Adequate System-Level Testing of Distributed Systems

by

Matthew J. Rutherford

B.S.E., Princeton University, 1996

M.S., University of Colorado, 2001

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2006

This thesis entitled:
Adequate System-Level Testing of Distributed Systems
written by Matthew J. Rutherford
has been approved for the Department of Computer Science

Prof. Antonio Carzaniga

Prof. Alexander L. Wolf

Date

Rutherford, Matthew J. (Ph.D. Computer Science)

Adequate System-Level Testing of Distributed Systems

Thesis directed by Prof. Antonio Carzaniga and Prof. Alexander L. Wolf

Software testing is about risk management. Typically, engineers use test adequacy criteria to balance the cost and efficacy of the testing activity. Test adequacy criteria are rules that provide an objective stopping condition on test input creation by defining a finite set of test requirements that must be satisfied. While adequacy criteria have been a focus of research activity for many years, existing testing criteria do not address the unique features of distributed applications. The contributions of this dissertation are: (1) a study of reported failure scenarios of seven distributed applications; (2) a novel testing technique based on discrete-event simulations that serves as a basis for adequacy criteria for distributed systems; (3) a fault-based analysis technique that allows testers to addresses the fundamental risks associated with using adequacy criteria; and (4) a case-study evaluation of the simulation-based and fault-based techniques.

The failure study involves the categorization of test inputs and observations needed to replicate failures reported by users. The results show that failure-producing scenarios are amenable to categorization, that simple system topologies are likely to be quite effective, and that a significant proportion of failure-producing scenarios involve distributed inputs. Thus, the results confirm our intuition that distributed systems need their own class of adequacy criteria.

Rather than inventing a new specification formalism, we instead adapt the common practice of using discrete-event simulations for the design and understanding of distributed systems to testing. Our key observation is that these simulations can be viewed as specifications of the expected behavior of the system. Using simulations to test the implementation of a system is therefore a matter of selecting inputs to cover the simulation according to some criterion, and then mapping the inputs into the implementation domain. As simulations are sequential programs themselves, virtually all black- and white-box criteria can be used with a simulation-based technique.

When using any adequacy criterion for testing, there is generally no way for engineers to know a priori how effective a test suite or the criterion itself is going to be on their system. To mitigate this risk within the context of simulation-based testing, we propose a fault-based analysis technique that uses code mutation operators to create a set of incorrect simulations. Candidate test cases are executed against each of these specification mutants and the number killed is used as a surrogate measure of effectiveness.

We evaluate the simulation-based technique and the companion fault-based analysis method on 28 implementations of three distributed systems. The results of these experiments are striking. First, we confirm that discrete-event simulations can indeed be used in testing, and that white-box techniques based on the simulation are both effective, and cost-effective when compared to randomly selected test suites of the same size. Second, in a significant advancement of the state of the art, we demonstrate the power of the fault-based analyses by improving the effectiveness of adequate suites within each criterion, and by predicting exactly the overall relationships between different criteria.

Dedication

I dedicate this dissertation to my family and friends who have always been so supportive of everything I do.

First, I dedicate my dissertation to my daughter Maya, my son Elliot and their amazing mother, my wife, JJ. Thank you all for your love and support and for helping put everything else in the proper perspective.

I would also like to dedicate this work to my parents, John and Cynthia Rutherford, who have been so supportive of me and who have worked so hard to give us amazing opportunities in life. I thank my sister, Anna Rutherford, my brother-in-law, Brendan Everett, and their beautiful daughter Amelia for their love and support. Also, I would like to thank my parents-in-law, Kathy and Sandy Lonsinger, for their generosity, hospitality and love.

Finally, I would like to thank Dan Weiler for being such a great friend and always helping to keep this, and everything else, "real."

Acknowledgements

I would like to sincerely thank all of the people who have provided their support, friendship, advice, and feedback to me while working on this dissertation.

First, I would like to thank Antonio Carzaniga and Alex Wolf for their constant support and attention. By my count we have worked together for at least six years, and I hope that the collaboration does not end here.

I would also like to thank my committee members: Amer Diwan, Shiv Mishra, and Alex Orso for their time and energy and excellent feedback and guidance.

I would like to thank Yanyan Wang for being such a wonderful collaborator and confidant around the lab. I wish her the best of luck with her future endeavors.

Next, I would like to thank other CU faculty (in alphabetical order) who have been particularly supportive and instrumental in providing me with an excellent computer science education: Ken Anderson, Liz Bradley, Dan Connors, Amer Diwan, Hal Gabow, Dennis Heimbigner, and Jim Martin.

Finally, I want to thank the rest of my fellow SERL members for their comradeship and support: Naveed Arshad, John Giacomoni, Cyrus Hall, Van Lepthien, Mette Moffett, Aubrey Rembert, Nathan Ryan, Susanne Sherba, and, of course, Laura Vidal.

Contents

Chapter

1 Introduction 1

1.1 Hypotheses and Contributions 4

1.2 Failure Study 7

1.3 Simulation-based Testing 8

1.4 Experimental Evaluation 11

1.5 Related Work 12

1.6 Document Organization 13

2 Failure Study 14

2.1 Subject Systems 17

2.1.1 Peer-To-Peer 18

2.1.2 Federated 19

2.1.3 Client/Server 20

2.1.4 Distributed Component 21

2.2 Empirical Method 22

2.2.1 Report Classification 22

2.2.2 Architecture 25

2.2.3 Inputs 27

2.2.4 Observations 30

2.3 Analysis and Discussion 32
2.3.1 Implications for Testing 37
2.3.2 Threats to Validity 38
2.4 Summary 39

3 Simulations and Testing 41
3.1 Simulation-Based Testing 44
3.1.1 Modeling 45
3.1.2 Simulation Environment 47
3.1.3 Configuration 53
3.1.4 Simulation Execution 56
3.1.5 Aggregation 56
3.2 Fault-Based Analyses 57
3.3 Usage Scenarios 61
3.3.1 Conventional 61
3.3.2 Boosting 62
3.3.3 Ranking 63

4 Empirical Evaluation 65
4.1 Subject Systems 66
4.1.1 GBN 67
4.1.2 LSR 69
4.1.3 DNS 71
4.2 Experimental Method 73
4.2.1 Preparation 75
4.2.2 Implementation Fault Characterization 80
4.2.3 Technique Simulation 83
4.3 Experiment 1: Effectiveness 84

4.4 Experiment 2: Boosting Criteria 89
4.5 Experiment 3: Ranking Criteria 98
4.6 Summary of Results 101
4.7 Threats to Validity 102

5 Related Work 104
5.1 Foundations 104
5.1.1 Fault Studies 105
5.1.2 Test Adequacy Criteria 106
5.1.3 Mutation Analysis 112
5.1.4 Empirical Evaluation of Testing Techniques 113
5.2 Related Work 114
5.2.1 Specification-Based Testing 114
5.2.2 Fault-based Testing 116
5.2.3 Testing Distributed Systems 117
5.2.4 Distributed Testing Frameworks and Testbeds 118
5.2.5 Early-phase Performance Evaluation 119
5.2.6 Distributed Component Unit Testing 119

6 Conclusion 121
6.1 Future Work 123
6.1.1 Test Adequacy Criteria Selection Methods 124

Bibliography 127

A Elvin Bug Report 1008 136

Appendix

B Example: GCDService 139

B.1 Euclid's Algorithm . . . 139

B.2 Messages . . . 140

B.3 `GCDclient` Processing . . . 140

B.4 `GCDserver` Processing . . . 142

B.5 CacheTest Configuration . . . 142

Tables

Table

2.1 Classified Report Types 23

2.2 In-Scope Reports by System 26

2.3 Architectural Summary 26

2.4 Input Categories 28

2.5 Observation Categories 31

2.6 Distributed Scenarios by System 33

2.7 Architectural Complexity for Distributed Scenarios 34

2.8 Scenario Diversity 34

3.1 Example Boosting Scenario 63

4.1 Implementation Failure Rates 77

4.2 Generated Mutants 78

4.3 Total Preparation Time 79

4.4 GBN Distributed Input Characteristics 80

4.5 LSR Distributed Input Characteristics 81

4.6 DNS Distributed Input Characterization 82

4.7 Adequacy Targets 83

4.8 White-Box Effectiveness and Size 85

4.9 GBN White-Box Cost Effectiveness 87

4.10 LSR White-Box Cost Effectiveness 89

4.11 DNS White-Box Cost Effectiveness 90

4.12 Boosting GBN Criteria 95

4.13 Boosting LSR Criteria 96

4.14 Boosting DNS Criteria 96

4.15 GBN Experiment 3 Data 98

4.16 LSR Criteria **p-values** 99

4.17 DNS Criteria **p-values** 100

Figures

Figure

1.1 Idealized Criteria Effectiveness Distributions 6

2.1 James Report #66 16

2.2 James Report #5 24

2.3 Squid Report #198 25

2.4 Observation Complexity versus Distribution 35

2.5 Generic Observations versus Distribution 36

3.1 Specification-Based Testing Process 42

3.2 `GCDclient` Component 49

3.3 GCDService Messages 50

3.4 **GCDserver** Component 52

3.5 Example Configuration 54

3.6 ClientPairTest_A Coverage Report 58

3.7 **GCDserver** with Line 17 Mutation 60

3.8 **GCDserver** with Line 45 Mutation 60

4.1 LSR Topologies 69

4.2 Absolute White-Box Effectiveness 86

4.3 GBN Criteria Effectiveness Distributions 92

4.4 LSR Criteria Effectiveness Distributions 93

4.5 DNS Criteria Effectiveness Distributions 94

4.6 Criteria Relationships 99

B.1 Euclid's Algorithm 139

B.2 Message Implementations 140

B.3 `GCDclient` Component 141

B.4 `GCDserver` Component 143

B.5 `CacheTest` Configuration 144

Chapter 1

Introduction

> Software testing consists of the **dynamic** verification of the behavior of a program on a **finite** set of test cases, suitably **selected** from the usually infinite executions domain, against the **expected** behavior [BD04].

Software testing is about risk management. No testing technique, method, tool or process can guarantee that a particular software system is fault free; software cannot be exhaustively tested. All software producing organizations attempt (rationally or otherwise) to balance the potential costs of failure of a fielded system and the real costs of software testing. Concretely, the risk of testing too little is software failure, while the risk of testing too much is waste of resources. There is no one solution to this problem, each organization must navigate these waters alone.

A fundamental contribution made by Goodenough and Gerhardt in 1975 to the practice of software testing was the idea of test adequacy criteria [GG75]. Test adequacy criteria are rules that provide an objective stopping condition on test input creation by defining a finite set of test requirements that must be satisfied. For software producers, test adequacy criteria help decision makers by providing a menu of rational testing techniques to choose from.

At present, this menu is quite large. Existing criteria operate at all three of the canonical levels of software testing: unit, integration, and system. Black-box criteria are based on the external interfaces of a program. White-box criteria are based on both formal models derived from a program's source code and specifications of various styles.

In a perfect world, criteria would be organized into a clear hierarchy in terms of effectiveness, and the costs associated with each would be well understood and predictable. However, the effectiveness of adequacy criteria cannot be determined analytically, and empirical data comparing effectiveness and cost is relatively scarce, somewhat inconclusive, and not generalizable. In practice, the choice of adequacy criterion is more art than science which is precisely the situation that adequacy criteria are supposed to remedy.

Within this context, we are concerned with developing techniques to help organizations that develop distributed software systems test them rationally through the use of test adequacy criteria. Part of this work is at a relatively low-level and is devoted to the motivation and creation of a specification-based testing technique that provides a powerful basis for the definition and application of test adequacy criteria for distributed systems. At a higher level, the technique that we have developed enables a style of analysis that supports the engineer in deciding which criterion to use.

Definitions

As our work is primarily concerned with selecting adequate **test inputs** for the testing of **distributed systems**, we begin by defining these terms.

Our view of distributed systems is quite broad and includes any system of interconnected components that communicate via message passing over a latent and unreliable network. In the literature, distributed systems are understood to have the following unique characteristics [Sch93, WWWK94]:

separate address spaces – components do not run in the same memory address space;

communication latency – communication channels between components exhibit significant and variable delays;

coarse-grained concurrency – differences in execution speed and latency cause issues of concurrency to appear when components interact;

partial failure – the lack of central control combined with latent and unreliable communications implies that components can fail without other components realizing it, and also appear to have failed when in fact they are temporarily separated from the rest of the system, or just operating slowly; and

variable configuration – many distributed systems can be configured at a high level with different quantities of components, different component types, and different interconnections between them. This is not generally recognized as one of the fundamental characteristics of distributed systems, perhaps because it is shared with non-distributed component-based systems.

To account for these unique features of distributed systems we take a broad view of inputs. Traditionally in testing, "inputs" are the set of values that are passed into a program during execution. In this work we take the view that inputs include items that do not take on a particular value, but are nevertheless needed to execute the system. These non-standard inputs include things like system topology, network behavior, component operating environment, dynamic events like component failures, etc. Essentially, we include all aspects of a system's environment and execution that are controllable by the tester, and that affect the results of the computation performed by the system. This broad view of inputs is common practice when testing complex systems, though the nature of the inputs varies. For example, when testing compilers the inputs consist of traditional valued inputs for options and configuration directives, and also a set of interrelated source code files.

Note that we do not view distributed systems as completely controllable. In particular, we do not assume that test executions are deterministic; there are still aspects of the environment (e.g. the timing of kernel interrupts) that are outside of the control of the tester.

In general, inputs to distributed systems can be organized into four categories:

(1) **application** – valued inputs provided to each component upon initialization. These typically include data and configuration values.

(2) **topological** – aspects of the high level configuration of the system: how many components of each type are needed?

(3) **environmental** – parameters affecting the operating environment of the whole system and/or individual components. A ubiquitous environmental input for distributed systems is the behavior of the network: drop rates, latency, bandwidth limitations, etc.

(4) **temporal** – a crosscutting aspect is the timing of events. Since distributed systems are reactive in nature, and because many distributed protocols use time explicitly, changes in the timing of the above inputs can result in a completely different execution.

In this document, we refer to all inputs in the last three categories as being "distributed" inputs. Additionally, some application inputs are related to distribution when they are protocol parameters, or the like. It is distributed inputs that are our focus in this work. Whenever system-level testing is undertaken for distributed systems, inputs like these must be selected; this work attempts to move this selection from an **ad hoc** process to a rational one.

1.1 Hypotheses and Contributions

Our initial ideas in this area are driven by personal experience developing and validating distributed systems. We faced the challenges associated with deciding which topologies to use and which network behaviors to emphasize for testing. Based on this experience we hypothesize that:

Hypothesis 1 "Distributed" inputs to a software system play an important role in causing failures.

To back up this hypothesis, we present the results of a failure study conducted to understand the failure-producing scenarios of several real systems. From this study, we contribute a general method for categorizing test cases through the analysis of bug reports, and trends and observations that emerge from the data.

The use of discrete-event simulations in the design and development of distributed systems is widespread. For example, they are used to understand network protocols [AF99], engineer distributed systems [LC97, SMLN$^+$03], and improve distributed algorithms [CMH$^+$02]. They are appealing to developers because of their inherent efficiency and scalability, and because their core abstractions of **process** and **event** map neatly to the **component** and **message** concepts inherent to modern-day distributed systems. Unlike many other development artifacts, simulations seem to be used, and therefore well maintained, throughout the development process, both as early design tools and as late evaluation tools.

Given the effort invested in the construction and maintenance of simulations, and the degree to which developers trust in them, we ask whether they can increase the rigor with which distributed systems are tested, and make the following hypothesis:

Hypothesis 2 Simulations can be used within a specification-based testing regime to help developers of distributed systems define and apply effective system-level test suites.

From the research literature, it appears that there are two risks inherent to the use of adequacy criteria. Part of the challenge of empirically evaluating testing techniques is the variability in effectiveness exhibited by test suites adequate for a particular adequacy criterion. This variability is also a problem for practitioners since they generally have no reliable knowledge of the effectiveness of the suite they select. Similarly, in the published empirical comparisons of test adequacy criteria [FI98, FWH97, FW93a, BLW04, BPL04]

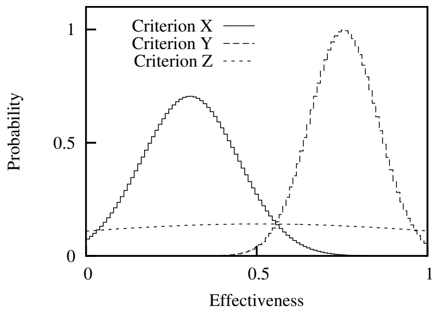


Figure 1.1: Idealized Criteria Effectiveness Distributions

there are few conclusive results in which one criterion is better than another for all experimental subjects.

Figure 1.1 depicts an idealized view of the effectiveness distributions of three fictional adequacy criteria. In this figure, the x-axis represents effectiveness or fault-detecting ability, and the y-axis shows the probability. The lines in this figure represent the probability that a test suite adequate with respect to a criterion has a given effectiveness. For example, Criterion X has an average effectiveness of 0.3, but some adequate suites have zero effectiveness, while others go as high as 0.75. The other two criteria have higher median values, but their distributions have different shapes. Using this figure the risks are defined as:

Intra-criterion – the risk of selecting an adequate test suite whose effectiveness is on the low side of the distribution. Since there is generally no way of determining the effectiveness of a selected suite, developers typically use the first adequate suite they select and consequently run the risk of selecting a suite that has low effectiveness.

Inter-criteria – the risk of selecting a criterion whose effectiveness is lower than another applicable criterion. For a given system, there are generally many adequacy criteria that could be used and their relative effectiveness is not predicable. The choice of criterion determines the location on the effectiveness axis, and therefore is central in dictating the quality of the testing process.

The execution efficiency and relative simplicity of simulation code versus implementation code enables the consideration of techniques to mitigate these risks. Therefore, we contribute a technique based solely on analysis of a DES specification to predict the relative effectiveness of test suites and criteria. To address **intra-criterion** risk, we claim that:

Hypothesis 3 Fault-based analysis of discrete-event simulations can be used to predict the relative effectiveness of adequate test suites.

For **inter-criteria** risk, we hypothesize that:

Hypothesis 4 Fault-based analysis of discrete-event simulations can be used to predict the relative effectiveness of competing criteria.

1.2 Failure Study

To motivate and inform our work, we conducted a study of failures experienced by the users of seven open-source distributed systems. The goal of our study is to understand what patterns exist in failure scenarios to guide the definition of an improved testing method for distributed systems. The raw data are the reports contained in the publicly available bug tracking databases of the systems considered. As an initial preprocessing step, we performed a coarse-grained categorization of about 1700 bug reports in order to separate those describing an actual failure of the system from those submitted for other purposes. Then, we examined the failure reports more carefully in order to categorize the nature of the inputs and observations described within.

At a high level, our results indicate that: there is empirical evidence to support the consideration of a new generation of testing methods that address the failures due to distribution; the configurations that cause user-reported failures are reasonably straightforward to construct; and generic failure observations (i.e., those for which reusable techniques can be developed) are strongly correlated to the distributed nature of system failures. The second two results in particular imply that there is a reasonable bound on the effort required to organize the testing activity.

1.3 Simulation-based Testing

A key component of this dissertation is the claim that simulations can be treated as a form of specification, and thereby used within a specification-based testing regime to provide developers with a basis for defining and applying system-level test adequacy criteria. The intuition leading to this is based on several observations.

- Simulations are used to understand and evaluate the functionality and performance of complex inter-component protocols and algorithms. They abstract away low-level details of the implementation of a distributed system, as well as details of the operational environment, yet still provide a faithful model of the expected behavior of the system in its environment.

- Simulations embody abstractions for the underlying mechanisms and environmental conditions that affect the distribution properties of systems. In addition to operating on the normal functional inputs of a system, simulations are parameterized by a set of inputs for controlling a wide range of environmental phenomena, such as message sequences, delays, and bandwidths.

- Recent frameworks for discrete-event simulation encourage simulations to be written in one or another common imperative programming language, such as C++ or Java.[1] Therefore, the simulation code itself is amenable to common program analysis techniques and tools.

[1] There are many such simulation frameworks. Examples can be found at `http://www.j-sim.org/` and `http://www.ssfnet.org/`.

Thus, a simulation is an abstract, executable specification of a distributed system, where the specification language happens to be a programming language.

In this dissertation, we demonstrate the power of a simulation-based approach to distributed-system test adequacy criteria by showing how to: (1) define adequacy criteria with respect to a simulation and (2) evaluate the criteria, again with respect to a simulation, to determine their effectiveness at causing fault-revealing failures in an implementation.

Notice that this conforms to the general idea of specification-based testing, where a fundamental premise is that a specification-adequate test suite can lead to effective testing of the implementation.

For example, consider the use of finite state machine (FSM) specifications in protocol testing [BP94, Lai02], where adequacy is established by measuring the extent to which a test suite exercises the structural or behavioral elements of the FSM specification. A state-coverage adequacy criterion may require that all states be visited at least once. Another criterion might require that the test suite produce all possible outputs by visiting all arcs in the FSM. Once the adequacy of a given test suite is established against the specification, it is simply applied to the implementation to perform the actual tests.

In our case, the structural and behavioral elements of the specification are embodied in the program code of the simulation. Therefore, as a logical first step, we examine familiar and simple adequacy criteria based on white-box code-coverage metrics, such as block coverage. (Notice the analogy to FSM-based coverage.) However, we do not imply nor require that these specification-code-coverage adequacy criteria correlate with similar implementation-code-coverage criteria. So, for example, a test suite that has an adequate coverage of the blocks in a simulation-based specification may or may not have adequate coverage of the blocks in the implementation. Any relationship is irrelevant, since the program code of each differs substantially from the other. Instead, we are interested in the relationship of simulation-code coverage to measures of **effectiveness** in causing fault-revealing implementation failures.

As a second, higher-level step in this work we also address the practical question of how to use simulation-based criteria in the most cost-effective way for each particular system. The result is a method for evaluating the relative effectiveness of competing criteria. The method is once again based on the simulation code. In particular, we perform a fault-based analysis of the simulation code to rank the relative effectiveness of multiple test suites. Then, by systematically analyzing the test suites that are adequate with respect to some criteria, we derive the ranking of the relative effectiveness of the criteria themselves.

To help understand how these techniques can be used in practice, we describe three usage scenarios:

Conventional The conventional way to use simulation-based testing is to choose a general-purpose adequacy criterion defined against the simulation code, and select a single test suite that is adequate with respect to it. In this scenario, the tester is exposed to both types of risk discussed above. The cost in this scenario is simply the cost of simulating test cases until an adequacy value is achieved.

Boosting In this scenario, the tester has somehow chosen a particular adequacy criterion, as before, but here they want to reduce the risk of picking an ineffective, adequate test suite. Thus, they select multiple adequate test suites, use fault-based analysis to rank the suites by mutant score, and apply the highest-ranked to the implementation. This usage is more costly than the conventional usage since multiple adequate suites must be selected, and each selected test case must undergo a fault-based analysis, but **intra-component** risk is reduced.

Ranking In this scenario, the tester is looking to rationally decide between multiple criteria. So, they create many adequate test suites for each candidate criterion, and use fault-based analysis to determine which criterion is likely to be the most effective. At this point, the tester simply applies the boosting usage to the adequate test suites already created for the highest-ranking criterion.

1.4 Experimental Evaluation

We validated our hypotheses and substantiated our claims through a series of experiments on three distributed systems. Two of the systems involve a set of faulty student implementations of the well-known distributed algorithms “go-back-n” and “link-state routing”. The third is MaraDNS, which is an open-source implementation of a recursive, caching Domain Name System (DNS) resolver. We experimented with public releases of MaraDNS, which consists of between 15,000 and 24,000 lines of code, depending on the version.

In our studies, we use the comprehensive experimentation and analysis method introduced by Frankl and Weiss [FW93a]. Their method involves sampling a large universe of test cases to randomly construct test suites that are adequate with respect to different criteria. Statistical inference is then used to test hypotheses about the relative fault-detecting ability of the criteria. To evaluate the different criteria, we employ a standard technique in which different testing strategies are simulated, once the failure data for each test case has been collected. The relative ranking of criteria is established by using hypothesis testing on the average mutant scores of each pair of criteria.

In support of these experiments we developed two sets of tools. The first was a thin adapter layer over the core discrete-event simulation engine provided by simjava[2] . This layer provided a more natural environment for developing simulations of distributed systems. As part of this package, we also developed analysis tools for instrumenting the simulation code so that block, branch, and definition-use test requirements could be

[2] `http://www.dcs.ed.ac.uk/home/hase/simjava/`

determined and traced during execution. The second tool set was used for executing the simulation runs, mutated simulation runs, test executions and gathering failure and cost data. Additionally, we developed a program to simulate random, black-box, and white-box testing procedures and to perform the statistical analysis.

The results of the experiments clearly show that even under the most simplistic usage scenario our approach performs significantly better than a random selection process for test suites. Moreover, the data show clearly that (1) the relative effectiveness of simulation-based criteria can be determined with a high degree of confidence by analyzing the simulation alone; and (2) this predictive power can be harnessed by a tester to make rational decisions about the expected effectiveness of test suites and criteria. This is an important result that gives developers a powerful new tool for organizing the testing activity and for tailoring that activity to the distributed system at hand.

1.5 Related Work

The ideas presented in this dissertation build upon several fields of software testing and software engineering research.

The failure study is unique in that it is aimed at analyzing failure-producing scenarios, not the root causes of bugs as in studies by Fenton and Ohlsson [FO00] and Ostrand and Weyuker [OW02]. In orthogonal defect classification (ODC) [$\text{CBC}^{+}92$], part of the classification addresses the failure-causing scenario which is similar in intent to the classification we perform, but at a coarser level.

Our use of operational models for testing is inspired by the extensive literature on protocol testing surveyed by Bochmann and Petrenko [BP94] and Lai [Lai02], The adequacy criteria we use are discussed in the literature on low-level white-box test adequacy criteria [ZHM97, FW88].

Our experimental setup is a direct instantiation of the method promoted by Frankl and Weiss [FW91] and Briand et al. [BLW04].

Finally, our initial understanding of the risks and limitations of testing with adequacy criteria comes from theoretical work by Hamlet [Ham94] and Hamlet and Taylor [HT90], and by analytical studies by Ntafos and colleagues [DN84, Nta98] and Frankl and Weyuker [FW93c].

1.6 Document Organization

The rest of this document is organized as follows. The next chapter presents the failure study. Chapter 3 describes our simulation-based testing technique, and the related fault-based analyses. Chapter 4 describes the experimental evaluation of these ideas. Chapter 5 provides background information on the use of test adequacy criteria, specification-based testing, fault-based analyses and testing of distributed systems. Chapter 6 concludes and describes some directions for future work.

Chapter 2

Failure Study

This chapter presents a study aimed at addressing two fundamental questions about the development of testing techniques specific to distributed systems: Is it necessary? Is it feasible?

