed programs using models built from data collected in-house. As illustrated in the upper part of Figure 3.1, the models would be built by collecting execution data in-house and using an accurate oracle to label these executions. The oracle could be of different kinds, such as a golden version of the program, an ad-hoc program, or a human oracle. This scenario makes sense when attaching oracles to a deployed program is too demanding in terms of resources (space, computational, or both) or setup costs, which is typically the case for accurate oracles. In particular, and fairly obviously, oracles that require human checking could not be used on remote executions. Thus, our technique needs to operate on readily-collectible exe-

Version	Faults	Total failures		Non-fatal failures	Fatal failures
1	1	0	(0%)	0	0
2	2	12	(1.7%)	0	12
3	3	0	(0%)	0	0
4	4	372	(52.6%)	372	0
5	5	138	(19.5%)	138	0
6	6	0	(0%)	0	0
7	7	144	(20.4%)	144	0
8	8	0	(0%)	0	0
9	9	86	(12.2%)	0	86
10	10	0	(0%)	0	0
11	11	69	(9.8%)	57	12
12	12	4	(0.6%)	4	0
13	13	38	(5.4%)	38	0
14	14	14	(2%)	0	14
15	15	0	(0%)	0	0
16	16	30	(4.2%)	22	8
17	17	105	(14.9%)	0	105
18	18	0	(0%)	0	0
19	19	21	(3%)	0	21
20	1 3	0	(0%)	0	0
21	2 17	113	(16%)	0	113
22	9 12	90	(12.7%)	4	86
23	12 13 19	60	(8.5%)	39	21
24	7 13 16 17	231	(32.7%)	118	113
25	5 7 11 19	207	(29.3%)	174	33
26	2 5 9 12 19	206	(29.1%)	94	112
27	9 12 13 16 17	200	(28.3%)	20	180
28	5 9 11 12 13 19	244	(34.5%)	133	111

Table 3.1: Errors associated with each version. Versions 1–19 are single-fault versions, whereas versions 20–28 contain multiple faults. Versions with failure rate greater than 8% are highlighted in boldface.

cution data and use these data to train the models and classify executions according to the models.

As a specific instance of this general problem, we consider classification of remote executions as passing or failing executions, that is, the behavior that we are interested in classifying is functional correctness. The other two aspects that we need to define within our approach are the machine-learning technique to use and the kind of execution data to consider. The family of learning techniques that we use to define our approach is tree-based classifiers. In terms of execution data, we consider different control- and value-related types of execution information and assess their predictive power using an empirical approach.

In the following sections, we first describe our approach based on tree-based classifiers. Then, we discuss the empirical studies that we performed to assess and refine the approach.

3.4.1 Random Forests Classifiers

Tree-based classifiers are an especially popular class of learning techniques with several widely-used implementations, such as CART [15] and ID4 [6]. These algorithms typically follow a recursive partitioning approach which subdivides the predictor-space into (hyper-)rectangular regions, each of which is assigned a predicted outcome label (i.e., "pass" or "fail"). The resulting models are logically tree-structured (see Figure 3.3). Each node denotes a predicate involving one predictor; each edge represents the *true* or *false* value of the predicate. Each leaf represents a predicted outcome. Based on a training set of data, classification trees are built as follows:

1. For each predictor, partition the training set based on the observed ranges of the predictor data.

2. Evaluate each potential partition based on how well it separates failing from passing runs. This evaluation is often realized based on an entropy measure [15].

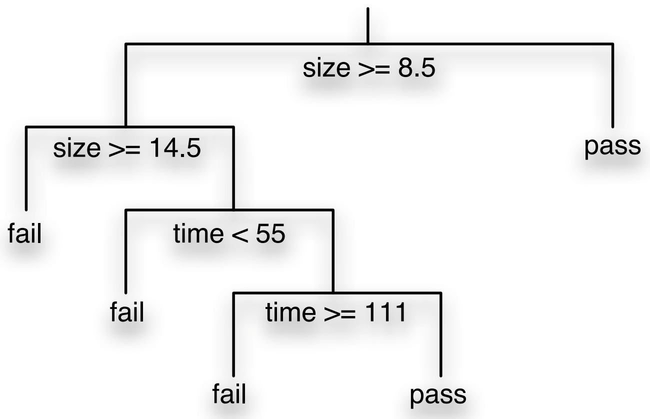

Figure 3.3: Example of tree classifier for executions with two features (i.e., predictors), *size* and *time*, and two possible labels, "pass" and "fail."

3. Select the range that creates the best partition and make it the root of the tree.

4. Add one edge to the root for each subset of the partition.

5. Repeat the process for each new edge. The process stops when further partitioning is impossible or undesirable.

To classify new observations (i.e., to a take a vector of features and predict their associated outcome), tree classifiers identify the rectangular region to which the considered observation belongs; the predicted outcome is the outcome label associated with that particular region. Specifically, for each execution we want to classify, we begin with the predicate at the root of the tree and follow the edge corresponding to the value of the associated predictor in the execution data. This process continues until a leaf is encountered. The outcome label found at the leaf is interpreted as the predicted outcome for the new program run.

For example, Figure 3.3 shows a hypothetical tree-classifier model that predicts the "pass"/"fail" outcome based on the value of the program running time and input size. The decision rules prescribed by the tree can be inferred from the figure. (While traversing the tree, by convention, we follow the left edge when the predicate is true and the right edge otherwise.) For instance, an execution with $Size < 8.5$ would be predicted as "pass", while if $(8.5 \leq Size < 14.5)$ AND $(Time < 55)$, the execution would be predicted as "fail."

The main reason why we chose tree-based classifiers over other classification approaches is that they provide a prediction model that is easy to interpret. However, because they are constructed using a greedy procedure, the fitted model can be quite unstable. That is, fitted classification trees can be very sensitive to minor changes in the training data [16]. To address these problems, we use a generalization of tree-based classification, called random forests, as our classification technique. Random forests is an ensemble learning method that builds a robust tree-based classifier by integrating hundreds of different tree classifiers via a voting scheme. This approach maintains the power, flexibility and interpretability of tree-based classifiers, while greatly improving the robustness of the resulting model.

Consider the case in which we have M predictors and a training set with N elements. We *grow* (i.e., incrementally create) each tree classifier as follows:

1. Sample n cases at random with replacement (bootstrap sample), from the original data. This sample will be the training set for growing the tree.

2. Specify a number $m << M$ such that, at each tree node, m variables are

selected at random out of the M, and the best split on these m is used to split the node.[2]

The forest consists of a large set of trees (500 in our approach), each grown as described above. For prediction, new input is fed to each tree in the forest, each of which returns a predicted classification label. The most frequently selected label is returned as the predicted label for the new input. In the case of a tie (i.e., two or more outcomes are predicted by the same number of trees), one of the outcomes is arbitrarily chosen.

Random forests have many advantages [16]. One is that they efficiently handle large numbers of variables. Another is that the ensemble models are quite robust to outliers and noise. Finally, the random forests algorithms produce error and variable-importance estimates as a byproduct. We use the error estimates to study the accuracy of our classifiers and use the variable importance estimates to determine which predictors must be captured (or can be safely ignored) in the field in order to classify executions.

3.4.2 Experimental Refinement of the Technique

To evaluate and refine the initial definition of our technique for identifying deployed programs' failures, we applied our approach to the subject and data presented in Section 3.3. As discussed above, our technique could be instantiated in many ways, depending on the different types of execution data considered. Instead

[2]A split is simply a division of the samples at the node into subsamples. The division is done using simple rules based on the m selected variables.

of simply picking a possible instance of the technique and studying its performance, we used an empirical approach to evaluate the different aspects of the technique. To this end, we designed and conducted a multi-part empirical study that explored three main research questions:

REQ1: Can we reliably classify program outcomes using execution data?

REQ2: If so, what kinds of execution data should we collect?

REQ3: Is all the data we collect actually needed to produce accurate and reliable classifications?

We addressed REQ1 by measuring classification error rates. We addressed REQ2 by examining the relationship between classification error rates and the type of execution data used in building classification models. We addressed REQ3 by examining the effect of predictor screening (i.e., collecting only a subset of the predictors) on classification error rates. In the following sections, we describe the design, methodology and results of our exploratory studies in detail.

3.4.2.1 Empirical Design

Initially, we considered only single-fault versions of the subject program (see Table 3.1). (The study involving multiple faults is discussed in Section 3.4.2.5.) For each program version and type of execution data collected, we fit a random forest of 500 classification trees using only predictors of that type. We then obtain the most important predictors by using the variable importance measures provided automatically by the random forest algorithm, and find the smallest subset of important predictors that achieves the minimal error rate.

For the study, we excluded versions with error rates below an arbitrarily chosen cutoff of 8%. Since many of the versions do not have their faults exposed by our test cases, this decision effectively removed only 4 versions from consideration. Admittedly, an 8% failure rate is much higher than what we would expect to see in practice, so the generalizability of our results will clearly be limited. Nevertheless, we felt we had to take this step for several reasons. In particular, we would have needed many more test cases (which would have been prohibitively expensive to generate) in order to reliably classify program versions with lower failure rates. In addition, special statistical and machine learning techniques have been developed to handle this specific situation. As we discuss in Section 3.8 these techniques can easily be grafted onto our technique in future studies, but at this point would only make our initial analyses more complicated to interpret. As discussed in Section 3.3 and shown in Table 3.1, six versions made the 8% cutoff. Of these, versions v4, v5, and v7 produced no fatal failures, while v9, v11, and v17 produced 86, 12, and 105 fatal failures, respectively.

For each resulting classification model we computed an error estimate, called the *Out Of Bag* (*OOB*) errors estimate. To compute this quantity, the random forest algorithm constructs each tree using a different bootstrap sample from the original data. When selecting the bootstrap sample for the k^{th} tree, only two-thirds of the elements in the training set are considered (i.e., these elements are in the bag). After building the k^{th} tree, the one-third of the training set that was not used to build the tree (i.e., the OOB elements) is fed to the tree and classified. Given the classification for an element n obtained as just described, let j be the class that got most of the votes every time n was OOB. The OOB error is simply the proportion of times that j is not equal to the actual class of n (i.e., the proportion of times n is misclassified) averaged over all elements. This evaluation method has proven to be unbiased in many studies. More detailed information and an extensive discussion of this issue is provided in Breiman's original paper [16].

3.4.2.2 Study 1 – Research Question 1

The goal of this first study is to assess whether execution data can be used at all to predict the outcome of program runs. To do this, we selected one obvious type of execution data–statement counts (i.e., the number of times each basic block is executed for a given program run), and used it within our technique. We chose statement counts because they are a simple measure and capture diverse information that is likely to be related to various program failures. For each JABA version there are approximately 12,000 non-zero statement counts, one for each executed basic block in the program.[3] Following the methodology described in Section 3.4.1, we built a classification model of program behavior for each version of the subject program. We then evaluated those models by computing OOB error estimates and found that statement counts were nearly perfect predictors for this data set: almost every model had OOB error rates near zero. This result suggests that at least

[3]The random forest algorithm automatically discards the counts for blocks that were never executed because they always have value zero and, thus, have no predictive power.

this one kind of execution data might be useful in predicting program execution outcomes.

Although statement counts were good predictors, capturing this data at user sites can be expensive. For our subjects, instrumentation overhead accounted for an increase of approximately 15% in the total execution time. While this may be acceptable in some cases, it is still a considerable slowdown that may not be practical for many applications. Moreover, the amount of information collected, one integer per basic block, can add considerable memory and bandwidth overhead for large programs and large numbers of executions.

3.4.2.3 Study 2 – Research Question 2

In Study 2, we investigate whether other kinds of (more compact and cheaper to collect) execution data can also be used to reliably estimate execution outcomes. Using statement counts as a starting point, we investigated whether other data might yield similar prediction accuracy, but at a lower runtime cost. Note that because statement counts contained almost perfect predictors, we did not consider richer execution data, such as data values or paths. Instead, we considered three additional kinds of data that require the collection of a smaller amount of information: throw counts, catch counts, and method counts.

Throw Counts and Catch Counts Throw counts measure the number of times each throw statement is executed in a given run. Analogously, catch counts measure the number of times each catch block is executed. Each version of JABA has approximately 850 throw counts and 290 catch counts, but most of them are always zero (i.e., the corresponding throw and catch statements are never exercised). This is a typical situation for exception handling code which is supposed to be invoked only in exceptional and often rare situations.

As with statement counts, we built and evaluated classification models using throw counts as predictors. We found that throw counts are excellent predictors for only one version (v17), with error rates well below 2%, but are very poor predictors for all other versions. Further examination of the fault in v17 provided a straightforward explanation of this result. Fault #17 causes a spurious exception to be thrown almost every time that the fault is executed and causes a failure. Therefore, that specific exception is an almost perfect predictor for this specific kind of failure. Most of the other throw counts refer to exceptions that are used as shortcuts to rollback some operations when JABA analyzes certain specific program constructs. In other words, those exceptions are used to control the flow of execution and are suitably handled by `catch` blocks in the code, so they are typically not an indicator of a failure. Note that throw counts did not perform well for the other versions that always fail with uncaught exceptions (i.e., v2, v9, v14, and v19) because the uncaught exceptions are runtime exceptions—exceptions that are not explicitly thrown in the code. Therefore, there is not a throw statement (and count) related to those exceptions.

The results that we obtained using catch count predictors were almost identical to those obtained using throw counts. Overall, it appears that, for the data considered, throw and catch counts do not by themselves provide wide predictive ability for different failures. Although this may be an artifact of the specific subject considered, we believe that the results will generalize to other subjects. Intuitively, we expect throw (and catch) counts to be very good predictors for some specific failures (e.g., in the trivial case of executions that terminate with fatal failures related to explicitly-thrown exceptions). We do not expect them to predict reliably other kinds of (more subtle) failures and to work well in general, which is consistent with what we have found in our study.

Method Counts Method counts measure the number of times each method has been executed in a given run. For each version of JABA considered, there are approximately 3,000 method counts—one for each method in the program. As with statement counts, the random forest algorithm considered, among these 3,000, only the 1,240 non-zero method counts. The models built using method counts performed extremely well for all program versions. Similar to statement counts, method counts led to models with OOB error rates near zero. Interestingly, these results are obtained from models that use only between two and seven method count predictors (for each program version). Therefore, method counts were as good predictors as statement counts, but had the advantage of being much less expensive to collect.

More generally, these results suggest there are several kinds of execution data that may be useful for classifying execution outcomes. In fact, in our preliminary investigations, we also considered several other kinds of data. For example, we considered branch counts and call-edge counts. Branch counts are the number of times each conditional block (i.e., method entries and outcomes of decision statements) is executed. Call-edge counts are the number of times each call edge in the program is executed, where a call edge is an edge between a call statement and the entry to the called method. Both branch counts and call-edge counts were as good predictors as statement or method counts.

Note that the execution data that we considered are not mutually independent. For example, method counts can be computed from call-edge counts; and throw counts are a subset of statement counts. It is also worth noting that we initially considered value-based execution data, and captured data about the values of specific program variables at various program points. However, we later discarded these data from our analysis because the compact, easy to gather count data defined above yielded almost perfect predictors. In particular, because method counts are excellent predictors and fairly inexpensive to collect, we decided to consider only method counts for the rest of our investigation. (There is an additional reason to use method counts, which is related to the statistical validity of the results, as explained in Section 3.4.2.5.)

3.4.2.4 Study 3 – Research Question 3

The results of Study 2 show that our approach could build good predictors consisting of only a small number of method counts (between two and seven). This finding suggests that, at worst, our technique needs to instrument less than 130 of the 3,000 methods (assuming 7 different methods over 19 faulty versions) to perform an accurate classification. We use this result as the starting point for investigating our third research question.

One possible explanation for this result is that only these few counts contain the relevant "failure signal." If this is the case, then choosing exactly the right predictors is crucially important. Another possibility is that the signal for program failures is spread throughout the program, and that multiple counts carry essentially the same information (i.e., they form, in effect, an equivalence class). In this case, many different predictors may work equally well, making it less important to find exactly the right predictors. Moreover, if many different predictor subsets are essentially interchangeable, then lightweight instrumentation techniques are more likely to be widely applicable to other remote analysis and measurement applications.

To investigate this issue, we randomly sampled a small percentage of the method counts and then investigated the predictive power of this small subset. More precisely, we:

1. randomly selected 1% (about 30) and 10% (about 300) of the method counts,
2. built a model based only on these counts, and
3. validated the model as described in Section 3.4.2.1.

We repeated this experiment 100 times, selecting different 1% and 10% subsets of method counts every time. This approach is an effective way to discover if there are many sets of common predictors that are equally significant. Without random sampling, it is easy to be misled into identifying just a few, important predictors, when in fact there are other predictors which have comparable predictive capabilities. Also, random sampling provides a way of assessing how easy (or difficult) it may be to find good predictors. For instance, if 1% subsamples return a good subset of predictors 90% of the time, the number of equally good predictors is very high. On the other hand, if 10% subsets contain good predictors only 5% of the time, we would conclude that good predictors are not as easily obtained.

We found that the randomly selected 1% and 10% of method counts invariably contained a set of excellent predictors over 80% and 90% of the time, respectively. This result suggests that many different predictor subsets are equally capable of predicting passing and failing executions. This result is interesting because most previous research has assumed that one should capture as much data as possible at the user site, possibly winnowing it during later post-processing. Although ours is still a preliminary result, it suggests that large amounts of execution data might be safely ignored, without hurting prediction accuracy and greatly reducing runtime overhead on user resources. (We actually measured the overhead imposed by collecting such small subsets of execution data, and verified that it is almost negligible in most cases.)

3.4.2.5 Possible Threats to the Validity of the Studies

Generality Issues All the results presented so far are related to the prediction of the outcomes within single versions. That is, each model was trained and evaluated on the same version of the subject program. Although a classification approach that works on single versions is useful, an approach that can build models that work across versions is much more powerful and applicable. Intuitively, we can think of a model that works across versions (i.e., a model that can identify failures due to different faults) as a model that, to some extent, encodes the concept of "correct behavior" of the application. Conversely, a model that works on a single version and provides poor results on other versions, is more likely to encode only the "wrong behavior" related to that specific fault.

Since one of our interests is in using the same models across versions, we also studied whether there were predictors that worked consistently well across all versions. We were able to find a common set of 11 excellent predictors for all of the programs versions that we studied. Classification using these predictors resulted in error rates below 7% for all versions of the data. Moreover, the models that achieved these results never included more than 5 of those 11 predictors.

Another threat to our results, in terms of generality, is the fact that we only considered versions with a single fault (like most of the existing literature). Therefore, we performed a preliminary study in which we used our technique on the versions of JABA that contain multiple faults (see Table 3.1). In the study, we selected predictors that worked well for single-error versions and used them for predicting versions with multiple errors. We found that, although there are instances in which these predictors did not perform well and produced error rates around 18%, the predictors worked well most of the time, with error rates below 2%. In Sections 3.5 and 3.6, we further discuss the use of our approach in the presence of multiple errors. In this first set of studies, our focus is mostly on investigating and defining techniques that can successfully classify program executions, rather than techniques specifically designed to recognize failures under different conditions.

