Simulation-Based Testing of Distributed Systems

Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf

Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado, 80309-0430 USA
{rutherfo,carzanig,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-1004-06 January 2006

© 2006 Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf

ABSTRACT

Developers of distributed systems routinely construct discrete-event simulations to help them understand and evaluate the behavior of inter-component protocols. Typically written using an imperative programming language, these simulations capture basic algorithmic functionality at the same time as they focus attention on properties critical to distribution, including topology, timing, bandwidth, and overall scalability. We ask the following question: Can simulations also be used to help in the testing of distributed-system implementations? Because simulations amount to specifications of intended behavior, the code of a simulation can be viewed as an operational, albeit non-traditional, formal model. We claim that this kind of model, when used within a specification-based testing regime, provides developers with the foundations of a powerful new method for selecting effective test suites. The primary tool used in our method is a fault-based analysis of the simulation code in which a set of mutants are generated using standard code-mutation techniques. The analysis can be used to rate the effectiveness of a test suite, as well as the criterion used to form it. We substantiate our claim through experiments performed on the simulations and implementations of two different distributed systems.

1 Introduction

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 [1], engineer distributed systems [20, 33], and improve distributed algorithms [8]. 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.

Simulations are used to understand and evaluate the functionality and performance of complex inter-component protocols and algorithms. Typically, they are written in an imperative programming language, such as C++ or Java. Simulations allow the developer to capture basic algorithmic functionality at the same time as they focus attention on topology, timing, bandwidth, overall scalability and other properties characteristic of distribution. Simulations embody abstractions for the underlying mechanisms and environmental conditions that affect these properties, providing parameters for exploring the functionality and performance space. 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 wonder whether there are other uses to which they can be put, in particular whether they can be used to increase the rigor with which distributed systems are tested. Toward that end, we make the following observation: *the code that implements a simulation of a distributed system is a formal specification of the intended functional behavior of that system, one whose behavior is parameterized by a well-defined set of controllable distribution properties in addition to normal inputs.* As such, the simulation code can be viewed as an operational, albeit non-traditional, formal model of the system.

Based on this observation, we make the following claim: *simulations can be used within a specification-based testing regime to provide developers of distributed systems with a method for selecting effective test suites.* The intuition leading to this claim is that, since the simulation code is a specification of intended behavior, it can be used to analyze test suites to predict their effectiveness at detecting failures in the implementation. The analysis takes particular advantage of the fact that the specification is executable, that it amounts to program code, and that it makes visible the environmental phenomena (message sequences, delays, throughput, etc.) of concern in distributed systems. We are assuming, of course, that the simulation code is a correct specification.

Test suites are typically constructed with respect to adequacy criteria. Adequacy criteria are studied as a means of organizing the testing activity, serving both as stopping conditions on testing and as measures of progress toward that goal. To date, however, adequacy criteria have not been studied specifically in the context of distributed systems.

At this point we are not concerned with inventing new criteria, but rather in helping the developer to determine whether a given test suite is adequate for a given criterion. Further than that we are interested in helping the developer to predict the relative effectiveness of adequate test suites—in other words, to select the most effective adequate test suite. Going still further, because it is essentially impossible to prove the universal effectiveness of an individual criterion, it is highly probable that different criteria will be effective for different systems. This should be especially true of distributed systems, whose differences are only exaggerated by the complicating factors of topology, timing, and the like. Therefore, we would like to provide developers with a method for evaluating the relative effectiveness of competing criteria. Discrete-event simulations provide a foundation for carrying out all these tasks.

We note that adequacy and effectiveness are separate, and separable, testing goals. The ultimate goal of any testing strategy is, of course, effectiveness at causing bug-revealing failures in an implementation. On the other hand, in a specification-based testing method, adequacy is defined in terms of the specification, not the implementation; for example, one might define some form of coverage criterion for the syntactic structures found in the specification. The question that must be answered, then, is whether the adequacy criterion defined on the specification leads to effectiveness in testing the implementation.

The primary tool used in our method is a *fault-based analysis* of the simulation code. The basic idea is to generate a set of mutants from the simulation code using standard code-mutation techniques [9]. Andrews et al. [3] showed the high fidelity of mutation-generated faults compared to real faults. Our method relies on this result, using mutants in the role of faulty "implementations" against which the developer can

evaluate and predict effectiveness. The effectiveness of an individual test case is measured by executing it on each mutant in turn (i.e., by running a simulation using each mutated version of the simulation code) and counting how many mutants are killed. A test case fails for a given mutant and a kill is recorded if the simulation enters an invalid state or does not terminate. A test suite is given an overall *mutant score* that is computed as the percentage of mutants killed by its test cases.