In the distributed testing literature, it is sometimes asserted that distributed software is inherently different to test than non-distributed software [GM99]. Some of these differences are related to problems in selecting effective test data for non-standard inputs, such as the topology of the distributed components and network characteristics of bandwidth or latency. Other difficulties arise from the challenges associated with defining and observing system failures in the face of nondeterminism and a possibly heterogeneous distributed testbed. However, in one of the few published case studies where traditional software testing techniques are applied to a distributed system [LS01], the techniques perform reasonably. In light of this, we first wanted to determine how frequently failures were caused by the effects of distribution.

Furthermore, in considering a testing method, we must find some structure to the problem that will reduce the effort required of a tester to navigate through an otherwise intractable space of potential test cases. If no such structure can be found, then there is simply no basis for defining a method. Thus, we seek to understand the extent to which there are patterns in failure scenarios, such that one could justifiably define a testing method for distributed systems.

Conducting a study of distributed-system failures presents a number of challenges, but perhaps none more so than simply finding concrete data. While gaining access to details about any aspect of an organization's software development process is not easy (particularly for a commercial organization), reviewing product failure information is virtually impossible due to its sensitive and potentially damaging nature. Fortunately, many popular distributed applications are developed as open-source projects having associated with them publicly accessible defect-tracking systems. It is the failure reports contained in the defect databases of open-source distributed systems that are the raw data of this study. The study population consists of seven systems representing a broad spectrum of distributed-system styles: peer-to-peer, federated, and client/server, as well as a component library.

The use of open-source software is not only convenient, it is an asset, since most of these projects function with little or no dedicated QA personnel, in contrast to traditional closed-source commercial systems. Therefore, open-source software is generally released having only been tested by developers, providing us with a unique opportunity to isolate failures that escape this class of testing. Furthermore, although many open-source developers consistently write and apply unit tests, they rarely if ever document and publish them, even within the defect-tracking systems. Under these circumstances, the failure reports originate mainly with users, who naturally concentrate on the features and configurations of most importance to them. In a sense, the users are acting as surrogate QA system-level testers for the purposes of this study, admittedly providing a broadly user-centric, as opposed to a strictly requirements-centric, prioritization of test cases. The question here is not how well those pseudo-QA testers may have performed their job, but rather whether there is something we can learn from the failures they did manage to uncover.

The testing activity involves two key tasks: (1) selecting inputs that are effective at causing failures and (2) constructing and applying oracles that are capable of recognizing that a failure has occurred. The method we developed for this study requires us to identify and classify each scenario described in a failure report in terms of the overall configuration of components, the specific inputs needed to drive the system to failure, and the technique used to observe the failure. Further, we attempt to determine whether each failure scenario was truly distributed or simply multi-component. A multi-component scenario involves components providing input to each other, while a truly distributed scenario is a multi-component scenario that also involves properties such as concurrency, platform heterogeneity, unreliable communication, delays, timeouts, or the like. This is a critical distinction that helps shed light on the underlying nature of a failure and the testing techniques that might expose it. We also distinguish between failure observations that are generic, in the sense that they are largely applicable to most kinds of systems (e.g., detecting system crashes), and those that are specific, relating to the functionality of the system (e.g., incorrect content of a message). Clearly, generic observations can be supported by reusable techniques, while specific observations often need purpose-crafted techniques.

```
MX chaining in RemoteDelivery is broken.  Basically, the code in place assumes
that a given MessagingException generated while sending mail doesn't merit a
retry unless it encapsulates an IOException.  This doesn't appear to be the
exception generated when a server refuses a connection.  So the exception is
rethrown, and the other SMTP servers are never tried.
```

Figure 2.1: James Report #66

As an example of the classification we perform, consider the mail server bug report contained in Figure 2.1. This figure displays the text of James report #66, entitled “MX Chaining in the RemoteDelivery mailet is broken”. The underlying fault is in the logic that deals with retrying email delivery when a domain has multiple mail exchangers (MX DNS records). The steps needed to reproduce this failure, inferred from the description, are:

(1) start an SMTP server (not necessarily James);

(2) configure a DNS server with a test domain that has two MX records: the first points to an invalid address; the second points to the server started in step (1);

(3) run a James server configured to use the DNS server started in step (2);

(4) send an email, through the James server, to an account at the test domain using an SMTP client;

(5) observe that the SMTP server started in (1) does not receive the email.

The analysis of this report includes the identification of the four inputs and classification of the observation. We refer to this report and its classification throughout the chapter.

Our observations resulting from this study hold several important implications for the development of advanced testing methods. First, there is empirical evidence to support the consideration of a new generation of test methods that address the significant number of failures due to distribution. Second, the configurations that cause user-reported failures are reasonably straightforward to identify. This means that testers are not required to find exceedingly complex or exotic configurations in order to expose the failures that users would typically encounter. This, in turn, means that there is a reasonable bound on effort. Third, generic observations are strongly correlated to the distributed nature of system failures, which means that there is again a reasonable bound on effort, in this case the effort required to define and construct test oracles.

These results give us some confidence that it is both necessary and feasible to consider a testing method that targets distributed systems.

2.1 Subject Systems

Our study reviewed failure reports from seven distributed systems. We selected these systems to represent a broad spectrum of distribution styles: peer-to-peer (P2P), federated, and client/server. We also included in our study population a component from a distributed component library.

2.1.1 Peer-To-Peer

For our purposes, the defining characteristic of P2P applications is their ability to operate without the resources of a centrally controlled server application. Peers generally discover each other, and then cooperate to deliver services.

The nature of P2P applications exacerbates the already complex task of conducting system-level tests of distributed systems. For example, schemes for operating in the presence of routers acting as firewalls and employing network address translation (NAT) are almost always adopted; testing these features requires a fairly complicated testbed. Also, since peers are discovered instead of being manually configured, a tester must be careful that the desired topology is detected when arranging a scenario.

2.1.1.1 Gtk-Gnutella

Gnutella is a popular protocol for decentralized file sharing. Gtk-Gnutella is a Unix-based Gnutella client. A client joins the network by locating a single existing member who can then help bootstrap the new node by sharing its list of known addresses. A decentralized system of web applications collectively known as the Gnutella Web Caching System (GWebCache) stores lists of active hosts and other caches; this too is used for bootstrapping.

The Gnutella protocol runs over both TCP and UDP and includes control messages for joining the network, issuing queries, and responding to queries. HTTP is used to download files from peers. Gtk-Gnutella also supports the GWebCache protocol for retrieving and updating lists of active peers.

In Gnutella, nodes can operate in "ultrapeer" mode or "leaf" mode. Ultrapeers are typically run on machines having access to large amounts of bandwidth and other resources. Ultrapeers are able to respond to query messages on behalf of leaf nodes to which they are connected, thereby reducing the network usage for leaf nodes.

Gtk-Gnutella is written in C, and supports a rich graphical user interface via the GTK library. It has a large number of configuration options related to interacting with peers, joining the network, and controlling the amount of bandwidth used for uploads and downloads. It is the only application we considered with a dedicated (i.e., non-web-based) graphical user interface.

2.1.1.2 JBoss Clustering

JBoss is a fully featured enterprise Java (J2EE) application server. A single instance of JBoss can be used as a servlet and Enterprise JavaBean (EJB) container, and provides the typical services of a J2EE container, including management of transactions, security policies, and threading. An advanced feature of JBoss is its support for clustering of multiple JBoss instances to provide load balancing and redundancy. While JBoss itself is typically considered to be a canonical server application, we consider its clustering feature in the P2P category, since the clustering module is completely decentralized and supports automatic discovery of cluster members who cooperatively provide a service.

JBoss clustering supports replication of HTTP session and EJB state, Java Naming and Directory Interface (JNDI) replication, failover and load balancing of JNDI and EJBs, and cluster-wide deployment of J2EE components. JBoss is written in Java.

2.1.2 Federated

Federated applications are similar to P2P applications in that they are able to cooperate with other programs to collectively deliver a particular service. They differ from P2P applications in that they also provide services when running alone, and they must be manually configured to cooperate with other programs.

2.1.2.1 Elvin

Elvin[1] is a content-based publish/subscribe communication middleware application [SA97, SAB$^+$00]. In its most basic configuration, an Elvin application consists of a publishing client and a subscribing client both connected to a single router. The subscriber provides the router with information about its content preferences, and whenever a publisher sends a notification that matches these preferences, the router forwards it to the subscriber. Elvin supports federation of routers located in different administrative domains. It also supports failover, in which routers can cooperate to provide replication for advanced quality-of-service features. Each Elvin router can optionally be managed through a web interface. The core Elvin router is written in C and runs on both Windows and Unix platforms. Client bindings are available in a number of languages.

2.1.2.2 Squid

Squid is an advanced caching proxy written in C that is able to proxy communications for the HTTP/HTTPS and FTP protocols, among others. Squid provides advanced support for integration with other web caches to improve performance and reliability. Squid can also operate as an HTTP accelerator for a standard web server. It includes built-in support for management via SNMP, and is able to operate as a caching DNS resolver.

2.1.3 Client/Server

In the traditional client/server model, a single server program services multiple client programs without cooperation from other programs. However, there are few server programs that are truly independent and do not use other services, such as DNS,

[1] Depending on the interpretation of the term, Elvin may not be considered truly "open-source". However, it has a publicly accessible defect tracking system, which is why it was included in this study.

sometime during their operation. Therefore, the two applications we selected as servers primarily operate alone to provide a service, but in some configurations may also act as clients of other servers.

2.1.3.1 AOLserver

AOLserver is a web server designed to be highly scalable and configurable. It is developed and used by America On-Line. The server core is written in C and is configured and programmed using the Tcl API that it exposes. AOLserver also supports server-side programming through Tcl, and integrates with various popular relational databases. Management of AOLserver is accomplished through a telnet-like interface.

Through the server-side programming API, AOLserver can act as an email client, and as an HTTP/HTTPS client to other (not necessarily AOLserver) web servers.

2.1.3.2 James

James is an email server written in Java. It supports SMTP, POP3, and NNTP. James can also be configured to act as a POP3 client, and supports the use of popular relational databases for storing email and metadata. Management of a running James instance is accomplished through a telnet-like interface. As an email server, James makes extensive use of the DNS system, and has a built-in DNS resolver/cache for this purpose. James can be customized through its Mailet interface.

2.1.4 Distributed Component

With the advent of server frameworks, distributed application programming is often accomplished through the development of components designed to operate within those frameworks. The components are similar to basic code libraries, but they often support special interfaces required by the framework and they must often be aware of the design of the advanced services provided by the framework.

2.1.4.1 OpenSymphony Workflow

The OpenSymphony Workflow component provides advanced support for workflows. The component is customized to a particular workflow through a configuration file, and supports various popular databases for persistence of workflow histories. The component is written in Java.

OpenSymphony Workflow can be executed in a standalone fashion, but it is also intended to operate within a J2EE container as a web application (servlet) or EJB.

2.2 Empirical Method

The raw data used in the study are the reports contained in the defect-tracking systems associated with the systems we considered. The central challenge faced in conducting the study was to transition from the informal application-specific descriptions submitted by users to generalizable descriptions of a test scenario in an objective manner. The method we used consists of four distinct activities. First, among all user reports, we isolate the ones describing failures that are suitable for our study; then, for each suitable report, we describe the architecture of the system in the reported scenario; we classify the inputs that cause the failure; and finally we classify the observations that highlight the failure.

2.2.1 Report Classification

Defect-tracking systems are used for many things, and particularly in open-source projects some of the uses are not necessarily related to reporting failures. Therefore, our first challenge was to identify a subset of the reports that are relevant to the scope of our study. For a report to be included in our study it must satisfy the following criteria:

- it describes a system-level failure;

- it describes a run-time failure; and
- it is verified by a developer.

Our initial set of reports for each application consisted of all “closed” reports. Using the querying features of the defect-tracking systems we were able to eliminate some extraneous reports by only considering those marked as “fixed”. However, such queries were not available for all defect-tracking systems, so for these systems all closed reports were classified manually.

Report Type	**Description**	%
`INSCOPE`	in-scope report	34.17
`IMPROVEMENT`	code improvement	21.96
`STATIC`	non-run-time failure	14.47
`NEI`	not enough information	11.18
`RFE`	request for enhancement	7.83
`UEI`	user error / ignorance	3.92
`UNIT`	unit test	2.84
`WORKSFORME`	developer cannot replicate	2.72
`3RDPARTY`	failure is in external software	0.91

Table 2.1: Classified Report Types

Table 2.1 shows the report types we used for this initial preprocessing step. In general, reports are eliminated from consideration because they (1) do not describe a failure (marked as `IMPROVEMENT`, `RFE`, `WORKSFORME`, `3RDPARTY`); (2) describe a compilation or documentation defect (marked as `STATIC`); (3) do not provide enough details about the behavior that causes the failure (marked as `NEI`); or describe replication that cannot be readily mapped to system-level actions (marked as `UNIT`).

Most report-type classifications are a straightforward matter of reading through the initial report and the subsequent dialog between developers and users. However, there is some subjectivity in making a `NEI` or `UNIT` classification. The following two examples should help clarify how these categorizations are used.

```
The following code generates a java.lang.StringIndexOutOfBoundsException
exception: -

MailAddress mailAddr = new MailAddress("A@B.");

The exception is as follows: -

Jul 05 16:59:59 2001: java.lang.StringIndexOutOfBoundsException: String index
out of range: 4
Jul 05 16:59:59 2001: at java.lang.String.charAt(String.java:503)
Jul 05 16:59:59 2001: at org.apache.mailet.MailAddress.<init>
(MailAddress.java:102)

I can only generate this error if the last character is a fullstop.
```

Figure 2.2: James Report #5

Figure 2.2 contains the initial description of a defect report for James. This report was classified as UNIT because it makes no mention of the user-level inputs needed to reproduce the failure or the user-level observations that would be needed to recognize the failure.

```
#0  0x40197d21 in __kill () from /lib/libc.so.6
#1  0x40197996 in raise (sig=6) at ../sysdeps/posix/raise.c:27
#2  0x401990b8 in abort () at ../sysdeps/generic/abort.c:88
#3  0x8061bce in xassert (msg=0x809c4e2 "auth_user_request != NULL",
    file=0x809c4c0 "authenticate.c", line=478) at debug.c:250
#4  0x8050778 in authenticateAuthUserRequestLock (auth_user_request=0x0)
    at authenticate.c:478
#5  0x8058fd3 in clientRedirectDone (data=0x845de90,
    result=0x8211f68 "http://127.0.0.1:2968/") at client_side.c:314
#6  0x8080f34 in redirectHandleReply (data=0x845e600,
    reply=0x8211f68 "http://127.0.0.1:2968/") at redirect.c:70
#7  0x806e7e4 in helperHandleRead (fd=7, data=0x8211f20) at helper.c:691
#8  0x8061050 in comm_poll (msec=878) at comm_select.c:434
#9  0x807a8a8 in main (argc=2, argv=0xbffff7ac) at main.c:720
```

Figure 2.3: Squid Report #198

Similarly, Figure 2.3 shows a report that simply consists of a debugger stack trace that was classified as NEI. In this report, there is enough detail for the developers to fix the problem, but there is not enough information to reconstruct the scenario that leads to the failure.

Table 2.2 shows that after preprocessing we are left with 602 in-scope failure reports.

There are three major aspects of each scenario that are summarized for each report: (1) architecture, (2) inputs, and (3) observations. The remainder of this section describes them in detail.

2.2.2 Architecture

The architecture of a scenario refers to the number and organization of any components involved in the failure. The architecture is separated into the configuration of

System	**Total Reports**	**In-scope Reports**	**%**
AOLserver	196	69	35
Elvin Router	149	67	45
Gtk-Gnutella	337	115	34
James	203	85	42
JBoss Clustering	36	22	61
OpenSymphony	289	37	13
Squid Cache	552	207	38
	1762	602	34

Table 2.2: In-Scope Reports by System

the application itself, the configuration of any supporting components, collectively referred to as the **harness**, and properties of the underlying network. The harness further is divided into **drivers** and **services**, where a driver component is one that initiates communication with the system, while a service component instead awaits communication from the system. In James #66, shown in Figure 2.1, the scenario architecture consists of one instance of James, one SMTP client driver, one SMTP service, and one DNS service. There are no special network properties described.

In a few scenarios, the underlying network was required to have certain characteristics. For example, a JBoss Clustering scenario requires that the underlying network not support IP multicast. We believe that manipulation and configuration of the network is an important class of inputs to these systems that is probably not sampled extensively by end users of these systems.

# Components	**%**
1	19.60
2	41.20
3	33.39
4	4.82
5	1.00

Table 2.3: Architectural Summary

The scenario architectures we identified involved between one and four application instances, between zero and three driver components, and between zero and three service components. Only seven scenarios required manipulation of the underlying network. Table 2.3 shows a summary of the architectures described in the reports. The first column in this table reflects the total number of components involved in a scenario. For example, a scenario involving an AOLserver instance and an HTTP client, and a different scenario requiring two Gtk-Gnutella instances are both considered to involve two components. One notable feature of the data in Table 2.3 is that over 80% of the scenarios involved more than one component.

2.2.3 Inputs

In addition to the scenario's architectural elements, we also identify and categorize the scenario's inputs. We take a broad view of "inputs" in which we consider all the installation/configuration settings and run-time actions needed to reproduce the reported failure. As with the architectural classification, the inputs are separated by whether the locus of the input is the system, one of the harness elements, or the underlying network. The input categories presented in Table 2.4 were derived from an examination of the reports themselves. In particular, new categories were created as needed, to accommodate an aspect of a scenario that did not fit neatly within the existing ones. Although it is not possible for us to claim that the categories we came up with are sufficient to represent all possible inputs for distributed systems, they were sufficient to handle the disparate systems included in the study. Furthermore, the set of categories is quite stable in that most new categories were created to accommodate the first few systems; very few categories were added late in the process.

We consider both setup inputs and run-time inputs since both of these affect the behavior of the system. Setup inputs are those steps that are taken to get components installed and configured as required by the scenario. This includes building the system, installation, configuration, customization and finally, execution. Run-time inputs are

Locus	**Category**	**Description**
System	Configuration	configuration activities
	Customization	API programming activities
	Execution	execution style or command line options
	FileData	data files contents
	FileSystem	changes to files or file system structure
	Installation	compilation and installation activities
	OperatingSystem	changes to the operating system
	Platform	specific software platform
	UserInterface	interaction with the system via interface
Harness	Behavior	specific network-related behavior
	Configuration	driver/service configuration
	Platform	specific software platform
	SendMessage	send a particular message
Network	Manipulation	alter underlying network

Table 2.4: Input Categories

any actions performed after the programs have been started.

Most categories listed in Table 2.4 are self explanatory. “Customization”, which appears as a system input, requires further explanation. Customization refers to the creation of any plug-in-like software that runs within a component in the system or the harness. For example, many of the failures associated with AOLserver required that a server-side script be created with particular contents, and hence this was classified as a customization.

Referring to the report in Figure 2.1, the inputs categorized for James #66 are: (1) **Harness:SendMessage** for the DNS response and email and (2) **System:Configuration** for the DNS configuration of James.

The initial categorization left us with a brief description of each scenario and the input categories that are present in the scenario. Next, we consider each input independently and determine whether it was truly “distributed”. The purpose of this categorization is to enable us to distinguish between scenarios whose architecture is multi-component, simply because the system accepts input at run time through the network interface, and scenarios that actually involve the distributed nature of the system. Inputs that are considered to be distributed fell into these three rough categories.

- **Inherent:** e.g., concurrency; relative differences in processing speed, latency, and bandwidth; unreliable communication; platform heterogeneity.
- **Accidental:** e.g., addresses, ports and well-known port assignments; networking resource limits.
- **Error simulation:** e.g., connection timeouts and termination; invalid or truncated messages.

A scenario was classified as distributed if any of its constituent inputs are distributed. For example, a number of defects in AOLserver are demonstrated by creating a server-side script with particular characteristics and then executing the script by sending the appropriate HTTP request to the server; these are all classified as non-distributed because the HTTP client program is only used as a vehicle with which to send inputs to the server. By contrast, an AOLserver scenario that required a client to connect to the HTTPS port and simply wait for the connection to timeout is classified as distributed since in this scenario the HTTP client is used to exercise an aspect of the protocol.

2.2.4 Observations

For a scenario to produce a failure, the anomalous behavior must be observed during execution. For each scenario, we described and classified the observation that was made. The categories used are summarized in Table 2.5. As we did for inputs, we separated our observation categories by whether they are applied to the system components, to the harness components, or to the network. For example, if a failure is observed by noticing that the structure or content of a message exchanged by **system components** the observation is classified as being applied to the “System”; if, on the other hand, the content or structure of a message provided to an entity outside the system boundary (i.e. part of the harness), then the observation is labeled “Harness”.

The fourth column in Table 2.5 shows the percentage of all 602 observations that fell into each category.

In our initial observation classification, we also produced a succinct summary of each observation. These summaries were then used in a second refinement phase to further categorize each observation category. In all, we identified 29 different subcategories of observations. Due to its length we do not show the entire list. As an example, the subcategories of the **Message** category are as follows.

- **Absence:** a message was not sent or received when it should have been.
- **Content:** the data transmitted in a message were somehow invalid.
- **Format:** the message format was incorrect.

Locus	**Category**	**Description**	**%**
System	Process	application process	28.57
	LogOutput	log or console messages	17.77
	State	internal state	12.96
	File	files or directories	6.48
	Performance	resource usage	2.16
	OperatingSystem	operating system elements	<1
	Message	network messages	<1
Harness	Message	network messages	25.41
	Networking	networking API	1.82
	State	internal state	1.16
	Performance	resource usage	<1
	LogOutput	log or console message	<1
Network	Transport	transport-layer aspects	<1

Table 2.5: Observation Categories

- **Presence:** a message was sent or received when it should not have been.
- **Type:** a message of a particular type was was observed.

The observation in James #66 was classified as a **Message:Absence**, since the SMTP service should have received an email message.

Once we identified all the subcategories, each subcategory was labeled as either **generic** or **specific**, according to the level of specificity of observation that was needed. Generic observations are those that can be applied to virtually any software system (e.g., application crashes, errors appearing in standard system logs, or memory leaks), and observations that are tied to the application being tested, but still at a high level (e.g., the presence or absence of network messages). By contrast, specific observations involve detailed judgments about computed data values (e.g., contents of message fields). These observation labels are the basis for our analysis of observations discussed in the next section.

2.3 Analysis and Discussion

The analysis presented in this section is centered around (1) determining what general trends exist with respect to the complexity of the failure scenarios and (2) examining the specificity of observations that detect failure. We examine both of these aspects from the perspective of two measures of a scenario's distributedness: (1) how many different components must cooperate to cause the failure and (2) whether or not the scenario requires any "distributed" inputs as described in Section 2.2.3. This leads us to distinguish between multi-component and distributed scenarios in the discussion below.

After classifying each input as distributed or not we were able to determine that 17% of the 602 scenarios required truly distributed inputs. While 17% is not a particularly high value, since the scenarios in this study were reported by users it is not

System	**# Dist.**	**# Scenario**	**%**
AOLserver	14	85	16
Elvin Router	13	69	18
Gtk-Gnutella	23	115	20
James	12	67	18
JBoss Clustering	2	37	5
OpenSymphony	1	22	4
Squid Cache	38	207	18
	103	602	17

Table 2.6: Distributed Scenarios by System

unreasonable to treat this as a lower bound. Failures involving concurrency and timing issues are notoriously hard to replicate even for professional testers. Additionally, as the data in Table 2.6 show, the percentage of scenarios that are distributed is relatively consistent across all the systems we studied, lending credence to our belief that this class of failures is fundamental to distributed systems. In any case, our conclusion is that a significant fraction of all the failures is directly related to the distributed nature of the systems. This observation suggests that specialized testing methods and adequacy criteria are needed to address distribution.

The core assumptions underlying our analysis are that a failure scenario with a larger number of components is inherently more difficult for a tester to perform, and that the space of possible test cases grows rapidly as scenarios of higher complexity are considered since each component can be configured independently. Table 2.3 shows the overall architectural complexity of the reported scenarios in terms of the number of components involved. One interesting observation from the data in this table is that approximately 75% of the scenarios we encountered involved either two or three components, and that 93% of the scenarios can be accounted for with simple scenario architectures involving three or fewer components.

Table 2.7 shows the same data for the distributed scenarios. It should be pointed out that just over 12% of the distributed scenarios actually only required a single system

# Components	**%**
1	12.62
2	36.89
3	41.75
4	6.80
5	1.94

Table 2.7: Architectural Complexity for Distributed Scenarios

instance. Most of these defects were related to misconfiguration of ports, addresses and DNS related settings and are thus part of the "accidental" distributed inputs. At a high level, comparing the values in Table 2.7 with those in Table 2.3 confirms the observation made above that even for the scenarios requiring distributed inputs, more than 90% of the scenarios we encountered involved three or fewer components.

# App. Instance	**# Drivers**	**# Services**	**%**
1	1	1	27.57
1	1	0	24.92
1	0	0	19.93
2	0	0	12.62
1	2	0	3.49
1	0	1	3.16
1	1	2	2.33
2	1	0	2.16

Table 2.8: Scenario Diversity

Table 2.8 shows the top 8 scenario architectures as represented by the mix of application instances, driver instances, and service instances. Altogether, these eight scenarios account for 96% of those encountered, while the top four alone account for 85%. It is encouraging to note that the four most common scenario architectures involve only simple harnesses, if at all.

While the inputs to a test determine if a fault will occur, for the scenario to generate a failure, the fault must be observed. Figure 2.4 shows the percentage of observations at each level of architectural complexity for the two main observation labels described

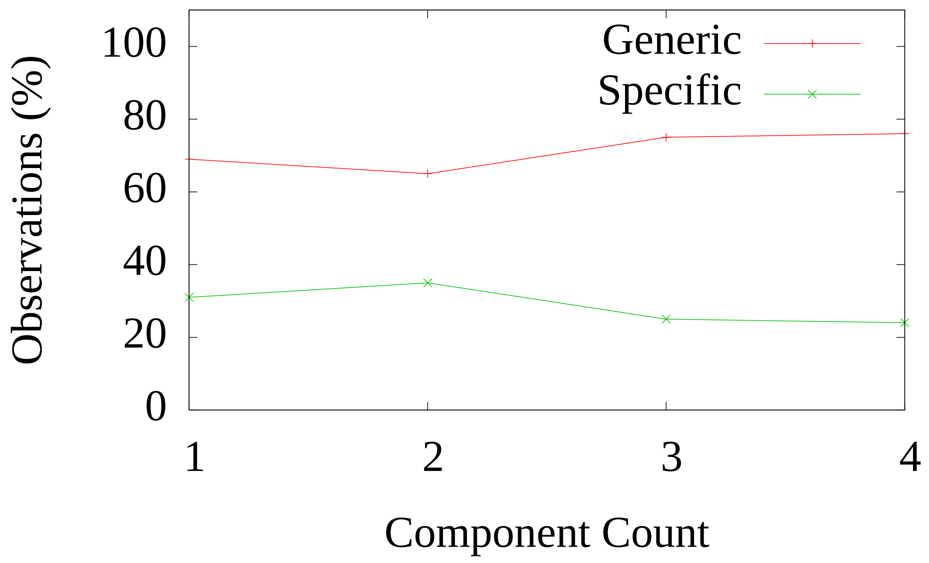


Figure 2.4: Observation Complexity versus Distribution

in Section 2.2.4. Data are only shown for scenarios with four or fewer components since there are so few scenarios with five components. This figure shows the encouraging result that the proportion of **generic** observations and **specific** observations remain steady as the scenarios become more distributed.