Multiplicity Issues When the number of predictors is much larger than the number of data points (test cases, in our case), it is possible to find good predictors purely by chance. When this phenomenon happens, predictors that work well on training data do not have a real relationship to the outcome, and therefore perform poorly on new data. If the predictors are heavily correlated, it becomes even more difficult to decide which predictors are the best and most useful for lightweight instrumentation. Inclusion of too many predictors may also have the effect of obscuring genuinely important relationships between predictors and outcome. This problem is a fundamental one because multiplicity issues can essentially mislead statistical analysis and classification. Unfortunately, this issue has been overlooked by many authors in this area.

Our first step to deal with multiplicity issues was to reduce the number of potential predictors by considering method counts, the execution data with the lowest number of entities. Furthermore, we conducted a simulation study to understand how having too many predictors may result in some predictors that have strong predictive powers purely by chance. The simulation was conducted as follows. We selected a version of the subject and treated the method counts for that version as a matrix (i.e., we arranged the counts column by column, with each row representing the counts for a particular test case). To create a randomly sampled data set, we then fixed the row totals (counts associated with each test case), and randomly permuted the values within each row. In other words, we shuffled the counts among methods, so as to obtain a set of counts that does not relate to any actual execution.

We repeated this process for each row in the data set to produce a new randomly sampled data set. With the data set so created, we then randomly sampled 10% subsets of the column predictors. Our simulations of 100 iterations each showed that a random 10% draw never (for all practical purposes) produced a set of predictors able to classify executions as “pass”/“fail” in a satisfactory manner. We therefore concluded that the probability of obtaining, purely by chance, good predictors from a 1% random subset of the predictors 80% of the time (which is what we observed in our study on the real data) is very slim. We can thus conclude that the results we obtained are due to an actual relation between the occurrence of a failure and the value of the method counts.

3.4.3 Summary Discussion

So far, our studies suggest that, for the subject and executions considered, we could:

1. reliably classify program outcomes using execution data,
2. do it using different kinds of execution data, and
3. at least in principle, do it while ignoring a substantial percentage of potential data items.

According to these results, we can use our technique based on random forests and on the collection of method counts to classify remote executions. This technique would be useful as long as we can build our classification models through in-house training under the developer supervision.

3.5 Classification Using Lightweight, In-The-Field Data Collection

In contrast to the previous technique which used data collected in-house, our goal here is to define a technique that can classify executions of deployed programs using models built from data collected in the field. As illustrated in the middle panel of Figure 3.1, the models would be built by collecting execution data in the field while using a lightweight, built-in oracle to label these executions. The oracle could be based on various mechanisms, such as assertion checking, monitoring of error handling code, or crash detection. As mentioned before, this scenario refers to situations where we want to train the models in the field (e.g., because the number of configurations/parameters is huge and we want to focus on the ones actually used by the users) and more accurate oracles are too expensive or impossible (e.g., in the case of human oracles) to attach to a deployed program. In these cases, a classification technique needs to operate on readily-collectible execution data, collected in the field, and use these data to train the models first and classify executions according to the models later on. The initial data collection would typically involve a subset of software instances (e.g., instances used by beta testers), whereas the later classification could be performed on execution data coming from any instance and would not require the use of a built-in oracle.

As a specific instance of this general problem, we again consider classification of remote executions as passing or failing executions. The machine-learning technique that we use to define our approach is, in this case, a learning technique that we have invented called *association trees*. In terms of execution data, we consider just the method entry counts that proved effective when used with our first technique (see Section 3.4).

In the following sections, we first describe our approach based on association tree classifiers. Then, we discuss the empirical studies that we performed to assess and refine the approach.

3.5.1 Association Trees

Our first approach, based on random forests, created very accurate models for the system and test cases studied. It requires, however, that every program instance capture the same, large set of features. When the training-data collection is performed completely in house, and on sufficiently powerful computers, this will not be a problem. In other situations, however, the data collection and transmission overhead could easily become unacceptable. Consider the remote failure modeling scenario of Section 3.2.2, in which developers may wish to capture low-level execution data from fielded instances, hoping that the resulting classification models will allow for precisely locating potential failure causes. Capturing lots of low-level data increases overhead, which will eventually affect system performance in an unacceptable way.

In the random forest studies, we found that from a large (thousands) pool of potential predictors only a very small fraction (less than 1%) were necessary for classification. These results suggest that an alternative approach—assuming one can be found—could eliminate a substantial amount of the data collection overhead, be it measured in data transmission bandwidth, runtime overhead, or code bloat. Such an approach would also save considerable data-analysis costs, which tend to grow polynomially with the number of potential predictors. All of these costs are substantial, especially when training data is captured in the field. The problem with the approach based on random forests is that it must actually collect the data and conduct the analysis before it can determine which predictors are actually useful.

To address this problem, we first changed our instrumentation strategy. Instead of capturing each potential predictor in each execution, we capture only a small subset (e.g., in our later feasibility studies we randomly selected 8%) of predictors in each instance. This sampling drastically reduces the data collection overhead, but creates a dataset in which nearly all of the entries are missing, which greatly reduces the performance of tree-based techniques—traditional tree-based classifiers, like random forests, don't work well when many predictors are missing. For example, one tree-based algorithm applied to this reduced data for one JABA version produced an error rate of 8.5%, compared to an error rate below 1% when applied to the complete data sets. To solve this problem, we developed a new classification technique, called *association trees*, that works well even when different instances capture different predictors.

The training set for the association trees algorithm is, as usual, a set of labeled execution data. The set of execution data consists of one vector of features (i.e., predictors) for each program run, and the associated label indicates the run's outcome. Like for our tree-based technique, in this case we consider only two outcomes: "pass" or "fail". Each data vector has one slot for each potential predictor. If a predictor is collected during a given run (i.e., the corresponding program entity is instrumented), its value is recorded in the vector; otherwise a special value (NA) is recorded, indicating that the predictor was not collected for that run. Also as usual, the algorithm's output is a model that predicts the outcome of a run based on the contents of an execution data vector. However, unlike for our tree-based technique, in this approach the models will predict one of {"pass", "fail", "unknown"}, where "unknown" means that the model is unable to make a clear prediction (because of the lack of data).

The association trees algorithm has three stages: (1) transform the predictors into items that are present or absent, (2) find association rules from these items, and (3) construct a classification model based on these association rules.

1. In this first stage, the algorithm transforms the predictors into items that are considered either present or absent, in two steps. *First*, it screens each predictor and checks that the Spearman rank correlation between the predictor and the observed outcomes exceeds a minimum threshold. (Spearman's rank correlation is similar to the traditional Pearson's correlation, except that the actual value of each variable is replaced with a rank: 1 for the largest value, 2 for the second largest value, and so on, which makes the correlation measure less sensitive to extreme values.) The goal of this step is to discard predictors

whose values are not correlated with outcomes and, thus, are unlikely to be relevant for the classification.

Second, the algorithm splits the distribution of each remaining predictor into two parts, such that the split point is the point that minimizes the p-value of the t-test for outcome. Ideally, after the split all runs with one outcome would have values above the split, while all runs with the other outcome would have values below it. If the p-value is below a maximum threshold of .0005, the algorithm creates two items: the first item is present in a given run if the predictor's value is below the split point; the second item is present if the value is above the split point. Neither item is present if the corresponding predictor was not being measured during the run. We also represent outcomes as separate items ("pass" and "fail" for the data used in this chapter).

2. After the algorithm completes Stage 1, the original set of training data has been transformed into a set of observations, one per run, where each observation is the set of items considered present for that run. The goal of the second stage of the algorithm is to determine which groups of frequently occurring items are strongly associated with each outcome. To this end, it applies the well-known apriori data-mining algorithm [8], where it sets outcome as the item to predict. The apriori algorithm is used extensively by the data-mining community to efficiently find items that frequently occur together in a data set (e.g., to discover which items, such as peanut butter and jelly, are frequently purchased together). The algorithm can then determine which of these fre-

quently occurring sets of items are good predictors of the presence of another item of interest (e.g., if someone buys peanut butter and jelly, then they often buy bread as well). In our case we want to know which sets of items are correlated with successful executions or failures. These sets of items, together with their correlations with outcomes are called *association rules.* Association rules are of the form A implies B. We call A the *antecedent*, and B the *consequent* of the rule. Each rule needs a minimum *support*: the antecedent must appear in a certain fraction of all observations for the rule to be considered potentially valid. Each rule also needs a minimum *confidence*: when the antecedent is present in an observation, the consequent must also be present in the observation for a predefined fraction of the observations (the value of this fraction represents the confidence). Typically, we set the confidence level to 1 if we assume outcomes are deterministic. The confidence level can be decreased if we wish to consider non-deterministic outcomes as well. As explained in Section 3.5.2, in our empirical evaluation of the approach, we vary the required supports to experiment with different settings, but typically set the support for rules predicting successful executions to be several times greater than that for predicting failed ones (because failures tend to occur far less often than successes). Finally, to reduce computation times, we choose to limit the length of association rules to three. Therefore if four items must be present to perfectly predict an outcome, our algorithm will not be able to find the corresponding rule (in these studies).

3. The third and last stage of the algorithm performs outcome prediction using the association rules produced in the previous stage. Given a new run, the algorithm finds the rules that apply to it. If all applicable rules give the same outcome, then it returns that outcome as the prediction. If there is disagreement, or there are no applicable rules, the algorithm returns a prediction of "unknown." We chose this unanimous voting scheme to be conservative. One might also decide to use simple majority voting or a weighted voting scheme (if the penalties for incorrectly predicting different outcomes are uneven).

3.5.2 Empirical Evaluation

3.5.2.1 Set-up

In this study, we use the data discussed in Section 3.3. The instrumentation measured the 1,240 method-entry counts greater than zero as possible features. From this complete data set, we created simulated program instances that represented a hypothetical situation in which each instance is instrumented to collect measures for 100 features. To this end, for each simulated instance, we randomly selected 100 features; the remaining ones correspond to features whose instrumentation was not activated and, thus, whose measures were not collected for that instance. We then applied the association tree algorithm to this data under various parameter settings. Note that the goal of these initial tests is simply to determine whether some points in the parameter space yield good classifications models. We leave a more exhaustive analysis of parameter effects and tradeoffs to later investigations.

We list and discuss the parameters considered.

- *Test suite size*: we assigned a random sample of b test cases to each simulated instance. Therefore, each instance executed b test cases, producing 100 unique predictors for each test case run. For each program version, we executed 18,000 total test runs, with test suite sizes of $b = 6, 12$ *or* 24 across 3,000, 1,500, and 750 simulated instances, respectively. One half of the data was used as a training set. The rest was used for cross validation.

- *Support and confidence*: we used a minimum support of 1 for both failures and successes because we consider the outcomes to be deterministic. We set minimum confidences of .0066, .0033, and .0016 for rules predicting success, and .001 and .0005 for rules predicting failure.

- *Correlation thresholds*: In Stage 1 of the algorithm, we discarded predictors from consideration if they did not have a minimum Spearman rank correlation with outcomes of at least 0.4, 0.3, or 0.2.

Performance measures: For every run of the algorithm, we captured several performance measures:

- *Coverage*: The percentage of runs for which the model predicts either "pass" or "fail" (i.e., $1 - percentage\ of\ \text{``}unknown''$).

- *Overall misclassification*: The percentage of runs whose outcome was incorrectly predicted.

- *False positives*: The percentage of runs predicted to be "pass" that instead failed.
- *False negatives*: The percentage of runs predicted to be "fail" that instead passed.

3.5.2.2 Results

We constructed about 90 different association tree models across the single- and multiple-fault versions of JABA used in the previous study. Figure 3.4 depicts the various performance metrics. For each metric, the figure shows the aggregated results for all versions (All Data), the results for single-fault versions only ($1-Fault$), and the results for multiple-fault versions only ($N - Fault$). Across all versions and settings, we found coverages ranging from 2% to 95% with a median of 63%. Coverage was substantially higher for single-fault versions (median of 74%) than for multiple-fault versions (median of 54%).

Overall misclassification ranged from 0% to 10%, with a median of 2%. There was little difference between single- and multiple-fault versions.

False positives had a median of 0% and a 75th percentile of 3%. Due to a few outliers, however, the distribution ranged from 0% to 100%. We found that all 7 of these outliers occurred on multiple-fault versions, with correlation thresholds of .3 or .4, and when coverage was less than 20%. Also, in each case, the accompanying false negative percentage was always 0%. In these cases, the high correlation threshold caused many predictors to be discarded. Few rules for passing runs were generated

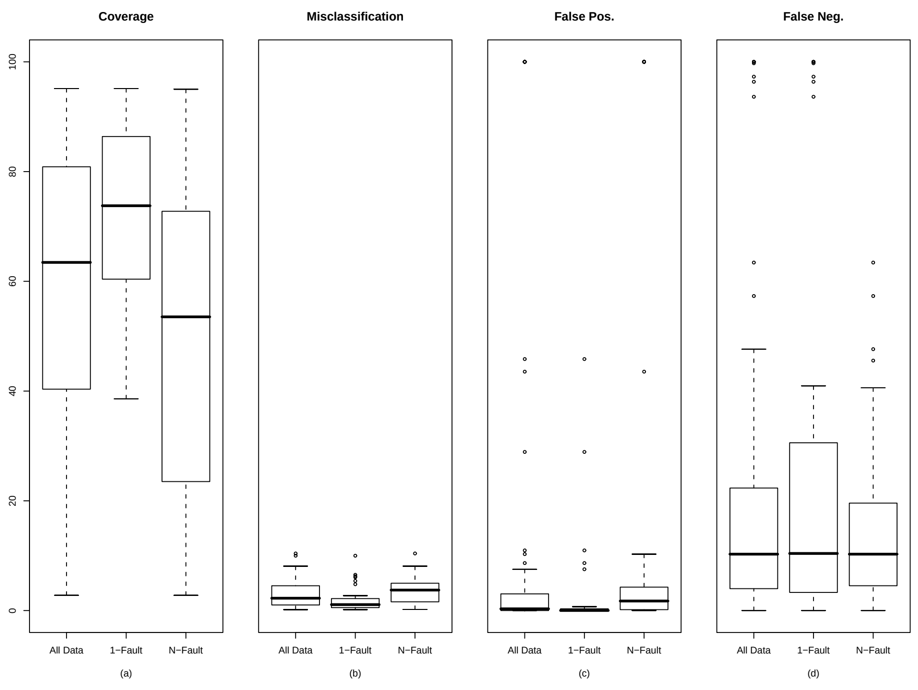

Figure 3.4: Association tree results.

and, thus, all of the passing runs in the test data were predicted to be "unknown" or "fail."

False negatives, like false positives, were generally low. The median false negative percentage was 10%. The 75th percentile was also low, 22%. Again, there were some (3) outliers with 100% false negative percentage and 0% false positive percentages. This time however, the outliers were all for runs of version 11, a single-fault version, and had 70% or more coverage.

Because one of the goals of this evaluation is to refine and tune our approach, we have also analyzed and made some tentative observations concerning the parameters used in building association trees.

Test suite size did not appear to be influential. We found results of various quality with every size. Given that we restricted the length of association rules to three and that we observed between 750 and 3000 instances, with each instance having roughly 1/12th of the possible predictors in it, then we should expect to have good coverage of all potential items by chance.

Increasing support for passing runs had a strong effect on coverage, but much less effect for the failing runs. This situation occurs because, with a lower support for passing runs, we find more rules for predicting "pass" (the much more frequently appearing class) and can thus predict correctly in more cases. The main effect of increasing support for failing runs was to decrease false negatives and to increase false positives.

The coarsest tuning parameter was the minimum correlation needed between the predictors and the outcomes. When it was set too low (0.1), far too many rules were found, which can greatly slow down the algorithm and even make it run out of memory while trying to find rules. Conversely, when the minimum correlation is too high, too few rules may be found. In our experiment, settings of 0.2 or 0.3 tended to give good results, while settings of 0.1 or 0.4 tended to perform much worse.

3.5.3 Summary Discussion

Although still preliminary, our results suggest that our association-tree technique performed almost as well as our tree-based technique defined in Section 3.4, albeit with a few outliers. In terms of data collection costs per instance, however, this new approach is considerably less expensive. Our random-forests technique, like all classification techniques that we know, measures all possible predictors across all instances. (Even techniques that do sampling, such as the one from Liblit and colleagues [54, 58], still collect data for each predictor, therefore missing opportunities for reducing instrumentation and data-transfer costs.) Conversely, our association trees approach instrumented only about 8% of the potential predictors in any given instance. Therefore, this second technique is likely to be applicable in cases where lightweight instrumentation is preferable (or necessary), and where a slightly less accurate classification is acceptable.

This new approach, although successful, has some issues that may limit its applicability in some contexts.

- First, the technique requires all (sampled) data to be collected before modeling begins, but gives no help in determining exactly how many instances and test cases are needed to build adequate models. In our studies, we arbitrarily chose to use a number of instances and executions (test cases, in the study) that goes from 3000 instances running 6 test cases each to 750 instances running 24 test cases each. We currently lack theoretical or heuristic support for deciding how much data to collect.

- Second, the technique gives no help in selecting the modeling parameters. Therefore, developers must rely on trial and error guessing, which can be problematic in some situations. For example, in some cases we created good models with a correlation threshold of 0.1, but in other cases we ran out of memory on a machine with 1 GB of RAM with the same value of that parameter.

- Third, the technique may under-sample useful predictors. This issue is related to the first problem, in that the algorithm does not help in determining the minimum number of runs needed to build good models. If there are very few useful predictors and not enough runs, then it is possible to miss important predictors (or combinations thereof).

- Finally, the technique does not adapt easily to changes in the observed systems and in their usage; it must be rerun from scratch when changes occur. It would be useful and much more cost-effective if the models could be adapted incrementally.

3.6 Classification Using Adaptive Sampling

As in the previous section, our goal here is to define a technique that can classify executions of deployed programs using models built from data collected in the field. The key difference is that here we also want our data collection strategy to be adaptive: information gleaned from early runs will help determine which data to collect during later runs. As we further discuss later, this approach can be used to reduce the process' time to completion and also reduce the total amount of data collected. All other aspects of the overall technique such as oracle characteristics, classification tasks, type of data collected, and underlying assumptions are the same as for the previous technique.

3.6.1 Adaptive Sampling

One limitation of the basic association tree technique described in Section 3.5 is that the algorithm treats all potential predictors as equally important, all of the time. That is, the algorithm selects which predictors to collect in a given instance by taking uniform random samples of all possible predictors. Because of this, if it takes a long time to collect execution data from deployed instances (as it might happen in the scenario about performance modeling of fielded executions in Section 3.2.2) and/or if only a small percentage of the predictors are actually useful for building good classification models, then the basic association tree algorithm may take a considerable time to complete and may fail to capture important association rules.

Suppose, for instance, that for a given system runs fail only when method x and method y are each executed more than 64 times in the same run. Suppose further that there are many possible predictors, a small percentage of them are enabled on each instance, and there are relatively few failing runs. In this case, it is possible that methods x and y are rarely sampled together in the same run. Therefore, the failure cause will not be identified or will at least take a long time to identify. Basically, by giving all predictors the same likelihood of being collected, the algorithm can spread resources too thinly, leading to poor models.

In these situations, finding ways to rule out useless predictors early could greatly improve the algorithm's costs effectiveness. To tackle this problem, we created an incremental association tree algorithm called *adaptive sampling association trees*. This algorithm incrementally learns which predictors have demonstrated predictive power in the past and then preferentially instruments them in future instances, while deemphasizing the other predictors. Important expected benefits of such an approach are that it should allow for:

1. reducing the amount of data collection required,

2. eliminating the need to guess at many of the parameter settings

3. naturally adapting models over time (instead of requiring complete recalibration every time the system, its environment, or its usage patterns change).