It is important to recall that for a simulated distributed system, a test case consists of normal functional input, plus input values for each of the simulated distribution and environmental parameters. Mutation of the simulation code potentially affects how the specified system is seen to deal with any of these inputs. This in turn is how we are able to specifically address the testing of distribution properties.

The goal of the work presented here is to substantiate our claim. To do so we have defined a simulation-based testing method and explained how it can be used to aid the developer in carrying out a variety of tasks of varying sophistication having to do with selecting effective test suites. In more concrete terms, we also performed experiments on the simulations and faulty implementations of two distributed systems. For purposes of these experiments, the adequacy criteria were drawn mainly from conventional testing strategies, specifically block coverage, branch coverage, and all-uses coverage (all of which are viable, due to the use of an imperative programming language for writing the specification), as well as from several input-partitioning methods.

We used the comprehensive experimentation and analysis method introduced by Frankl and Weiss [13]. 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 competing suites and criteria. To evaluate the different criteria, we employ a technique introduced recently by Briand et al. [5], in which different testing strategies are simulated, once the failure data for each test case has been collected.

The results of the experiments clearly show that even under the most simplistic usage scenario our method performs significantly better than a random selection process for test suites. Moreover, we are able to show that the method can successfully establish an effectiveness ranking among adequate test suites, as well as among adequacy criteria themselves. This presents the developer with a powerful new tool for organizing the testing activity and for tailoring it to the distributed system at hand.

In the next section we present our simulation-based testing method. In discussing the method, we take the perspective of the developer of a distributed system and consider several ways that they might approach the problem of testing their implementation. Section 3 reviews the experimental setup and subjects. The details of the experiments, their results, and threats to validity are presented in Section 4. Section 5 reviews related work. Section 6 concludes with a brief look at future work.

2 Simulation-Based Testing

As noted above, discrete-event simulations 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 are used by processes to exchange information. When simulating distributed systems, processes are used to represent the core components of the system, as well as environmental entities such as the underlying 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 connected into a particular configuration that is then executed.

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 represented in the simulation by having the *Network* process control the scheduling of event deliveries. The unreliable nature of the network is represented by having the *Network* process randomly drop events by not scheduling them at

all. 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?

2.1 Basic Concepts

Simulation-based testing is used within a simple and generic testing process. As a first step, the developer assembles a *test suite*. As usual, 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. Then, the developer determines whether the suite is adequate, and if it is, they use it to test the implementation.

The simulation code plays the role of the specification in specification-based testing. Therefore, simulation is used to decide the adequacy of the test suite. The process by which individual test cases are created or generated is outside the scope of this paper. Similarly, we do not 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, the simulation-based testing method we propose 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.

This use of simulation requires that the developer define one or more adequacy criteria. 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 generate multiple adequate suites, and might want to predict their relative effectiveness to maximize the overall effectiveness of the testing process.

Therefore, the second idea is to provide the developer with a general ranking mechanism to: (1) "bootstrap" the testing process in the absence of a specific adequacy criterion; (2) guide the selection of the most effective criterion for the system at hand; and (3) fine tune the selection of the most effective suite within the set of adequate suites, given a chosen criterion. 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.

2.2 Fault-Based Analysis

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 specification in well-defined ways to produce a set of incorrect versions of the specification. These incorrect versions, or *mutants*, can be used to compare testing strategies.

For example, an implementation might have a fault that causes a particular state 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 the incorrect specifications from the correct one 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. We discuss fault-based testing in greater detail in Section 5.

A prerequisite of fault-based analysis is the existence of a set of mutation operators that can be applied to the specification being analyzed. Simulations are typically coded in imperative programming languages

and are therefore well suited to the code-mutation operators developed in the context of mutation testing [9]. 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 simulation code, thereby obtaining a set of faulty specifications. To evaluate a test suite, we run the all the test cases against all the mutant simulations. For each run, one of several things may happen:

- the simulation terminates normally with reasonable results;
- the simulation terminates normally with unreasonable results;
- the simulation does not terminate; or
- the simulation is invalid and terminates abnormally.

In most mutation analyses, output from the original version is used as an oracle against which mutant output is compared. In discrete-event simulations of distributed systems, this is not always possible because the discrete-event core uses separate threads for executing each process and is therefore nondeterministic. In practice, we use assertions and sanity checks in the simulation code to determine which results are considered "reasonable".

For all but the first situation listed above, the test case is recorded as having killed the mutant. The *mutant score* of a test suite is computed as the percentage of all mutants killed by at least one test case in the suite.