Within the generic observations, we further identified three groupings: (1) catastrophic, (2) universal, and (3) general. Observations of catastrophic failures are the easiest to make since, by definition, the application stops running. Universal observation are those that can be applied to virtually any software system (e.g. error messages in system logs), while general observations are system-specific, but still broadly applicable within the space of possible behaviors of the system (e.g. message format violations).

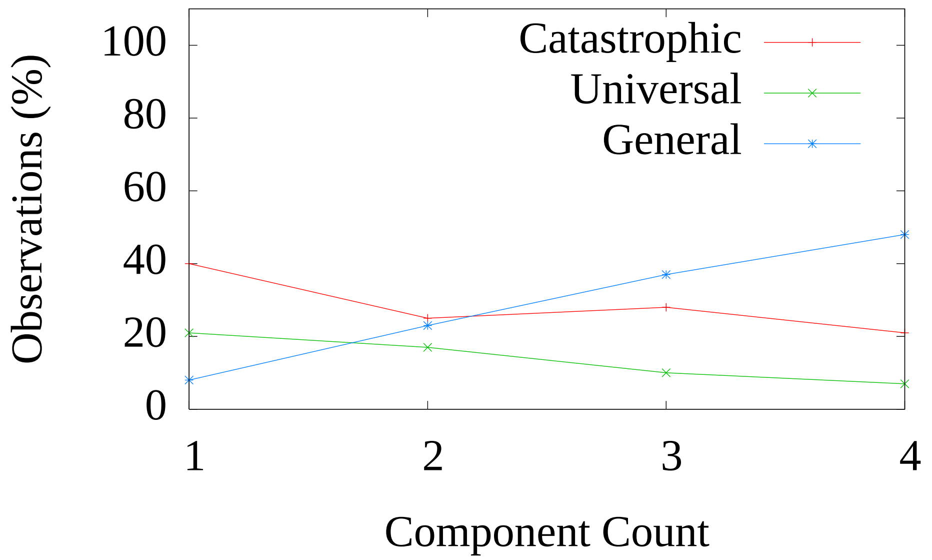


Figure 2.5: Generic Observations versus Distribution

Figure 2.5 shows that observations of catastrophic failures are most prevalent in single-component scenarios, and that their frequency levels off at roughly 25% for scenarios with more components. The most striking feature of Figure 2.5 is the rise of the general observations as the scenario complexity increases.

2.3.1 Implications for Testing

The prevalence of failures that are directly attributable to the distributed nature of systems suggests that specialized testing methods are needed to address distribution. As for the complexity of these methods, the implications of our two main findings are quite encouraging.

First, our analyses of the structural complexity of scenarios uncover the encouraging finding that the vast majority of scenarios are accounted for with relatively simple architectures. This is even true of scenarios that explicitly include inputs related to the distributed nature of the system. This implies that testers can reasonably prioritize their attention on the simple configurations.

Next, while the proportion of very specific observations appears to remain relatively constant at about 30%, the need for general purpose oracles for a particular system appears to increase as scenarios involve more components. This finding also bears out common sense which dictates that the failures in systems with more communication and interaction will be more subtle to identify, and that techniques and methods for developing system-level test oracles become more important in these cases.

Finally, at a higher level, the experience of conducting this study and attempting to impose some level of structure and categorization on the space of distributed system failures is encouraging. We were able to analyze a large number of failure reports from seven disparate systems using quite a small set of categories and a relatively simple model of the scenario architectures. All of this implies that attempting to develop systematic and rational approaches to high-level testing of distributed systems is not futile, and that general-purpose methods, tools and techniques can be developed with at least the possibility of making a positive impact on the quality of such systems.

2.3.2 Threats to Validity

While we feel confident that the results and their implications found through this study are reasonable, it is important to recognize threats to their validity.

We state explicitly that we are not claiming that our results can be used to draw conclusions about the testing of all distributed systems; we are only claiming that our study shows that there are some systems to which our analysis applies, although we have tried to look at systems that represent a diversity of distribution styles. There are a number of reasons why our results might not generalize. For example, the systems considered in this study all result from open-source development projects that would be expected to devote fewer resources to quality assurance than would commercial projects. Thus, failures that may be caught through standard means might have escaped into the field.

The failures are only those that have been reported using the defect-tracking systems; it is highly likely that failures are found and fixed by developers while using the system themselves, or that failures are seen by users, but they simply upgrade to the newest release or avoid the problem in some way without reporting it. Therefore, the study is limited to drawing conclusions about the failures that were actually reported by users. We believe that this probably limits the complexity of the scenarios that are included in the study, and it is the primary reason that we consider the results to be applicable to a test goal of exposing major user-visible failures. In other words, even if all the failures mentioned in this study were fixed, there will likely still be failures in the systems that can only be found through scenarios that are more complex than those examined here.

A major threat to validity is the objectivity of the analyst. As pointed out by El Emam and Wieczorek [EW98], this is a problem with all empirical defect classifications, since it is by nature a subjective activity. In our study, all classifications were performed by a single analyst, and the results might have been unwittingly skewed in some direction. The process that we have established guards against this by breaking the classification into small steps so that the analyst can consider each piece independent of its scenario description. In the future, we hope to use a collaborative approach to defect-report classification, where different analysts classify reports independently, and conflicts are resolved anonymously.

Another major threat to validity is the content of the defect reports themselves. It is possible that a report contains too much or too little information. If too many details about a scenario are included in an initial report, and the developer does not make a note of this, the analysis of the report will include more specifics than are really warranted. Conversely, if the initial report does not contain enough detail about the scenario to reliably duplicate the failure, but the developer does not use the defect-tracking system to gather the necessary information from the reporting user, then the report will be given a less-specific classification than is warranted.

2.4 Summary

In this chapter we present the results of an empirical study undertaken to understand and structure the space of failure scenarios reported by users of distributed systems. Our intent is to motivate new testing techniques tailored to distributed systems by presenting patterns, commonalities, and correlations in the user inputs and observations described in defect reports. A secondary goal of our study is to determine how frequently the distributed nature of an application comes into play when system failure occurs. Our results indicate that: a new generation of test-adequacy criteria are needed to address the failures that are due to distribution; the configurations that cause user-reported failures are reasonably straightforward to identify; and generic failure observations are strongly correlated to the distributed nature of system failures.

A secondary contribution of this work is the empirical method we developed. As far as we know, this is the first empirical study of its kind and the method is generally applicable to many varieties of software. In particular, we believe that the classification activities and input and observation categories identified could provide a basis for other defect report analyses aimed at understanding failure scenarios. Additionally, critical examination of defect reports is not a purely academic endeavor; practical techniques derived from our empirical method might provide QA managers with a means of understanding shortcomings of their team's testing processes, analogous to the way root-cause analysis can help development teams improve and evolve their processes.

Chapter 3

Simulations and Testing

Discrete-event simulations (DES) are commonly used during the design and development of distributed systems. Traditionally, simulations are used to help understand the behavior and performance of complex systems. Here we are interested in using them to help guide testing.

Discrete-event simulations are organized around the abstractions of **process** and **event**. Briefly, processes represent the dynamic entities in the system being simulated, while events represent a stimulus applied at a particular time to one of the running processes. When simulating distributed systems, processes are used to represent the core components of the system, as well as environmental entities such as the network or external systems. Events represent messages exchanged by the components and can be thought of as generic structured data types. Virtual time is advanced explicitly by processes to represent "processing time" and advanced implicitly when events are scheduled to occur in the future. To run a simulation, processes are instantiated, initialized, and arranged into a particular configuration that is then executed. A simulation executes until there are no longer events scheduled to occur.

As a brief example, consider a simple client/server system designed to operate in a network environment with unreliable communication. The simulation of this system consists of three process types, **Client**, **Server**, and **Network**, and two event types, **Request** and **Response**. The **Network** process is used as an intermediary through which events between clients and servers are scheduled. Network latency is implemented in the simulation by having the **Network** process control the scheduling of event deliveries. The unreliable nature of the network is implemented by coding the **Network** process to probabilistically drop events by ignoring them. A given configuration might include four process instances: **s:Server**, **c1:Client**, **c2:Client**, and **n:Network**, communicating using an arbitrary number of **Request** and **Response** events.

Clearly, the simulation code of this example system can be used to experiment with network latencies and drop rates under different configurations, as a means to predict overall performance, and to evaluate scalability and other properties. But, how can the simulation code be used for testing?

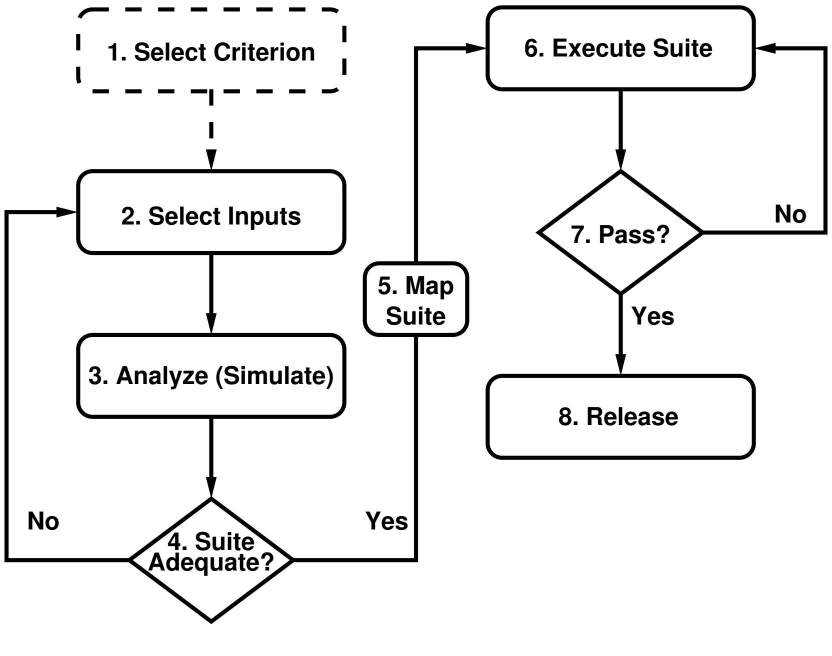


Figure 3.1: Specification-Based Testing Process

Figure 3.1 depicts a simple and generic specification-based testing process. As a first step, the tester selects a particular adequacy criterion to organize the rest of the process; we defer discussion of this difficult decision until later, for now assume that a criterion is being used and the tester must select a test suite that will satisfy it. A test suite is composed of **test cases**, each one consisting of an input vector that includes direct inputs to the system, representing functional parameters, as well as inputs to the environment, representing environmental conditions. In the figure, actions 2–4 are repeated until the criterion is satisfied.

Once a suite has been selected, the tester uses it to test the implementation by first mapping input vectors into the implementation domain (step 5), and then executing each test case on the implementation (step 6). The system is ready for release once it passes all test cases.

The simulation code plays the role of the specification in specification-based testing (i.e., during analysis). Given a candidate test case, an corresponding simulation is configured and executed to gather coverage data. These data are then aggregated for the test suite to determine adequacy. The process by which individual test cases are created or generated is outside the scope of this dissertation. Similarly, we neither propose nor discuss any specific strategy by which the developer might search the space of test suites to find an adequate one; our concern is with the decision process, not the search process.

At a high level, our approach rests on two ideas. The first idea is to use the simulation code and simulation executions as a basis to formulate general-purpose and/or system-specific test adequacy criteria. For example, a general-purpose criterion might call for statement coverage of the simulation code of all non-environmental processes (**Client** and **Server** in the example above), or a system-specific criterion might require that each event type be dropped at least once during a simulation run. Once a criterion is defined, the developer can evaluate the adequacy of a test suite by running the test cases in a suitably instrumented simulation.

Step 1 in Figure 3.1 requires that the developer choose a single adequacy criterion. However, the developer may not have enough information to make this decision with confidence. This might be because no good criteria are available or known, due to the lack of experience with a particular system, or because the developer cannot decide which criteria to adopt out of a set of plausible candidates. Even when a criterion has been selected, the developer might be concerned with selecting an ineffective test suite.

Therefore, the second idea is to provide the developer with a general ranking mechanism to: (1) fine tune the selection of the most effective suite within the set of adequate suites, given a chosen criterion; and (2) guide the selection of the most effective criterion for the system at hand. This ranking mechanism is also based on the simulation code, and in particular it is derived from a fault-based analysis of the simulation code.

3.1 Simulation-Based Testing

As with all specification-based testing techniques, we analyze a specification to select test cases for the implementation. Analysis, in our approach, involves setting up a simulation configuration and executing it. In the following discussion, we distinguish four parts of the analysis process:

(1) **modeling**: the behavior of a system is coded within a discrete-event simulation environment;

(2) **configuration**: parameterized configurations of the simulated system are created;

(3) **simulation execution**: parameter values (i.e., test cases) are picked, the configuration is executed, and coverage data are collected; and

(4) **aggregation**: coverage data are post-processed and aggregated to determine the adequacy of a collection of executions (i.e., a test suite).

Each set of parameter values derived from this analysis constitutes the input vector of a test case that is later applied to the implementation during test execution by a suitable test harness.

It is important to understand that the analysis process we outline is a conceptual framework, not a specific tool-set. The concepts central to **simulation-based** testing can be applied within many different environments. In fact, part of the power of this approach is the manner in which it leverages existing tools and techniques. The only constraint we place on the DES environment is that it is programmed in a language that is amenable to coverage analysis and mutation. If a common general-purpose programming language is used for simulation, then there are a wide variety of coverage analysis tools that can be used off-the-shelf. There is less selection of mutation tools, but the major imperative languages are supported. Furthermore, the process we outline above makes no assumptions and imposes on restrictions on the implementation style of the system.

In order to conduct our experiments, we were compelled to choose a specific DES environment, specific coverage analysis techniques, and a specific code mutation tool. Without loss of generality, we describe the technique of specification-based testing in the context of our experimental environment. Also, to make the discussion more concrete, we use an example distributed application, GCDService, is a service that computes the greatest common divisor of two integers upon request. **GCDclient** components construct a request message containing the integers of interest and send it over an unreliable network to a server. A **GCDserver** component waits for requests and provides responses either by looking up a cached copy of the result from previous request of the same integers; or by computing it directly through the application of Euclid's algorithm.

Appendix B contains the code for Euclid's algorithm and the full simulation code for GCDService. Some of this is duplicated here for clarity.

3.1.1 Modeling

The natural way to represent a distributed software system within a DES is to map components of the system to DES processes, and to use events to represent the arrival of messages at the components. This model is used ubiquitously by distributed-system researchers and practitioners. With this basic model, it is relatively straightforward to preserve and represent the fundamental properties of distributed systems.

- **Latency**: the behavior of the network can be implemented as a process. Message propagation delays are implemented by altering the scheduled time of message arrival events.
- **Separate address spaces**: processes in the simulation should not communicate directly (i.e., by invoking methods), but instead use events.
- **Partial failure**: within a DES environment, processes can stop at any time, but other processes have no direct way of determining this.
- **Concurrency**: within DES environments, processes execute independently, using simple directives to simulate delays or pauses in processing.

For a developer, the challenge of modeling a distributed system is in determining the level of abstraction at which to implement algorithms and behavior. There is no rulebook about how to proceed and to a large degree it depends on the intended usage of the simulation and experience of the simulation developer. Part of the power of using such a flexible paradigm is that it can easily be tailored to the situation at hand. For example, with **GCDService**, the core functionality is provided by Euclid's GCD algorithm, and the distributed functionality related to the caching behavior, sending and receiving messages, etc. is wrapped around it. When modeling this system, the developer must decide whether to include Euclid's algorithm and thus have a fairly complete simulation, or to exclude it to focus on the distributed processing done by the **GCDserver** component. This decision is influenced by, and impacts the testing process. If the simulation developer knows that Euclid's algorithm is undergoing extensive testing in isolation, they might abstract away this logic in the simulation, and thereby limit the scope of what system-level tests will address. On the other hand, if the developer performing system-level testing is worried that the testing of this module is incomplete at lower levels, they can include the algorithm in the simulation and ensure that its functionality is adequately exercised.

Modeling the network is key decision that developers must make. There are several possibilities. At one extreme, the developer may model each physical network link that connects programs in the system, the routers and switches and the like. This would be appropriate in situations where the behavior of the system was influenced by individual link properties. At the other extreme, the developer may represent the entire network as a single process and manipulate delays and drops globally. The DES paradigm is flexible enough to account for either extreme, and virtually anything in between.

To simulate the systems we consider in this work, we developed a simple simulation environment that encourages a particular programming style and makes certain modeling decisions. Foremost, we represent the network as a single process that has homogeneous drop and delay rates; we do this because our experimental method requires that we can automate the selection of simulation inputs, and more detailed network implementation would make this difficult. We also chose to include the notion of “ports” with communication channels to make the simulation implementation have the flavor of low-level sockets programming. Finally, we adopted a coding style in which message objects (i.e., events) are limited to being structured data types without methods of their own. In the next section, we describe in our simulation environment in more detail.

3.1.2 Simulation Environment

In our experiments we use the Java-based DES library simjava[1] . From this library, we use classes representing processes and events, virtual time management, and

[1] `http://www.dcs.ed.ac.uk/home/hase/simjava/`

the event scheduling framework. We adapt these domain-independent mechanisms to provide a simulation environment more appropriate for simulating distributed software systems that exchange messages over an unreliable and latent network.

The primary function of our simulation environment is to provide a more intuitive interface for representing communication between processes. This is enabled by implementing a pluggable framework for coding network behavior. With this in place, the simulation developer interacts with the network in an intuitive way at appropriate places within the component logic.

3.1.2.1 Process Model

There are two different paradigms for implementing a DES system: event-based and process-based, the difference primarily being how the behavior of a process is implemented. In an event-based simulation, process behavior is implemented within callback functions that are invoked when an event occurs for the process; this accentuates the reactive nature of the processes. Conversely, in a process-based environment, the behavior is implemented within a main routine, and event occurrences must be polled explicitly. We use the process-based approach, primarily because this results in a more natural implementation of each process' behavior.

3.1.2.2 Communication

Many different approaches to representing the network are possible, depending on the focus of the simulation. We chose to represent the entire network as a single process that acts like a switchboard between the components. This makes it quite simple to implement a global behavior for the network.

The simulation environment provides implementations of different communication paradigms: datagram based, and stream based. During the modeling stage the specifier need only decide which paradigm to operate within for each interaction; similar to the decision to use either the UDP or TCP transport mechanisms. In this work we consider only systems whose communications can be modeled with datagrams. The behavior of the network can be specified when a simulation is configured, and we discuss this below.

Components interact with the network through channels. **DatagramChannels**, reminiscent of UDP sockets, are bound to a port of the local component either explicitly or implicitly. Typically, port numbers are explicitly provided for channels that will act as server ports, if a port number is not provided, an unused one is selected. Messages arriving at the port are received through a channel object, and the same object may be used to send messages to other channels. Incoming messages are queued and retrieved in the order in which they were received. If no messages are available, the receive operation blocks (with an optional timeout). The send operation does not block.

A component can instantiate as many channels as necessary to implement the behavior correctly and a select mechanism is provided so that a component can wait for incoming messages on multiple channels simultaneously.

```
public class GCDclient extends Component
{
  public int main(Address server, int a, int b)
    throws NoResponseError
  {
    DatagramChannel ch = newDatagramChannel();

    Request req = new Request(a, b);

    for(int i=0; i<3; ++i){
      ch.send(server, req);
      try {
        return ((Response)ch.recv(0.5)).gcd;
      }
      catch(TimeoutException e){}
    }
    throw new NoResponseError();
  }
}
```

Figure 3.2: `GCDclient` Component

Figure 3.2 contains the full implementation of the **GCDclient** component. Since this component is a client (i.e., it initiates connections) it does not provide a port number in its call to **newDatagramChannel** in line 6. In line 11, the send operation of DatagramChannel is used to simulate the sending of a message to the server address, and in line 13, the channel is used to receive a **Response** object with a timeout of 0.5.

```java
public class Request extends Message
{
  public final int a;
  public final int b;

  public Request(int a, int b){
    this.a = a;
    this.b = b;
  }
}

public class Response extends Message
{
  public final int gcd;

  public Response(int gcd){
    this.gcd = gcd;
  }
}
```

Figure 3.3: GCDService Messages

3.1.2.3 Messages

Messages are implemented as immutable structures. Aside from application-specific data fields, message objects also contain the source and destination addresses after being sent across the network. Figure 3.3 contains the code implementing the **Request** and **Response** messages needed to simulate **GCDService**. Within our simulation environment, message classes are limited to being structured data types, they are explicitly prevented from defining methods to operate on their member variables.

In our experience, it is convenient to represent messages at a high-level using arrays and collections and other advanced data structures as necessary. However, this is a modeling decision that results in message parsing and formatting logic being left out of the simulation and therefore unable to be addressed by simulation-based criteria. The simulation developer has complete control over the level of abstraction of the simulation code, and our environment does not preclude, for example, the implementation of all message content as an unformatted byte array.

3.1.2.4 Components

The core behavior of the system is coded in the components. Components interact with each other by sending message objects through the communication channels discussed earlier. Within the component, virtually any data and control structures can be used.

Figure 3.2 contains the full implementation of the **GCDclient**. This component is parameterized by the address of its companion **GCDserver** and the two integers for which the GCD is desired. Its behavior is quite simple: it instantiates a request object and then tries three times to send the request to the server and receive the response, waiting 0.5 virtual time units between each attempt. If a response is received, the GCD value it contains is returned, otherwise an error is signaled.

As a more complicated example, the implementation of **GCDserver** is shown in Figure 3.4. This component is parameterized by the port to listen on, and the upper size bound of its internal cache. Initially, a channel is bound to the specified port and the cache data structure is instantiated (lines 5 and 6). Within the main loop, the logic is divided between the caching functionality and the actual computation of GCD values with Euclid's algorithm. For caching, lines 11–14 construct the cache key out of the values contained in the **Request** object so that the key will accommodate different orderings of variables; the code in lines 16–20 checks the cache for a previously computed result. Lines 37–50 are devoted to cache cleanup logic wherein the least recently used cache item is removed.

Euclid's algorithm is implemented in lines 21–33. Line 25 demonstrates the use

```
public class GCDserver extends Component
{
  public void main(int port, int cacheSize)
  {
    DatagramChannel ch = newDatagramChannel(port);
    Map cache = new HashMap();

    while(running()){
      Request req = (Request)ch.recv();

      String key = req.a + "," + req.b;
      if(req.a > req.b){
        key = req.b + "," + req.a;
      }

      Integer gcd = null;
      if(cache.containsKey(key)){
        CacheValue cv = (CacheValue)cache.remove(key);
        gcd = cv.gcd;
      } else {
        // Begin Euclid's Algorithm
        int x = req.a;
        int y = req.b;
        while(x > 0 && y > 0){
          pause(0.1);
          if(x > y) {
            x = x - y;
          } else {
            y = y - x;
          }
        }
        gcd = new Integer(x + y);
        // End Euclid's Algorithm
      }
      cache.put(key, new CacheValue(now(), gcd));

      if(cache.size() > cacheSize){
        Iterator itr = cache.entrySet().iterator();
        Object minKey = null;
        double minValue = Double.MAX_VALUE;
        while(itr.hasNext()){
          Map.Entry entry = (Map.Entry)itr.next();
          CacheValue value = (CacheValue)entry.getValue();

          if(value.usedAt < minValue){
            minKey = entry.getKey();
          }
        }
        cache.remove(minKey);
      }
      assert cache.size() <= cacheSize;
      Response resp = new Response(gcd.intValue());
      ch.send(req.getSourceAddress(), resp);
    }
  }
}
```

Figure 3.4: **GCDserver** Component

of the explicit time advancement operation “pause”. From its location in the code, the intent is that the processing time of this computation is proportional to the number of time through the loop started on line 24. As mentioned earlier, the inclusion of this algorithm is another modeling decision. If the simulation developer knew that this logic was the focus of special attention from other testing techniques, then the whole algorithm could be replaced with:

```
   ...
21 // Begin Euclid's Algorithm
   gcd = new Integer(-1);
   // End Euclid's Algorithm
   ...
```

Throughout the rest of this chapter we use code examples that are specific to our simulation environment. The code is clearly not general, but the concepts related to configuration, execution and aggregation are.

3.1.3 Configuration

Before a simulation can be run, components must be instantiated and organized into a particular configuration that represents the real-world situation being simulated. In our environment, this is accomplished by extending the `Simulation` class and adding component instances. In our discussion of configurations, we differentiate between **abstract** configurations which are parameterized, and therefore unable to be directly executed, and **concrete** configurations that supply values for all parameters. In testing terminology, abstract configurations are test harnesses, and the values included as part of a concrete configuration comprise the input vector of a test case.

Figure 3.5 contains an abstract configuration, **ClientPairTest** that consists of a single server component and two client components. **ClientPairTest** is parameterized

```
public abstract class ClientPairTest extends Simulation {
  public ClientPairTest(int a1, int b1, int gcd1,
                        int a2, int b2, int gcd2)
  {
    Address addr = new Address("server", 1234);

    add(new GCDserver(addr.name, addr.port, 10));

    add(new GCDclient("c1", addr, a1, b1),
        new Integer(gcd1), 0.0);

    add(new GCDclient("c2", addr, a2, b2),
        new Integer(gcd2), 5.0);

    setNetworkBehavior(new NetworkBehavior(0.1));
  }
}

public class ClientPairTest_A extends ClientPairTest {
  public ClientPairTest_A(){
    super(3, 5, 1, 2, 4, 2);
  }
}

public class ClientPairTest_B extends ClientPairTest {
  public ClientPairTest_B(){
    super(3, 5, 1, 3, 5, 1);
  }
}

public class ClientPairTest_C extends ClientPairTest {
  public ClientPairTest_C(){
    super(3, 5, 1, 5, 3, 1);
  }
}
```

Figure 3.5: Example Configuration

by the values requested by each of the components. Figure 3.5 also contains three concrete configurations that extend **ClientPairTest** by supplying parameter values.