To do adaptive sampling, our algorithm first associates a weight with each predictor. Initial weights can be set in many different ways. For example, they can be based on the developers' knowledge of interesting (i.e., problematic) modules or paths. In the experiments described below, we simply used a uniform weighting of 1 for each possible predictor.

When a new instance is ready to run, the algorithm queries a central server for the k predictors to be collected in the instance (which, in turn, define which measurement instrumentation will be enabled in that instance). The server then selects without replacement the k features to be measured from the set of possible features. Unlike the basic association trees algorithm, which gives equal weights to all predictors, the adaptive sampling association trees algorithm sets the selection probability of each predictor to be the predictor's weight divided by the total weight of all predictors. Next, the measurement of the selected features is enabled in the instance so that, when it is executed, the resulting execution data are returned to a central collection site.

At that point, the algorithm tests whether the collected predictors are related to the outcome ("pass" or "fail," in this case). For each predictor measured, the algorithm computes the Spearman rank correlation between the values observed for this predictor and the outcomes of all runs in which the predictor was collected. If the Spearman's rank correlation is above a minimum threshold, the algorithm increases the predictor's weight by one.

If, over the universe of all possible inputs, the Spearman rank correlation of any useful predictor is above our threshold, it is easy to show that, asymptotically in the number of executions, the algorithm will sample it at a high rate. Similarly, if any predictor has a Spearman rank correlation below the threshold, it will eventually drop out of the set of predictors sampled.

Once the algorithm has collected data from a sufficient number of instances, it creates association rules following the same approach as the basic association trees algorithm. The key difference is that, by using a non-uniform sampling strategy, the algorithm is more likely to have heavily sampled the useful predictors, while lightly sampling less useful ones. Consequently, we expect to find more correct rules and fewer incorrect ones at any support level with this new algorithm.

3.6.2 Empirical Evaluation

3.6.2.1 Set-up

We used the same data in this study that we used in the previous studies: method counts as execution features measured, and "pass" and "fail" as the possible behaviors. We then simulated 500 instances of each of the single- and multi-fault JABA versions (see Table 3.1) as follows. For each instance, we randomly selected 100 features (i.e., method counts) to collect and k runs (i.e., test case executions), where k is a random number following a Poisson(12) distribution. Half of the runs were allocated to the training set, and the rest were used for cross validation. As discussed above, we assigned to all features an initial weight of 1. After each run, the algorithm increased by 1 the weights of any features that were strongly correlated with the outcome (i.e., Spearman rank correlation above 0.3 in absolute value).

Overall, our training and our test sets comprised an average of 3,000 test cases, each collecting 100 predictors. Basic association trees, in contrast, executed 9,000 test cases with 100 predictors each. (Note that we actually considered 18,000 test runs for the basic association trees, but used only half of the data as a training set, as discussed in Section 3.5.2). We measured the performance of each adaptive sampling association tree using the same four measures used in the previous study (see Section 3.5.2).

3.6.2.2 Results

Using the adaptive sampling algorithm, we constructed models (i.e., association rules) for each of the JABA versions used in earlier studies. In the few cases where a given model performed poorly (i.e., produced a false positives rate over 20% or coverage below 80%), we allowed ourselves to change support levels for failure rules and/or for success rules.

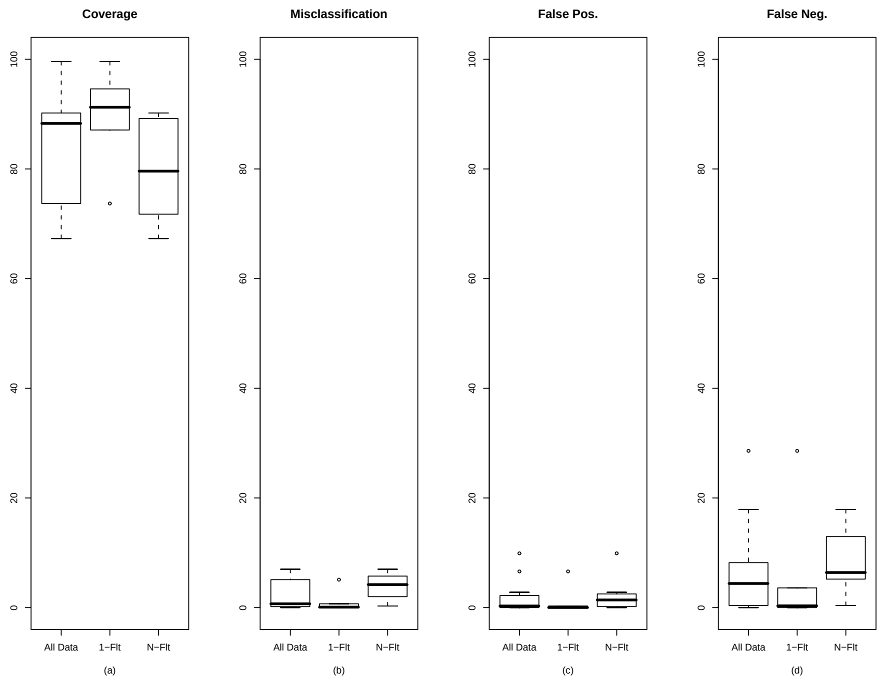

Figure 3.5: Adaptive sampling association tree results

This strategy mimics the tuning process a developer could perform in practice. There was some variability in the number of iterations needed for the results to converge. This variability is present because the time to discover relevant features varies—if it takes a while to discover good features, then, obviously, more iterations of the process are needed before obtaining good results.

Figure 3.5 depicts the various performance metrics for the final models. Across all models, coverage ranged from a minimum of 67.3% to a maximum of 99.6%. The median coverage was 88.3%. As the figure shows, coverage was substantially higher for single-fault versions (median of 91.25% for 1-Flt) than for multiple-fault versions (median of 79.86% for N-Flt).

Overall misclassification ranged from 0% to 7%, with a median of 0.7%. Misclassification was lower for single-fault versions than for multiple-fault versions

(0.09% median versus 4.2% median), although both are quite low in practical terms.

False positive percentage ranged from 0% to about 10% due to a few outliers. The 3^{rd} quartile, for instance, is 2.2% false positives. Also in this case, single-fault versions had fewer false positives than multiple-fault versions (0.02% median versus 1.4% median), but both were low in practical terms.

Finally, false negative percentages range from 0% to 28.6%. The median was 4%, and the 3^{rd} quartile was 8.2%. Single-fault versions had fewer false negatives than multiple-fault versions (0.33% median vs. 6.4% median).

To examine the scalability of this approach, we also applied it to the much more voluminous statement count data (about 12,000 possible features as opposed to about 1,240 possible features in the case of non-zero method counts). We applied the technique to the 6 single-fault JABA versions using less than 450 instances and collecting less than 600 predictors (less than 5% of the total) per run. The results are shown in Table 3.2. As the table shows, coverage is over 90% on average, and false positives, false negatives, and overall misclassification are close to zero in every case. These results suggest that adaptive sampling may scale nicely and, thus, allow for classification of the larger programs and much larger data sets that we would expect to see in practice.

3.6.3 Summary Discussion

Our results suggest that the adaptive sampling association trees approach can perform almost as well as the random forest approach used in Section 3.4 and as

Version	Coverage	False.pos	False.neg	Misclass
v4	0.87	0.02	0.01	0.01
v5	0.97	0.00	0.00	0.00
v7	0.98	0.00	0.00	0.00
v9	0.89	0.00	0.00	0.00
v11	0.79	0.00	0.00	0.00
v17	0.98	0.00	0.00	0.00

Table 3.2: Adaptive sampling association trees built with statement counts

well or better than the basic association tree approach from Section 3.5.

Figure 3.6 depicts the results obtained for the basic association trees side by side with those obtained for the adaptive sampling association trees. To perform this comparison as fairly as possible, we selected the "best" basic association trees results for each version. This is necessary because our basic association trees study included models built with non-optimal parameter settings. Note, however, that since we have four different performance measures, the definition of best is necessarily subjective. In this case, we have preferred models with higher coverage and lower false negatives because software developers are often more interested in identifying failing runs than passing runs. As shown in the figure, by sampling adaptively we generally achieved higher coverage and lower misclassification rates, with significantly less variability between versions.

In addition, in terms of data collection costs, our new approach is significantly cheaper than the previous two approaches. Random forests instrumented all possible 1,240 predictors over 707 test case runs. Association trees instrumented only 100 predictors, but used 9,000 runs. Our final approach instead instrumented only 100 predictors over 3,000 runs. Moreover, since the adaptive sampling approach

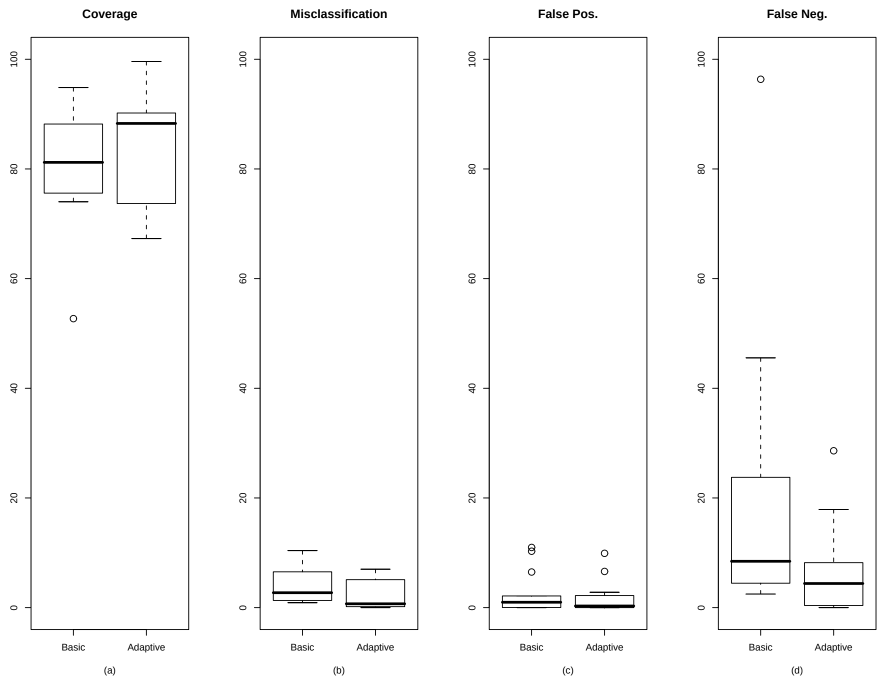

Figure 3.6: Basic association trees vs. adaptive sampling association trees.

often converged well before processing the execution data from all scheduled runs, this result is an upper bound on the number of runs needed by the algorithm. One important implication of these savings is that we were able to generate all desired models, whereas the basic association trees algorithm occasionally ran out of memory. Finally, adaptive sampling allowed us to scale up to more voluminous, lower-level data (statement counts instead of method counts) easily and successfully.

Although quite effective in our tests, adaptive sampling has some limitations as well. The key limitation we found is that the approach necessarily introduces a sequential dependence among instrumented instances. That is, in its simplest implementation, adaptive sampling would not select the predictors for instance i and deploy it until (1) instance $i - 1$ had returned its data and (2) predictor weights had been are updated (we call this dependence *lag-1 dependence*). There are many workarounds to this issue. For example, we could instrument instances in batches or loosen the dependences by considering lag-k rather than lag-1 approaches. Nevertheless, some sequential ordering must remain (or the technique simply degenerates to basic association trees).

Such dependences might render adaptive sampling unacceptable in certain situations, which include cases where many instances are deployed at the same time and it is impossible or undesirable to update them in the field, and cases where observation periods are relatively short, but many runs must be instrumented. In the first case, adaptive sampling is impossible. In the second case, it might result in unacceptable slow-downs of the deployed instances. In both cases, thus, developers might prefer using the uniform sampling offered by basic association trees.

3.7 Overhead Weighted Adaptive Sampling

The previous section of this chapter present and evaluate an improved implementation of adaptive sampling association trees. While this approach builds accurate classification models, requiring less training data than other existing techniques; one problem is that while on average this approach reduces runtime data collection overhead, it may still be very expensive. This is especially true when overheads are non-uniform. For instance, a counter on a frequently executed path will create much more overhead than a similar counter on the first line of the main program.

This section presents an empirical evaluation of an improvement to the previous approach. This improvement generalizes the criteria for adjusting sampling weights from benefit only (rewarding predictive power) to cost and benefit (rewarding predictive power and low observed runtime overhead). In other words our new adaptation goal is to maximize information content, while minimizing runtime overhead, given a fixed number of measurement instruments.

First, the new approach is compared to the performance of the previously presented adaptive sampling association trees. Then, we examine in detail how the new approach affects instruments sampling frequencies and analyze how this affects the runtime overhead of the modified approach.

3.7.1 Weighted Adaptive Sampling

The studies discussed in Section 3.6.1 show that adaptive sampling association trees create effective classification models, while requiring substantially less data than other existing techniques. Despite these successes, however, it has some limitations as well. In particular, since the algorithm solely considers the correlation of a predictor to the outcomes (a benefit-only measure), it can heavily sample predictors that incur high run-time overheads. In addition, even among "good" predictors, some predictors are better than others. To improve this situation, we have developed an alternative weighting criteria that factors in both the strength of a predictor effectiveness and its observed cost history.

3.7.2 The Cost-Benefit Function

When updating a predictor's weight we first determine whether its values are strongly correlated with execution outcome. As in Section 3.6.1, if the absolute value of the Spearman rank correlation is less than 0.3, the predictor is considered to be poor and its weight remains unchanged.

Otherwise, the weight is incremented by the ratio of benefit to cost. Our cost function is the proportion of the total number of measurements made by each instrument to the total number made by all instruments. (I.e. the frequency of this predictor relative to all measurements instrumented in the test instance.) The goal is to compute the average relative overhead incurred whenever this predictor is enabled.

We used a function of the correlation to measure the benefit of each predictor. We wanted a benefit function with a large range, easily distinguish amongst "good" predictors. However, the absolute value of correlation is in $[0, 1]$. We use a modification of the logistic function to map from the unit interval to the positive real line. The logistic function is

$$\text{logit}(x) = \log(\frac{x}{1-x})$$

and maps $[0, 1]$ to the positive real line. If the absolute value of the correlation is r, we use

$$x = (r + .99)/2$$

and then use the logistic function as above. Since r is guaranteed to be greater than .3 to be qualified as "good," this transform will result in a benefit on the interval

$[.597, 5.29]$. If we used 1 instead of .99, the transform would be unbounded, allowing for infinite weights, and the resultant computational issues.

Every time we identify a "good" predictor, we increase the weight by the benefit divided by the cost. This gives the potential for very large weights since, with over 1,000 predictors, the average cost of a predictor is less than 1/1000. In contrast to our previous system of trying uniform weights, this system will preferentially select the better of the "good" predictors, and the "good" predictors which give the least overhead in collecting the data. Also, the chosen predictors are stable; once we find a "good" predictor, we are very likely to keep measuring it in future iterations.

3.7.3 Empirical Evaluation

3.7.3.1 Set-up

We once again used the same data used in previous sections this time focussing on the much more voluminous statement count features to examine the scalability issues with our overhead weighted approach. As in Section 3.5.2, the JABA data set was using, with about 12,000 possible predictors (as opposed to about 1,240 possible features using method counts). We applied the technique to the 6 single-fault JABA versions using no more than 450 instances and k runs, where, again, k is a random number following a Poisson(12) distribution. For each instance we randomly selected no more than 600 predictors (less than 5% of the total). Again half the data is assigned to the training set and half to the test set. Outcomes were again limited to the same two "pass" and "fail" possible outcomes. As discussed above, after each run, the overhead-weighted approach increases the weights of strongly correlated features (i.e., Spearman rank correlation above 0.3 in absolute value) using the cost-benefit function defined in Section 3.7.2. Overall, our training and our test sets each comprised an average of roughly 3,000 test cases, each collecting no more than 600 statement count features for each program instance.

Performance measures: For every run of the algorithm, we captured the same performance measures as before:

- *Coverage*: The percentage of runs for which the model predicts either "pass" or "fail" (i.e., $1 - percentage\ of\ "unknown")$.
- *Overall misclassification*: The percentage of runs whose outcome was incorrectly predicted.
- *False positives*: The percentage of runs predicted to be "pass" that instead failed.
- *False negatives*: The percentage of runs predicted to be "fail" that instead passed.

3.7.3.2 Results

We constructed models (i.e. association rules) using our new overhead weighted adaptive sampling algorithm for each JABA version used in the previous sections. Again if a model performed poorly, we allowed ourselves to tune the support levels– mimicking common real-world development practice. Figure 3.7 depicts the various

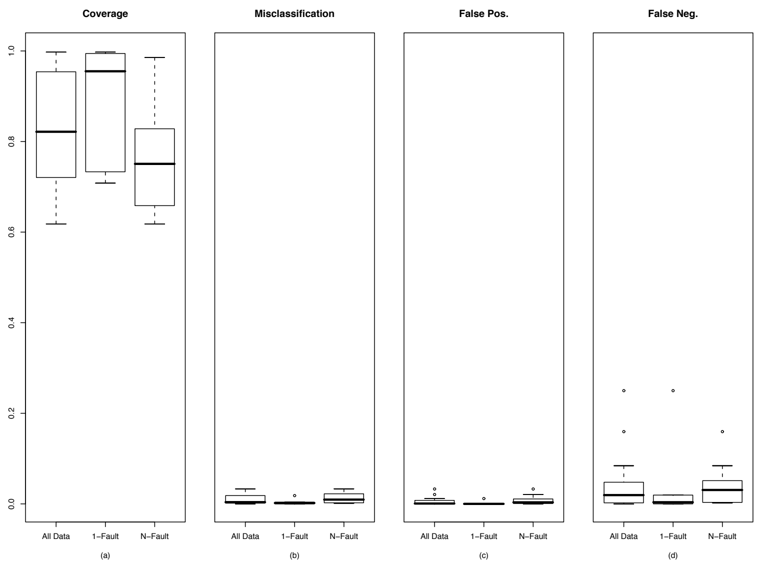

Figure 3.7: Weighted utility adaptive sampling association tree results.

performance metrics for these models. Across all data, coverage ranged from a minimum of 62% to a maximum of 99.7%. The median coverage was 83%. As the figure shows, coverage was substantially higher for single-fault versions (median of 96% for 1-Fault) than for multiple-fault versions (median of 75% for N-Fault).

Examining the results in detail, overall misclassification ranged from 0% to 3%, with a median of 0.3%. Misclassification was acceptably low for single-fault versions and for multiple-fault versions (0.1% median versus 0.9% median). False positive percentages range from 0% to 3%. The median was 0%. Single-fault versions again did better, but both single- and multi-fault instances performed well (0% and 0.8% median values, respectively). Looking at false negative percentages, they ranged from 0% to about 25% due to two outliers. The 3^{rd} quartile, for instance, is 4% for false negatives. Also in this case, single-fault versions performed measurably better than multi-fault versions, having fewer false negatives (0.3% median versus

3% median for the multi-fault versions). But, both measures were low in practical terms.