In the abstract configuration, the **GCDserver** instance is added to the simulation in line 7. In our environment, all components must be instantiated with a unique name as their first constructor argument, the remaining constructor arguments are passed to the main routine by our framework. In this case, the server's port number and maximum cache size are passed.

GCDclient instances are added to the simulation in lines 9 and 12. These are instantiated with a name, the server address, and the request values. The second argument to the **add** method is the expected return value of the component. This is used as a high-level oracle in the simulation. The last argument is the start time for the component; in this example the first component starts at a virtual time of 0.0, while the second component delays until 5.0.

Finally, on line 15 the configuration specifies the behavior of the network during simulation. In this case, a simple delay-only network is used, with a small latency. More advanced behaviors can be specified by extending the **NetworkBehavior** class; we have developed generic probabilistic behaviors that drop, delay, and duplicate messages by arbitrary amounts. We have also implemented more explicit behaviors that drop specific messages.

The concrete configurations in Figure 3.5 have supplied values for all parameters, so they are essentially executable simulations. In **ClientPairTest_A**, two different integer pairs are used; in the second two concrete configurations, the same two integers are used in requests (3 and 5), but in **ClientPairTest_B** the order is the same, while in **ClientPairTest_C**, the order is different. These three concrete configurations are candidate test cases that must be executed before their adequacy as a suite can be determined.

3.1.4 Simulation Execution

The actual means of execution is specific to the DES framework being used, but all simulations must somehow be executed. In ours the name of a concrete configuration class is passed to the simulation engine. The simulator loads the class using a custom class loader that first checks the bytecode to ensure conformance to the implementation limitations mentioned above, and then instruments the classes so that block, branch, and definition-use coverage values can be determined.

The simulation classloader uses the Bytecode Engineering Library[2] to analyze bytecode. Using this library, instrumentation for block and branch coverage is essentially a matter of inserting tracing method calls before each branch instruction, and before each instruction that is the target of a branch.

Complete data-flow analysis of Java code is quite difficult to accomplish in the general case. In our implementation, we take advantage of some of the constraints imposed on the component coding style by our framework to make this task easier. Therefore, to instrument for definition use coverage, tracing method calls are inserted before each store instruction, and each load instruction. Method calls are conservatively treated as object definitions. To conservatively determine the definition-use pairs, we developed some supporting classes to execute the same data-flow analysis performed during bytecode verification by the Java compiler [LY99]. The coverage and performance information collected during a simulation execution is stored in a compressed XML format.

3.1.5 Aggregation

Each simulation run generates coverage data for a single test case, to determine adequacy of a test suite, the coverage data must be aggregated properly. To do this with our simulation environment, we created a reporting tool that parses the simulation-

[2] `http://jakarta.apache.org/bcel/`

trace files created during execution, and records the coverage of each block, branch and definition-use pair. By separating the execution and reporting functions in this way, each simulation is executed once even if the adequacy of different combinations of simulations are computed multiple times.

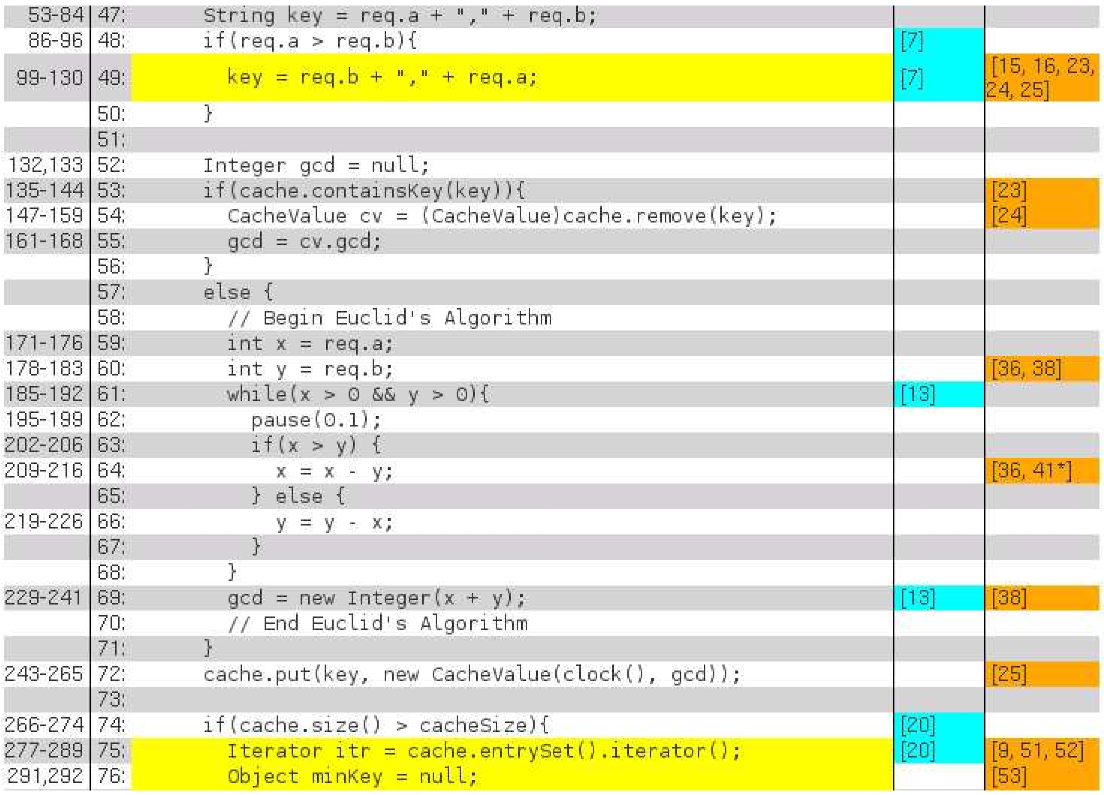
The reporting engine provides output in both HTML and XML. The XML output is used during experimentation when there is no need for graphical examination of the results. During development of the simulations and especially during the development of configurations, the HTML output enabled us to graphically examine the white-box coverages.

Figure 3.6 shows the HTML report created after executing the concrete configuration **ClientPairTest_A** detailed in Figure 3.5. The report is organized into four columns. The first column lists the bytecode PC ranges for each line of source code (the reporting tool also generates a similar report for the bytecode representation of the software which is sometimes useful for determining exactly which elements are not being covered). The second column has the source line number and the text of the source code. Lines with a yellow background contain statements that have not been executed (block coverage). The third column contains information about uncovered branches. Any line with a blue background in the third column is either the source or destination of a branch that is not executed; the bracketed numbers are used to match branch pairs. Similarly, column four contains information about uncovered definition use pairs.

3.2 Fault-Based Analyses

In fault-based analysis, testing strategies such as adequacy criteria are compared by their ability to detect fault classes. Fault classes are typically manifested as mutation operators that alter a correct formal artifact in well-defined ways to produce a set of incorrect versions. These **mutants**, can be used to compare testing strategies.

For example, an implementation might have a fault that causes a particular state



53-84	47:	String key = req.a + "," + req.b;		
86-96	48:	if(req.a > req.b){	[7]	
99-130	49:	key = req.b + "," + req.a;	[7]	[15, 16, 23, 24, 25]
	50:	}		
	51:			
132,133	52:	Integer gcd = null;		
135-144	53:	if(cache.containsKey(key)){		[23]
147-159	54:	CacheValue cv = (CacheValue)cache.remove(key);		[24]
161-168	55:	gcd = cv.gcd;		
	56:	}		
	57:	else {		
	58:	// Begin Euclid's Algorithm		
171-176	59:	int x = req.a;		
178-183	60:	int y = req.b;		[36, 38]
185-192	61:	while(x > 0 && y > 0){	[13]	
195-199	62:	pause(0.1);		
202-206	63:	if(x > y) {		
209-216	64:	x = x - y;		[36, 41*]
	65:	} else {		
219-226	66:	y = y - x;		
	67:	}		
	68:	}		
229-241	69:	gcd = new Integer(x + y);	[13]	[38]
	70:	// End Euclid's Algorithm		
	71:	}		
243-265	72:	cache.put(key, new CacheValue(clock(), gcd));		[25]
	73:			
266-274	74:	if(cache.size() > cacheSize){	[20]	
277-289	75:	Iterator itr = cache.entrySet().iterator();	[20]	[9, 51, 52]
291,292	76:	Object minKey = null;		[53]

Figure 3.6: ClientPairTest_A Coverage Report

change to be missed, where such state changes are represented as transitions in a finite-state specification. This **missing transition** fault class is then represented in the specification domain by all specifications that can be obtained from the original specification by removing one of the transitions. Testing strategies that are able to distinguish incorrect from correct specifications are said to cover that particular fault class. The underlying assumption of fault-based analysis is that simple syntactic faults in a specification are representative of a wide range of implementation faults that might arise in practice, so a testing strategy that covers a particular fault class is expected to do well at finding this class of faults in an implementation.

A prerequisite of a fault-based analysis is the existence of a set of mutation operators. Simulations are coded in imperative programming languages and so well suited to the code-mutation operators developed in the context of mutation testing [DLS78]. These operators make simple syntactic changes to code that may result in semantic differences.

In our fault-based analysis, we apply standard code-mutation operators to the component code. Each concrete simulation is then executed against each mutant component in turn. A simulation may (1) terminate normally with reasonable results, (2) terminate normally with unreasonable results, (3) not terminate, or (4) terminate abnormally. For all but the first situation, the test case (configuration) is recorded as having killed the mutant. The **mutant score** of a test suite is computed as the percentage of mutants killed by at least one test case in the suite.

In most mutation analyses, the exact output from the original version is used as an oracle against which mutant output is compared. This is not always possible in this case because simulations of distributed systems are naturally non-deterministic. In practice, we use assertions and sanity checks in the simulation code to determine which results are considered "reasonable".

As an example, consider the implementation of **GCDserver** shown in Figure 3.4.

```
17        if(! cache.containsKey(key)){
18          CacheValue cv = (CacheValue)cache.remove(key);
19          gcd = cv.gcd;
20        } else {
```

Figure 3.7: **GCDserver** with Line 17 Mutation

Line 17 in this figure contains an if-statement that checks if the cache contains the key corresponding to the request being processed. If a mutation operator inserts a logical **not** in this boolean expression the code might appears as shown in Figure 3.7. When this mutated code is executed, the **GCDserver** component would retrieve a **null** reference from the cache for the very first request processed, ultimately resulting in a **NullPointerException** being thrown in line 52, aborting the simulation. Since this particular mutant would cause the simulation to fail under virtually any configuration, it is an example of a "pathological" mutant, and is therefore not useful for ranking.

```
40          double minValue = Double.MAX_VALUE;
41          while(itr.hasNext()){
42            Map.Entry entry = (Map.Entry)itr.next();
43            CacheValue value = (CacheValue)entry.getValue();
44
45            if(value.usedAt > minValue){
46              minKey = entry.getKey();
47            }
48          }
49          cache.remove(minKey);
50        }
51        assert cache.size() <= cacheSize;
```

Figure 3.8: **GCDserver** with Line 45 Mutation

Figure 3.8 shows the code obtained from changing the arithmetic operator in line 45 from $<$ to $>$. In this case the resulting mutant component would never remove items from the cache, causing the assertion in line 51 to fail. This mutant is only detected if a configuration causes a **GCDserver** component to process more requests than the upper bound on the cache size.

While fault-based analysis is not a new technique, it is typically not used in practice because of the computational cost associated with obtaining mutants and executing test cases against each one. However, in our case, two fundamental features of simulation-based testing enable the use of fault-based analysis. First, when compared to the execution cost of deploying and executing the real (i.e. non-simulation) implementation of a distributed system, simulation execution is efficient. Second, because of the use of abstraction when developing implementations, simulation code is significantly smaller and simpler than the code needed to implement the actual system (for DNS, described in the next chapter, the size of simulation code is several orders of magnitude smaller than the implementation of just one part of the system). In combination, these two features result in a use fault-based analysis techniques that is actually feasible in practice.

3.3 Usage Scenarios

We use simulation-based adequacy criteria and fault-based analysis of the simulation code, individually or in combination, to support the identification of effective test suites. We describe this approach through three different usage scenarios.

3.3.1 Conventional

The conventional way to use simulation-based testing is to choose a an adequacy criterion defined against the simulation code, and select a single test suite that is adequate with respect to it. In this scenario, the developer has a number of options for which adequacy criteria to use.

The developer might choose a criterion that depends on observable events from the environment. For example, the criterion could require that communication links be used up to their maximum capacity. Or, the developer might choose a classical white-box code coverage adequacy criterion defined over the simulation. However, in both cases, the simulation must be suitably instrumented and then executed for the adequacy to be determined.

In this scenario, the tester is exposed to both **intra-criterion** and **inter-criteria** risk. The cost in this scenario is simply the cost of simulating a number of test cases until an adequacy value is achieved.

3.3.2 Boosting

In this scenario, the tester has somehow chosen a particular adequacy criterion, as before, but here they want to reduce the risk of picking an ineffective, adequate test suite. Thus, they select multiple adequate test suites, use fault-based analysis to rank the suites by mutant score, and apply the highest-ranked to the implementation.

This usage scenario obviously applies to the code or environmental criteria described in the previous scenario. In this case, the simulation may be used in a first stage to identify a number of adequate suites, and in a second stage to rank the individual suites through fault-based analysis.

But, the boosting technique can also be applied to a criterion that depends exclusively on the input vector. In this case, the simulation code is of no use in evaluating adequacy, and the developer must do that through other means. However, fault-based analysis of the simulation can still contribute in this scenario by providing a relative ranking of adequate suites based on mutant score.

This usage is more costly than the conventional usage since multiple adequate suites must be selected, and each selected test case must undergo a fault-based analysis, but **intra-component** risk is reduced.

To make this more concrete, assume that a system has a simulation from which five mutants are derived. First, the engineer uses the basic specification-based testing process to independently select two adequate test suites, each of which consist of three test cases. The rows in Table 3.1 are labeled with the test cases 1.1 – 1.3 and 2.1 – 2.3. The columns in this table contain the results of executing the test cases against

	Mut1	Mut2	Mut3	Mut4	Mut5	
Test 1.1	$\surd$	$\times$	$\surd$	$\surd$	$\times$	
Test 1.2	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	
Test 1.3	$\surd$	$\surd$	$\surd$	$\times$	$\times$	
	$\surd$	$\times$	$\surd$	$\times$	$\times$	60%

	Mut1	Mut2	Mut3	Mut4	Mut5	
Test 2.1	$\surd$	$\surd$	$\times$	$\surd$	$\times$	
Test 2.2	$\surd$	$\times$	$\surd$	$\surd$	$\times$	
Test 2.3	$\surd$	$\surd$	$\times$	$\times$	$\times$	
	$\surd$	$\times$	$\times$	$\times$	$\times$	80%

Table 3.1: Example Boosting Scenario

each of the five mutants. A $\surd$ in a cell indicates that the test passed successfully when executed against the mutant, while an $\times$ symbol indicates that the test failed (i.e. the mutant was killed).

The rightmost column contains the overall mutant score for both test suites. The first suite has a mutant score of 60% because three of the five mutants are killed; the second suite has a score of 80% since four of the five mutants are killed. Based on these results, the engineer decides that suite #2 is more likely to be effective, and chooses to map tests 2.1, 2.2 and 2.3 into the implementation domain and then execute them.

3.3.3 Ranking

The developer wants to choose a criterion among many applicable criteria for the particular system under test. Once again, the developer can turn to fault-based analysis of the simulation code to rank the different criteria. Specifically, the developer creates adequate suites for each criterion and then selects the suite with the highest mutant score. At this point, the tester simply applies the boosting usage to the adequate test suites already created for the highest-ranking criterion.

This scenario is even more costly than the boosting, because many adequate suites are both selected (using the simulation or not), and then executed against simulation mutants.

In summary, through these scenarios, a DES can be used directly to evaluate the adequacy of a test suite with respect to criteria based on the environment and on the simulation code. This requires running the test suite through an instrumented simulation. In addition, the simulation can be used to improve the effectiveness of any criterion and to guide the programmer in selecting a criterion. This is done by means of a fault-based analysis of the simulation code.

Chapter 4

Empirical Evaluation

To evaluate our ideas, we follow the experimental method introduced by Frankl and Weiss [FW93a]. At a high level, this involves sampling a large universe of test cases to randomly construct test suites that are adequate with respect to different criteria. Statistical inference is then used to test hypotheses about the relative fault-detecting ability of competing criteria and their relative cost. We instantiate the different criteria by employing a technique introduced by Briand et al. [BLW04], in which different testing strategies are simulated once the failure data for the universe of test cases has been collected.

We conducted three separate experiments, aimed at evaluating the usage scenarios described in Section 3.3. In the first usage scenario, code-based or environmental adequacy criteria are instantiated against a simulation which is used to select an adequate test suite. There is an almost infinite variety of specialized adequacy criteria that could be defined against a simulation, but we believe the typical developer interested in this conventional scenario would be most interested in applying a simple, general-purpose simulation-code-coverage criterion. Thus, in Experiment 1, we compare the cost and effectiveness of three white-box adequacy criteria to randomly selected test suites of varying size.

Experiment 2 is aimed at determining our ability to improve the effectiveness of individual adequacy criteria, both white-box and black-box, through fault-based analysis. This corresponds to the **boosting** usage scenario in which a developer is willing to incur the cost of creating multiple adequate suites and applying fault-based analysis, with the expected benefit of improving the effectiveness.

In the third usage scenario, the developer uses fault-based analysis to rank candidate criteria. Thus, Experiment 3 is aimed at determining the accuracy with which we can predict the actual effectiveness relationships between criteria used in the first two experiments.

4.1 Subject Systems

Our case-study evaluation uses simulations and implementations of three subject distributed systems. For each system, we created simulations from informal specification documents describing their intended inputs and behaviors, and experimented with available implementations. The first system, GBN, is the "go-back-n" algorithm, which is used for reliably transferring data over an unreliable communications layer. The second, LSR, is a link-state routing scheme that uses Dijkstra's algorithm in each component of a decentralized collection of routers to compute local message-forwarding behavior. The third, and most significant, is a recursive, caching Domain Name System (DNS) resolver.

Implementation of the first two systems were given as programming assignments in an introductory undergraduate networking course taught at the University of Lugano. We used the assignment handout provided to students and the Kurose and Ross networking textbook [KR04] as source descriptions. For the DNS resolver we used historical releases of MaraDNS,[1] an open-source resolver that implements the core DNS functionality as described by RFCs 1034 and 1035.

[1] `http://maradns.org`

4.1.1 GBN

As described by Kurose and Ross, GBN involves two processes: a sender that outputs data packets and waits for acknowledgments, and a receiver that waits for data packets and replies with acknowledgments. Both processes maintain sequence-number state to ensure data packets are received in the proper order. The sender also maintains a sliding window of sent packets from which data can be retransmitted if acknowledgments are not received within a certain time period. As an additional wrinkle, the student assignment required the sending window size to grow and shrink as dictated by the history of acknowledgment packets received.

The **go-back-n** algorithm is implemented within a simple client/server file transfer application. The client program is invoked with the path to an existing file to be read and transferred, and the server program is provided the path to a file to be created and populated with the received data.

4.1.1.1 Simulation

The GBN simulation code consists of two message classes, **ACK** and **DAT**, which represent acknowledgments and data packets, respectively. The **Sender** component represents the client program and implements the functionality of a GBN sender. Similarly, **Receiver** implements the server side of the algorithm. All together, the specification consists of approximately 125 non-comment, non-blank lines of Java.

We define an abstract GBN configuration consisting of one sender and one receiver. This configuration is parameterized by the following three values:

(1) **NumBytes**: the number of bytes to be transferred, sampled uniformly between $[1, 335760]$.

(2) **DropProb**: the probability of a packet being dropped by the network, distributed uniformly in $[0.0, 0.20]$.

(3) **DupProb**: the probability of a non-dropped packet being duplicated, distributed uniformly in $[0.0, 0.20]$.

NumBytes is passed directly to **Sender** and also used in an assertion within **Receiver** to check that the proper number of bytes is actually received. DropProb and DupProb are used by the datagram network process to randomly drop and duplicate events. The network process delays all events by the same amount.

Using MuJava [MOK05], we generated 488 mutants of the simulation; 74 mutants were from the **Receiver** component, and the remaining 414 were from the **Sender**. The majority of the 488 mutants were caused by mutation operators that inserted and altered arithmetic and logical operators on integers. After eliminating pathological and equivalent mutants, we were left with 72 mutants to analyze.

For GBN, the universe of test cases consists of 2500 randomly created parameter tuples.

4.1.1.2 Implementations

On average, the 19 student implementations of GBN were coded in 129 lines of C using a framework, itself approximately 300 lines of C code, provided by the course instructor. This framework handles application-level file I/O and the low-level socket API calls. The students were responsible for implementing the core **go-back-n** algorithm within the constraints imposed by these layers.

We also used our own Java-based implementation of this system, and created 192 faulty versions using the MuJava automatic mutation tool [MOK05]. This implementation uses the basic sockets facilities provided by the **java.net** package and consists of 24 classes containing 565 lines of code.

The test harness randomly populates a temporary file with NumBytes bytes and starts an instance of a simple UDP proxy that mediates packet exchange and drops and duplicates packets in the same way the network process does in the simulation. The test fails if either program exits with an error, or if the file was not duplicated faithfully.

4.1.2 LSR

A link-state routing scheme is one in which each router uses complete global knowledge about the network to compute its forwarding tables. The LSR system described in the student programming assignment utilizes Dijkstra's algorithm to compute the least-cost paths between all nodes in the network. This information is then distilled to construct the forwarding tables at each node. To reduce complexity, the assignment statement stipulates that the underlying network does not delay or drop messages.

4.1.2.1 Simulation

The LSR simulation code consists of three events, several supporting data structures, a class implementing Dijkstra's algorithm, a **Router** process type, and a **Client** process type to inject messages, for a total of approximately 180 lines of Java code. A router takes as input a list of its direct neighbors and the costs of the links to each of them.

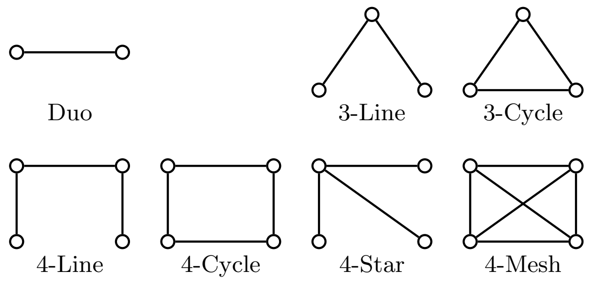


Figure 4.1: LSR Topologies

We defined a simulation of LSR, parameterized by the following values:

(1) **Topology**: an integer in the range $[1, 7]$, representing a particular arrangement of routers and costs (Figure 4.1 shows each possibility graphically).

(2) **Message count**: an integer in the range $[0, 2n]$, representing the number of messages to be sent, where n is the number of routers in the topology.

(3) **Message source and destination**: for each message, the source router and destination router is selected randomly from the range $[1, n]$ with local messages allowed.

Hard coded into the simulation are the details of each topology, including statically computed shortest-path costs for all router pairs. In the simulation, n routers are instantiated and arranged according to the specified topology. Then, mc client instances (where mc is equal to the message count) are created and scheduled for execution at regular intervals with the specified source and destination router. Each client publishes a short text string. As each publication is propagated by a router, a path-cost variable is updated with the costs of the links traversed. When a router receives a publication to be delivered locally, it saves the string and its total path cost. When the simulation terminates, assertions ensure that each router received the expected number of publications, and that the path cost for each is correct.

The universe of test cases for LSR consists of 7000 parameter tuples, 1000 for each topology, with the message count, sources, and destinations selected randomly.

4.1.2.2 Implementations

Students were again provided with an application framework within which they implemented the algorithm. On average, the 16 students implemented the LSR logic within 120 lines of Java and the framework itself is about 500 lines of code.

In the course assignment, this framework merely simulated the network interaction. For our experiments, we reimplemented the framework to run over the (unreliable) network. This change required some of the supporting classes written by students to be altered to implement the **java.io.Serializable** interface. But the interface defines no methods, so no algorithmic changes were made to any of the student implementations.

The test harness duplicates the functionality of the simulation setup code faithfully. A test fails if any of the router or client programs terminates with an error code, or if a router does not receive the expected messages.

4.1.3 DNS

The Domain Name System is one of the fundamental underpinnings of the Internet, providing a distributed database that maps names to resources of different types; the most common mapping is used to translate names into network addresses, but there are many others. The core algorithms and resource types are defined in RFCs 1034 and 1035. These documents differentiate among several conceptual components of DNS, including servers, stub resolvers, and recursive resolvers. Briefly, a **server** has primary responsibility for a particular subset of the name space. It is capable of responding directly to queries about names in this space, and providing delegation information for other names. A **stub resolver** is a program that delegates the resolution of user queries to other resolvers. A **recursive resolver** is a more sophisticated program that is able to perform the multiple queries needed to successfully resolve names to resources. Typically this involves following a trail of delegations from the top-level domain servers (servers responsible for .com and .net, for example) to the "authoritative" servers actually responsible for the name in question.

Of these three components, the recursive resolver is the most complex, involving message exchanges with multiple different servers, caching responses, processing name aliases, and the like. For this reason, we focus on testing the behavior of a recursive resolver.

4.1.3.1 Simulation

The simulation code for DNS consists of 10 structures representing the basic DNS resource record types, a single message type that represents both requests and responses (as in DNS), 11 low-level procedures for manipulating and comparing names and resource records, and three classes representing stub resolvers, recursive resolvers, and servers, for a total of approximately 900 lines of Java code.

Inputs to the simulation consist of the following:

(1) **Name space**: a generated name space populated randomly with resources, divided into between 2 and 5 administrative domains randomly assigned to authoritative servers.

(2) **Queries**: name and type queries to be issued by stub resolvers randomly selected to include valid mappings, unknown names, and invalid resource types.

(3) **DropProb**: the probability of a packet being dropped by the network, distributed uniformly in $[0.0, 0.20]$.

(4) **DupProb**: the probability of a non-dropped packet being duplicated, distributed uniformly in $[0.0, 0.20]$.

During simulation, authoritative servers are instantiated with relevant portions of the name space; some of these servers are root servers that can act as a starting point for the recursive resolvers. The network error probabilities only affect packets traveling between recursive resolvers and servers.

Assertions within the simulation ensure that the response to each query is correct given the generated name space and records. The universe for DNS consists of 2000 test cases.

4.1.3.2 Implementations

For implementations of the recursive resolver experimental subjects we used 34 public releases of MaraDNS, starting with version 1.0.0.0 and ending with 1.2.07.1. MaraDNS is implemented in C. The source code for the entire MaraDNS package ranges from roughly 15KLOC to 24KLOC, depending on the release. MaraDNS also provides the functionality of a server, with a lot of the supporting code is shared between the resolver and server subsystems.