Finally, we compare the new overhead weighted adaptive sampling approach to the previous adaptive sampling approach. Referring to Table 3.2 for a summary of the statement count results for basic adaptive sampling association trees – we saw that overhead weighted approach made fewer predictions (median coverage was 91% vs 83%). However, the overhead weighting shows comparable prediction quality as the previous approach. We see that the new approach has similar overall misclassification rates (0.3% vs 0.2%). Overhead weighting produces slightly lower false positive percentage (0.0% vs 0.3%), but higher false negative rates (0.7% vs 0.1%). Overall these results suggest that in terms of performance overhead weighted adaptive sampling was competitive with basic adaptive sampling.

3.7.3.3 Weighting Schemes Analysis

So far we have seen that the accuracy of models built using overhead weighted adaptive sampling is very close to those built with our initial adaptive sampling approach. Next we examine whether our original goal for moving to the new weighting scheme – to reduce runtime overhead – has been met.

Our first analysis of this question was to examine the 100 highest weight predictors under the old and new weighting strategies. We then went back to the raw data and computed the average number of times each of the high-weight predictors was executed in runs in which it was enabled. This computation gives a picture of the average overhead incurred by each predictor. We find that the old adaptive sampling approach incurs substantially higher overhead than does our new, overhead weighted approach; a median of 16626 executions versus just 268 executions with the new weighting algorithm.

Our second analysis focused on versions specific versions of JABA (namely, 4, 17, and 28) to see how the weights selected by our overhead weighted adaptive sampling technique compared to other measures of the variable *importance*. In particular, the random forests [16] learning technique returns an importance measure of each predictor which helps to quantify how much the quality of the prediction would be degraded if each predictor were removed. We compared our weights to the random forest importance measure using Spearman's Rank Correlation (while the scales of the two systems aren't directly comparable, the rankings of the predictors are). Looking at the set of all predictors, we found a correlation of 0.15, which indicates a weak relationship. We then looked at the correlation of predictors which were declared "good" by one method or the other – where "good" is defined to be a weight above 1, an random forests importance above 0, or a random forests importance above 2. In all three cases, the relationship was weakest when looking at the set of predictors with weights above 1, where the Spearman Rank Correlation is negative for versions 4 and 17 (-0.13 and -0.31), and 0.05 for version 28. The correlation was larger on the set of predictors with importance above 0 (0.07, 0.01, and 0.15 respectively), and largest on the set of predictors with importance above 2 (0.14,0.26, and 0.30). Version 28 is notable for having nearly all its predictors having importance above 0, as opposed to the other versions which had only a third of the predictors qualify.

We illustrate this in figure 3.8; the figure plots weights from version 17, where the dark circles represent the points that have random forest importance measures above 2. Note that while the two methods agree on the best predictors, most of the other highly weighted predictors have importance measures between 0 and 2.

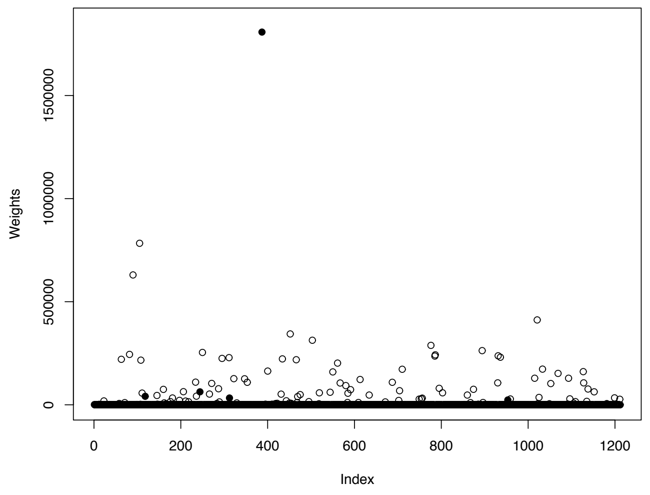

Figure 3.8: Overhead Weighted Sampling Weights for Version 17. Open circles have Random Forest Importance measures below 2, filled circles have Importance Measure greater than 2

3.7.4 Summary Discussion

Our results suggest that the overhead weighted adaptive sampling approach produces reasonably good results (in comparison to adaptive sampling association trees), but somewhat sacrifices coverage. But the initial results don't tell the whole story; empirical analysis of the weighting scheme shows that our utility function lowered costs by over 98%, substantially decreasing the processing and transmission overhead of the instrumentation data. Our analysis finds that overhead weighted adaptive sampling takes advantage of the redundancy in the data to discover the "good" predictors with low cost. Given that we have previously shown that there are multiple sets of predictors which can be used to good effect; it preferentially selects cheaper predictors than those selected if guided by standard techniques like random forests or even pure correlation as in the basic adaptive sampling association trees.

In particular, this new approach addresses one of the limitations of adaptive sampling association trees; namely, that the resulting instrumentation would lead to unacceptable slow-downs in fielded instances. With only 2% of the runtime overhead of the our previous approach, this is distinctly less likely to be a problem with the new weighting algorithm.

Although overhead weighted adaptive sampling association trees are more efficient than our previous methodology, one of the same limitations presented in Section 3.6.3 remains a concern. The approach is necessarily dependent on results from previous instrumented test instances (lag-1 dependence). While the same mitigation techniques could be used (batch instrumentation or using historical instrumentation data), in circumstances where the incremental process of adaptive association trees are unacceptable, overhead weighting may not be appropriate either.

3.8 Related Work

Several software engineering researchers have studied techniques for modeling and predicting program execution behaviors. Researchers in other fields have also studied one or more of the general techniques underlying this problem. In this section we discuss some recent work in these areas.

Classifying Program Executions Podgurski and colleagues [29, 30, 39, 52, 72] present a set of techniques for clustering program executions. The goal of their work is to support automated fault detection and failure classification. For example, in one effort they use cluster analysis to group execution profiles taken from fielded programs. They show that the resulting clusters help to partition execution profiles stemming from different failures. This work, like ours, considers different execution data and selects the ones with most predictive power, but it has two main limitations when compared with our approach. First, their model construction requires the collection of all predictors for all executions. Second, their work assumes that a program's failure status is either obvious (e.g., a crash) or is provided by an external source (e.g., the user).

Bowring, Rehg, and Harrold [13] classify program executions using a technique based on Markov models. Their models consider only one specific feature of program executions: program branches. Our work considers a large set of features and assesses their usefulness in predicting program behaviors. Also, the models used by Bowring and colleagues require complete branch-profiling information, whereas we found that our approaches can generally perform well with only minimal information.

Brun and Ernst [17] use two machine learning approaches to identify program execution behaviors that are likely to be present during faulty program runs. This approach first generates the program abstractions based on test runs and then uses machine learning techniques to find those abstractions that are predictive of program failures. This approach is related to ours because it also uses machine-learning techniques to classify some aspects of program executions. However, their approach does not consider the implications of acquiring the necessary execution data from fielded programs.

Execution Profiling Our research draws heavily on the methods and techniques of run-time performance measurement, particularly issues related to program instrumentation. One simple way to instrument a program is to place probes (instrumentation code) at all important program points (e.g., entrances to basic blocks). As the programs runs, instrumentation code executes each time control reaches them. Executing the probes causes data to be calculated and recorded. Of course, this simple approach can generate enormous amounts of data. For instance, Hollingsworth [47] presents an example from a distributed application that generated over 2Mb of data per second per node! Moreover, such instrumentation can substantially perturb the very performance developers are trying to observe. Consequently, researchers have explored techniques to lower instrumentation overhead.

Arnold and Ryder [11] present an approach for reducing the cost of instrumented code using sampling. The approach is based on (1) having two versions of the code, one instrumented and one non-instrumented, and (2) switching between the two versions of the code based on sample frequency that can be dynamically modified. They investigate the tradeoff between overhead and accuracy of the approach and show that it is possible to collect accurate profiling information with a low overhead. This approach is related to ours, in that it also tries to infer information about the program behavior (profiles, in this case) through partial instrumentation.

Traub et al. [79] present an approach in which they use few probes and collect data infrequently. With this approach, called *ephemeral instrumentation*, probes and groups of probes that work together are statically inserted into the host program. The probes cycle between enabled and disabled states. Traub et al. use this approach to capture branch biases, while adding only 1%–5% overhead to the program and while computing estimated results that are close to the actual ones.

Miller, Hollingsworth and colleagues [63] have developed a dynamic run-time instrumentation system called ParaDyn and an associated API called DynInst. This system allows developers to dynamically change the location of probes and their functionality at run-time. This general mechanism can be used to implement strategies like ephemeral instrumentation, and allows probes to change functionality on the fly.

Anderson et al. [9] present work in which they use no software probes and collect data infrequently. While the program is running, they randomly interrupt it and capture the value of the program counter (or other available hardware registers). Using this information they statistically estimate the percentage of time each instruction is executed. Assuming they run the program long enough, they claim that they can generate a reasonably accurate model with overheads of less than 5%. This approach lets them reduce overheads by putting no probes directly into the host program. The approach simulates the effect of changing the probe locations, whereas sampling lets them limit the amount of data they capture. A chief disadvantage of this approach is that it is very limited in the kind of data it can gather.

There are numerous other approaches to instrumentation; among the most prominent are techniques such as ATOM [77], EEL [50] ETCH [75], and Mahler [80].

Machine Learning and Statistical Analysis Our work also relies heavily on techniques for learning models that distinguish binary outcomes, such as passing runs from failing runs. One particularly important issue is dealing with data sets in which failure is rare. For example, if a system's failure rate is 0.1%, then a data set of 1M runs would be expected to contain only 100 failing runs. In these situations, it can be hard to classify the rare outcome.

Several general strategies have been proposed to deal with this problem. One approach, called boosting, involves multiphase classification techniques that place special emphasis on classifying the rare outcome accurately. For example, Joshi and colleagues [48] developed a two-phase approach in which they first learn a good set of overall classification rules, and then take a second pass through the data to learn rules that reduce misclassifications of the rare outcomes. Similarly, Fan and colleagues [35] developed a family of multiphase classification techniques, called AdaCost and AdaBoost, in which each instance to be classified has its own misclassification penalty. Over time the penalties for wrongly-classified training instances are increased, while those of correctly-predicted instances are decreased.

An alternative approach is to build a probabilistic model of the frequently-occurring outcome and then use an anomaly detection approach to identify instances that deviate significantly from normal as producing the rare outcome. This approach, called anomaly detection, is frequently used in problem areas such as computer and network security [51] and autonomic computing [40].

Partial Data Collection Our research is also related to approaches that aim to infer properties of executions by collecting partial data, either through sampling or by collecting data at different granularities.

Liblit and colleagues [54, 58] use statistical techniques to sample execution data collected from real users and use the collected data to perform fault localization. Although related to our work, their approach is mostly targeted to supporting debugging and has, thus, a narrower focus. Furthermore, although their data collection approach is time efficient because it aggressively samples the execution data, they still may incur space-related issues: they must add instrumentation to measure all predictors (predicates, in their case) in all instances, which may lead to code bloat, and collect one item per predictor for each execution, which may result in a considerable amount of data being collected from each deployed instance. Our techniques based on association trees do not have these problems because they can build reliable models while collecting only a small fraction of execution data from each instance. Finally, like Podgurski's work described above, Liblit and colleagues require failure status to be provided by an external source. In fact, their approach could probably leverage the classification techniques presented in this chapter.

Pavlopoulou and Young [70] developed an approach, called residual testing, for collecting program coverage information in program instances deployed in the field. This approach restricts the placement to instrumentation in fielded instances to locations not covered during in-house testing. Although related to ours because it also partially instruments deployed software, this approach has very different goals and uses different techniques. In particular, the approach does not perform any kind of classification of program outcomes.

Elbaum and Diep [34] perform an extensive analysis of different profiling techniques for deployed software, including the one from Pavlopoulou and Young discussed above. To this end, they collect a number of execution data from deployed instances of a subject program and assess how such data can help quality-assurance activities. In particular, they investigate how the granularity and completeness of the data affect the effectiveness of the techniques that rely on such data. This work provides information that we could leverage within our work (e.g., the cost of collecting some kinds of data and the usefulness of such data for specific tasks) and, at the same time, could benefit from the results of our work (e.g., to use predicted outcomes as triggers for the data collection).

We conclude this section by noting that several methodological and statistical issues that we addressed in this chapter have not been adequately covered in the existing software engineering literature so far. In particular, we describe some useful heuristics for separating genuine relationships between predictors and outcomes from spurious predictor-response relationships. Considering this issue is particularly important when the classification techniques are used to support software-engineering tasks: for these tasks, it is easy to define vast numbers of predictors, but it is typically quite costly to actually collect a sufficiently-large number of data points.

3.9 Conclusion and Future Work

In this chapter, we have presented and studied four techniques—random forests, association trees, adaptive sampling association trees, and overhead weighted adaptive association trees–whose goals are to automatically classify program execution data as coming from program executions with specific outcomes. More specifically, in this investigation we focused on a binary outcome: passing versus failing. These techniques can support various analyses of deployed software that would be otherwise impractical or impossible. For instance, they can allow developers to gather detailed information about a wide variety of meaningful program behaviors. They can also allow programs to trigger targeted measurements only when specific failures are likely to occur.

The empirical evaluation and investigation of our techniques suffers, like all empirical studies, from various threats to validity. The main threats are that we studied only a single system, considered only two possible outcomes, exercised the system with a limited set of test cases, and focused only on versions with failure rates higher than 8%. Nevertheless, the system is sufficiently large and complex to have non-obvious behaviors, the faults are real (in that they originally occurred during the system's development), and the test suite includes tests (obtained from users) that represent real usages of the system. Therefore, we can draw some initial conclusions about the effectiveness and applicability of our techniques.

In our initial studies, we examined three fundamental questions about classification techniques. Our results showed that, for the cases considered, we could reliably classify binary program outcomes using various kinds of execution data (e.g., statement counts). We were able to build accurate models for single system versions with one fault, single system versions with multiple faults, and for multiple system versions. We also found that models based on method counts were as accurate as those built from statement counts. Finally, we conducted several further analyses that suggested that (a) our results are unlikely to have occurred by chance and (b) many predictors were correlated (i.e., many predictors were irrelevant for predicting). An important implication of this last finding is that the signal for failure seemed to be spread through the program rather than associated with a single predictor or small set thereof.

In general, all four techniques performed successfully with overall misclassification rates typically below 2% for all program versions (except for some basic association-trees models built with poorly-performing parameter settings). The key differences between the techniques lie in (1) how much data must be collected and (2) whether the training phase is conducted in a batch or sequential fashion. These differences make each technique more or less applicable in different scenarios and contexts.

For example, the random-forests technique requires for the training to be performed in-house, by using an accurate oracle and capturing all predictors in every program instance. Therefore, it may not be applicable in cases in which the training must be performed in the field (e.g., because different configurations must be considered) but the overhead imposed by the oracle and/or the data collection cannot be tolerated on fielded instances. However, when overhead is not an issue and an oracle is available, such as when testing typical systems in-house, this technique may be preferable to the other three.

The basic association trees algorithm randomly and uniformly instruments a small percentage of all potential predictors (8% in our study), but still allows reliable predictive models to be built. Our evaluation suggests that this technique can be competitive with random forests, while greatly lessening the instrumentation costs suffered by each instance. The tradeoff is that we may need to observe more instances than we would when training completely in-house in order to guarantee coverage of the various predictors (and their combinations). Thus, basic association trees may be appropriate when overhead is a concern, many program instances can be instrumented, and some imprecision in the predictions is tolerable.

Although successful, our basic association trees algorithm had several limitations, such as the fact that it requires substantial tuning, requires all data to be collected before starting to build the models, and needs to be rerun from scratch when the underlying system or its usage change. Our third technique, called adaptive-sampling association trees, addresses these issues. Our empirical evaluation of this third technique strongly suggests that this third technique can also be almost as accurate as random forests, while improving on the basic association trees algorithm and being much cheaper than both. This approach is especially suitable for scenarios in which data collection overhead is a concern, the number of available instances is limited, and/or per-instance data collection is expensive or time-consuming. The limitation of this technique is that it introduces sequential dependences among instances in terms of their instrumentation and execution, which might be problematic in some situations.

Finally, we investigated a revised weighting scheme for use with adaptive sampling association trees to reduce the runtime burden imposed by our previous techniques. By taking into account the runtime overhead imposed by the measurement and collection of each predictor. By favoring good predictors with low cost we developed highly accurate classification models that required just 2% of the runtime overhead imposed by adaptive sampling association trees. Unfortunately, the same sequential dependency limitation of adaptive sampling still applies to overhead weighting adaptive sampling association trees.

In future work, we will continue our investigations in several directions. First, we will study our techniques in increasingly more realistic situations and on increasingly larger systems. Specifically, we will apply these techniques to modeling faults in several large, open source systems. We will also extend our investigation beyond binary classifications to assess whether our techniques can capture behaviors other than passing and failing (e.g., performance).

Second, we will investigate the application of our technique to continuous data streams, rather than just static data collected at the end of program runs. We expect that analyzing such time series data will better allow us to do proactive failure detection (i.e., online prediction of impending failure). Similarly, such an approach should allow for detecting more dynamic phenomena, such as intrusions and system overloads.

Finally, we intend to explore the analysis of multiple, simultaneous data streams, as opposed to a single data stream as we do now. In particular, we want to collect distinguished data streams from multiple components in component-based or service-oriented systems. Our goal would be not only to detect problems, but also to localize them to specific components.

Chapter 4

Test Scheduling and Failure Characterization with Incremental Covering Arrays

4.1 Overview

Today's software systems are increasingly built from flexible combinations of components that can be configured in a multitude of different ways. While a high degree of configurability has many benefits, it can also add combinatorial complexity to the already difficult testing problem as, in the worst case, the *full* configuration space of a system is the exponentially large cross-product of all possible option settings. Consider, for example, a typical web server; these systems can contain a number of compile- and run-time options for configuring OS dependent code, the maximum number of concurrent connections, which commands are enabled, and so on. Each specific configuration is potentially a unique system, with its own unique features, flaws, performance profiles and constraints [73].

Since any one of these configurations might be used in the field ,testing processes for configurable systems increasingly focus on discovering subtle *interaction* failures – failures due to specific combinations of option settings. However, given the large numbers of configurations, exhaustive testing is infeasible. In practice software developers usually limit their testing efforts to a relatively small portion of the configuration space. Historically, this has been done by testing just a small set of default or popular configurations, randomly choosing some subset of configurations for testing, or choosing test configurations based on developer intuition and experience. While all of these methods can find flaws in a software system; they have drawbacks. First, they cover a limited, perhaps biased, sub-space of configurations. Furthermore, unless the test configurations are updated over time, testing will continue in already debugged configurations while leaving large sections of the configuration space untested.

To improve on these methods, researchers have studied configuration sampling strategies based on computing mathematical objects called covering arrays [24, 28, 49, 73]. This approach generates a test schedule that satisfies specific coverage metrics, particularly that of testing all t-way combinations of the configuration options.

Yilmaz et al. [84] expanded this general approach to support fault classification of large-scale configurable systems. In that work they used covering arrays to generate test schedules that were then executed in parallel across a grid of computers. The results were returned to central servers, and observed failures were automatically classified to help developers find their underlying causes. The overall process was managed by the Skoll system [61] – a distributed continuous quality assurance (DCQA) environment that allows for highly parallel execution of QA processes. The results suggested that the covering array test schedules produced better classification models than equivalently-sized random samples and that the process scaled reasonably well to large configuration spaces.

The work described in this chapter began when we decided to apply Yilmaz et al.'s approach within the live software development process of MySQL [64]. MySQL is a widely-used database system with around 150 configuration options. In our work we focused on about 25 of those configuration options, each with 2–5 option settings. Since just this small slice of the configuration space contains over 72,000,000 unique configurations and since compiling and testing just one MySQL configuration took an average of about 3 hours, the covering array-based approach seemed appropriate.

Although, we expected the approach to be straightforward, we quickly discovered at least three critical stumbling blocks once we tried to apply it to the live system. First, covering arrays depend on developer insight to select the key sampling parameters. In order to reliably classify failures that are caused by t configuration options, samples must, at a minimum, test all t-way combinations of these options. This means the tester must know *a priori* what *strength*—value of t—to use. If they set t too large, resources will be wasted or the process may not complete before the next release. On the other hand, setting t too small, may result in poor classification and require the process to be repeated with a higher value of t. Since systems often have multiple failures with different causes, either—or even both—of these situations is virtually guaranteed. Our experience with MySQL confirmed this.

Second, a t-way covering array, by design, computes a sparse sample of a configuration space. These samples have the property that they can *detect* interactions between any t options. They cannot be relied on, however, to definitively *classify* which specific options and settings are actually interacting (because of ambiguities in the sparse data). As a result, classification techniques can sometimes mistake coincidental relationships for actual failure causes. In addition, problems involving more than t options may be incorrectly classified as can non-deterministic failures. Incorrect classifications misdirect debugging attempts wasting developer time and effort. Although not widely discussed in the research literature, a common way to limit these problems is to run multiple different covering arrays, hoping that the differences in the specific configurations tested can disambiguate actual from incidental relationships. Therefore as part of this process we found we sometimes need to choose how many covering arrays to generate at each level of t.

Third, because it is not generally possible to use an arbitrary subset of a $t-$way covering array to reliably classify failures caused by fewer than t options, developers should ideally run the covering array as a unit, waiting until all tests have been run before attempting to classify failures. This leads to several problems. For example, it delays results even for easy to classify failures. In some cases, because inter-release testing time can be unpredictable, the test schedules may not even complete before the next release, potentially wasting all the testing effort.

In summary, we found that covering array approaches can run more tests than necessary, can mis-correlate failures with configuration parameters, can duplicate work, and can delay classification results.

To deal with these limitations we ultimately developed a new approach called *incremental covering array failure classification.* The key feature of our redesigned approach is that it is *incremental.* It begins by building a low strength covering array, testing the indicated configurations, doing automatic failure classification, and providing those results to developers. It then uses incrementally stronger covering arrays as time and testing resources allow. A central tactic of this approach is to lower the overhead of incremental execution by carefully reusing results from earlier test runs. This, in effect, efficiently increases covering array strength as far as time and resources allow. We have applied our incremental approach across a large configuration space ($\sim$ 72M configurations) of the open source database, MySQL [64], and evaluated it across three consecutive releases, each with varying characteristics.

Based on initial work presented in Fouché et al. [37], this chapter extends that work by improving the underlying algorithms, by weakening their operating assumptions, and by deepening our empirical evaluation. Specifically, we improved our algorithms by relaxing several constraints present in the previous work. We have also generalized the algorithms, allowing engineers greater freedom to tailor the algorithms to their systems. This more general algorithm can thus be applied across a wider variety of software configuration models and testing environments. We also extended our empirical evaluation in several ways. First, our previous work used an imprecise approximation for calculating the accuracy of fault classification. In this chapter we built evaluation tools that directly compare classification trees to more accurately evaluate the success of each approach. Additionally, we have improved the classification evaluation metric to better reflect how the data would be used for debugging. Classification trees are now evaluated on the configuration options and settings used instead of on pure lexical matching or tree height. Finally, we've added new and deeper analyses – examining how traditional and incremental covering arrays perform as the testing process unfolds. That is, we evaluate how classification accuracy changes as each new configuration is tested as well as at the

Number	Config			Result
	o1	o2	o3	
1. *	0	0	0	FAIL
2.	0	0	1	FAIL
3.	0	0	2	FAIL
4.	0	1	0	PASS
5. *	0	1	1	PASS
6. *	0	1	2	PASS
7.	1	0	0	PASS
8. *	1	0	1	PASS
9. *	1	0	2	PASS
10.*	1	1	0	PASS
11.	1	1	1	PASS
12.	1	1	2	PASS

Table 4.1: An exhaustive schedule

end of the build lifetime.

The remainder of this chapter is organized as follows: Section 4.2 briefly reviews the mathematical tools and process we used; Section 4.3 provides a field study

that motivates our process. Section 4.4 describes the revised covering array algorithm

Section 4.5 describes our empirical evaluations; Section 4.6 compares our

results to other scheduling policies; and Section 4.7 presents concluding remarks and possible directions for future work.

4.2 Background

In this section we provide some background on our techniques. First, failure classification is explained. Next, we describe our approach for selecting which configurations to test, i.e. covering arrays. Finally, we present, Skoll, the distributed quality assurance environment we use to execute failure classification on a large scale.

4.2.1 Failure classification

Failure classification aims to provide developers with compact and accurate descriptions of the configuration subspaces in which failures occur. This section details both the process and how we evaluate its performance. Table 4.1 depicts the results of exhaustively testing a hypothetical system. This system has three configuration options ($o1$, $o2$, and $o3$), where the first two options are binary (0, 1) and the third is ternary (0,1,2). There are no constraints among the options, so there are 12 valid configurations. Each configuration is tested and the test outcome is indicated as either PASSed or FAILed. To automatically determine the likely cause of the three observed failures, we feed the test outcome data to a machine learning technique called classification tree analysis. Classification tree analysis uses a recursive partitioning approach to build a tree-structured model [83, 15] that correlates a configuration's test result (e.g., passing or failing) with the settings of its options. Each node in the model denotes an option, each edge represents an option setting, and each leaf represents a test outcome or set of outcomes (if there were, for example, different failure types). The classification tree generated for the sample system is shown in Figure 4.1 and correctly indicates that the failures appear whenever $o1 == 0$ and $o2 == 0$.

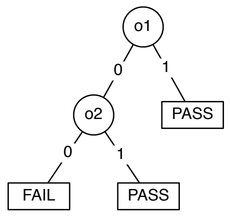

Figure 4.1: An example classification tree

4.2.2 Covering Arrays

While the model in Figure 4.1 explains the failures in the underlying dataset, it does so at the cost of performing exhaustive testing. We could, however, have obtained the same model using only a subset of the exhaustive data. For example, the starred configurations in Table 4.1 make up only one-half of the full configuration space, but generate the same classification tree model. We selected these starred configurations because they constitute a 2-way *covering array* (CA) [24, 25] of the configuration space; i.e., all pair-wise combinations of configuration option settings appear in at least one of the starred configurations. More generally, t-way covering arrays "cover" all possible combinations of option settings between any t options in the configuration space. Importantly, they do this while also limiting the total number of configurations tested. The approach takes k configuration options (or *factors*) each with v_i settings (or *values*), and produces an $N \times k$ array over the values of v_i in the corresponding columns. The resulting covering array $CA(N; t, k, (v_1 v_2 ... v_k))$

has N configurations with the property that every $N \times t$ sub-array contains all ordered subsets of size t from each of the v_i's at least once. The parameter t, the covering array *strength*, corresponds to the number of options whose combinations of settings must be covered. For instance, if we set $t = 2$ we would need to cover all combinations of settings for every pair of options; if we set $t = 3$ we would need to cover all possible combinations of settings for every trio of options. The starred configurations in Table 4.1 can be written as a $CA(6; 2, 3, (2, 2, 3))$ or more compactly as a $CA(6; 2, 2^2 3^1)$, a shorthand notation that drops the variable k– it is implicit in this notation– and that represents the number of times a particular v is repeated as an exponent.

Covering arrays can have an arbitrarily large number of rows (configurations), N, but there are a number of algorithms for limiting the size of covering arrays [20, 21, 23, 24, 25, 43, 46, 65, 78]. We construct covering arrays using algorithms devised by Cohen et al. [27] in this chapter because they support the seeding of test cases (a requirement for our incremental approach) and because they handle inter-option dependencies.

Covering arrays have been used in many domains of software testing including testing system inputs [19, 24, 28, 33], testing configurations [49, 73], testing graphical user interfaces and databases[22, 82, 87], and for web testing[81].

4.2.3 Skoll

Skoll (chapter 2) is a process and infrastructure that simplifies executing QA tasks across a grid of computing resources. It uses a server component to plan and distribute the QA process. Skoll efficiently leverages computing resources by dividing global QA processes into multiple subtasks which can then be distributed to client machines and executed. The results, when returned, are fused together to complete the overall QA process. Skoll maintains a formal model of the QA processes' configuration space that captures configuration options and their settings as well as constraints (refer to [61] for further details).

In this work, we create covering arrays for a configuration model and then use Skoll to distribute and test individual configurations. For a given configuration, a client computer configures and compiles the source code, runs tests, and sends the results back to the Skoll server. We then store the results in a database which are then used to build classification trees that isolate observed failures.

4.3 An Initial Field Study

As described earlier, we implemented the failure classification approach of Yilmaz et al. [84] within the Skoll environment. We then executed the approach as part of a continuous build, integration and test (CBIT) process for the MySQL database project, focusing on a partial configuration space of MySQL, with over 72,000,000 unique configurations. While doing this we uncovered several limitations of this approach, which motivated us to create our incremental covering array approach.

In the remainder of this section, we describe our experiences and highlight each of the limitations we observed.

4.3.1 Our MySQL CBIT Process

MySQL is an open-source, multi-threaded, SQL database management system (DBMS) [64]. Initially released over 10 years ago, it now contains over 2 million lines of code. It has been downloaded over 10 million times and is available for use on over 20 platforms. MySQL has a significant number of test cases (including both installation tests and generic SQL tests), and it enjoys a large developer community that actively updates and tests the system. In short, MySQL represents the type of large-scale, highly-configurable system for which covering arrays have been shown to be an effective technique [84].

Configuration Model. MySQL runs on a myriad of hardware architectures and operating system combinations and allows extensive customization of functionality. This is supported through the use of configuration options (over 120 of them). For example, MySQL can be compiled with support for differing character sets and differing back-end storage engines. In this chapter, we consider only a partial, but still significant, subset of MySQL's configuration options.

Continuous Build, Integration and Test. Our Skoll-based MySQL CBIT process configures, builds and integrates a MySQL instance, and then executes a standard battery of 772 tests against the resulting executables. Multiple configurations are tested in parallel using a grid of computing resources primarily located at the

University of Maryland. Because our approach shares CBIT effort across a large grid, we could test many more system configurations than was possible with MySQL's limited in-house testing resources.

4.3.2 Problems Encountered

Our initial implementation of this process computed a traditional covering array test schedule whenever new code was checked into the main developer repository. With this method, we first determined the desired strength of the covering array test schedule (i.e., t=2,3, etc.) and then ran the schedule generated for this strength. Once the covering array test schedule was finished, we classified the observed failures. Applying this approach to the live MySQL development process, however, we very quickly ran into problems. Specifically, the standard covering array approach required us to make several problematic decisions/assumptions. Each such decision forced us to statically fix something that varied in reality; and each such mismatch implied unnecessary costs and/or reductions in effectiveness. We describe these in detail below.

Problem #1: Variability of Test Time and Resources. The time needed to test one MySQL version in one configuration varies depending on the features and failure rates present in that configuration. Successful test runs on limited feature sets (which skip some unsupported tests) complete in as little as 45 minutes. Other test runs, exercising larger feature sets and possibly failing after long timeout periods, take upwards of 4 hours to execute, with the average being about 3 hours across all configurations. Thus, for a given amount of testing time, it was impossible to know just how many configurations could be tested. This hindered our ability to choose the largest strength t that would fit into a given period of time.

Second, and more importantly, the time available for testing a given version release varies considerably. During a six month period, we saw releases that were current for a few minutes, some for a few days and others for more than a month. In fact, the total time available for testing a release is not known until the *next* source change is committed. As a result, we often selected the wrong value of t and committed to test schedules that were either too large or too small for a particular release period. Some typical interrelease times can be seen in the following snippet taken from MySQL's change log:

```
 ChangeSet@1.2527, 2007-07-02 17:45:46+02:00 ChangeSet@1.2528,
2007-07-02 17:55:24+02:00 ChangeSet@1.2529, 2007-07-02 18:21:52+02:00
ChangeSet@1.2531, 2007-07-03 18:31:31+02:00 ChangeSet@1.2533, 2007-07-05
13:39:12+02:00
```

(Note: build 1.2530 & 1.2532 were never made available for external use, and therefore were excluded from Skoll testing). Here we see three builds submitted in less than 40 minutes, the next one submitted a day later, and the last submitted almost 2 days after that. Consequently, our use of a rigid test schedule was inappropriate in this live development environment.

Third, because some of our resources are volunteered or shared, the number and capacity of test resources could not be known *a priori*. Again our test planning was adversely affected because we never knew how many CPUs would be available for testing and for how long we could use them.

Given these issues, we needed a more flexible approach that would test the system as comprehensively as possible given the unknown testing time.

Problem #2: Variability Caused by Non-Deterministic Failures. In our work with MySQL, we found that a single configuration would sometimes pass and sometimes fail, despite apparently identical test conditions. I.e., some test case failures were nondeterministic. This is problematic for a traditional approach because it only runs a single covering array test schedule at the chosen strength, and thus cannot reliably detect nondeterministic failures.

Note that classification trees can model nondeterministic failures. For instance, one classification rule we found for MySQL, after running the same covering array schedule multiple times, was: Test `main.func_in` always fails if `sql_mode` is `ANSI` or `sql_mode` is `TRADITIONAL`, and passes over 99% of time if `sql_mode` is `NULL` or if `sql_mode` is `STRICT_ALL_TABLES`). Here, we see that the rule does not perfectly explain the observed failures. It does, however, give developers a starting point for further investigation.

Nondeterministic failures can only be detected and analyzed if we make multiple observations of a given configuration (or sub-configuration). Therefore, any new approach needs to accommodate multiple observations of certain configurations or sub-configurations.

Problem #3: Variability of Failure Characteristics. Finally, the characteristics of the underlying failures can change from release to release. For instance, in one random sample involving 14 releases, checked in over about 2-3 weeks, we saw a variety of failures whose root causes involved anywhere from 1 to 5 configuration options. These patterns and their distributions also changed across releases. For example, in one release we observed failures caused by 1 to 4 options, distributed respectively as: 24%, 38%, 34%, and 2% of the total failures. In another release we documented failures caused by 1,2,3 or 5 options, distributed respectively as follows: 5%, 48%, 45% and 2%. Therefore, it was once again impossible for us to choose, *a priori*, the optimal covering array strength for testing.

4.3.3 Problem Analysis

Traditional covering arrays force developers to commit to a particular test schedule without knowing whether it has the right sampling characteristics and if they can finish it before the next version arrives. As mentioned earlier, in Section 4.1, choosing incorrectly can have severe consequences. For example, when a release has simple failures, but large test schedules are used, then testing will do much more work than strictly necessary. If the test schedule, however, is too large to finish before the next version arrives, then failure classification performance may be negatively affected by only having a partial covering array worth of data. Also, if failures are complex, but the test schedule is too small, failure classification will suffer and the process will need to be repeated at a higher strength.

Since the update frequency for specific MySQL versions can vary (sometimes multiple times a day, sometimes every few days, sometimes every few weeks)the

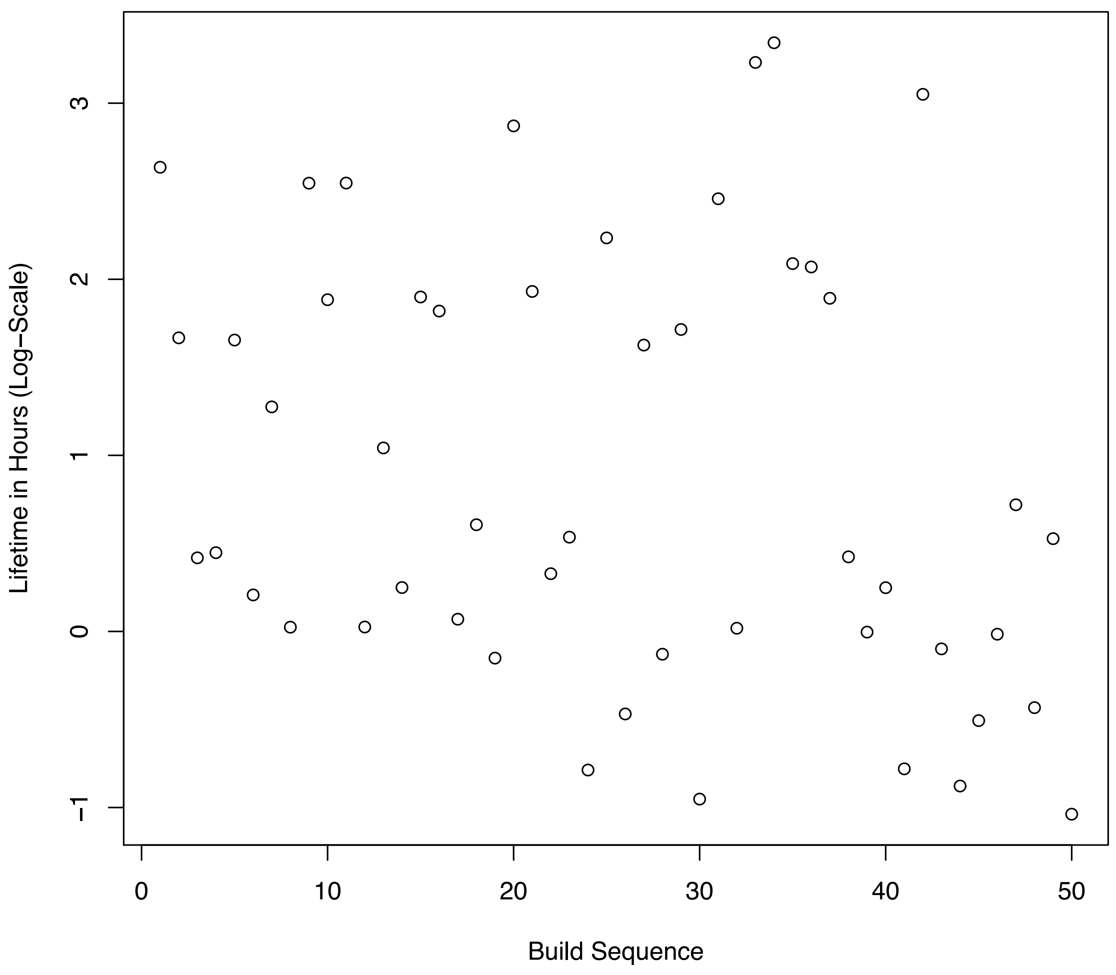

Figure 4.2: Lifetime of 50 releases of MySQL

possible testing time per release varies as well. This variance in release lifetime was a major motivating factor for our approach. To better understand this issue, we examined historical build data from the MySQL database project. Figure 4.2 depicts the lifetime in hours (on a log_{10} scale) for 50 consecutive builds of MySQL 5.1 between March 2007 and January 2008. We can see a range of release lifetimes ranging from less than one hour to a maximum of about 2,000 hours. The differences in release lifetimes have multiple causes. Sometimes geographically-distributed developers happen to check in code at nearly identical times. Other times a check-in is so obviously flawed that it is pulled quickly. Often the mix of development activities vary – sometimes developers are more focused on cleaning up existing bugs, while at other times they are more focused on adding new features (which tend to arrive at a slower pace). It is clear that we cannot always predict beforehand, what time frames are available for testing, or what the complexity of failures will be.