During test executions we used version 2.0.1 of **dnsjava**[2] as a stub resolver implementation, and the **tinydns** server included with version 1.0.5 of **djbdns**[3] as a server implementation; we assume these programs are correct.

For each simulation run, a control file is generated that contains the definition of the name space, server configurations, recursive resolver configuration, and the number and behavior of stub resolvers. This file is also read by the test harness that sets up the system using a localhost network, and executes the tests.

4.2 Experimental Method

We primarily compare adequacy criteria by their effectiveness, E. Effectiveness is measured as the average proportion of faulty implementations found by adequate test suites; a test suite that fails on 6 out of 10 subject implementations has $E = 0.6$. Others define and measure effectiveness on a per-implementation basis as the proportion of adequate test suites that fail. Our metric is simply an extension to this that is more appropriate for specification-based testing since it accounts for the breadth of a suite's effectiveness. In other words, since adequacy is measured with respect to coverage of the specification, an adequate test suite should perform well against **any** implementation of the specification. Therefore, we consider a suite that finds three faulty implementations

[2] `http://www.dnsjava.org/`
[3] `http://cr.yp.to/djbdns.html`

to be three times as effective as a suite that finds just one.

For cost, C, we use total execution time, including both analysis time (c_a) and test execution time (c_e). As we assume the existence of a suitable simulation, we do not consider the cost of developing this. In many situations a specification-based test suite will be executed multiple times (e.g., by different testers during QA, as regression tests, or as part of a regularly scheduled automated test strategy). To account for this, we define M, the multiplier; we fix $M = 10$ for these experiments. The overall cost for a particular test suite is given by equation 4.1.

$$C = c_a + M * c_e \tag{4.1}$$

In the literature, cost is measured in a number of different ways, with suite size being the most common. In our experiments, resource usage (i.e., time, memory, cycles, and bandwidth) varied significantly among test cases, so we did not feel that suite size would capture a suite's cost meaningfully. Also, by using execution time, we are able to include the analysis cost naturally.

Of course, random and input-partitioning criteria have $c_a = 0$ since input values can simply be picked that satisfy a particular criterion. For white-box criteria, c_a consists of the total simulation execution time of all test cases **considered** plus the time needed to compute aggregated coverage for each of the intermediate suites examined. For example, supposed we consider five test cases, but discard the second and third because they do not cover any new elements. The analysis cost is the time needed to run five simulations plus the time taken to determine coverage of the combinations: $(1, 2), (1, 3), (1, 4), (1, 4, 5)$. Note that this puts white-box criteria at a disadvantage, since it assumes the tester has no intuition about the relation between input values and coverage, which is not usually the case in practice.

To determine statistically significant effectiveness relationships, we apply hypothesis testing to each pair of criteria and compute the **p-value**. The **p-value** can be interpreted as the smallest α at which the null hypothesis would be rejected, where α is the probability of rejecting the null hypothesis when it in fact holds. For example, to determine if criterion A is more effective than criterion B, we propose a null hypothesis H_0 and an alternative hypothesis H_a:

$$\begin{aligned} H_0 &: A \leq B \\ H_a &: A > B \end{aligned}$$

where $>$ means "more effective than". Although we do not know the actual distribution of effectiveness values, we take advantage of the central limit theorem and assume that the distribution of the normalized form of our test statistic E approximates a normal distribution. We can use this theorem comfortably with sample sizes larger than 30 [Dev95]. Therefore, we compute the z value for this hypothesis and use the **p-value** formula for high-tailed hypothesis tests (i.e., when the rejection region consists of high z values):

$$\begin{aligned} z &= \frac{\bar{E}_A - \bar{E}_B}{\sigma_{E_A}/\sqrt{n}} \\ p &= 1 - \Phi(z) \end{aligned}$$

where $\bar{E}_A$ and $\bar{E}_B$ are the average effectiveness values for criteria A and B respectively, σ_{E_A} is the sample standard deviation of effectiveness values of A, n is the sample size, and Φ is the standard normal cumulative distribution function. Typically, with **p-values** less than 0.05 or 0.01 one rejects H_0 and concludes that $A > B$.

4.2.1 Preparation

Following the terminology of Briand et al. [BLW04], we first **prepare** our data by executing and analyzing all test cases individually. This entails the management of three different executions: (1) simulations, (2) implementations, and (3) mutants.

Test cases are generated randomly from the input domain of each system and are applied to the simulation code through concrete configurations, as described in Section 3.1.3. For the implementations, we developed test harnesses that mimic the simulation configuration faithfully. These test harnesses contain high-level test oracles. For example, the service provided by the GBN system is file transfer. The high-level oracle for this system is simply a direct comparison between the original file and the copy. The sophistication of the oracle plays a significant role in determining the absolute effectiveness of a testing technique. The experimental method we adopt requires oracles to be automated which is why we could not employ more detailed oracles.

To manage the preparation step, we developed a sophisticated execution script that enabled us to stop and start the process as necessary, and that helped organize and optimize the large amount of data generated during this step. Careful management of the implementation and mutant executions is especially important since both executions can enter infinite loops or otherwise hang. We used timeouts to handle these situations. For the mutant executions, the timeout we used is triple the longest non-mutant simulation run. For the implementations, we determined timeout values conservatively starting with high timeouts (e.g., 5 minutes) and gradually lowering the values.

4.2.1.1 Simulations

Executing the simulation tests is simply a matter of instantiating each concrete configuration within the simulation environment and running it. During execution, coverage information for blocks, branches and definition-use pairs is recorded along with the wall-clock time used by each simulation run.

4.2.1.2 Implementations

Next, we execute each test case against all implementations. We record the result of each test case in terms of success or failure, along with the test execution time.

As the goal of these experiments is to differentiate between testing techniques, we eliminate implementations with high failure rates since a test suite of any reasonable size will always detect failures, regardless of the technique used to select its cases. We do this by initially executing 100 randomly selected test cases against each implementation and eliminate from further consideration those with failure rates higher than 20%. The same approach is used by others in empirical studies [FI98] to focus attention on faults that are hard to detect. After executing all test cases, we also excluded correct implementations (i.e., those without failures) from further analysis.

IUT	**%**
stud07	0.28
stud06	0.32
stud08	0.40
stud15	3.16
stud09	3.40
stud18	4.72
mutROR15	16.96
mutLOI35	17.00

(a) GBN

IUT	**%**
stud07	5.04
stud16	5.15
stud03	5.21
stud08	7.70
stud01	7.94
stud06	8.02
stud14	8.48
stud15	12.17
stud09	12.42

(b) LSR

IUT	**%**
1.0.29	9.15
1.1.59	15.25
1.1.91	17.30
1.2.03.3	17.45
1.1.60	17.55
1.2.01	18.00
1.2.03.5	18.20
1.2.07.1	18.40
1.2.00	18.45
1.1.61	18.70
1.2.03.4	18.90

(c) DNS

Table 4.1: Implementation Failure Rates

Table 4.1 lists the 28 remaining faulty, non-pathological implementations along with the percentage of test cases that fail on each. The GBN and LSR implementations with names beginning with "stud" are student implementations. For GBN, the two beginning with "mut" are mutants of our own Java implementation.[4] For DNS, the names correspond the version numbers of the releases. In these tables, the implementations are ordered by failure rate.

[4] On principle we avoid using implementation mutants as subjects, since we use the same operators to mutate simulation code. We generated mutants of our GBN implementation only after determining that there were not enough student implementations with appropriate failure rates.

4.2.1.3 Mutants

Mutants are generated from the simulation code by applying all conventional mutation operators provided by the MuJava tool [MOK05] to the component classes and supporting logic classes (excluding message structures) in the simulation code. As with simulation executions, here we simply instantiate each concrete configuration and run it. We use Java's class loader path to ensure that the compiled mutants are loaded instead of the original, correct version.

This step is quite computationally expensive, and as we did with implementations, we mitigate this by aggressively excluding mutants with high failure rates. "Pathological" mutants are identified by initially simulating a small sample of test cases against all mutants. Any mutants with failure rates higher than 50% are excluded from further consideration. Eliminating these mutants is justified because we measure the effectiveness of test suites, not test cases, and mutants with high failure rates are killed by virtually all test suites with more than a few members. After this initial sampling, we examined the code of mutants with zero kills and eliminated any ones semantically equivalent to the original code.

System	N_M	N_e	N_p	N_u
GBN	488	74	342	72
LSR	58	2	41	15
DNS	234	0	99	135

Table 4.2: Generated Mutants

Table 4.2 contains the details of mutants generated for each system. The second column, labeled N_M, is the total number of mutants generated by MuJava. The third and fourth column, labeled N_e and N_p, are the number of equivalent and pathological mutants. The fifth column, N_u, contains the number of mutants for each system used during analysis. We attribute the comparatively large N_M for GBN to the amount of integer arithmetic used in this algorithm. The most striking aspect of these data is the number of pathological mutants we were able to eliminate. For GBN and LSR, these mutants account for 70% of the total, for DNS 42% of the mutants were pathological. By eliminating these early in the execution process, we were able to realize significant savings in terms of execution time. Interestingly, there were no equivalent mutants for DNS. We confirmed this by examining the differences between each mutant and the original and determined that all mutants were in face semantically different.

System	N	N_m	N_i	T_s	T_i	T	T_e
GBN	2500	146	8	4.12 (s)	35.2 (s)	25 (d)	17 (d)
LSR	7000	17	8	2.66 (s)	6.9 (s)	8 (d)	4.5 (d)
DNS	2000	135	11	7.37 (s)	11.8 (s)	26 (d)	23 (d)

Table 4.3: Total Preparation Time

Still, the preparation phase of our experimental method is quite expensive computationally. Table 4.3 gives a rough estimate of the amount of computation time required. The columns in this table are: N, the universe size; N_m, the number of non-pathological mutants; N_i, the number of faulty, non-pathological implementations; T_s, the average simulation time; T_i, the average test execution time; T the total computation time as computed by equation 4.2.

$$T = N * T_s * (N_m + 1) + N * T_i * N_i \tag{4.2}$$

The total numbers are quite large. Fortunately, test-execution is highly parallelizable, especially simulation, and we executed on two machines in parallel, one for simulations and mutants, and the other for tests. Thus, the effective time, T_e is the longer of the two terms in equation 4.2. For GBN and DNS, the simulation and mutant execution was significantly more expensive than the test execution time. For LSR, the two terms were about equal. In practice, we developed the simulation of each system early in the process, and let the simulations run while working on the test harnesses for test execution.

4.2.2 Implementation Fault Characterization

In the failure study, a scenario's architecture and inputs are classified as either "distributed" or not. Section 2.2.3 describes in detail what input types are classified as distributed. There, we found that a significant proportion of user-reported failures required distributed inputs and scenarios.

Here we are concerned with determining the extent to which the failures we encountered during preparation depend on the distributed nature of the inputs that caused them. To do this we take a careful look at the the distributed inputs for each system, and examine whether there is a correlation between an increase in a test case's "distributedness" and the probability that it causes a failure.

IUT	**Med. DropProb**	**Med. DupProb**
stud07	0.17	0.14
stud06	0.17	0.13
stud08	0.17	0.12
stud15	0.03	0.13
stud09	0.12	0.11
stud18	0.18	0.1
mutROR15	0.14	0.1
mutLOI35	0.13	0.1

Table 4.4: GBN Distributed Input Characteristics

Of the three inputs to GBN, we consider NumBytes, the number of bytes transferred from sender to receiver to be a non-distributed input, while DropProb and DupProb, the probability that a message is dropped and duplicated, to be distributed. DropProb and DupProb values are uniformly selected from the range $[0.0, 0.20]$, where 0.0 corresponds to a scenario in which no messages are dropped or duplicated and 0.20 corresponds to a scenario where on average one in five messages are dropped or duplicated. For these two input parameters, the level of distributedness increases as the values approach 0.2. Thus, we seek to determine if the failures that occurred are more likely to occur in scenarios with values on the high side (i.e., > 0.1) of the range.

Table 4.4 contains the median DropProb and DupProb values for failing test cases of the eight implementations. The data in this table are computed by first separating out the the failing test cases for each implementation, and then computing the median of the range of these two inputs. Considering the DupProb values in the second column, all implementations except **stud15** have a median value that is greater than 0.1, and several have median DropProb values quite close to the upper bound. Recalling the definition of median, a DropProb value of 0.17 as a median means that 50% of the failing test cases have DropProb values above 0.17. Similarly, for most of the implementations, the median DupProb value is above 0.1. In particular, the median DupProb for **stud15** is above 0.1. Thus, all eight GBN implementations are more likely to fail when tested with high values for at least one of the distributed inputs.

IUT	**Overall %** (7000)	**N-D %** (2000)	**D %** (3000)	**H-D %** (2000)
stud07	5.04	0.45	10.83	0.95
stud16	5.15	0.30	11.23	0.90
stud03	5.21	0.45	11.37	0.75
stud08	7.70	0.35	3.63	21.15
stud01	7.94	0.65	3.83	21.40
stud06	8.02	0.45	3.80	21.95
stud14	8.48	0.45	4.60	22.35
stud15	12.17	0.40	2.13	39.00
stud09	12.42	0.65	2.60	38.95

Table 4.5: LSR Distributed Input Characteristics

For LSR, the distributed nature of a scenario is largely dictated by the topology used to test the system. The seven topologies considered are shown in Figure 4.1. Of these, we consider **Duo** and **3-Line** to be non-distributed, **3-Cycle**, **4-Line** and **4-Star** to have average distribution, and **4-Cycle**, and **4-Mesh** to be highly distributed. Thus for LSR, we determine for each implementation, the failure rates within the non-distributed, distributed, and highly-distributed regions of the input space. These data are shown in Table 4.5. In this table, the second column contains the overall failure rate for each implementation, duplicated from Table 4.1 for comparison. The third column, labeled **N-D**, contains the failure percentage for non-distributed topologies. The fourth column, labeled **D** contains the failure percentage of averagely distributed topologies, and the fifth column contains the failure percentage for highly-distributed topologies which are labeled **H-D**. The number in parentheses underneath the column labels is the number of test cases that fall within the each of the distribution regions; thus for each row, the value in the second column is the weighted average of the three columns to the right. For six of the nine implementations, test cases selected from the highly-distributed partition have a significantly higher probability of causing a failure. For the other three implementations, topologies with medium distribution have the highest probability of failure. Non-distributed topologies are not effective at causing failures for any of the implementations.

IUT	**Med. DropProb**	**Med. DupProb**
1.0.29	0.11	0.10
1.1.59	0.12	0.10
1.1.60	0.11	0.10
1.1.61	0.11	0.10
1.1.91	0.11	0.10
1.2.00	0.11	0.10
1.2.01	0.11	0.10
1.2.03.3	0.11	0.10
1.2.03.4	0.11	0.10
1.2.03.5	0.11	0.10
1.2.07.1	0.11	0.10

Table 4.6: DNS Distributed Input Characterization

Due to the relative simplicity of the GBN and LSR systems, the inputs are fairly easy to classify as either distributed or functional. For DNS, the core inputs are not as easy to classify, so here we concentrate, as we did with GBN, on the network DropProb and DupProb parameters. These parameters are selected from the same range as for GBN, and have the same effect in the simulation. Table 4.6 contains the median values for DropProb and DupProb. For DNS, we see that for all implementations the median DropProb value is slightly higher than 0.1, indicating that this region of the input space has a better chance at causing failures. Interestingly, the median DupProb value is exactly 0.1 for all implementations. This makes sense when you consider that the DNS protocol has no special provisions for handling duplicate packets.

4.2.3 Technique Simulation

The white-box criteria used in our experiments are the well-known **all-blocks**, **all-branches**, and **all-uses** coverages [FW88] applied to simulation code. Specifically, we apply these criteria in aggregate to the entire simulation code base, excluding classes that are part of the discrete-event simulation core and our API layer.

System	**all-blocks**	**all-branches**	**all-uses**
GBN	100.0%	96.9%	91.5%
LSR	98.4%	94.6%	96.3%
DNS	96.0%	92.0%	95.0%

Table 4.7: Adequacy Targets

During simulation execution, coverage data are collected and stored for each test case. In order to determine target values for each criterion (100% was typically not possible due to infeasible paths), we created a test suite out of 50 test cases and verified that any uncovered elements are truly infeasible. We did this iteratively during the generation of test cases to ensure that our random generator was not missing subtle cases. Table 4.7 contains the target adequacy values used in the experiments.

We create adequate suites for these criteria by randomly selecting test cases from the universe and including them if they improve the coverage value. Specifically, for white-box techniques we employed the following algorithm for each suite:

(1) initialize coverage-value to zero

(2) randomly select a candidate test case from the universe

(3) determine the aggregate coverage value for the suite consisting of previously selected members plus the candidate

(4) if the computed coverage value improves with the addition of the candidate, add it to the suite, otherwise discard the candidate and try again

(5) stop when the coverage value meets the target value

As mentioned above, the analysis cost for white-box techniques includes the simulation time for each candidate test case selected in step #2, plus the time needed to compute the aggregate coverage for each candidate suite in step #3.

For fault-based experiments, we use system-specific black-box criteria that are defined with respect to the input domain of the simulations. We construct test suites for these criteria similarly by randomly adding test cases that will cover a previously uncovered partition. This does not result in minimal test suites, but we feel that it is a reasonable approximation of the way in which a developer would try to accomplish this task without the aid of a specialized tool.

For the **null** criterion (i.e., random testing) all suites of a particular size are adequate. Construction of a random suite of size n is simply a matter of selecting n unique test cases randomly from the universe.

Once a test suite has been selected, its effectiveness score is computed by determining the proportion of implementations that fail on at least one test case in the suite. Analogously, the mutant score is the proportion of mutants for which at least one test case fails. In our experiments we generated 200 adequate suites for each criterion, and compute the sample average and standard deviation for effectiveness and cost.

4.3 Experiment 1: Effectiveness

The first experiment we describe is aimed at verifying that test suites adequate with respect to simple white-box simulation-code-coverage criteria are effective at testing implementations. To do this, we compare white-box techniques to randomly selected test suites of fixed size in several different ways.

System	**Criterion**	**Size**	E
GBN	all-blocks	3.1	0.19
	all-branches	4.1	0.22
	all-uses	4.7	0.35
LSR	all-blocks	1.60	0.46
	all-branches	3.05	0.64
	all-uses	2.92	0.61
DNS	all-blocks	9.92	0.936
	all-branches	12.76	0.98
	all-uses	12.21	0.933

Table 4.8: White-Box Effectiveness and Size

Table 4.8 contains the initial data obtained from analyzing adequate suites for each criterion. In Table 4.8, the second column contains the average size of the adequate suites, measured in test cases, and the third column contains the average effectiveness of the same suites. The differences in effectiveness shown in these tables are all statistically significant, except for that between **all-branches** and **all-uses** for DNS. Therefore, for GBN the criteria ranking is the canonical ranking one would expect from the literature:

$$\textbf{all-blocks} < \textbf{all-branches} < \textbf{all-uses}.$$

For LSR the ranking is:

$$\textbf{all-blocks} < \textbf{all-uses} < \textbf{all-branches}.$$

Finally, for DNS the ranking is:

$$\textbf{all-blocks} < \textbf{all-branches}$$

$$\textbf{all-uses} < \textbf{all-branches}.$$

We believe that **all-uses** is not significantly more effective than **all-branches** for LSR and DNS because the simulations of these two systems make extensive use of collections such as lists and maps to store data resulting in an imprecise data-flow analysis. We do not believe that these inconsistent relationships are due to a failing in specification-based testing, but rather are fundamental to the use of adequacy criteria. Experimental results reported by others support this observation: **all-uses** is shown to be better than **all-branches** on average, but not universally [FW93a].

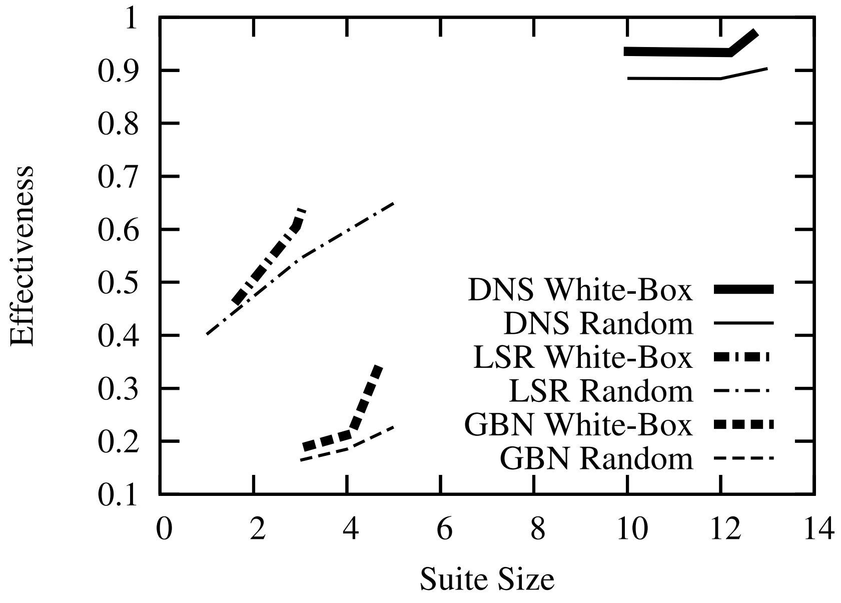


Figure 4.2: Absolute White-Box Effectiveness

To evaluate the absolute effectiveness of the white-box techniques, we generate random suites with sizes corresponding to the average sizes of the white-box techniques. Figure 4.2 shows plots of suite size versus effectiveness for the random and white-box criteria. In this figure, the DNS plots are on the upper right-hand side, GBN is on the lower left, and LSR is in the middle left. Notice that in each case the white-box line is above the random line, graphically demonstrating our claim. Hypothesis testing rigorously corroborates this result by showing that the improvement in effectiveness of white-box techniques compared to same-sized random suites is statistically significant.

Random testing is a general approach to testing and can be considered an adequacy criterion only with respect to a particular universe of test cases and a particular target suite size. In other words, "random testing" provides no way to objectively determine if test cases are suitably broad (i.e., that the test-case generation method is good), and it gives no means of rationally deciding when enough testing has been done. When using random testing, a software system is tested until it must be released, as determined by other factors; this is the inverse of criteria-based approaches in which software is released when objective stopping conditions are satisfied. However, because both the cost and effectiveness of a randomly selected suite is determined completely by its size, it can be used as a scale against which to fix the effectiveness of other, more rational, techniques.

So, to evaluate cost effectiveness, we first randomly select 200 random suites for a range of different suite sizes and determine their effectiveness and cost. Then, for each white-box criterion we use statistical inference to determine the relationship, if any, that exists between the white-box technique and the random suites.

	Random Size									
Criterion	3	4	5	6	7	8	9	10	11	12
all-blocks C	-	+	+	+	+	+	+	+	+	+
E	+	=	-	-	-	-	-	-	-	-
all-branches C	-	-	+	+	+	+	+	+	+	+
E	+	+	-	-	-	-	-	-	-	-
all-uses C	-	-	-	-	+	+	+	+	+	+
E	+	+	+	+	+	+	+	=	=	-

Table 4.9: GBN White-Box Cost Effectiveness

Table 4.9 contains the results of this analysis for GBN. In this table, the columns represent random suites of the labeled size. The rows represent either the cost, labeled C, or effectiveness, labeled E for each criterion. The cells of these tables contain either "-", "=", or "+" which are interpreted generally as, "worse than", "equivalent to", and "better than", respectively. So, for cost rows, cells having a "-" symbol indicate that the white-box technique is worse (i.e., more expensive) than the random suite of that size. Similarly, in effectiveness rows, cells marked with a "+" symbol, indicate that the white-box technique is significantly better than random suites. Finally, cells marked with "=" symbols indicate that there is no statistically significant relationship between the two criteria, in either direction.

In Table 4.9 there are some general trends that help understand what is being displayed. First, each of the white-box criteria are worse in terms of cost than small random suites. Recall from Section 4.2, white-box techniques have an additional cost component due to analysis, C_a. This means that all else being equal, white-box techniques do worse in terms of cost than random suites of the same size. Moving to the right in the cost rows, there is eventually a point at which the white-box technique breaks-even or is better than random suites. Intuitively, this happens when the random suites become sufficiently large that their execution cost overcomes the analysis cost incurred by the white-box technique.

Conversely, in terms of effectiveness, the white-box techniques start off better than the random suites. This is a combination of the white-box suite being bigger, and also being inherently more effective. Moving to the right, there comes a point at which the two techniques are equivalent and white-box techniques are worse than random ones. Intuitively, this occurs when the random suites become large enough to overcome the inherent advantage of the white-box analysis.

The data shown in Figure 4.2 already establish that white-box techniques are inherently more effective than random suites of the same size. Here we are interested in determining if cost of analysis needed to obtain the white-box suite is so large that it would be more cost effective to simply use randomly selected suites. In other words, we are trying to determine if the tester would have a more effective test suite had they simply used the analysis time needed to select white-box suites to instead execute more tests.

Reading this from the tables is simply a matter of determining if the point at which the white-box technique breaks even in terms of cost is at a suite size less than or equal to the break-even point of effectiveness. If so, then the technique is not only effective, but also cost effective. For example, **all-blocks** with GBN becomes cheaper at suites of size 4, and at size four **all-blocks** has equivalent effectiveness, therefore it is more cost effective than random testing. Following this procedure, **all-branches** for GBN has equivalent cost effectiveness since both measures break even at the same point. Lastly, **all-uses** for GBN is much more cost effective than random testing since its cost breaks even at size 7, but not until random suites contain 12 test cases do they become more effective.

	Random Size					
Criterion	1	2	3	4	5	6
all-blocks C	-	+	+	+	+	+
E	+	-	-	-	-	-
all-branches C	-	-	-	-	+	+
E	+	+	+	+	=	-
all-uses C	-	-	-	-	+	+
E	+	+	+	=	-	-

Table 4.10: LSR White-Box Cost Effectiveness

The same data are shown for LSR in Table 4.10. These data show that **all-blocks** has equivalent cost effectiveness, **all-branches** is more cost effective than random, and **all-uses** is less cost effective. Again, we believe that **all-uses** is not effective for LSR because of the use of collections.

Similarly, the cost effectiveness data for DNS is listed in Table 4.11. Here, **all-blocks** and **all-branches** are significantly more cost-effective, and **all-uses** is more cost-effective. This is interesting because in terms of absolute effectiveness, **all-blocks** and **all-uses** are statistically indistinguishable, but when the analysis cost is considered, **all-blocks** is better.

4.4 Experiment 2: Boosting Criteria

Experiment 2 is aimed at determining if the fault-based analysis technique can be used to improve the effectiveness of white-box and black-box criteria. This experiment

	Random Size										
Criterion	12	13	14	15	16	17	18	19	20	21	...	26
all-blocks C	-	+	+	+	+	+	+	+	+	+	+	+
E	+	+	+	=	=	-	-	-	-	-	-	-
all-branches C	-	-	-	-	-	+	+	+	+	+	+	+
E	+	+	+	+	+	+	+	+	+	=	=	-
all-uses C	-	-	-	-	+	+	+	+	+	+	+	+
E	+	+	+	=	=	-	-	-	-	-	-	-

Table 4.11: DNS White-Box Cost Effectiveness

corresponds to the **boosting** usage scenario in which a developer selects more than one adequate suite during analysis, and then uses the mutant score to decide which one to use on the implementation. There are two costs associated with this usage, the first comes from having to select multiple adequate test suites, and the second is the cost of performing the fault-based analysis by executing each test case on all specification mutants. Thus, the main goal of this experiment is to determine the smallest number of additional suites at which the developer sees a statistically significant improvement in effectiveness.

For this experiment, we devised two black-box test criteria for each system. The intent with these criteria is that they are plausible alternatives to white-box techniques that a developer might consider using. Details of these black-box criteria are provided below. These criteria are all similar in that they define their test requirements with respect to the function and distribution inputs spaces rather than the code of the simulation. Nevertheless, the presence of the simulation code is critical to the technique, as we explain below.

The black-box criteria are:

- **GBN IP-1**: This criterion simply divides value ranges of each parameter into four equally sized partitions. A suite is adequate for this criterion when each partition is represented by at least one test case value.

- **GBN IP-2**: This criterion divides the NumBytes value range into seven equally sized partitions. For DropProb and DupProb, the value range [0,0.1) is split in half, while the range [0.1,0.2) is split into four bins, thereby placing an emphasis on higher drop and duplication rates. Again, a test suite is adequate for this criterion when each partition is accounted for. The underlying intuition here is that higher drop and duplication rates make a test exercise more of the retransmission and window manipulation code.

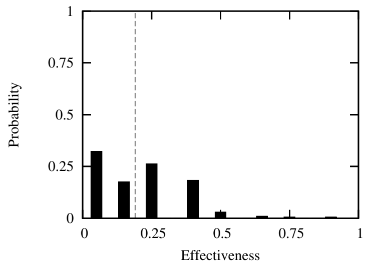
- **LSR Pair-50**: This criterion ensures that paths between pairs of routers are well represented by a test suite. A test suite is adequate for this criterion if all topologies are represented and 50% of the possible router pairs are accounted for.

- **LSR Quartet**: This criterion simply requires that all 4-router topologies in Figure 4.1 are represented. The intuition behind this is that 4-router topologies are more complicated and therefore will make better test scenarios.

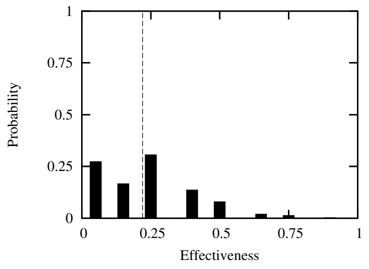
- **DNS All-resource-types**: This criterion requires that a query is made for each of the core DNS resource types. Intuitively, this exercises the breadth of the user inputs.

- **Typical usage**: This criterion attempts to exercise the DNS features that are used most often. Thus, it requires a valid query, a name-error query, and a no-data query to be issued for A records (maps names to addresses) and MX records (maps domain names to mail exchanges). These two record types are those most often used, and so this criterion targets this narrow usage pattern.

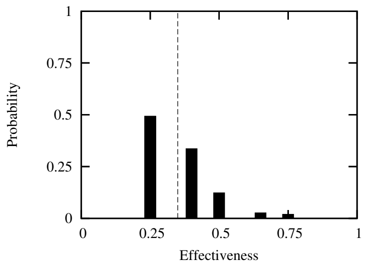
For this experiment we use the 200 adequate suites for the white-box techniques that were created for experiment 1, and we create 200 new suites for each of the black-box techniques. To underscore the inherent variability in the adequate test suites, we



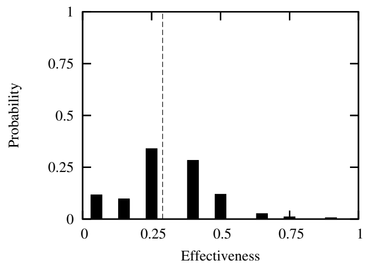
(a) all-blocks



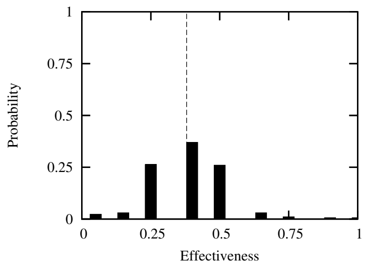
(b) all-branches



(c) all-uses

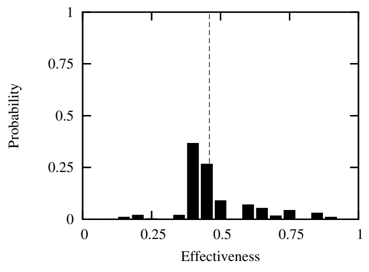


(d) IP-1

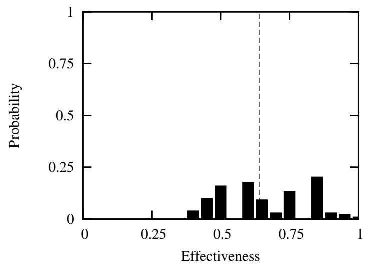


(e) IP-2

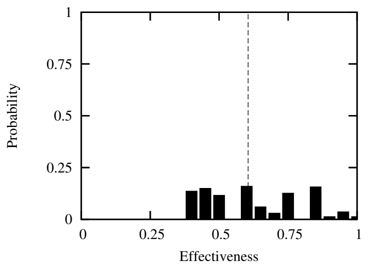
Figure 4.3: GBN Criteria Effectiveness Distributions



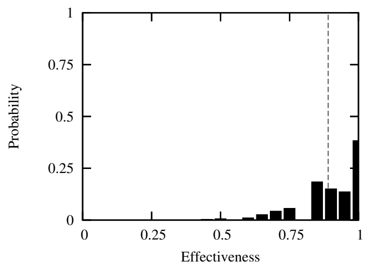
(a) all-blocks



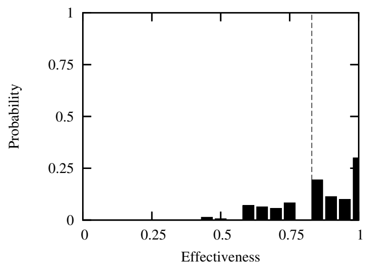
(b) all-branches



(c) all-uses

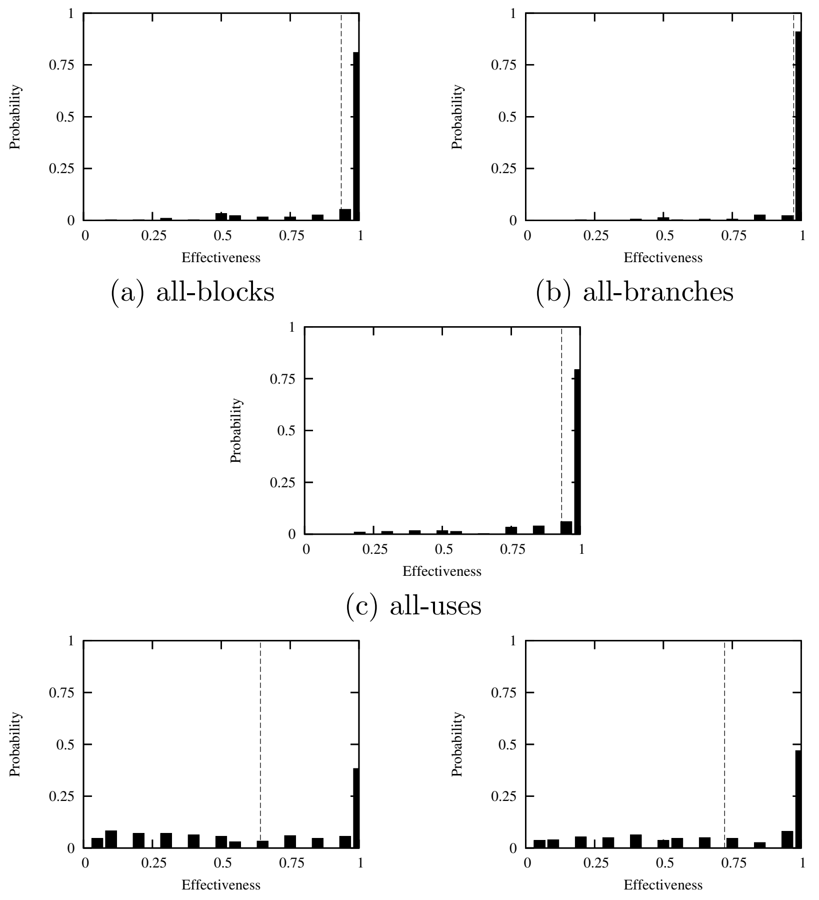


(d) IP-1



(e) IP-2

Figure 4.4: LSR Criteria Effectiveness Distributions



(a) all-blocks (b) all-branches

(c) all-uses

(d) Resources (e) TypicalUsage

Figure 4.5: DNS Criteria Effectiveness Distributions

show the probability distributions for each of the criteria in Figures 4.3, 4.4 and 4.5. Each of these plots has effectiveness on the x-axis, and probability on the y-axis, the average effectiveness is shown as a vertical dashed line. The height of each bar in the plots represents the probability that an adequate test suite has the corresponding effectiveness. For instance, consider the plot shown in Figure 4.3a, the distribution for **all-blocks** for GBN. The bar at 0.25 on the x-axis has a height of 0.25 indicating that for this criterion, there is a 25% chance that an adequate suite will have an effectiveness score of 0.25. Since the average effectiveness of **all-blocks** for GBN is only 0.19, selecting a suite, by chance, that actually had 0.25 effectiveness would result in a significant benefit for the tester. This is exactly the kind of improvement that we are trying to achieve by using fault-based analysis to predict where candidate test suites fall in the effectiveness distribution.

To reiterate, the procedure we follow for this experiment is as follows:

(1) Choose M suites at random from the set of suites adequate for the criterion in question.

(2) Select the suite with highest mutant score, and determine its effectiveness.

We perform this procedure 100 times each with values of M starting at 2 and stopping at a value at which the fault-based technique is statistically better than the baseline effectiveness of the criterion (with $\alpha = 0.05$).

Criterion	**min** M	E	$E+$
all-blocks	2	0.19	0.27
all-branches	2	0.22	0.29
all-uses	–	0.34	0.34
IP-1	2	0.29	0.32
IP-2	3	0.37	0.41

Table 4.12: Boosting GBN Criteria

Tables 4.12, 4.13 and 4.14 shows the results of these experiments. In these tables,

Criterion	**min** M	E	$E+$
all-blocks	2	0.46	0.51
all-branches	3	0.64	0.69
all-uses	2	0.61	0.68
Quartet	3	0.84	0.87
Pair-50	3	0.89	0.92

Table 4.13: Boosting LSR Criteria

Criterion	**min** M	E	$E+$
all-blocks	8	0.936	0.9545
all-branches	–	0.973	0.973
all-uses	2	0.933	0.9678
Resources	4	0.642	0.69
TypicalUsage	4	0.722	0.823

Table 4.14: Boosting DNS Criteria

the second column contains the number of adequate suites needed to achieve a significant boost in the effectiveness, the third column contains the baseline effectiveness and the fourth column the boosted effectiveness. The **min** M values reported in the second column of these tables is the total number of adequate suites needed, including the suite needed for the conventional usage scenario.

For GBN, shown in Table 4.12, three of the five criteria needed only one additional adequate suite to improve the effectiveness and one, **IP-2** needed two additional suites. For **all-uses**, we examined up to seven additional adequate suites without achieving a significant boost in effectiveness.

The data shown in Tables 4.13 and 4.14 for LSR and DNS are very similar to this. For many of the criteria, the minimum multiplier M is 3 or less, meaning that the developer first sees **significant** improvement of effectiveness with only 2 additional adequate suites. For DNS, **all-blocks** is harder to boost than the others, requiring seven additional test suites before a significant improvement is achieved. Also for DNS, we were unable to boost **all-branches**, even trying up to seven additional suites. For DNS, **all-branches** is the most effective criterion we evaluated with an effectiveness of 0.98. Referring to the distribution of adequate suites shown in Figure 4.5(b), it is understandable that this criterion would be difficult to boost since virtually all adequate test suites are in the rightmost bin.

While conducting this experiment, we also tested the opposite hypotheses, which is that the effectiveness drops when increasing the number of suites considered. This never occurred.

An important aspect of this result is that the technique works for both black-box and white-box adequacy criteria. If a developer prefers not to use the simulation code as a basis for defining adequacy, then they can still use it to improve the effectiveness of any other test suites with which they are working.

Hypothesis	m. **p-value**	e. **p-value**
IP-1>all-blocks	< 0.001	< 0.001
IP-1>all-branches	< 0.001	< 0.001
IP-2>all-uses	< 0.001	0.004
all-branches>all-blocks	< 0.001	0.002
all-uses>all-blocks	< 0.001	< 0.001
IP-2>all-blocks	< 0.001	< 0.001
all-uses>all-branches	< 0.001	< 0.001
IP-2>all-branches	< 0.001	< 0.001
IP-2>IP-1	< 0.001	< 0.001
all-uses>IP-1	< 0.001	< 0.001

Table 4.15: GBN Experiment 3 Data

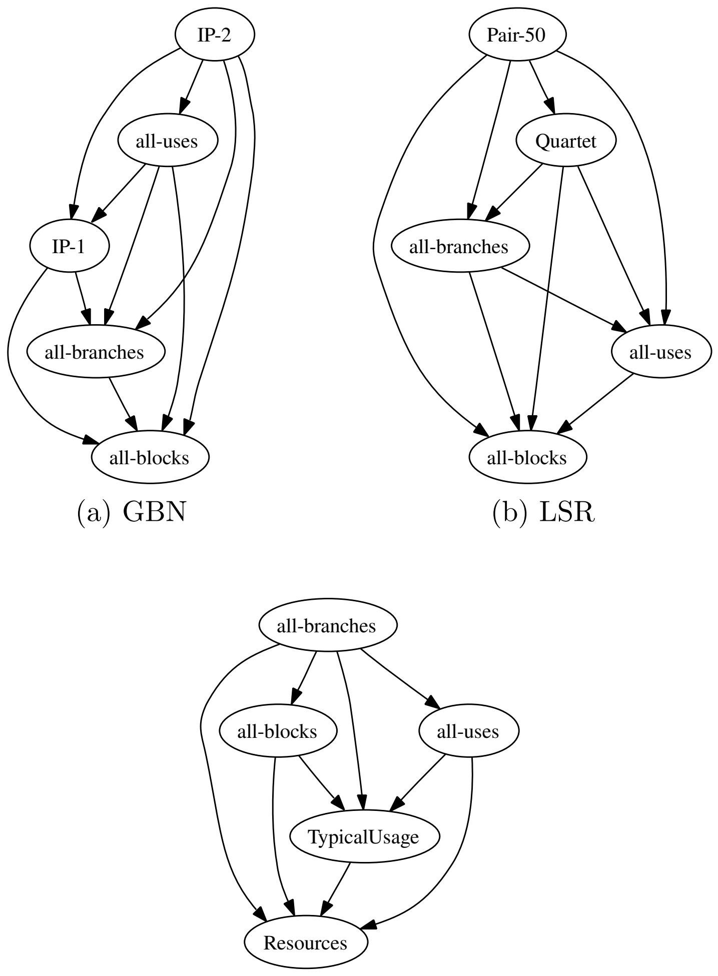
4.5 Experiment 3: Ranking Criteria

Our final experiment, Experiment 3, is aimed at determining how effective the fault-based analysis technique is at predicting relationships between **criteria**. This experiment corresponds to the **Ranking** usage scenario in which a developer is interested in evaluating several plausible adequacy criteria to determine which to use.

From the previous two experiments we have 200 adequate suites for 5 criteria for each of the systems. Figure 4.6 depicts all statistically significant relationships with $\alpha = 0.05$. For example, Figure 4.6a indicates that **all-uses** has a significantly higher effectiveness than **IP-1**, which is itself higher than **all-branches** and **all-blocks**, etc. This figure underscores the difficulty practitioners face when choosing between criteria: there is no rational way for a developer to know where **IP-1** and **IP-2** fall with respect to the white-box criteria.

To determine our ability to predict these same relationships, we simply compute the mutant scores for each of adequate suites, and apply hypothesis testing using the average and standard deviation of these values instead of effectiveness.

Our experiments show that the fault-based analysis predicts the actual relationships almost perfectly. Since the predicted relationship graphs are virtually identical



(a) GBN (b) LSR

(c) DNS

Figure 4.6: Criteria Relationships

Hypothesis	m. **p-value**	e. **p-value**
Quartet>all-blocks	< 0.001	< 0.001
Pair-50>all-blocks	< 0.001	< 0.001
all-branches>all-uses	< 0.001	0.01
all-branches>all-blocks	< 0.001	< 0.001
Quartet>all-uses	< 0.001	< 0.001
all-uses>all-blocks	< 0.001	< 0.001
Pair-50>Quartet	0.02	< 0.001
Pair-50>all-uses	< 0.001	< 0.001
Quartet>all-branches	< 0.001	< 0.001
Pair-50>all-branches	< 0.001	< 0.001

Table 4.16: LSR Criteria **p-values**

Hypothesis	m. **p-value**	e. **p-value**
all-branches>Resources	< 0.001	< 0.001
all-uses>Resources	< 0.001	< 0.001
all-branches>TypicalUsage	< 0.001	< 0.001
all-uses>TypicalUsage	< 0.001	< 0.001
TypicalUsage>Resources	< 0.001	< 0.001
all-branches>all-uses	< 0.001	< 0.001
all-blocks>Resources	< 0.001	< 0.001
all-blocks>TypicalUsage	< 0.001	< 0.001
all-branches>all-blocks	< 0.001	< 0.001
all-uses>all-blocks	< 0.001	0.5980946

Table 4.17: DNS Criteria **p-values**

to those shown in Figure 4.6, we do not report them here. Instead, Tables 4.15, 4.16 and 4.17 show the computed **p-values** for relationships between the criteria. Each row Table 4.15 corresponds to an edge in Figure 4.6a.

The hypothesized relationship between the criteria are listed in the first column. For each relationship, the second column reports the **p-value** for the mutant score (simulation) while the third column reports the **p-value** for the effectiveness (implementation). Notice that the correspondence is virtually exact with **p-values** all lower than 0.02. In particular, for LSR the fault-based analysis accurately predicts the reversal of the relation between **all-uses** and **all-branches**.

For DNS, the relationship between these two criteria represents the only situation in which the predicted relationship does not match the actual relationship; this is highlighted in the bottom row of Table 4.17. In this situation, the fault-based analysis predicts, with a strong level of confidence, that **all-uses** will be more effective than **all-blocks**. However, this relationship is not borne out by the effectiveness data from the implementations. So, in statistical terms the fault-based technique would have caused the engineer to favor a criterion that was not actually more effective than another one considered. However, we should point out, that in real terms, the average effectiveness of the two criteria is practically identical, as show in Table 4.8.

4.6 Summary of Results

The results of the three experiments presented above are compelling and provide strong validation for the ideas we present in this dissertation. The results of Experiment 1 show that discrete-event simulations can be used effectively as the basis for definition and evaluation of adequacy criteria for implementation testing. The three white-box techniques we evaluated are some of the simplest available, and the data show that they are clearly more effective than the baseline provided by randomly selected suites. Additionally, our cost effectiveness analysis shows that time spent to simulate and select white-box test suites is small enough that for most of the criteria it does not put it at a disadvantage compared to random testing.

The results from Experiment 2 show that for virtually all of the white-box and black-box adequacy criteria we evaluated, a fault based analysis of the simulation code can be used to boost effectiveness significantly.

Finally, we show that for the collection of arbitrarily selected adequacy criteria that we use here, we can predict the actual effectiveness relationships very accurately through analysis and execution of the simulations.

Collectively, these results validate our research hypotheses that (1) discrete-event simulations can be used effectively to test implementations, (2) fault-based analysis can be used to predict the relative effectiveness of adequate test suites, and (3) the fault-based technique can also be used to predict the relative effectiveness of adequacy criteria.

4.7 Threats to Validity

While we feel confident that the experimental method we used in conducting this research is sound and that the results are valid, we highlight potential threats to our conclusions.

The chief threat to the construct validity of our approach is in the definition of effectiveness that we adopt. We assume that a specification-based testing technique is most effective when it is able to identify a broad range of faulty implementations, but others might see this differently. For example, implementation failure rates could be used to include the relative difficulty of finding bugs in the effectiveness score.

The main internal threat is that our experiments validate our claims because the simulation code mimics the structure of implementation code closer than it would in practice. This is not likely, considering that the simulations were created by the author, who had no contact with the students or the materials they were presented with before receiving the implementations of GBN and LSR. Similarly for the implementation of DNS.

Finally, as the scope of our empirical study is limited to three systems, it is difficult to argue that our results are externally valid. However, we view this work as a necessary first step in establishing the utility of simulation-based testing for distributed systems, and make no claims about its broad applicability. More work is required to understand the conditions under which our techniques are applicable and effective, but it seems clear that it is both applicable and effective on the systems described here.

Chapter 5

Related Work

The ideas presented in this dissertation are primarily based on a rich corpus of research related to test adequacy criteria. Since the seminal papers by Goodenough and Gerhart [GG75, GG77], the development of techniques and theories of test adequacy has been a major focus of the research community; important results in this area are presented in Section 5.1.2. Other foundational research areas are empirical fault studies, mutation analysis and experimental evaluation of testing techniques which are discussed in Section 5.1.1, Section 5.1.3 and Section 5.1.4 respectively.

Other specification based techniques and system-level testing techniques, and testing methods specifically targeted at distributed systems are discussed in Section 5.2.

5.1 Foundations

The work represented by this dissertation is built upon several different areas of software engineering. The fault study we conducted was inspired and assisted by the reports of others who have attempted to classify defects in an effort to inform and improve software testing. The literature in this area is surveyed in Section 5.1.1. For our main idea of simulation-based testing, the rich literature on test adequacy criteria was critical, and the fault-based analysis we performed is based on work in the area of mutation analysis. Finally, we base our entire experimental framework on long-established methods. We now discuss each of these areas in more detail.

5.1.1 Fault Studies

There is a large body of work devoted to the use of defect reports to inform the development of testing techniques and methodologies. For the most part, these are aimed at studying the faults themselves rather than the failure-producing scenarios that are the target of our study. For example, Fenton and Ohlsson [FO00] describe an extensive study of faults found both pre- and post-release in two releases of a large commercial software system. This is an interesting study in which the data are used to confirm and refute various hypotheses representing the conventional wisdom about fault location, module complexity and fault predictability. Similarly, Ostrand and Weyuker [OW02] describe a similar empirical study of many releases of a large commercial system. They present a number of interesting results related to correlations between module size and fault-proneness and the relationships between pre- and post-release faults in modules. In a subsequent study of the same system [OWB04] they develop a predictor for fault-prone files. As pointed out by Ostrand, Weyuker and Bell [OWB04], their research goal is to determine **where** to concentrate testing effort. By contrast, our study is aimed at understanding **how** to test, and for this the failure scenarios, rather than the underlying faults are considered.

In their study, Leszak, Perry, and Stoll examine both bug reports and subsequent change requests to determine the root cause of the defects across all phases and activities in the development process [LPS00, LPS02]. As part of their root-cause analysis, Leszak, Perry and Stoll describe the development of countermeasures to help improve the development effort. One of the countermeasures listed was improving unit testing. However, this is motivated by the root-cause information not the test scenario information that could have potentially been gleaned from the initial defect reports.

In orthogonal defect classification (ODC) [CHBC93, CBC$^+$92], both a **defect type** and a **defect trigger** are identified for each defect. The defect type is similar to a root cause, and is recorded by the software engineer making the changes needed to fix the defect. Conversely, the defect trigger is a coarsely grained categorization of the circumstances that lead to the defect being reported. At a high level, the defect trigger is used to improve the verification process by determining why the circumstances that caused the defect where not seen during testing. Our input classification is similar in intent to the defect trigger classification. However, ours is more detailed, as well as being more directly targeted at distributed systems. ODC is a general-purpose defect analysis technique that is targeted at improving the software development process for a project in general, while we are primarily concerned with exploring the feasibility of creating a new testing method.

More recently, Li et al. [LSH$^+$04] studied user defect reports from several open source projects, and fault and failure reports from commercial software, to study defect occurrence models for large, multi-release software systems, including operating systems and a distributed middleware system. While the raw data for their study are similar to ours, Li et al. are primarily interested in frequency data. Interestingly, they also note that user defect reports in open source systems often contain duplicates and invalid reports, though in their study, these reports are considered in scope, since a developer has to review each one, regardless of its ultimate utility.

5.1.2 Test Adequacy Criteria

In 1972 Dijkstra noted that while testing might demonstrate the presence of bugs in a software system, it could never demonstrate the absence of bugs [Dij72]. Following this, Goodenough and Gerhart [GG75, GG77] identified the problem of determining when "enough" testing has been done as a central problem in software testing. The development of test adequacy criteria, also called "coverage" criteria in the literature, has been a major focus of research. Adequacy criteria have been developed to structure the testing process in three ways: (1) providing quantitative measures of the effectiveness of existing test cases, (2) serving as objective stopping criteria for the test-case creation activity, and (3) specifying the observations that must be made to compute the metrics used in (1) [ZHM97]. In their 1997 survey, Zhu, Hall and May identified two orthogonal dimensions for test adequacy criteria: (1) source of information and (2) underlying testing approaches. While their survey is focused on unit testing criteria, we have adopted this terminology in the following sections, where we summarize important criteria at each of the common levels of testing: unit, integration, and system. The sources of information mentioned by Zhu, Hall and May are:

- **program** – test criteria are based on the full source of the program being tested;
- **interface** – criteria are based on only the interface information, namely the types and valid ranges of inputs to the software; and
- **specification** – test criteria are based features of the software's specification.

While the underlying testing strategies are:

- **structural** – criteria are based on elements of structures in the source of information being used;
- **fault-based** – criteria are based on the fault-detecting ability of the tests; and
- **error-based** – criteria are based on knowledge of where errors typically occur.