The next analysis we performed examined what would happen if we always chose to run the lowest strength covering array test schedule first. For instance, if we always select $t = 2$, then we can increase our test strength if additional time is left. In this approach, it might seem that we can reuse some of the already tested t-way test schedule. But in fact, if we use the traditional approach and build a new $t + 1$ covering array test schedule there is often little to no overlap between the old and the new test schedules. In fact, for our system we would have had to re-do almost all of the work from the prior test schedules. Most of the algorithms for building covering array test schedules make some random decisions in the array construction, meaning that each time the algorithm is run a different array is generated. Even in algorithms that use determinism, there is no guarantee that a $t + 1$-way array will have any resemblance to a t-way array. Although, one can use certain techniques – called *seeding* – to build upon lower strength arrays (and this will be leveraged in our new approach), this is not the ordinary construction method for covering arrays.

To measure the expected overlap in different strength covering arrays, we generated 10 covering arrays for MySQL at each strength of $2 \leq t \leq 5$. We then compared each of the resulting t-way arrays against all of the $t + 1$-way arrays to evaluate differences between consecutive strength covering arrays. This gave us 100 data points for each increment of t. In the analysis, we computed the number of configurations shared between the t-way and each of the 10 $t+1$-arrays. We found that there were zero configurations in common between any of the 10 t and $t+1$ arrays in this data set. Given that there are over $72,000,000$ possible configurations this is not too surprising. It highlights, however, the fact that choosing too low an initial strength causes redundant work and wasted effort.

Finally, we tried to determine what would happen if we simply chose the highest strength t we were likely to need and simply ran the computed test schedules *as far as possible* towards completion. In this scenario, we would be unlikely to finish many of our test schedules. As a result classification of t-way failures may be impossible because the data set is incomplete. However, since, by definition, for $t > 1$, a t-way covering array is also a $t-1$-way covering array, we might still get reasonable classification performance in practice. To better understand this issue, we examined the 10, 3- and 4-way covering arrays to see how quickly $t-1$-way combinations are covered when running a t-way covering array. We found that we generally needed to run substantially more configurations of the t-way array to achieve all $t-1$-way coverage than we would have needed by just running the $t-1$-way array initially. Although the coverage grows quickly there is a long plateau before full coverage is achieved. For instance with the 3-way arrays we needed to test 58 configurations on the average, to achieve 2-way coverage (versus 22 configurations for an average 2-way covering array). Running the 4-way arrays, we needed an average of 46 configurations to complete 100 percent of the 2-way coverage while we needed an average of 174 configurations to reach 100 percent of the 3-way coverage (versus 80 for an average 3-way covering array). The implication is that if we are unable to complete a large portion of a given test schedule, then we may not be able to classify even the lower strength failures because our information sample will be incomplete.

4.4 Proposed Solution

Based on our findings, we believe that the three central limitations of traditional covering array schedules for failure classification are (1) the lack of guidance for selecting the initial interaction strength (2) the inability to reuse information from prior test runs if the initial interaction level proves too low and (3) that non-deterministic failures can be difficult or impossible to identify and characterize. To address these issues, we have designed and evaluated an incremental process for creating and using covering array test schedules. With this new approach, we begin by testing at the lowest strength (i.e., t=2), and then continue by incrementally testing at successively higher strengths. To limit total costs, at each stage, we construct the next higher-strength covering array so that it incorporates, to the largest extent possible, configurations already run in lower-strength covering arrays. Thus, we need to run fewer completely new configurations in order to cover the current strength. We have designed two variants of this process: one that creates a single covering array at each strength and one that creates multiple covering arrays at each strength (for handling non-deterministic failures). Developers can choose between the variants based on available resources or initial results. This process allows classification as early as possible, and improves testing efficiency. It also allows us to generate test schedules in environments with unknown time and hardware resources.

Our key technical conjectures are that we can construct each covering array using a *seed* taken from already run lower strength arrays such that its size will be approximately that of a traditionally built covering array. A seed is a *fixed* set of configurations that must be included in the covering array. We construct covering arrays by adding new configurations to the seed set until all required t-way interactions appear at least once.

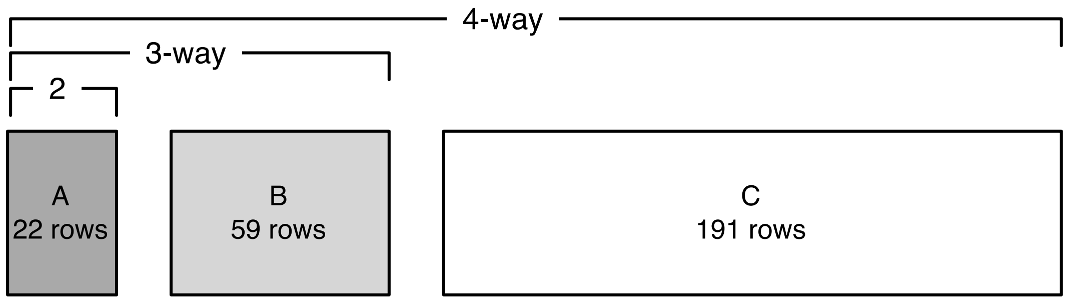

Figure 4.3: Seeding to create a covering array

4.4.1 The Use of Seeding in CA Construction

A variety of covering array construction techniques use seeds (both partial configurations and complete configurations) [24, 26, 78]. One approach allows for *default* or required configurations to be included as part of the testing process [18, 24]. However, these are selected ahead of time based on developer knowledge and are typically used to force inclusion of a specific, usually small, set of known configurations. Another use of seeds is to reduce the number of required t-way interactions that need to be covered by a specific algorithm, when constructing a covering array.

For example, Cohen et al. [26] used seeds with particular structural properties to build covering arrays that were then smaller than those created with non-seeded methods. The In Parameter Order (IPO) algorithm [78] uses a construction technique that explicitly adds new options (horizontal expansion) followed by a vertical expansion (additional configurations). This use of seeding is for the direct construction of covering arrays and is not aimed at re-use.

In this section we describe our seeding technique which has a different primary purpose: incremental construction of high-strength covering arrays. We begin with a single array at each strength, followed by an approach that allows for multiple arrays at each strength.

4.4.2 A Single Array at Each Strength

Figure 4.3 illustrates our approach in its simplest form. Using the model in Table 4.2 (where $k = 23, v_1..v_k = 2^{18}3^3 4^1 5^1$)), we start with an initial 2-way array called `A`, created using traditional techniques. To later create a 3-way array, `A`+`B`, we will use `A` as a seed and fill in the remaining rows, `B`. Similarly, we can build a 4-way covering array `A`+`B`+`C` using `A`+`B` as a seed.

If our incremental covering arrays are roughly the same size as traditional ones, then the 2-way array would have about 22 configurations and the 3-way, 80. We would execute the 2-way array first and then attempt to classify any failures involving two options. Because we build the 3-way covering array using the previous 2-way array as a seed, we reuse its 22 configurations and add about 58 new ones to get the complete 3-way coverage. We would be able to classify all 2-way failures after only 22 configurations and we could still classify all 3-ways after running only a total of 80 configurations, i.e., it costs us no more than before to execute the 3-way array, but we get early classification of the 2-way failures.

A key unanswered question is whether the covering arrays built with seeds will be comparable in size to the traditionally built ones. Little is currently known, however, about the size of covering arrays generated by large seeds. We have found heuristically that we can use a seed of approximately the same size as a $t-1$ array to build a t-way array for our problems. Beyond this, however, our solutions degrade. We use this heuristic in our case study and leave a generalization for future work.

4.4.3 Multiple Arrays at Each Strength

As mentioned earlier, a single covering array does not always provide enough information to definitively associate a failure with any particular t-way configuration option interaction. This is because the failure might be caused by: (1) an actual t-way interaction, (2) a higher-order interaction that is coincidentally being uncovered, or (3) a non-deterministic failure of the system that merely happens to occur on this execution. For example, a given 2-way covering array might indicate that activating option X and option Y simultaneously leads to a failure, but in reality the presence of options X, Y, T, and R might be required to trigger the failure, and this run just happens to exercise that configuration. Similarly, two covering arrays of any given strength may not be sufficient to disambiguate a low-strength, but non-deterministic failure from a higher-order failure.

In practice, testers may elect to run more than a single covering array test schedule at each strength, t, to deal with this situation. Therefore, we have extended our incremental approach to gather and distribute already tested configurations across multiple seeds when building multiple covering arrays at higher strengths.

4.4.4 The Incremental Covering Array Algorithm

The pseudocode for our algorithm is shown in Algorithm 1. It begins by allowing users to *optionally* specify the starting strength (t), the number of arrays to generate at each strength ($number$) and the maximum strength that will be built (max_t). Some default values we've used are: $t = 2$, $number = 1$, and $max\_t =$ the number of configuration options.

Next it generates the first t-way array using the traditional approach (Line 4) and then constructs arrays iteratively at higher strengths until it has reached the maximum strength (Lines 5-11). Each iteration starts by using the first t way array, CA_t_1 as a seed to build the first $t + 1$ array, $CA\_(t + 1)_1$ (Lines 6-7), e.g. it always begin by building a single higher strength array. This array is then used as a resource for seeding the remaining covering arrays at strength t. For each remaining array at strength t, $CA\_t_2..CA\_t_i$ (*for* loop: Lines 8-10), we gather all non-seeded configurations from $CA\_(t - 1)_i$ (Line 8) and some additional configurations from the higher strength covering array $CA\_(t + 1)_1$ (Line 9) to create the seed for CA_t_i. We have found that a good size for this seed is approximately the size of the

$CA_(t-1)$s. This provides as much re-use as possible while still yielding a small size for the resulting array. We then build CA_t_i using this seed (Line 10). Finally, we increment t (Line 11) and repeat the outer loop.

Figure 4.4 depicts an example of this strategy that starts at $t = 2$ and creates three arrays at each strength. The matching letters and arrows shows the seeding relationships. For instance, the 2-way array, `A`, becomes a seed for the first 3-way array and the configurations from the first 3-way array labeled as `B` and `C` are used as seeds for the second and third 2-way arrays. We can see that the second and third 3-way arrays have seeded configurations (`E,G,F,H`) that are taken from both the 2-way arrays (`E,F`) and from the first 4-way array (`G,H`) - not shown. In the first iteration, when $t = 2$ we see a special case. Since there is no $t-1$-way array, the seed set is initially empty (Line 8); in this case, all of the seeded configurations come from the higher strength array (`B` and C).

In essence, our algorithm iterates through a process of building a covering array of strength $t+1$, using a strength t array as a seed; it then uses configurations from both previously run $t-1$-way arrays and from the newly created $t+1$ array to seed the building of the remaining t-way arrays. Thus, after running some or all of the $t-$way covering arrays, a large portion of the initial $t+1$-way array has already been run. At this point, t-way failures can be classified and developers can begin to fix the underlying faults. If and when the developers so desire, t can be incremented by 1 and the process can repeat. In this case, a new $t+1$-way array will be constructed using the first t-way array as a seed. Part of that $t+1$-way array is combined with the configurations for the $t-1$-way arrays and used as seeds to generate the rest of the t-way arrays.

```
1: set $t$
2: set $number$
3: set $max\_t$
4: generate base $t$-way array, $CA\_t_1$
5: while not $max\_t$ do
6:     $seed\_t_1 = CA\_t_1$
7:     generate $CA\_(t+1)_1$ from $seed\_t_1$
8:     for $i = 2$ to $number$ do
9:         get $seed\_t_i$ from $CA\_(t+1)_1$ and $CA\_(t-1)_i$
10:        use $seed\_t_i$ to generate $CA\_t_i$
11:    increment $t$
```

Algorithm 1: INCREMENTAL

4.5 A Case Study

To evaluate our incremental failure classification approach we applied it to three consecutive releases of MySQL (builds 1.2510 – 1.2512). We also applied the traditional covering array approach with varying strengths. Our goal was to compare the costs and benefits of the modified approach with those of the traditional one, which requires pre-selection of the covering array's strength, analysis of the resulting test data only after all tests are completed, and, in some cases, repetition of the process with a higher strength covering array.

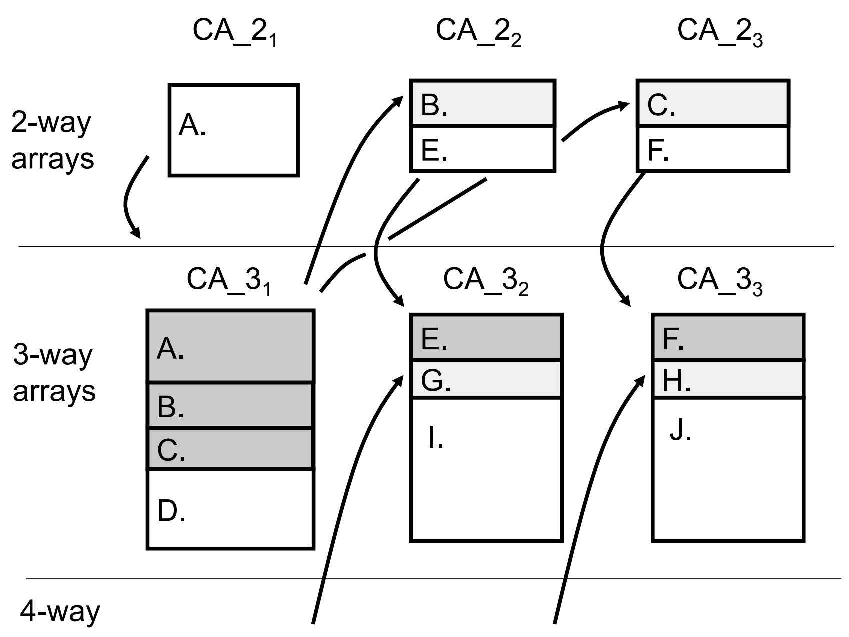

Figure 4.4: Constructing iterative CAs

4.5.1 Methodology

Our subject program for these studies is the MySQL client and server system [64], specifically focusing on 3 later releases of MySQL 5.1 that were not part of the field study described earlier in Section 4.3. For this study, our configuration space comprises 23 MySQL configuration options (18 binary, 3 with 3 values, 1 with 4 values and 1 with 5 values). There are several constraints on these options, which give rise to 8 explicit pair-wise combinations that cannot occur in valid configurations. The resulting test space contains 72,548,352 possible configurations. Table 4.2 gives more details of our configuration model and its constraints; the configuration options are split between compile time and run time options. For instance, `extra-charsets` is a compile time option with three possible values; while `sql-mode` is a run time option, also with three possible values.

We tested these configurations using 50 nodes of our Skoll Quality Assurance

Cluster–each running Red Hat 3.4.4-2 on a 2.8 GHz Pentium 4 with 1GB of memory. During testing, the configuration model, individual test plans and all results were stored in the Skoll Cluster Database Server, running MySQL 5.0.27. For these 3 releases (source control revisions: 1.2510, 1.2511, and 1.2512), we tested a total of 5994 compilable, valid configurations, running 772 developer-supplied regression tests on each config. Each test is designed to emit an error message in the case of failure, which we captured as well as recording all other test results. All told, this testing took 2 machine years to complete and included over 4.6 million individual test executions.

Covering Arrays. Using the configuration model previously described, we executed both the traditional and incremental covering array algorithms. Specifically, we created 3 incremental covering arrays at each strength t from 2 through 4, and a single incremental 5-way covering array (which was needed to generate the incremental 4-way arrays). Specifically we computed a $CA(N; t, 2^{18}3^{3}4^{1}5^{1})$ for each value of t. To evaluate our incremental covering arrays, we also generated three traditional covering arrays for each value of t from 2 through 4. Since our algorithms make some random decisions, the sizes of the individual covering arrays can vary slightly which is shown as a range in table 4.3.

Table 4.3 summarizes the covering array sizes for each value of t, the number of reused results in the incremental approach, and the total number of configurations for each covering array approach. Since some of the incremental covering arrays are generated from seeded rows, the sizes of the resulting arrays vary and are slightly larger than the fixed t-way arrays. As can be seen, if we use the incremental ap-

Compile Time Options	
Binary Options (Enabled/NULL)	
assembler, local-infile, thread-safe-client,archive-storage-engine, big-tables, blackhole-storage-engine, client-ldflags, csv-storage-engine, example-storage-engine, fast-mutexes, federated-storage-engine, libedit, mysqld-ldflags, ndbcluster, pic, readline, config-ssl, zlib-dir	
Non-binary Options	Values
extra-charsets innodb	–with-extra-charsets=all, –with-extra-charsets=complex, NULL –with-innodb,–without-innodb, NULL
Run Time Options	
transaction-isolation innodb_flush_log sql-mode	READ-UNCOMMITTED, READ-COMMITTED, REPEATABLE-READ, SERIALIZABLE, NULL 0,1,2, NULL ANSI,TRADITIONAL, STRICT_ALL_TABLES
Constraints	
innodb **requires:**	–without-innodb –transaction-isolation=NULL –innodb_flush_log=NULL
libedit **cannot occur with**	Enabled –readline=enabled

Table 4.2: MySQL configuration options

proach, starting at 2-way and successively increasing to 4-way for this model, then we need to run 834 configurations (68, 2-way + 180, 3-way + 586, 4-way). This is nearly identical to the size of the traditional 4-way covering array which has 832 configurations. It is also considerably more efficient (27%) than sequentially using the traditional approach starting at $t = 2$, then running $t = 3$ and then $t = 4$. Under that scenario, we would run 1136 configurations.

CA Strength (t)	size per trad. CA (N)	total size of trad. CA	size per incr. CA (N)	total size of incr. CA	num. incr. rows reused
2	22	66	22-23	68	0
3	78-80	238	81-84	180	68
4	271-281	832	272-283	586	248
totals		1136		834	316

Table 4.3: Size of covering arrays for $2 \leq t \leq 4$.

Process. For each release we use the incremental approach and then two different usage scenarios with the traditional approach. For the incremental approach we start at $t = 2$, incrementing as we go until we run out of time. For the traditional, in one scenario we only use a single strength t. This mimics the traditional approach where we must select *a priori* which strength t to use. In the other scenario, we run traditional schedules that begin at a specified t, incrementing to higher strengths as time allows. This mimics a situation in which developers complete a low strength test schedule, but find they still have more time available for testing.

For each of the the three usage scenarios, we run two variants. The first one uses a single array at every given strength. The second uses 3 arrays at every strength. The incremental approach is illustrated for the single array case in Figure 4.3 (on page 136). We run the test schedule in the order that completes lower-strength covering arrays the fastest. For instance, while running incremental covering arrays at strength 2 with 3 arrays at each strength we would execute (in order) array parts : A, B, E, C, and finally F. To increase to strength 3, we would then begin by running the configurations in part D, even though they were generated before parts E & F.