There are a few important terms associated with adequacy criteria that we introduce before proceeding. First, an important property of an adequacy criteria is its **finite applicability** [ZH93]. A criterion is finitely applicable if it can be satisfied by a finite test set. This obviously has bearing on the practicality of a criterion, since the use of testing goals that cannot be attained in a finite amount of time do not make sense from a management perspective. As an example, structural program-based criteria (discussed below) are not finitely applicable in the presence of code that cannot be exercised by any valid inputs, often called **infeasible** or **dead code** [ZHM97]. These criteria are usually redefined to cover only the so-called **feasible** elements, which brings them into conflict with **undecidability** since it is not decidable if a particular program element is feasible [ZHM97].

Second, when comparing different criteria, the **subsumes** relation often comes into play. A criterion A subsumes another criterion B if B is always satisfied when A is satisfied.

5.1.2.1 *

Unit Test Adequacy Criteria

Unit testing focuses on evaluating software at a low enough level that each unit can be tested in isolation [BD04]. Historically, test adequacy criteria for unit testing has received the most attention from the research community. As mentioned earlier, a comprehensive summary of this topic was produced by Zhu, Hall and May [ZHM97], so this section merely summarizes the structural criteria that have received the most attention and are most directly related to this work.

Structural criteria come in two flavors, program-based and specification-based. Program-based structural criteria are typically based on the flow-graph model of program structure. Briefly, a flow-graph is a graph representing the flow of control through a procedure. Nodes in the flow-graph represent basic blocks of statements, while edges represent transfer of control between the basic blocks. Generally, each edge is associated with a predicate that guards the transfer of control between the nodes it connects.

The most basic adequacy criterion is **statement coverage** [Het84]; this requires that each node in a program's flow-graph be executed, corresponding to each individual statement in the source being executed. **Branch coverage** is slightly more stringent and requires that each edge in the control flow graph be exercised by a test suite. Branch coverage subsumes statement coverage [Het84]. Both statement and branch coverage can be made finitely applicable by excluding dead code, as mentioned above.

The next criterion in this progression is **path coverage** which requires that all distinct paths through the flow-graph are exercised. This criteria is not finitely applicable and for this reason is not generally useful. Instead, heuristic approaches were created to test some "interesting" set of paths. These approaches include strategies to limit the length of the paths [Gou83], their level [Pai75, Pai78], and to find "linearly independent" paths through the code [McC76, McC83, MS83].

The criteria mentioned above are collectively referred to as "control based" since they are defined with respect to the basic flow-graph. The other major family of unit test adequacy criteria are termed "data based" since they refer to a flow-graph that has been augmented with information about the data definitions and uses. Informally, a variable is defined when it appears on the left-hand side of an assignment operator, and it is used when it appears on the right-hand side of an assignment, as an actual parameter to a procedure call (computation use), or in a predicate (predicate use). With these definitions, a flow-graph is augmented so that each node can be marked as a definition or use node with respect to a particular variable.

Basic data flow coverage criteria are **all-definitions**, **all-uses** and **all-definition-use-paths (all-DU-paths)** [FW88]. In **all-definitions** coverage, the set of execution paths through the augmented flow-graph must contain at least one path from each definition node to a use node that is covered by that definition. Similarly, **all-uses** coverage requires that the set of execution paths covers all use nodes of variables; this subsumes **all-definitions**. Finally, **all-DU-paths** requires that all paths between definitions and uses of variables are covered. **All-DU-paths** coverage subsumes **all-uses**, and is subsumed by **path** coverage. Rapps and Weyuker [RW85], also defined coverage criteria that emphasize computation and predicate uses differently, these are **all-c-uses/some-p-uses**, **all-p-uses/some-c-uses**, **all-p-uses**, and **all-c-uses**.

Other families of data-flow criteria are based on the interactions of variables [Nta84], and on combinations of definitions [LK83]. Podgurski and Clarke extended control- and data-based testing strategies by introducing the notions of semantic and syntactic dependence and then defining coverage of these relations [PC89, PC90].

Other structural testing strategies are based on specifications and we discuss these techniques in Section 5.2.1.

While we primarily focused on the use of discrete-event simulations because of practitioner acceptance, it seems clear from the long and rich history of research into unit-level adequacy criteria that testers have more experience testing software artifacts, and thus would be more comfortable creating tests for a relatively simple program like a discrete-event simulation than other formal artifacts. While this is a qualitative argument that is hard to justify, clearly any of the unit-level techniques discussed in this section could be applied directly to the code of a discrete-event simulation.

5.1.2.2 *

Integration Test Adequacy Criteria

As the name implies, integration testing involves the verification of combinations of units and components and is typically performed after the units have been tested. Faults that are due to problems in the interfaces between units cannot be identified by any amount of unit testing; so integration testing techniques are typically geared towards these faults.

One approach to integration testing involves testing combinations of procedures (the typical "units" of many procedural languages) [HS89, HS91]. This technique is based on augmenting traditional flow-graphs with interprocedural information related to procedure call and return sites. With these interprocedural flow-graphs in place, the basic coverage criteria used for intraprocedural flow-graphs can be used for procedure combinations.

Memon, Soffa and Pollack have proposed a suite of coverage criteria for testing graphical user interfaces (GUIs) [MSP01]. They present a model of GUIs that divides the interface into components that react to events. They define three criteria for testing individual components: **event**, **event-interaction**, and **length-n-event-sequence**. **Event** coverage specifies that the set of event sequences used to test a GUI must contain all events at least once. **Event-interaction** coverage requires that after a particular event, e, all events that interact with e are exercised. **Length-n-event-sequence** coverage requires that all possible event sequences of length n are used to test the GUI. Memon, Soffa, and Pollack also define three criteria for testing combinations of components: **invocation**, **invocation-termination**, and **inter-component-length-n-event-sequence**. Informally, **invocation** coverage ensures that each component can launch all other components that it interacts with. Similarly, **invocation-termination** coverage ensures that each component can launch all other components that it interacts with, and that those components can terminate immediately. Finally, **inter-component-length-n-event-sequence** coverage requires that all event sequences that start in one component, and end in another component of length n are included in the test inputs.

Kapfhammer and Soffa [KS03], have defined a family of data-flow based criteria for database applications. These criteria are defined with respect to a "database interaction control flow graph" (DICFG). Kapfhammer and Soffa use static analysis to determine a set of database interaction points that can be categorized as either **defining**, **defining-using**, or **using**. The SQL commands `DELETE` and `UPDATE` are categorized as either **defining** or **defining-using**, `INSERT` interaction points are **defining**, and `SELECT` interaction points are considered to be **using**. Using this information the coverage criteria are similar to the traditional **all-DU-paths** criterion mentioned above, except that instead of being defined with respect to different variables, these criteria are defined at different levels of granularity. For instance, at the "database" level, an `INSERT` into any table of a database is considered to **define** that database, and any subsequent **SELECT** from any table in that database is a **use**. The other levels of granularity used are relations, attributed, records, and attribute values.

Jin and Offutt considered adequacy criteria for integration testing based on classes of coupling between software units [JO98]. They defined four types of coupling that are useful for testing: **call coupling**, **parameter coupling**, **shared data coupling**, and **external device coupling**. Using these coupling types and the data-flow graphs for each unit individually, they define a number of criteria that refer to procedure calls, shared data, shared media and the definitions/uses between them.

While not specifically addressed in our experiments, we feel that applying some of the low-level integration techniques to pairs or clusters of processes in the discrete-event environment could be a powerful means of selecting tests that address the specific interactions that exist between components of distributed systems.

5.1.3 Mutation Analysis

In mutation analysis, models of likely or potential faults are used to guide the testing process. In mutation testing [DLS78], the engineer applies mutation operators [OLR+96] to the source code to systematically create a set of programs that are different from the original version by a few statements. A mutation-adequate test suite is one that is able to “kill” all of the non-equivalent mutants.

Mutation testing is based on two complementary theories [DLS78]. The **competent programmer hypothesis** states that an incorrect program will differ by only a few statements from a correct one; intuitively, this means that in realistic situations a program is close to being correct. The **coupling effect** states that tests that are effective at killing synthetic mutants will also be effective at finding naturally occurring faults in an implementation.

Traditionally, mutation analysis has been targeted at syntactic structures within the body of procedures or methods. Recently, work has been done to define mutation operators for object-oriented (i.e., inter-class) constructs as well [MKO02, MOK05].

5.1.4 Empirical Evaluation of Testing Techniques

The comparison of competing testing strategies has been the focus of research for decades. Theoretical studies using simulation-based [DN81, HT90, Nta98] and analytical [FW93b, FW93c, Hie02, WJ91] methods have been conducted. Definitive results for general relationships among testing strategies are not generally available, and the focus of more recent work has been empirical evaluations.

An experimentation and analysis method was presented by Frankl and Weiss 15 years ago [FW91, FW93a]. This method relies on the generation of a large universe of test cases from which many adequate suites can be selected by the criteria being evaluated. For each criterion, the proportion of fault-detecting suites is computed, and statistical inferences is used to test hypotheses about the relative effectiveness of criteria. Frankl, Weiss, and colleagues have used this method to compare **all-uses** and **all-edges** [FI98, FW91, FW93a], and **all-uses** and **mutation** testing [FWH97]. Hutchins et al. [HFGO94] used a similar empirical framework to compare **all-edges** and **all-DUs**. Recently, Briand et al. [BLW04] extended the basic experimental approach by introducing a simulation step in which different testing strategies are simulated after the initial data are collected. In this study, we employ the Frankl and Weiss experimental method with the test strategy simulation introduced by Briand et al.

Briand and colleagues have also recently conducted a study of the use of automatically generated mutants as subjects in empirical studies. This is certainly a useful tool for researchers, since it provides the ability automatically generate a large number of potentially faulty implementations. Andrews et al. used some of the canonical examples from the testing literature for which there were naturally occurring bugs, hand-seeded bugs, and automatically generated mutants [ABL05]. The goal of their study is to determine which method of introducing faults is more representative of naturally occurring bugs: hand seeding or mutation. They conclude that using mutation operators results in subject systems that are more representative since hand-seeded faults are often more subtle and harder to uncover than natural faults.

5.2 Related Work

While there are few testing techniques that are in direct competition with the simulation-based technique we present here, there is a significant body of research on specification-based techniques, fault-based testing, and various methods related to testing distributed systems. We discuss each of these below.

5.2.1 Specification-Based Testing

The work of Richardson, O'Malley, and Tittle [ROT89] is generally accepted as the beginning of research into formal specification-based testing techniques. Earlier, interface-based techniques, such as random testing and the category-partition method [OB88], are also based on specifications, though not necessarily formal ones. In general, the appeal of specification-based testing is that tests can be constructed to check that an implementation does what it is required to do, rather than what programmers want it to do. However, these techniques are viewed as complementing implementation-based techniques, not replacing them.

There have been a number of studies of general-purpose specification-based testing techniques, including the work of Chang and Richardson [CR99, CRS96] on using a function-level assertion-based language to guide testing. Offutt and Liu [OL99] describe the generation of test data from a higher-level, object-oriented specification notation. Offutt et al. [OLAA03] describe the use of generic state-based specifications (e.g., UML and statecharts) for system-level testing. Harder et al. [HME03] describe the operational difference technique, which uses dynamically generated abstractions of program properties to aid in test selection. While these general-purpose techniques certainly can be applied to low-level testing of distributed systems, our focus is on system-level testing. Thus, we concentrate on higher-level specifications used in the areas of communication protocols and software architecture.

In protocol testing, each side of a two-party interaction is represented by a finite state machine (FSM) specification. Surveys by Bochmann and Petrenko [BP94] and Lai [Lai02] describe many of the algorithms that have been developed to generate test sequences for FSM specifications. These algorithms can be classified by the guarantees they provide with respect to different fault models (effectiveness), and by the length of sequences they create (cost). Fault models differ in the set of mutation operators they allow (e.g., output faults only) and in assumptions they make about implementation errors (e.g., by bounding the number of states that are possible in an implementation). Once abstract test sequences have been chosen using these algorithms, the test suite is adapted for a particular implementation and executed to demonstrate conformance.

The chief problem with these techniques is the limited expressivity of the FSM formalism. Extended FSMs, which are FSMs with minor state variables used in guard conditions and updated during state transitions, are used to represent protocol behavior more accurately, but as pointed out by Bochmann and Petrenko, these extensions are not handled by basic FSM techniques. The greater expressiveness of discrete-event simulations compared to FSM models could be what attracts practitioners to simulations.

More recently, software architectures have been studied as a means to describe and understand large, complex systems [PW92]. A number of researchers have studied the use of software architectures for testing. Richardson and Wolf [RW96] propose several architecture-based adequacy criteria based on the Chemical Abstract Machine model. Rice and Seidman [RS98] describe the ASDL architecture language and its toolset, and discuss its use in guiding integration testing. Jin and Offutt [JO01] define five general architecture-based testing criteria and applied them to the Wright ADL. Muccini et al. [MBI04] describe a comprehensive study of software architectures for implementation testing. Their technique relies on Labeled Transition System (LTS) specifications of dynamic behavior. They propose a method of abstracting simpler, abstract LTS (ALTS) from the monolithic global LTS in order to focus attention on interactions that are particularly relevant to testing. Coverage criteria are then defined with respect to these ALTS, and architectural tests are created to satisfy them. Finally, architectural tests are refined into implementation tests and executed against the implementation.

The main difference between our work and the approaches above is the nature of the specifications being used. Simulations are encoded in languages that are more expressive than FSMs, allowing more details of the system to be included in the analysis. Conversely, simulations operate at a lower level of abstraction than software architecture descriptions and use an imperative style to express functional behavior. Finally, and most importantly for distributed systems, simulations deal with such things as time and network behavior explicitly.

5.2.2 Fault-based Testing

Mutation testing is usually described in the context of implementation testing, but more recently researchers have proposed the application of mutation testing to specifications by defining mutation operators for statecharts [FMSM99] and Estelle specifications [SMFS99]. This work differs from ours in that their goal is specification testing, while ours is specification-based implementation testing. The mutation operators proposed in those papers could certainly be used to measure the effectiveness of test suites or testing techniques, but we know of no results in this area.

In closely related work, Ammann and Black [AB01] use a specification-based mutant score to measure the effectiveness of test suites. Their method employs a model checker and mutations of temporal logic specifications to generate test cases. They use this metric to compare the effectiveness of test suites developed using different techniques. This work differs from ours in two important ways: (1) their specification must be appropriate for model checking, namely it must be a finite-state description and the properties to check must be expressed in temporal logic, while our specification is a discrete-event simulation, and (2) their focus is solely on the comparison of candidate test suites, while ours also includes the comparison of testing criteria.

Finally, fault-based testing has been studied extensively with respect to specifications in the form of boolean expressions. In this context, a number of specification-based testing techniques have been experimentally evaluated [RT93, WGS94]. Recently, a fault-class hierarchy has been determined analytically and used to explain some of the earlier experimental results [Kuh99, TK02]. We are targeting a more expressive specification method whose fault classes (mutation operators) are not amenable to a general analytical comparison.

5.2.3 Testing Distributed Systems

A major motivating factor of our work is the lack of any general-purpose, disciplined, and effective testing method for distributed systems. This being said, a number of studies and research efforts are aimed at understanding the issues surrounding testing distributed systems and at developing methods and tools to improve the state of the art.

Several authors describe their experience and highlight the issues related to testing distributed systems [LS01, Dow89, GM99]. In particular, Ghosh and Mathur [GM99] list a number of differentiating factors between distributed and non-distributed software which are representative of the those mentioned by other authors: distributed systems have a larger scale, heterogeneity, monitoring and control is more difficult, nondeterminism, concurrency, scalability and performance, and partial failure (fault tolerance).

Of these, tool support for monitoring and controlling distributed tests has received considerable attention. Some tools support primarily functional testing [HGC04, GBCK01, GBGR02, HWS95]; others are specifically target at performance testing [DPE04, GCL05]. While this is important work, it addresses only the accidental issues associated with distributed system testing, and does not address the more fundamental questions having to do with what and how best to test a system.

Several studies concentrate on distributed component based systems. Gosh and Mathur present an adequacy criterion targeted at covering CORBA component interfaces. Krishnamurthy and Sivilotti also specifically target CORBA components and present a method for specifying and testing progress properties of components [KS01]. Williams and Probert apply techniques of pairwise interaction testing to component based systems [WP01, WP96]. By its nature, work targeting distributed component systems is restricted to a limited level of distribution since much of the complexity of developing these systems is handled by the container infrastructure.

We now discuss some of these areas in more detail.

5.2.4 Distributed Testing Frameworks and Testbeds

One focus with testing distributed applications has been the development of testbeds that are appropriate for evaluating large scale distributed applications.

PlanetLab [PACR03], is a popular global-scale testbed for distributed systems. To use PlanetLab, an engineer interacts with each of the hosts associated with their “slice”, using SSH to perform the necessary configuration and execution. There are hundreds of PlanetLab nodes spanning the globe. PlanetLab is relies on the actual conditions of the global Internet to provide variability and realistic (by definition) conditions.

Another fully featured testbed for distributed systems, Netbed/Emulab [WLS$^+$02], is designed to operate on a cluster of computer all occupying the same local network. In order to achieve “realistic” conditions within the cluster, Netbed provides an integrated emulation and simulation environment that support the use of traffic-shaping techniques. The Netbed management system provides ways for the engineer to distribute software onto the hosts used in their experiment.

Neither of these systems provides explicit support for testing, however, with the integration of some additional management and reporting software, they would provide test engineers with many of the tools they needed to exercise a distributed application under varying network conditions.

5.2.5 Early-phase Performance Evaluation

Another aspect of distributed application testing that has received significant attention from the research community lately is performance evaluation of systems based on architectural and design specifications. For example Grundy, Cai, and Liu [GCL01] discuss the SoftArch/MTE system. This system enables the automatic generation of prototype distributed systems based on high-level architectural objectives. These generated systems are deployed on a dedicated testbed and performance metrics are gathered while the system is exercised. The SoftArch/MTE system is targeted at prototype systems created early in the design process.

More recently, Denaro, Polini and Emmerich have presented similar ideas that are targeted at generating test cases for exercising middleware configurations as a surrogate for the functioning system during early phases of development [DPE04].

5.2.6 Distributed Component Unit Testing

As mentioned earlier, unit testing is intended to verify the most basic program units in isolation before more complicated and comprehensive tests at the integration- and system-level are conducted. Unfortunately, some modules simply cannot be tested in complete isolation since they depend heavily on other modules for a portion of their functionality. For example, in an object-oriented paradigm, if one class delegates part of its core functionality to another class, it is difficult to test this top-level object in isolation. A technique for dealing with this, called "mock object testing," has grown out of the extreme programming movement [MFC00]. In mock object testing, stub classes are created that allow the tester to control the behavior of a dependency so that all expected behavior of the low-level module can be mimicked.

While mock object testing was not specifically developed for distributed applications, it has proved quite useful for testing distributed component software where the modules are intended to run inside a container like a servlet engine or CORBA object request broker (ORB). In these types of systems, hooks into the container are usually provided through an object that is passed in at runtime. Testing inside a running container can be time consuming and complicated, so mock versions of the container hooks are created to allow unit testing of these classes without running any container software.

Chapter 6

Conclusion

Testing is a discipline of software engineering that, like all engineering activities, is governed by the tradeoff between time and resources on the one hand, and completeness and quality on the other. Given this, it is vital that methods be developed with an eye toward helping testers manage the risks associated with the testing activity. Through this dissertation we have attempted to develop a testing technique that is: (1) aimed at tackling a problem that exists in real world systems; and (2) that works in conjunction with tools and techniques favored by practitioners.

The failure study we undertook surveyed the failures reported by users of seven sophisticated, widely used, distributed systems. Through carefully examination of the failure reports we were able to categorize the architecture of the failure-producing scenarios and the lower level inputs and observations needed to replicate them. The results of this study convinced us that techniques were needed to rationally select distributed inputs for this class of systems.

Next, based on our experience designing, developing and researching distributed systems we were aware of the broad acceptance of discrete event simulations in this area. We observe that discrete event simulations of distributed systems can be viewed as specifications of the expected behavior of a system, and conjecture that they can be possibly be used within a specification based testing regime as the basis for definition and application of test adequacy criteria. We validate this claim by verifying that even the simplest white-box adequacy criteria when applied to simulation code are both **effective** and **cost-effective** at testing implementations.

However, although adequacy criteria are quite prevalent in the research literature they are much less widely used in practice. We believe one of the reasons for this is that there is little or no experimental data that can help guide testers to use adequacy criteria that are likely to be well suited to their system. There is also a disconnect between general-purpose technique like white-box adequacy criteria, and more system specific techniques that testers are tempted to use on their systems. This observation, in conjunction with much of the literature on analytical and experimental evaluation of testing technique effectiveness, led us to identify two risks that practitioners face when using adequacy criteria.

The first risk is that of selecting an ineffective, adequate test suite, purely by bad luck. Using fault-based analysis of simulation code and candidate test suites, a tester can expend the effort necessary to mitigate this risk.

The second risk is that the tester will choose, through no fault of their own, an adequacy criterion that happens to be ineffective for their system, and thereby limit the quality of their testing efforts. This risk can be addressed by applying still more effort to perform the same fault-based analysis on several candidate adequacy criteria to first determine their relative effectiveness.

The experiments we conducted, though necessarily limited in scope, are encouraging and show that techniques we developed can be used to produce their intended result. Although we believe this work can stand on its own, we are motivated to continue to refine and develop the ideas we begin here. In the following section, we describe some of the directions we hope to explore in the future.

6.1 Future Work

The work presented in this dissertation can be extended along several different avenues. Within the simulation-based paradigm, there are several direct extensions to the work presented here. At a higher-level, we propose several other research directions around the idea of providing techniques and rules for software producers to use when selecting which testing techniques to use.

6.1.0.1 Extensions to Simulation-Based Testing

The simulation-based testing technique presented in this dissertation is really a platform that can be used to advance several different areas of testing for distributed systems.

Advanced Adequacy Criteria

Perhaps the most direct extension to this work is in the definition of new criteria that utilize the other aspects of the simulation structure and dynamic nature. For example, as the simulation executes messages are exchanged between processes. Patterns in the interleaving of these messages could be used to define criteria targeting issues of concurrency. Another example would be to derive inter-process control- and data-flow graphs using message exchange as the procedure-call mechanism.

Probabilistic Oracles

The oracle problem is challenging, even for non-distributed systems; for distributed systems the influence of environmental inputs makes this even more difficult. For the experiments described in this dissertation, we used "high-level" oracles in which there is enough state at a particular component to determine if the system performed correctly. For instance, after a GBN data exchange we ensure that the data file written by the receiving process matches the original file exactly. This type of oracle works for some failures, but not others. If there was a fault in the implementation of the sizing of the sliding window that did not affect the overall data transfer, the oracle would not detect the failure.

For distributed systems, message exchange is appealing as a basis for defining oracles since it is straightforward to monitor and record network traffic during a test execution. A major complication, however, is the inherent variability in the message ordering that arises from latency and non-determinism in the network stack. Process-based discrete-event simulations (as opposed to event-based) exhibit non-determinism due to the thread scheduling algorithms typically used in their implementation. Therefore, it would be interesting to investigate the training of a hidden Markov model with the message streams obtained from simulation. If there was a direct mapping between simulated messages and implementation messages, these Markov models could then be used to determine the probability that a given implementation message exchange corresponds to a valid message exchange as determined by the simulation. We believe this idea holds a lot of promise and plan on pursuing it vigorously.

6.1.1 Test Adequacy Criteria Selection Methods

While the subject of test adequacy criteria is widely addressed by the research community, the issues associated with actually using adequacy criteria in practice are not addressed at all. There is concern in the research community that in the 20+ years since the most basic white-box test adequacy criteria were defined, there has been little adoption of advanced adequacy criteria by practitioners [LO96]. We believe that one reason for this is that in the absence of clear results or processes for selecting between adequacy criteria, practitioners prefer to use something that is clearly understood and widely discussed.

If there were some clear rules and/or experimental results that practitioners could use to make rational decisions about which advanced testing techniques to use, we believe that the impact of testing on practice could be raised significantly.

Application of Fault-Based Techniques

The fault-based analysis technique we present here is not limited in applicability to simulation-based techniques. It may be applied to any specification-based technique in which adequate test suites can be selected without executing the implementation. This restriction rules out techniques such as structural specification-based testing [CRS96] in which **test requirements** are derived from a specification and mapped to the implementation domain in the form of instrumentation since the implementation must be executed on each candidate test case to determine adequacy.

Another prerequisite of using our fault-based technique is the existence of a family of mutation operators for the specification language being used. Due to the popularity of mutation testing (among researchers, at least), there are a number of specification languages that have a defined set of mutation operators. Fabbri and colleagues alone have defined operators for Estelle [SMFS99], Statecharts [FMSM99], Petri nets [FMM+96], and finite-state machines [FDMM94]. Since these specification techniques are all executable, this would be an interesting place to start.

Broad Empirical Studies

While fault-based predictive approaches might work for some specification-based techniques, there is no substitution for extensive and broad empirical studies to establish some general results for the testing community.

In existing empirical studies, comparisons are made between testing techniques that operate at the same level (i.e., unit, cluster, integration or system). This experimental framework excludes comparisons of the type that are perhaps most useful to practitioners: what combinations of high-level (system, integration) and low-level (cluster, unit) techniques are most effective. To do this type of experimentation, we advocate an approach whereby entire **systems** are tested at each level by multiple different techniques. With this data at hand, comparison could be made within and between levels. Open-source projects would be a valuable source of systems to experiment with; indeed, savvy project leaders would recognize this as an opportunity to gain extensive testing of their software.

Once enough data had been collected, there would be an opportunity to look for models based on different aspects of implementations and specifications that could reliably predict which techniques work well for different systems.

Bibliography

[AB01] Paul E. Ammann and Paul E. Black, A specification-based coverage metric to evaluate test sets, International Journal of Reliability, Quality and Safety Engineering **8** (2001), no. 4, 275–299.

[ABL05] J.H. Andrews, L.C. Briand, and Y. Labiche, Is mutation an appropriate tool for testing experiments?, Proceedings of the 27th International Conference on Software Engineering (ICSE) 2005 (St. Louis, MO, USA), May 2005, pp. 402–411.

[AF99] Mark Allman and Aaron Falk, On the effective evaluation of tcp, SIGCOMM Comput. Commun. Rev. **29** (1999), no. 5, 59–70.

[BD04] Pierre Bourque and Robert Dupuis (eds.), Guide to the software engineering body of knowledge (swebok), IEEE Computer Society, 2004, available at http://www.swebok.org/.

[BLW04] L. C. Briand, Y. Labiche, and Y. Wang, Using simulation to empirically investigate test coverage criteria based on statechart, ICSE '04: Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 86–95.

[BP94] Gregor Bochmann and Alexandre Petrenko, Protocol testing: review of methods and relevance for software testing, Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, 1994, pp. 109–124.

[BPL04] Lionel C. Briand, Massimiliano Di Penta, and Yvan Labiche, Assessing and improving state-based class testing: A series of experiments, IEEE Transactions on Software Engineering **30** (2004), no. 11, 770–793.

[CBC^{+}92] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong, Orthogonal defect classification-a concept for in-process measurements, IEEE Transactions on Software Engineering **18** (1992), no. 11, 943–956.

[CHBC93] Jarir K. Chaar, Michael J. Halliday, Inderpal S. Bhandari, and Ram Chillarege, In-process evaluation for software inspection and test, IEEE Transactions on Software Engineering **19** (1993), no. 11, 1055–1070.

[CMH$^+$02] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley, Protecting free expression online with freenet, IEEE Internet Computing **6** (2002), no. 1, 40–49.

[CR99] Juei Chang and Debra J. Richardson, Structural specification-based testing: automated support and experimental evaluation, Proceedings of the 7th European Software Engineering Conference/7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 1999, pp. 285–302.

[CRS96] Juei Chang, Debra J. Richardson, and Sriram Sankar, Structural specification-based testing with ADL, Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis, 1996, pp. 62–70.

[Dev95] Jay L. Devore, Probability and statistics for engineering and the sciences, 4th ed., Brooks/Cole, 1995.

[Dij72] E.W. Dijkstra, Structured programming, ch. Notes on structured programming, Academic Press, 1972.

[DLS78] R. A. DeMillo, R.J. Lipton, and F.G. Sayward, Hints on test data selection: Help for the practicing programmer, IEEE Computer **11** (1978), no. 4, 34–41.

[DN81] Joe W. Duran and Simeon Ntafos, A report on random testing, Proceedings of the 5th International Conference on Software Engineering, 1981, pp. 179–183.

[DN84] J.W. Duran and S. Ntafos, An evaluation of random testing, IEEE Transactions on Software Engineering **SE-10** (1984), 438–444.

[Dow89] E. J. Dowling, Testing distributed ada programs, Proceedings of the conference on Tri-Ada '89, 1989, pp. 517–527.

[DPE04] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich, Early performance testing of distributed software applications, Proceedings of the fourth international workshop on Software and performance, 2004, pp. 94–103.

[EW98] K. El Emam and I. Wieczorek, The repeatability of code defect classifications, Proceedings of the The Ninth International Symposium on Software Reliability Engineering, 1998, pp. 322–333.

[FDMM94] S.C. Pinto Ferraz Fabbri, M.E. Delamaro, J.C. Maldonado, and P.C. Masiero, Mutation analysis testing for finite state machines, Proceedings of 5th International Symposium on Software Reliability Engineering, November 1994, pp. 220–229.

[FI98] Phyllis Frankl and Oleg Iakounenko, Further empirical studies of test effectiveness, SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering (New York, NY, USA), 1998, pp. 153–162.

[FMM$^+$96] Sandra Camargo Pinto Ferraz Fabbri, Jose C. Maldonado, Paulo Cesar Masiero, Marcio E. Delamaro, and E. Wong, Mutation testing applied to validate specifications based on petri nets, Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques VIII (London, UK, UK), 1996, pp. 329–337.

[FMSM99] Sandra Camargo Pinto Ferraz Fabbri, Jose Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero, Mutation testing applied to validate specifications based on statecharts, ISSRE '99: Proceedings of the 10th International Symposium on Software Reliability Engineering (Washington, DC, USA), 1999, p. 210.

[FO00] Norman E. Fenton and Niclas Ohlsson, Quantitative analysis of faults and failures in a complex software system, IEEE Transactions on Software Engineering **26** (2000), no. 8, 797–814.

[FW88] Phyllis Frankl and Elaine Weyuker, An applicable family of data flow testing criteria, IEEE Transactions on Software Engineering **14** (1988), no. 10, 1483–1498.

[FW91] Phyllis Frankl and Stewart Weiss, An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria, Proceedings of the Symposium on Testing, Analysis, and Verification, 1991, pp. 154–164.

[FW93a] ______, An experimental comparison of the effectiveness of branch testing and data flow testing, IEEE Transactions on Software Engineering **19** (1993), no. 8, 774–787.

[FW93b] Phyllis Frankl and Elaine Weyuker, An analytical comparison of the fault-detecting ability of data flow testing techniques, Proceedings of the 15th International Conference on Software Engineering, 1993, pp. 415–424.

[FW93c] ______, A formal analysis of the fault-detecting ability of testing methods, IEEE Transactions on Software Engineering **19** (1993), no. 3, 202–213.

[FWH97] Phyllis Frankl, Stewart Weiss, and Cang Hu, All-uses versus mutation testing: An experimental comparison of effectiveness, Journal of Systems and Software **38** (1997), no. 3, 235–253.

[GBCK01] S. Ghosh, N. Bawa, G. Craig, and K. Kalgaonkar, A test management and software visualization framework for heterogeneous distributed applications, HASE '01: The 6th IEEE International Symposium on High-Assurance Systems Engineering (Washington, DC, USA), 2001, p. 0106.

[GBGR02] Sudipto Ghosh, Nishant Bawa, Sameer Goel, and Y. Raghu Reddy, Validating run-time interactions in distributed java applications, ICECCS

'02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems (Washington, DC, USA), 2002, p. 7.

[GCL01] J. Grundy, Y. Cai, and A. Liu, Generation of distributed system test-beds from high-level software architecture descriptions, Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), 2001.

[GCL05] John Grundy, Yuhong Cai, and Anna Liu, Softarch/mte: Generating distributed system test-beds from high-level software architecture descriptions, Automated Software Engg. **12** (2005), no. 1, 5–39.

[GG75] J. B. Goodenough and S. L. Gerhart, Towards a theory of test data selection, IEEE Transactions on Software Engineering **SE-3** (1975).

[GG77] ______, Towards a theory of testing: Data selection criteria, vol. 2, Prentice-Hall, 1977.

[GM99] S. Ghosh and A. Mathur, Issues in testing distributed component-based systems, Proceedings of the First International ICSE Workshop Testing Distributed Component Based Systems, May 1999.

[Gou83] J. Gourlay, A mathematical framework for the investigation of testing, IEEE Transactions on Software Engineering **SE-9** (1983), no. 6, 686–709.

[Ham94] Dick Hamlet, Foundations of software testing: dependability theory, Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, 1994, pp. 128–139.

[Het84] W. Hetzel, The complete guide to software testing, Collins, 1984.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand, Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria, Proceedings of the 16th International Conference on Software Engineering, 1994, pp. 191–200.

[HGC04] Daniel Hughes, Phil Greenwood, and Geoff Coulson, A framework for testing distributed systems, P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing (P2P'04) (Washington, DC, USA), 2004, pp. 262–263.

[Hie02] R. M. Hierons, Comparing test sets and criteria in the presence of test hypotheses and fault domains, ACM Transactions on Software Engineering Methodology **11** (2002), no. 4, 427–448.

[HME03] Michael Harder, Jeff Mellen, and Michael D. Ernst, Improving test suites via operational abstraction, Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 60–71.

[HS89] M. Harrold and M. Soffa, Interprocedual data flow testing, Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification, 1989, pp. 158–167.

[HS91] M. J. Harrold and M. L. Soffa, Selecting and using data for integration testing, IEEE Software **8** (1991), no. 2, 58–65.

[HT90] Dick Hamlet and Ross Taylor, Partition testing does not inspire confidence (program testing), IEEE Transactions on Software Engineering **16** (1990), no. 12, 1402–1411.

[HWS95] Alex Hubbard, C. Murray Woodside, and Cheryl Schramm, DECALS: distributed experiment control and logging system, Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research, 1995, p. 32.

[JO98] Zhenyi Jin and Jeff Offutt, Coupling-based criteria for integration testing, Journal of Software Testing, Verification, and Reliability **8** (1998), no. 3, 133–154.

[JO01] ______, Deriving tests from software architectures, The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01), November 2001, pp. 308–313.

[KR04] James F. Kurose and Keith W. Ross, Computer networking: A top-down approach featuring the internet, Pearson Benjamin Cummings, 2004.

[KS01] Prakash Krishnamurthy and Paolo A. G. Sivilotti, The specification and testing of quantified progress properties in distributed systems, Proceedings of the 23rd international conference on Software engineering, 2001, pp. 201–210.

[KS03] G. Kapfhammer and Mary Lou Soffa, A family of test adequacy criteria for database-driven applications, Proceedings of ACM SIGSOFT Foundations of Software Engineering Conference, September 2003.

[Kuh99] D. Richard Kuhn, Fault classes and error detection capability of specification-based testing, ACM Transactions on Software Engineering Methodology **8** (1999), no. 4, 411–424.

[Lai02] R. Lai, A survey of communication protocol testing, Journal of Systems and Software **62** (2002), no. 1, 21–46.

[LC97] C. Liu and P. Cao, Maintaining strong cache consistency in the world-wide web, ICDCS '97: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97) (Washington, DC, USA), 1997, p. 12.

[LK83] J. Laski and B. Korel, A data flow oriented program testing stragegy, IEEE Transactions on Software Engineering **SE-9** (1983).

[LO96] et al Leon Osterweil, Strategic directions in software quality, ACM Comput. Surv. **28** (1996), no. 4, 738–750.

[LPS00] Marek Leszak, Dewayne E. Perry, and Dieter Stoll, A case study in root cause defect analysis, Proceedings of the 22nd International Conference on Software Engineering, 2000, pp. 428–437.

[LPS02] ———, Classification and evaluation of defects in a project retrospective, Journal of Systems and Software **61** (2002), no. 3, 173–187.

[LS01] Brad Long and Paul Strooper, A case study in testing distributed systems, DOA '01: Proceedings of the Third International Symposium on Distributed Objects and Applications (Washington, DC, USA), 2001, p. 20.

[LSH$^+$04] Paul Luo Li, Mary Shaw, Jim Herbsleb, Bonnie Ray, and P. Santhanam, Empirical evaluation of defect projection models for widely-deployed production software systems, Proceedings of the 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (New York, NY, USA), 2004, pp. 263–272.

[LY99] Tim Lindholm and Frank Yellin, The java virtual machine specification, second ed., Addison-Wesley Professional, April 1999.

[MBI04] H. Muccini, A. Bertolino, and P. Inverardi, Using software architecture for code testing, IEEE Transactions on Software Engineering **30** (2004), no. 3, 160–171.

[McC76] T. J. McCabe, A complexity measure, IEEE Transactions on Software Engineering **SE-2** (1976), no. 4, 308–320.

[McC83] T. J. McCabe (ed.), Structured testing, IEEE, 1983.

[MFC00] T. Mackinnon, S. Freeman, and P. Craig, Endo-testing: Unit testing with mock objects, Proceedings of eXtreme Programming Conference 2000 (XP2000), May 2000.

[MKO02] Yu-Seung Ma, Yong-Rae Kwon, and J. Offutt, Inter-class mutation operators for java, 13th International Symposium on Software Reliability Engineering, November 2002, pp. 352–363.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon, Mujava : An automated class mutation system, Journal of Software Testing, Verification and Reliability **15** (2005), no. 2, 97–133.

[MS83] T. J. McCabe and G. G. Schulmeyer, Combining testing with specifications: A case study, IEEE Transactions on Software Engineering **SE-9** (1983), no. 3, 328–334.

[MSP01] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack, Coverage criteria for GUI testing, Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, 2001, pp. 256–267.

[Nta84] S. C. Ntafos, On required element testing, IEEE Transactions on Software Engineering **SE-10** (1984), no. 6.

[Nta98] Simeon Ntafos, On random and partition testing, Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, 1998, pp. 42–48.

[OB88] T. J. Ostrand and M. J. Balcer, The category-partition method for specifying and generating fuctional tests, Communications of the ACM **31** (1988), no. 6, 676–686.

[OL99] A. Jefferson Offutt and Shaoying Liu, Generating test data from SOFL specifications, Journal of Systems and Software **49** (1999), no. 1, 49–62.

[OLAA03] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann, Generating test data from state-based specifications, Journal of Software Testing, Verification and Reliability **13** (2003), no. 1, 25–53.

[OLR$^+$96] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf, An experimental determination of sufficient mutant operators, ACM Transactions on Software Engineering Methodology **5** (1996), no. 2, 99–118.

[OW02] Thomas J. Ostrand and Elaine J. Weyuker, The distribution of faults in a large industrial software system, Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, 2002, pp. 55–64.

[OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell, Where the bugs are, ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (New York, NY, USA), 2004, pp. 86–96.

[PACR03] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe, A blueprint for introducing disruptive technology into the internet, ACM SIGCOMM Computer Communication Review **33** (2003), no. 1, 59–64.

[Pai75] M. R. Paige, Program graphs, an algebra, and their implications for programming, IEEE Transactions on Software Engineering **SE-1** (1975), no. 3, 286–291.

[Pai78] ______, An analytical approach to software testing, Proceedings of COMPSAC'78, 1978, pp. 527–532.

[PC89] A. Podgurski and L. A. Clarke, The implications of program dependences for software testing, debugging and maintenance, Proceedings of SIGSOFT Symposium on Software Testing, Analysis, and Verification 3, December 1989.

[PC90] ______, A formal model of program dependences and its implications for software testing, debugging and maintenance, IEEE Transactions on Software Engineering **16** (1990), no. 9, 965–979.

[PW92] Dewayne E. Perry and Alexander L. Wolf, Foundations for the study of software architecture, SIGSOFT Software Engineering Notes **17** (1992), no. 4, 40–52.

[ROT89] D. Richardson, O. O'Malley, and C. Tittle, Approaches to specification-based testing, Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification, 1989, pp. 86–96.

[RS98] Michael D. Rice and Stephen B. Seidman, An approach to architectural analysis and testing, Proceedings of the 3rd International Workshop on Software Architecture, 1998, pp. 121–123.

[RT93] Debra Richardson and Margaret Thompson, An analysis of test data selection criteria using the relay model of fault detection, IEEE Transactions on Software Engineering **19** (1993), no. 6, 533–553.

[RW85] Sandra Rapps and Elaine J. Weyuker, Selecting software test data using data flow information, IEEE Transactions on Software Engineering **11** (1985), no. 4, 367–375.

[RW96] Debra J. Richardson and Alexander L. Wolf, Software testing at the architectural level, Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops, 1996, pp. 68–71.

[SA97] B. Segall and D. Arnold, Elvin has left the building: A publish/subscribe notification service with quenching, Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG'97), 1997.

[SAB$^+$00] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps, Content based routing with elvin4, Proceedings of Australian UNIX and Open Systems User Group Annual Conference, 2000 (AUUG2k) (Canberra, Australia), June 2000.

[Sch93] Michael D. Schroeder, A state-of-the-art distributed system: Computing with BOB, Distributed Systems (Sape Mullender, ed.), Addison-Wesley, 2nd ed., 1993.

[SMFS99] Simone Do Rocio Senger De Souza, Jose Carlos Maldonado, Sandra Camargo Pinto Ferraz Fabbri, and Wanderley Lopes De Souza, Mutation testing applied to estelle specifications, Software Quality Control **8** (1999), no. 4, 285–301.

[SMLN$^+$03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, Chord: a scalable peer-to-peer lookup protocol for internet applications, IEEE/ACM Trans. Netw. **11** (2003), no. 1, 17–32.

[TK02] Tatsuhiro Tsuchiya and Tohru Kikuno, On fault classes and error detection capability of specification-based testing, ACM Transactions on Software Engineering Methodology **11** (2002), no. 1, 58–62.

[WGS94] E. Weyuker, T. Goradia, and A. Singh, Automatically generating test data from a boolean specification, IEEE Transactions on Software Engineering **20** (1994), no. 5, 353–363.

[WJ91] Elaine J. Weyuker and Bingchiang Jeng, Analyzing partition testing strategies, IEEE Transactions on Software Engineering **17** (1991), no. 7, 703–711.

[WLS$^+$02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar, An integrated experimental environment for distributed systems and networks, Proceedings of the Symposium on Operating Systems Design and Implementation (Boston, MA), December 2002, pp. 255–270.

[WP96] A. W. Williams and R. L. Probert, A practical strategy for testing pair-wise coverage of network interfaces, Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96), 1996, p. 246.

[WP01] Alan W. Williams and Robert L. Probert, A measure for component interaction test coverage, Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications, 2001, p. 304.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, A note on distributed computing, Tech. report, Sun Microsystems Laboratories, Inc., 1994.

[ZH93] Hong Zhu and P.A.V Hall, Test data adequacy measurement, Software Engineering Journal **8** (1993), no. 1, 21–29.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May, Software unit test coverage and adequacy, ACM Comput. Surv. **29** (1997), no. 4, 366–427.

Appendix A

Elvin Bug Report 1008

```
Location: http://www.mantara.com/support/bugzilla/show_bug.cgi?id=1008
Mantara

Bugzilla Bug 1008   slave dying causes master to die
     Query page      Enter new bug
---------------------------------------------------------------------------------------------
                                                                                             croy(at)
       Bug#: 1008                     Platform: [All      ]                       Reporter: dstc-dot-edu-dot-au-nospam
                                                                                             (Clinton Roy)
    Product: [Elvin Router    ]             OS: [All                         ]     Add CC: [                               ]
  Component: [Failover       ]          Version: [CVS     ]                             CC:
     Status: RESOLVED                   Priority: [P1]
 Resolution: FIXED                      Severity: [major      ]
             ilister(at)                   Target
Assigned To: mantara-dot-com-nospam   Milestone: [4.1b4]
             (Ian Lister)
        URL: [                                                            ]
    Summary: [slave dying causes master to die                            ]

+---------------------------------------------------------------------+
| Attachment        | Type    | Modified       | Status       | Actions  |
|----------------------------------------------------------+----------|
| Create a New Attachment (proposed patch, testcase, etc.)   | View All |
+---------------------------------------------------------------------+

Bug 1008 depends on:  [                    ] Show dependency tree
    Bug 1008 blocks:  [                    ] Show dependency graph

Additional Comments:

[                                                                                ]

(*) Leave as RESOLVED FIXED
( ) Reopen bug
( ) Mark bug as VERIFIED
( ) Mark bug as CLOSED
 Commit
```

```
View Bug Activity   |   Format For Printing

Description:                                                  Opened: 2002-11-07 15:24

----------------------------------------------------------------------------------------------
Set up a failover pair like thus:

[Master] <-------------- [Slave]

Kill the slave, the Master dies.

backtrace:

#0  strncat (s1=0xbfffdc08 "", s2=0x0, n=80) at ../sysdeps/generic/strncat.c:52
#1  0x080a6cc6 in elvin_i18n_message_set_string (msg=0xbfffdb30, argnum=0,
argval=0x0) at i18n.c:699
#2  0x0807bec6 in connection_failed (failover=0x820e8f8, error=0x81e68c0) at
failover.c:1026
#3  0x0807a94d in handle_Disconn (failover=0x820e8f8, packet=0xbfffeaa0,
error=0x81e68c0) at failover.c:555
#4  0x0807b743 in failover_recv_cb (endpoint=0x82115f0, recv_rock=0x820e8f8,
error=0x81e68c0) at failover.c:921
#5  0x0807bad3 in offer_recv_cb (endpoint=0x82115f0, recv_rock=0x820e8f8,
error=0x81e68c0) at failover.c:961
#6  0x080a28a6 in process_pending (endpoint=0x82115f0, error=0x81e68c0) at
endpoint.c:167
#7  0x080a298d in ep_recv_cb (rock=0x82115f0, reentry_is_safe=1,
buffer=0x820ce08 "\2202@\2202@\020", length=12, error=0x81e68c0) at endpoint.c:206
#8  0x080db70b in elvin_tcp_recv_cb (handler=0x8211cb8, fd=8, active_mask=8,
would_block_out=0xbffff5f8, rock=0x820f3a8, error=0x81e68c0) at tcp_transport.c:1971
#9  0x080a7699 in elvin_io_handler_dispatch (io_handler=0x8211cb8,
active_mask=8, would_block_out=0xbffff5f8, error=0x81e68c0) at io_handler.c:292
#10 0x080a78ff in io_handler_set_dispatch (set=0x81f5af0, error=0x81e68c0) at
io_handler.c:594
#11 0x080a7879 in elvin_io_handler_set_dispatch (set=0x81f5af0, error=0x81e68c0)
at io_handler.c:537
#12 0x08086576 in server_loop_once (error=0x81e68c0) at main.c:1268
#13 0x0808766b in server_main_loop (ee=0x81e68c0) at main.c:1847
#14 0x08087a83 in main (argc=3, argv=0xbffff6f4, env=0xbffff704) at main.c:1998
#15 0x4021c336 in __libc_start_main (main=0x8087994 <main>, argc=3,
ubp_av=0xbffff6f4, init=0x804a474 <_init>, fini=0x80ddf70 <_fini>,
rtld_fini=0x4000d2fc <_dl_fini>, stack_end=0xbffff6ec)
    at ../sysdeps/generic/libc-start.c:129

------- Additional Comment #1 From Ian Lister 2002-12-05 16:19 -------
I cannot reproduce this crash. Can you please provide more details on:

 * the failover.* lines in your elvind.conf files
 * the order in which you started the servers
 * how you killed the slave
```

```
Ian

------- Additional Comment #2 From Clinton Roy 2002-12-05 16:34 -------
For the Master:
failover yes
failover.protocol efo:/tcp,none,xdr/0.0.0.0:2915
failover.submissive no

For the Slave:
failover yes
failover.url efo:/tcp,none,xdr/localhost:2915
failover.submissive yes

I can crash the Master regardless of which server starts first.

I killed the Slave by ^C.

------- Additional Comment #3 From Ian Lister 2002-12-06 11:11 -------
Fixed in failover.c revision 1.31.

=====8<-----
--- failover.c  2002/11/14 01:33:35     1.30
+++ failover.c  2002/12/06 01:06:28
@@ -1026,7 +1026,7 @@
         ELVIND_LOG_FAILOVER_CONN_LOST(loghandle,
                                       ELVIN_LOG_WARNING,
                                       (self->peerident == NULL) ?
-                                      "<null>" : self->remote_url_str);
+                                      "<null>" : self->peerident);
     }

     /* Clean up the endpoint, if we still have one, but don't purge
=====8<-----

Ian
----------------------------------------------------------------------------------------------
     Query page      Enter new bug

This is Bugzilla: the Mozilla bug system. For more information about what Bugzilla is and what
it can do, see bugzilla.org.

Actions:   New | Query |  Find  bug # [      ] | Reports                   New Account | Log In
```

Appendix B

Example: GCDService

This example distributed application, GCDService, is a service that computes the greatest common divisor of two integers upon request. `GCDclient` components construct a request message containing the integers of interest and send it over an unreliable network to a server. A `GCDserver` component waits for requests and provides responses either by finding a cached copy of the result from previous request of the same integers; or by computing it directly through the application of Euclid's algorithm.

B.1 Euclid's Algorithm

```
public class GCD {
  public static int gcd(int x, int y) {
    while(x > 0 && y > 0) {
      if(x > y){
        x = x - y;
      } else {
        y = y - x;
      }
      return x + y;
    }
  }
}
```

Figure B.1: Euclid's Algorithm

Figure B.1 contains the code for a simple iterative implementation of Euclid's algorithm for computing the greatest common divisor of two integers.

B.2 Messages

There are two message types, show in exchanged by client and server components: Request and Response.

```
public class Request extends Message
{
  public final int a;
  public final int b;

  public Request(int a, int b){
    this.a = a;
    this.b = b;
  }
}

public class Response extends Message
{
  public final int gcd;

  public Response(int gcd){
    this.gcd = gcd;
  }
}
```

Figure B.2: Message Implementations

Request data members are simply the two integers for which the GCD is required. The Response contains the computed GCD result value.

B.3 `GCDclient` Processing

The processing performed by the `GCDclient` component is simple. The component is provided with the network address of the server to use and the integer values for which the GCD is desired. The client sends the request to the configured server and waits 500 milliseconds for a response; if a response is not received within the timeout, the client repeats this sequence 2 more times. The client terminates by returning the response or signaling an error if no response is received.

```
public class GCDclient extends Component
{
  public int main(Address server, int a, int b)
    throws NoResponseError
  {
    DatagramChannel ch = newDatagramChannel();

    Request req = new Request(a, b);

    for(int i=0; i<3; ++i){
      ch.send(server, req);
      try {
        return ((Response)ch.recv(0.5)).gcd;
      }
      catch(TimeoutException e){}
    }
    throw new NoResponseError();
  }
}
```

Figure B.3: `GCDclient` Component

B.4 `GCDserver` Processing

The processing performed by a `GCDserver` component is more complicated due to its caching behavior.

Each `GCDserver` has a cache with fixed upper size limit, `cacheSize`. The cache is a map from the input values to the computed result value. Since the GCD algorithm is symmetric with respect to the ordering of the two input integers, the cache key is an encoding of the values that matches either ordering. Once the upper size limit is reached, the least recently used entry is removed before the new entry is added. This implies that the `GCDserver` must keep track of the use time of each cache entry.

B.5 CacheTest Configuration

```
public class GCDserver extends Component
{
  public void main(int port, int cacheSize)
  {
    DatagramChannel ch = newDatagramChannel(port);
    Map cache = new HashMap();

    while(running()){
      Request req = (Request)ch.recv();

      String key = req.a + "," + req.b;
      if(req.a > req.b){
        key = req.b + "," + req.a;
      }

      Integer gcd = null;
      if(cache.containsKey(key)){
        CacheValue cv = (CacheValue)cache.remove(key);
        gcd = cv.gcd;
      } else {
        // Begin Euclid's Algorithm
        int x = req.a;
        int y = req.b;
        while(x > 0 && y > 0){
          pause(0.1);
          if(x > y) {
            x = x - y;
          } else {
            y = y - x;
          }
        }
        gcd = new Integer(x + y);
        // End Euclid's Algorithm
      }
      cache.put(key, new CacheValue(now(), gcd));

      if(cache.size() > cacheSize){
        Iterator itr = cache.entrySet().iterator();
        Object minKey = null;
        double minValue = Double.MAX_VALUE;
        while(itr.hasNext()){
          Map.Entry entry = (Map.Entry)itr.next();
          CacheValue value = (CacheValue)entry.getValue();

          if(value.usedAt < minValue){
            minKey = entry.getKey();
          }
        }
        cache.remove(minKey);
      }
      assert cache.size() <= cacheSize;
      Response resp = new Response(gcd.intValue());
      ch.send(req.getSourceAddress(), resp);
    }
  }
}
```

Figure B.4: `GCDserver` Component

```
public class CacheTest extends Simulation {
  public CacheTest(double latency) {
    Address addr = new Address("server", 1234);

    add(new GCDserver(addr.name, addr.port, 1));

    add(new GCDclient("c1", addr, 3, 5),
        new Integer(1), 0.0);

    add(new GCDclient("c2", addr, 5, 3),
        new Integer(1), 5.0);

    add(new GCDclient("c3", addr, 2, 4),
        new Integer(2), 10.0);

    setNetworkBehavior(new NetworkBehavior(latency));
  }
}
```

Figure B.5: `CacheTest` Configuration