We ran all of the possible schedules to completion, but for our analysis we limited the data to the part of the schedule that could have been run during the *actual* lifetime of the release. We compute this by figuring an average cost of 3 hours to test each configuration, assuming a computing grid containing 50 nodes, and never interrupting a test execution. Thus, test release 1 (build 1.2510) which was current for 159 minutes before being replaced by build 1.2511 would only complete 50 test executions. Test release 2 (build 1.2511), in contrast, was current for almost 2 days, allowing for the completion of almost 800 test executions.

Metrics. For each specific build, we use all of test data from both the traditional and incremental approaches to create classifications models of the observed failures. The resulting models are used as our oracle since they represent our most complete explanation of the underlying test case failures. We note that most of our F-measures (the weighted harmonic mean of precision and recall in the classifications) were very close to 1.0 for this data – meaning that the classifications appear to accurately predict test outcomes.

Next, for each usage scenario we executed its test schedule and computed classification trees after each configuration in the schedule completed. This is done cumulatively; we keep the results of previous configurations within each test schedule. Each configuration ran 772 independent (MySQL developer created) test cases. After each test configuration,all the existing test results were collated and used to create 772 new classification trees. Note that approximately 300 of the 772 tests could not be usefully classified in terms of MySQL's configuration options. In our study, this occurs for two reasons. One is that we don't observe at least one passing execution and at least one failing execution for a given test. For example, some tests failed every time they ran. A classification tree for this data, of course, would not associate the failure with any particular configuration option. The other reason is that rarely-occurring failures are easily confounded with transient failures and other problems not related to configuration options. Particularly, classification attempts on rarely occurring failures led to trees that merely predicted that all configurations passed with high (e.g. $> 99.5\%$) accuracy. Therefore, we considered only those failures that occurred on more than 1% of the test runs.

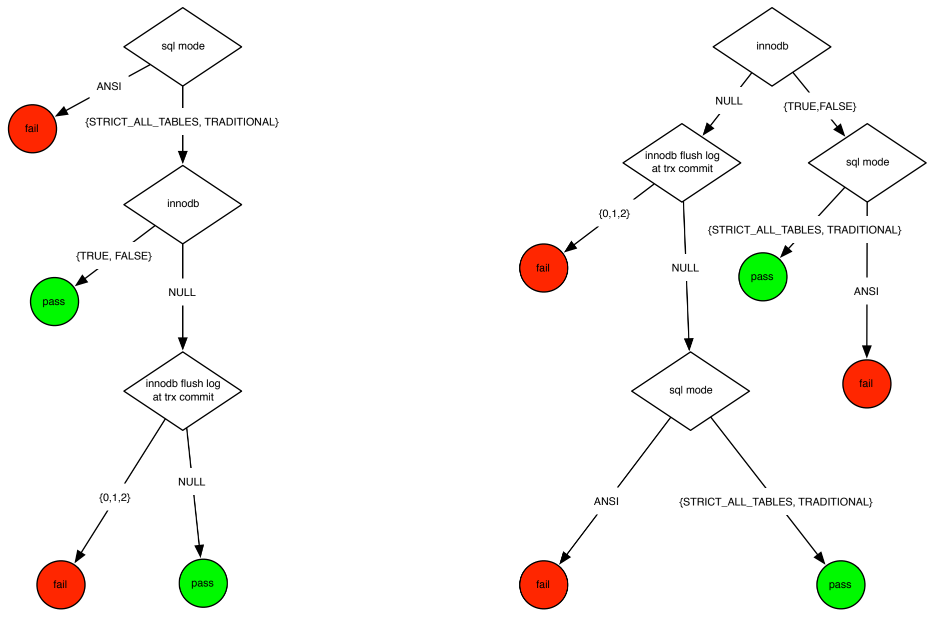
sql mode
ANSI
fail
{STRICT_ALL_TABLES, TRADITIONAL}
innodb
{TRUE, FALSE}
pass
NULL
innodb flush log at trx commit
{0,1,2}
fail
NULL
pass
innodb
NULL
{TRUE,FALSE}
innodb flush log at trx commit
sql mode
{0,1,2}
fail
NULL
{STRICT_ALL_TABLES, TRADITIONAL}
pass
ANSI
fail
sql mode
ANSI
fail
{STRICT_ALL_TABLES, TRADITIONAL}
pass

Figure 4.5: two example classification trees for test “ndb.ndb_gis” (configurations: #256 & #257)

For each test result and associated classification tree, we then compared it to a corresponding tree produced using the oracle data. During our initial analyses we considered the two trees to be identical only when their structures matched exactly. This proved too rigid as we found many examples of semantically equivalent, but structurally different classification trees. For example, the trees in figure 4.5 are structurally different, but give the same classifications for all inputs.

To remedy this, we “normalized” the classification trees before comparison. Specifically, we reduced the two trees into sets of option/settings pairs and considered the trees to be equivalent if they specified the same configuration options (or *features*) with the same settings (*values*). In other words, when the two trees made classification decisions based on the same options and settings, we considered them to be equivalent. We note that this normalization strategy is imperfect. It may decide that two trees are not equivalent, when they, in fact, are. However, as these measurements errors apply equally to all techniques and as normalization is only used internally for our experimental measurements, we decided it was sufficient.

Figure 4.5 includes an example using two trees for one particular MySQL test case (`ndb.ndb_gis`).In this example the trees are structurally dissimilar, but yield failure classifications that are the same. In fact, after normalization, these trees are both reduce to the following set of option=settings pairs:

```
 { (innodb=--with-innodb,--without-innodb),
(innodb=NULL), (innodb_flush_log_at_trx_commit=0,1,2),
(innodb_flush_log_at_trx_commit=NULL), (sql_mode=ANSI),
(sql_mode=STRICT_ALL_TABLES,TRADITIONAL) }
```

Finally, we compared the aggregated the number of equivalent trees across each test schedule to compute the percentage concordance with the oracle for each covering array algorithm. Intuitively, the more agreement between the more comprehensive oracle data and that generated by the proposed regime, the better we consider the proposed regime to be.

Threats to Validity. All experiments suffer from threats to validity. In this study we have used a single software system, with many native failures, and a large configuration space. Although MySQL shares many characteristics with other large configurable systems, the specific failures it experiences may differ in number or character. In our experiments we tried to select a broad configuration sample, but due to the sheer size of the entire configuration space for the software subject, we were only able to test a small portion of possible configurations. It is possible that choosing a different configuration space might alter the results. For instance, the failures that we are currently classifying as 2-way failures may indeed have shown themselves to be higher order failures if a larger configuration space was chosen. We do not believe, however, that this will change the results of cost comparison which are independent of the types of failures seen because we have considered several different classes of configuration options.

4.5.2 Results

In this section, we evaluate our approach on three releases of MySQL, each of which has a different lifetime. We begin with release 1.2510 which was current for only 159 minutes. We then examine release 1.2511 which was current for just under 2 days. Finally, we examine 1.2512 which was current for about 2 weeks.

4.5.2.1 Release 1: Lifetime of 159 Minutes

This release had a short lifetime during which we could test only 50 configurations. This was enough time to run 2 complete 2-way arrays, but not enough to run any complete 3-way arrays. Table 4.4 shows the overall classification data for this build. Each row represents a different covering array test schedule; rows are broken up into sections for the oracle, the traditional covering array approach, and our incremental approach. The first three columns show information regarding the complete test schedule; specifically, the strength of the arrays in the schedule (t), the number of arrays run at each strength (n), and the number of configurations in each schedule. The last three columns present the data stemming from the execution of the field study; namely, the percentage of the schedule executed (due to time allotted), the number of classification trees that matched the oracle, and ,finally, the percentage of the trees that were correctly classified according to the oracle data. Thus, the first line of data shows that for this study the oracle contained 3136 configurations, and contained 471 classification trees. The next line of data is for the traditional approach with strength $t = 2$ and a single array at each strength. Where the value for t is a list the strength of the covering array schedule was increased after the lower strength arrays completed.

As can be seen in Table 4.4, the traditional approach generally performed worse than the incremental approach for this release. For instance when we ran a single traditional array of strength $t = 3$ 84% of the resulting classification trees matched those created by the oracle data. With incremental arrays for $t = 2, 3$ the classification trees matched the oracle 94.7% of the time. The traditional approach that executed both a 2-way and a 3-way array performed even worse, correctly classifying only 55.2% of the faults. It should be noted that for single ($n = 1$) covering arrays with strength $t = 2$ the incremental approach does better, but this is an artifact of the data set since both test schedules were generated using the traditional approach. In general, there isn't enough data in these test schedules to classify the MySQL failures well.

t	**Num at t (n)**	**Configs scheduled**	**Percent Schedule Completed**	**Num Correctly Classified**	**Percent Correctly Classified**
Oracle					
		3136	100%	471	100
Traditional					
2	1	22	100%	93	19.7%
3	1	80	63%	397	84.3%
4	1	271	18%	95	20.2%
2	3	66	76%	343	72.8%
2,3	1	100	50%	260	55.2%
Incremental					
2	1	22	100%	319	67.7%
2,3	1	81	62%	446	94.7%
2	3	68	74%	438	93.0%

Table 4.4: Classification results for release 1: 159 min. execution time

Figure 4.6 shows the classification results obtained from covering arrays for strengths $t = 2, 3, 4$. Here, the classification trees are recomputed each time a configuration is tested. The horizontal axis represents the number of configurations tested during the study. The vertical axis denotes the percentage of classifications that were correct by comparison to the oracle. We see that the incremental approach never does worse than the traditional and, in this case, seems to do well early (at test 13). The end of the $t = 2$ portion each schedule is marked with a filled circle on the graphs. The remainder of the graph represents data from the incomplete 3-way covering array schedules.

t	**Num at t (n)**	**Configs scheduled**	**Percent Schedule Completed**	**Num Correctly Classified**	**Percent Correctly Classified**
Oracle					
		1600	100%	468	100%
Traditional (n=1)					
2	1	22	100%	91	19.4%
3	1	80	100%	384	82.1%
4	1	271	100%	465	99.4%
2,3	1	102	100%	389	83.1%
3,4	1	351	100%	467	99.8%
2,3,4	1	373	100%	467	99.8%
Traditional; (n=3)					
2	3	66	100%	450	96.2%
3	3	238	100%	466	99.6%
4	3	832	96%	467	99.8%
2,3	3	304	100%	466	99.6%
3,4	3	1070	75%	467	99.8%
2,3,4	3	1136	70%	467	99.8%
Incremental; (n=1)					
2	1	22	100%	322	68.8%
2,3	1	81	100%	465	99.4%
2,3,4	1	273	100%	466	99.6%
Incremental; (n=3)					
2	3	68	100%	458	97.9%
2,3	3	248	100%	465	99.4%
2,3,4	3	835	96%	464	99.1%

Table 4.5: Classification results for release 2: 1.9 day execution time

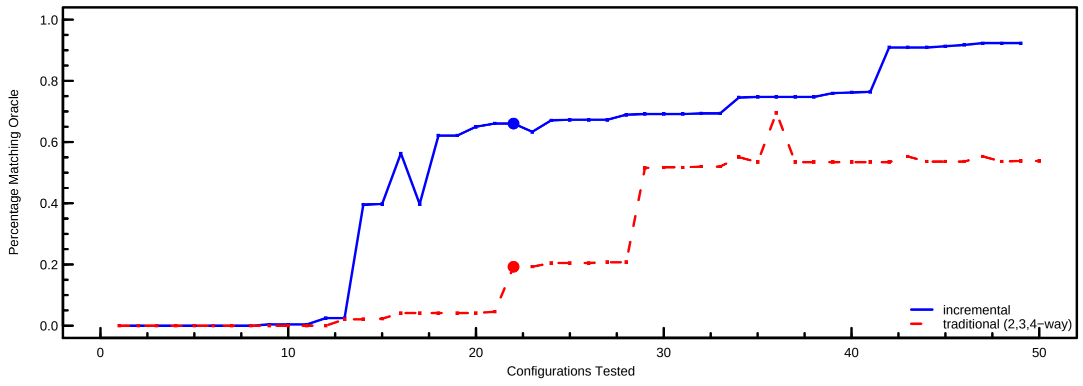

Figure 4.6: Graph comparing percentage of classifications matching the oracle as an increasing number of test executions are performed for incremental vs. 2,3,4-way traditional: release 1; 3 arrays at each strength

4.5.2.2 Release 2: Lifetime of 1.9 Days

This release was current for about 2 days, allowing for the testing of about 800 configurations. Thus, most but not all, schedules had enough time to complete. Table 4.5 presents some of the classification results. Here the poorest performer was the traditional, single 2-way array. The poor performance occurred because many 3-way failures were misclassified as having 2-way causes. This problem disappeared when using multiple traditional 2-way arrays and when using one or more 3-way covering arrays.

The incremental approaches match the oracle about as well as the corresponding traditional schedules, but often required significantly fewer configurations to complete the test schedule. The cost of the incremental approach – in terms of configurations tested – was equal to running a fixed traditional 4-way array; while the cost of running the traditional approach "incrementally" was considerably higher than the cost of using our incremental approach.

We also note that while running multiple covering arrays at a given strength does not greatly improve overall classification accuracy in this release, it did prevent misclassifications of a non-deterministic failure (detailed in problem 2 of section 4.3.2). In contrast, the single array approaches failed to characterize that particular failure properly.

Figures 4.7-4.9 show classification accuracy as the test process progresses. In each graph the incremental approach is represented by a solid line, the traditional approach by a dotted line, and filled circles mark the completion of a complete covering array (e.g., while executing a 4-way covering array, the process must at some point first complete a 2-way and then a 3-way covering array. Overall incremental covering arrays yielded better results earlier in the process. This may be useful for giving developers feedback as soon as possible. Additionally, it may be beneficial in the event that a test schedule is truncated early and classification must be done with the currently available data. Note that the incremental approach was not always better than the traditional approach. Specifically, Figure 4.7 shows a short time period (configs 30-42) where the traditional approach does better than incremental.

4.5.2.3 Release 3: Lifetime of 2 weeks

This release had the longest lifetime; long enough to allow all schedules to run completely. Some results for this release are shown in Table 4.6. Once again, we see that the failures are classified poorly when using only a single, lower strength (e.g.,

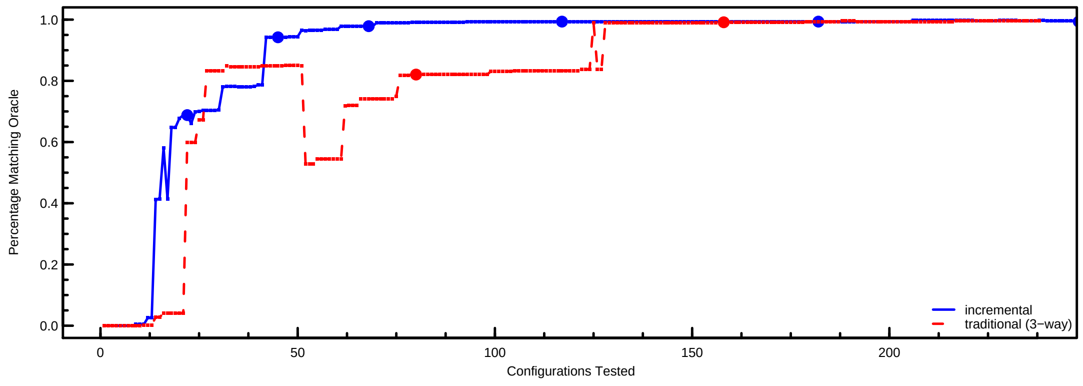

Figure 4.7: Graph comparing percentage of classifications matching the oracle as an increasing number of test executions are performed for incremental vs. 3-way traditional: release 2; 3 arrays at each strength

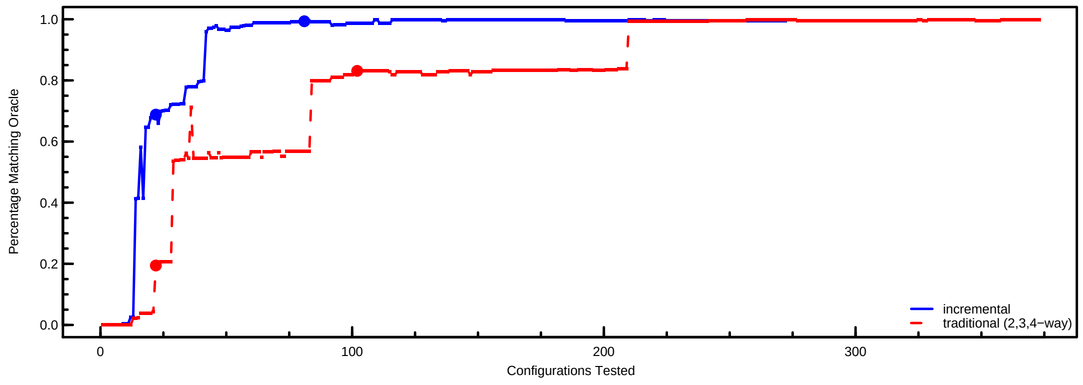

Figure 4.8: Graph comparing percentage of classifications matching the oracle as an increasing number of test executions are performed for incremental vs. 2,3,4-way traditional: release 2; 1 array at each strength

$t \leq 3$) covering array. Classification accuracy was generally greater than 98% when using multiple covering arrays at each strength or when using higher strength covering arrays (e.g., $t = 4$). Figures 4.10, 4.11 and 4.12 show classification accuracies as the process progressed under different usage scenarios. The incremental approach

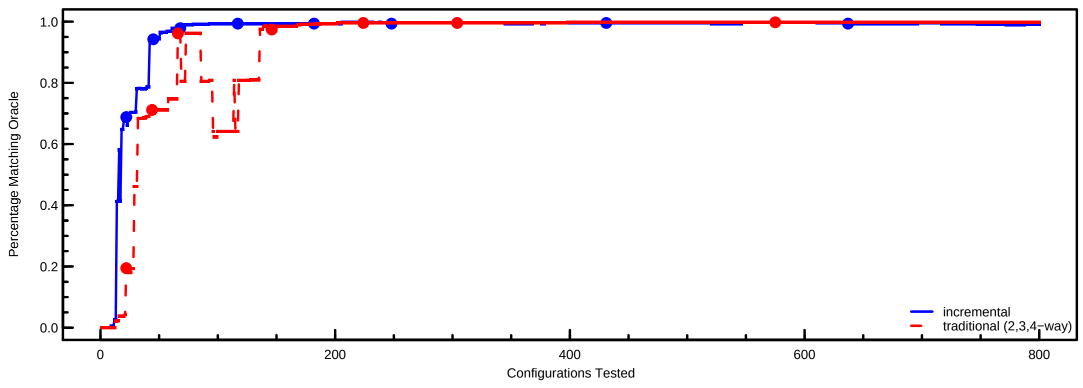

Figure 4.9: Graph comparing percentage of classifications matching the oracle as an increasing number of test executions are performed for incremental vs. 2,3,4-way traditional: release 2; 3 arrays at each strength

was generally comparable to the traditional approach at $t = 4$, but that the incremental approach produced good classifications earlier. For example, in figure 4.11 with the traditional approach, if t is initially chosen too low and the developers need to increase the covering array strength and continue testing. Meanwhile, the incremental approach automatically increases strength and converges to the oracle's results more quickly – achieving 95% accuracy with only 50 configurations tested (). The traditional approach completed a 2-way, a 3-way and half of the 4-way (approx. 225 configurations) to get the same level of accuracy. Similar results occurred for single 4-way covering array as well.

4.6 Related Work

Other techniques have been used to isolate failures in code during debugging. The cooperative bug isolation project [53], for example, uses code instrumentation

t	**Num at t (n)**	**Configs Tested**	**Num Correctly Classified**	**Percent Correctly Classified**
Oracle				
		1970	468	100%
Traditional (n=1)				
2	1	22	92	19.6%
3	1	80	377	80.6%
4	1	271	465	99.4%
2,3	1	102	378	80.8%
3,4	1	351	463	98.9%
2,3,4	1	373	463	98.9%
Traditional (n=3)				
2	3	66	450	96.2%
3	3	238	458	97.9%
4	3	832	467	99.8%
2,3	3	304	459	98.1%
3,4	3	1070	467	99.8%
2,3,4	3	1136	467	99.8%
Incremental (n=1)				
2	1	22	323	69.0%
2,3	1	81	466	99.6%
2,3,4	1	273	466	99.6%
Incremental (n=3)				
2	3	68	458	97.9%
2,3	3	248	465	99.4%
2,3,4	3	834	465	99.4%

Table 4.6: Classification results for release 3: 2 week execution time

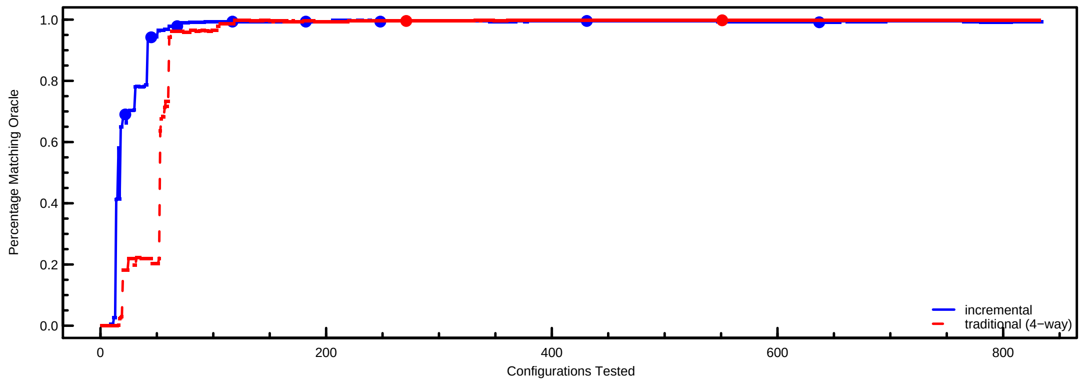

Figure 4.10: Graph comparing percentage of classifications matching the oracle as an increasing number of test executions are performed for incremental vs. 4-way traditional: release 3; 3 arrays at each strength

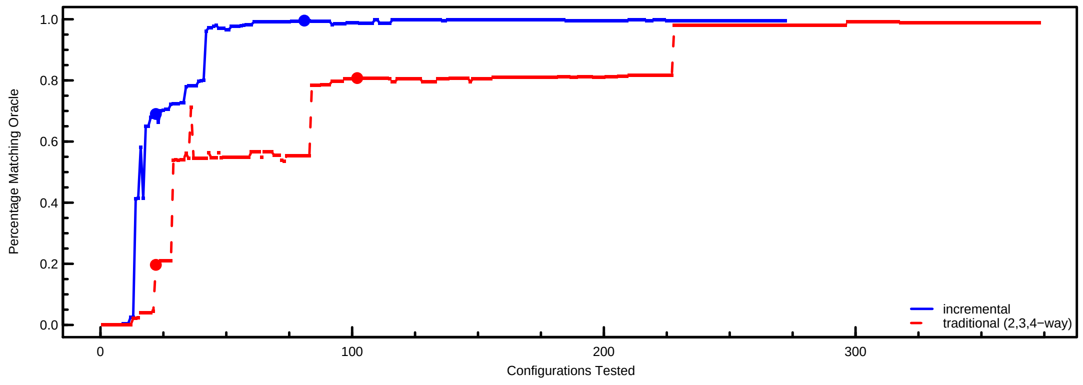

Figure 4.11: Graph comparing percentage of classifications matching the oracle as an increasing number of test executions are performed for incremental vs. 2,3,4-way traditional: release 3; 1 array at each strength

and statistical sampling to achieve failure characterization, while the delta debugging project isolates minimal subsets of tests that cause failures through successive elimination of the input space [88]. Neither of these projects address the configuration space directly. As mentioned earlier, covering arrays were used for failure

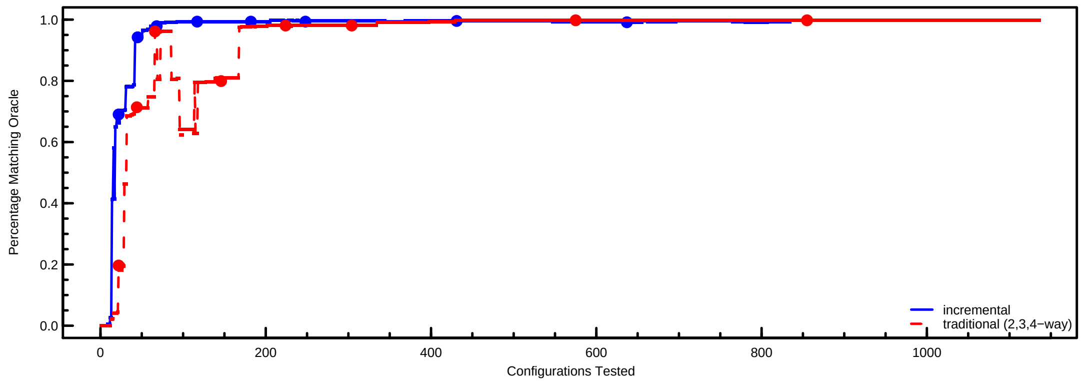

Figure 4.12: Graph comparing percentage of classifications matching the oracle as an increasing number of test executions are performed for incremental vs. 2,3,4-way traditional: release 3; 3 arrays at each strength

classification in [84], however this work assumes *a priori* knowledge about the types of failures that will be observed and assumes resources are available to run the selected strength arrays in their entirety. That work also used another type of covering array, called a variable strength covering array to support classification. The test schedules generated from variable strength arrays allow different subportions of the configuration space to be tested at higher strengths than other parts. The assignment of strengths to configuration subspaces is still done manually, however, and does not change during the test process. The work of Robinson and White [74] selects subsets of configurations for testing, but uses a different sampling technique.

Covering arrays have been used frequently to reduce the number of inputs [24, 28, 33] or configurations [49, 73] when testing a program; however, other than in [84] their primary purpose has been failure detection, not classification. Construction techniques for building covering arrays [24, 25, 26, 78] describe seeding of rows of the covering array, but for a different purpose than our approach. Seeding has been used either to allow testers to request a set of default configurations [18, 24] or as the basis for specialized constructions that generate smaller covering arrays [26]. The work of Tai *et al.* [78] uses a construction method that builds covering arrays by expanding the factors (i.e., the columns), but the purpose is to allow for new factors to be added, not to change the strength.

Our approach is unique in that we use covering arrays for failure classification, but do not require developer expertise or *a priori* knowledge in setting covering array strengths. Instead we employ an initial lightweight sample and then incrementally build using seeding as a both a construction technique, and as a mechanism to reuse information from already tested configurations.

4.7 Conclusion and Future Work

This chapter presents an automated algorithm for generating covering array test schedules that reduces costs and improves flexibility by incrementally constructing and executing covering arrays, and by carefully reusing tests from lower strength covering arrays to construct higher strength ones.

Our algorithm successfully addresses several serious limitations of current techniques. Specifically, developers must currently select a single strength for the covering array even though they have no reliable scientific or historical basis for doing so and even though there may no reason to believe that a single strength is sufficient. In practice, if developers choose too low an initial strength they will need to start the process from scratch at a higher strength. If they choose too high a strength, they waste resources and delay the arrival of lower strength results. Also, the typical practice of generating a single covering array at a given strength leads to complications when non-deterministic failures appear.

Our approach, incremental covering arrays, leverages information gained in previous test executions to generate future test schedules. This allows developers to choose the lowest practical value for t, 2 by default, because there is minimal penalty for starting too low. The process can move up to higher strength covering arrays only if warranted by test results or resource availability. Finally, running multiple covering arrays at each strength can better support identification of non-deterministic faults, while simultaneously providing data for higher strength arrays.

We also presented a large case study in which we evaluated this new algorithm to test a non-trivial open source software system. This case study compared our new approach to traditional covering arrays across a configuration space of $72M$ configurations of MySQL. For each tested configuration we ran nearly 800 test cases. Despite the limitations of our case study, we tentatively conclude that for this data:

• Classification models based on incremental covering array test schedules were no worse than those based on traditional covering array test schedules.

• In the worst case, the incremental approach starting at strength 2 and working up to strength t, was $\sim 5\%$ more expensive than using fixed strength traditional covering arrays of strength t.

• Compared to using traditional covering arrays incrementally, which typically involves running multiple covering arrays from scratch, our approach required up to

27% fewer configurations. It did this with no loss of quality.

• Our incremental approach was able classify failures earlier–using fewer test configurations–than traditional covering arrays. This is valuable because for many evolving systems undergoing continuous build and test where the lifetime of a particular source revision is unknown.

Our future work concentrates on validating, as well as refining and generalizing, the incremental covering array approach. First, we plan to expand our study to monitor the *live* MySQL continuous build, integration and test process. We will examine a broader configuration space than we are currently using and we will replicate this work on additional large, configurable software systems as well. Second we plan to examine alternative models for distributing seeds which will allow us to use differing numbers of covering arrays at each level of t. Finally, we are working on a method for automatically determining "when" the algorithm should adapt by moving to higher strength (versus generating more arrays at the current strength).

Bibliography

[1] CruiseControl. http://cruisecontrol.sourceforge.net/.

[2] Dart. http://www.itk.org/Dart/HTML/Index.shtml.

[3] GNU C Compiler. `http://gcc.gnu.org`.

[4] JABA: Java Architecture for Bytecode Analysis. http://www.cc.gatech.edu/aristotle/Tools/jaba.htm.

[5] Linux Test Project. `http://ltp.sourceforge.net/`.

[6] Rulequest.

[7] The R Project for Statistical Computing. `http://www.r-project.org/`.

[8] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of the 20th Conf. on Very Large Databases*, pages 487–499, 1994.

[9] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4), Nov 1997.

[10] Aristotle Research Group. Jaba: Java architecture for bytecode analysis. http://www.cc.gatech.edu/aristotle/Tools/jaba.html.

[11] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2001)*, pages 168–179, New York, NY, USA, 2001. ACM Press.

[12] James Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed sw using sw tomography. In *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for SW Tools and Eng. (PASTE 2002)*, pages 2–8, Charleston, SC, USA, november 2002.

[13] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of sw behavior, July 2004.

[14] Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. *SIGSOFT Softw. Eng. Notes*, 28(1):2–9, 2003.

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

[16] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[17] Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions, May 2004.

[18] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Journal of Information and Software Technology*, 48(10):960–970, 2006.

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

[20] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In *Tests and Proofs, Lecture Notes in Computer Science, 4966*, pages 66–83, 2008.

[21] A. Calvagna and A. Gargantini. Using SRI SAL model checker for combinatorial tests generation in the presence of temporal constraints. In *Workshop on Automated Formal Methods (AFM)*, pages 1–10, 2008.

[22] D. Chays, S. Dan, Y. Deng, F. I. Vokolos, P. G. Frankl, and E. J. Weyuker. AGENDA: A test case generator for relational database applications. Technical report, Polytechnic University, 2002.

[23] C. Cheng, A. Dumitrescu, and P. Schroeder. Generating small combinatorial test suites to cover input-output relationships. In *Proceedings of the 3rd Quality Software International Conference (QSIC)*, pages 76–82, November 2003.

[24] 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 Trans. on SW Eng.*, 23(7):437–44, 1997.

[25] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. In *Proc. of the Int'l Conf. on SW Eng., (ICSE)*, pages 38–44, 2003.

[26] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *Proc. of Int'l Symp. on SW Reliability Eng. (ISSRE)*, pages 394–405, November 2003.

[27] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Int'l Symp. on SW Testing and Analysis, (ISSTA)*, pages 129–139, July 2007.

[28] 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 Int'l Conf. on SW Eng., (ICSE)*, pages 285–294, 1999.

[29] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: the distribution of program failures in a profile space, September 2001.

[30] William Dickinson, David Leon, and Andy Podgursky. Finding failures by cluster analysis of execution profiles, May 2001.

[31] DOC group for DRE Systems. Real-time corba with tao (the ace orb). http://www.cs.wustl.edu/ schmidt/TAO.html, February 2006.

[32] Richard Duda, Peter Hart, and David Stork. *Pattern Classification*. Wiley, second edition, 2000.

[33] 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 Int'l Conf. on SW Eng., (ICSE)*, pages 205–215, 1997.

[34] Sebastian Elbaum and Madeline Diep. Profiling deployed sw: Assessing strategies and testing opportunities. *IEEE Trans. on SW Eng.*, 31(4):312–327, 2005.

[35] Wei Fan, Salvatore J. Stolfo, Junxin Zhang, and Philip K. Chan. AdaCost: misclassification cost-sensitive boosting. In *Proc. 16th Int'l Conf. on Machine Learning*, pages 97–105. Morgan Kaufmann, San Francisco, CA, 1999.

[36] Sandro Fouché, Myra B. Cohen, and Adam Porter. Incremental covering arrays failure characterization: a detailed case study.

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

[38] Sandro Fouché, Adam Porter, Michael Last, and Alessandro Orso. Low overhead classification of deployed software executions.

[39] Patrick Francis, David Leon, Melinda Minch, and Andy Podgurski. Tree-based methods for classifying sw failures, November 2004.

[40] Kenny C. Gross, Aleksey Urmanov, Lawrence G. Votta, Scott McMaster, and Adam Porter. Towards dependability in everyday sw using sw telemetry. In *EASE '06: Proc. of the Third IEEE Int'l Workshop on Eng. of Autonomic & Autonomous Systems (EASE'06)*, pages 9–18, Washington, DC, USA, 2006. IEEE Computer Society.

[41] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying Classification Techniques to Remotely-Collected Program Execution Data. In *Proc. of the ACM SIGSOFT Symp. on the Foundations of SW Eng.*, pages 146–155, Lisbon, Portugal, september 2005.

[42] Murali Haran, Alan Karr, Michael Last, Alessandro Orso, Adam Porter, Ashish Sanil, and Sandro Fouche. Techniques for classifying executions of deployed sw to support sw eng. tasks. Submitted for publication. Available as Technical Report cs-tr-4826 of the University of Maryland, Computer Science Department, September 2006.

[43] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math*, 284:149 – 156, 2004.

[44] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[45] David M. Hilbert and David F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, Dec 2000.

[46] B. Hnich, S. Prestwich, E. Selensky, and B.M. Smith. Constraint models for the covering test problem. *Constraints*, 11:199–219, 2006.

[47] J.K. Hollingsworth, J.E. Lumpp, and B.P. Miller. Techniques for performance measurement of parallel programs. In *Parallel Computers: Theory and Practice.* IEEE Press, 1995.

[48] Mahesh V. Joshi, Ramesh C. Agarwal, and Vipin Kumar. Mining needle in a haystack: classifying rare classes via two-phase rule induction. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):91–102, 2001.

[49] D.R. Kuhn, D. R. Wallace, and A. M. Gallo. Sw fault interactions and implications for software testing. *IEEE Trans. on SW Eng.*, 30(6):418–421, 2004.

[50] J.R. Larus and E. Schnarr. Eel: Machine-independent executable editing, 1995.

[51] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proc. of the 7th USENIX Security Symp.*, San Antonio, TX, 1998.

[52] David Leon, Andy Podgurski, and Lee J. White. Multivariate visualization in observation-based testing, May 2000.

[53] B. Liblit, A. Aiken, Z. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Conf. on Programming Language Design and Implementation*, pages 141–154. ACM, June 2003.

[54] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling, June 2003.

[55] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael 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.

[56] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Sampling user executions for bug isolation. In *First international workshop on remote analysis and measurement of software systems*, 2003.

[57] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. public deployment of cooperative bug isolation. In *Second international workshop on remote analysis and measurement of software systems*, 2004.

[58] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation, June 2005.

[59] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June12–15 2005.

[60] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[61] Atif Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, Douglas C. Schmidt, and Bala Nata rajan. Skoll: Distributed continuous quality assurance. In *Proc. of the Int'l Conf. on SW Eng., (ICSE)*, pages 459–468, 2004.

[62] Microsoft online crash analysis, 2004. `http://oca.microsoft.com`.

[63] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, Nov 1995.

[64] MySQL, 2006. `http://www.mysql.com`.

[65] K.J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138(1-2):143–152, 2004.

[66] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of the 9th European SW Eng. Conf. and 10th ACM SIGSOFT Symp. on the Foundations of SW Eng. (ESEC/FSE 2003)*, pages 128–137, Helsinki, Finland, september 2003.

[67] Alessandro Orso, James A. Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed sw. In *Proc. of the ACM symposium on SW Visualization (SOFTVIS 2003)*, pages 67–76, San Diego, CA, USA, june 2003.

[68] Alessandro Orso and Bryan Kennedy. Selective Capture and Replay of Program Executions, may 2005.

[69] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: continuous evolution of software after deployment. *SIGSOFT Softw. Eng. Notes*, 27(4):65–69, 2002.

[70] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring, May 1999.

[71] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *International Conference on Software Engineering*, pages 277–284, 1999.

[72] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda MinchJiayang Sun, and Bin Wang. Automated support for classifying sw failure reports, May 2003.

[73] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis*, pages 75–85, July 2008.

[74] Brian Robinson and Lee White. Testing of user-configurable software systems using firewalls. In *Symp. Softw. Rel. Eng.*, pages 177–186, November 2008.

[75] T. Romer, G. Voelker, A. Wolman, S. Wong, H. Levy, B. Chen, and B. Bershad. Instrumentation and optimization of win32/intel executables using etch, Aug 1997.

[76] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis.* Cambridge University Press, 2004.

[77] A. Srivastava and D.W. Wall. Link-time optimization of address calculation on a 64-bit architecture, 1994.

[78] K. C. Tai and L. Yu. A test generation strategy for pairwise testing. *IEEE Trans. on SW Eng.*, 28(1):109–111, 2002.

[79] O. Traub, S. Schechter, and M.D. Smith. Ephemeral instrumentation for lightweight program profiling. Unpublished technical report, Harvard University, Jun 2000.

[80] D.W. Wall and M.L. Powell. The mahler experience: Using an intermediate language as the machine description, Oct 1987.

[81] Wenhua Wang, Yu Lei, S. Sampath, R. Kacker, R. Kuhn, and J. Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 211 –220, 2009.

[82] L. J. White. Regression testing of GUI event interactions. In *International Conference on Software Maintenance*, pages 350 – 358, 1996.

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

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

[85] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. In *ISSTA '04: Proc. of the 2004 ACM SIGSOFT international symposium on SW testing and analysis*, pages 45–54, 2004.

[86] Cemal Yilmaz, Adam Porter, and Sandro Fouché. Distributed continuous qa. http://www.cs.umd.edu/projects/skoll/, June 2006.

[87] Xun Yuan, Myra Cohen, and Atif M. Memon. Covering array sampling of input event sequences for automated GUI testing. In *International Conference on Automated Software Engineering*, pages 405–408, 2007.

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