2.3 Method

We propose a method in which fault-based analysis of the simulation code and simulation-based adequacy criteria are used, individually or in combination, to support the identification of effective test suites. We describe this method through six different scenarios. In each scenario, we assume that the simulation code is available.

No information. The first scenario is one where the developer has little or no experience with the system under test. The developer is therefore unable to select any specific adequacy criterion and is looking for any guidance as to how to evaluate test suites. In this case, we propose to use fault-based analysis. The developer generates a number of test suites, obtains their mutant score, and picks the suite with the highest mutant score.

Criteria based on input values. The developer chooses a criterion that depends exclusively on the input vector. An example is a suite that is adequate if it covers the partitions of the input space. In this case, the simulation code is of no use in evaluating adequacy, and the developer must do that through other means. However, simulation-based testing can still contribute in this scenario by providing a relative ranking of adequate suites based on mutant score. Notice that the first scenario, above, could be seen as a special case of this scenario, where the "null" criterion is chosen.

Criteria based on environmental phenomena. The developer chooses a criterion that does not depend on the simulation code, but that depends on observable events from the environment. For example, the criterion could require that communication links be used up to their maximum capacity for more than a given amount of time during the execution of at least one test. In this scenario, the simulation can directly evaluate the adequacy of a test suite, but the developer must execute the test cases using simulation code specifically instrumented to record events of interest. For the example criterion to be applied to the client/server system, this would involve modifying the code of the *Network* process to monitor and log link utilization.

Criteria based on the simulation code. The developer chooses a criterion defined over the simulation code. In this case, too, the developer can use the simulation directly to evaluate the adequacy of a test suite. As in the previous scenario, it is necessary to instrument the simulation code in order to compute the adequacy. The difference in this case is that the instrumentation applies to the processes that execute the simulation code. For example, for the statement-coverage criterion mentioned in Section 2.1, this involves the insertion of instrumentation code to log the execution of each basic block.

Effectiveness of a single chosen criterion. The developer chooses a specific adequacy criterion, which may be based on input, observable events, simulation code, or any combination thereof. However, the developer notices that there is significant variability in the effectiveness of suites that are adequate for that particular criterion. 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.

Ranking candidate criteria. The developer wants to choose a criterion. However, for the particular system under test, there exist many applicable criteria whose relative effectiveness is not known. Once again, the developer can resort to fault-based analysis of the simulation code to rank the different criteria. Specifically, the developer creates adequate suites for each one and then selects the suite with the highest mutant score.

In summary, simulation 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, including the "null" criterion, and to inform the programmer in selecting a criterion. This is done by means of a fault-based analysis of the simulation code. In Section 4, we experimentally evaluate the benefits of these techniques.

In terms of cost, the test selection process we propose is advantageous because it only requires the execution of simulations, and therefore avoids the cost of setting up the system under test over complex distributed testbeds [35]. Specifically, deciding whether a test suite is adequate with respect to a given criterion requires only one execution of the simulation for each test suite. Fault-based analysis is more expensive, as it requires several simulation executions (one for each mutant) for each test suite.

3 Subject Systems

Our study uses simulations and implementations of two subject distributed systems realizing well-known distributed algorithms. The first, 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.

Implementation of both systems were given as programming assignments in an introductory undergraduate networking course taught at the University of Lugano. We used the assignment description provided to students and the Kurose and Ross networking textbook [18] to create the simulations.

Student implementations of each system were used as candidate subjects for our experimental evaluation. For GBN we also developed a Java-based implementation and created faulty versions using the MuJava automatic mutation tool [21].

For the simulations we used the *simjava* discrete-event simulation engine.[1] In the course of this work we developed a thin layer over the discrete-event core that provides a more natural way to schedule and receive events. This layer also includes a process implementing the behavior of a probabilistically unreliable network that is used by both systems.

3.1 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 assignment given to the students required the sending window size to grow and shrink as dictated by the history of acknowledgment packets received.

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

GBN 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.

3.1.1 Specification

The GBN simulation code consists of two event classes, `ACK` and `DAT`, which represent acknowledgments and data packets, respectively. The `Sender` process 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 code.

We defined a single GBN simulation consisting of one server and one receiver. This simulation 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 $[0.0, 0.20]$; and
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 unreliable network process to randomly drop and duplicate events. The network process delays all events by the same amount.

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

3.1.2 Implementations

Student implementations of GBN were coded in an average of 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.

Our stand-alone implementation, written in Java, 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.

3.2 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.

3.2.1 Specification

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.

We defined a single 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 1 shows each possibility graphically);

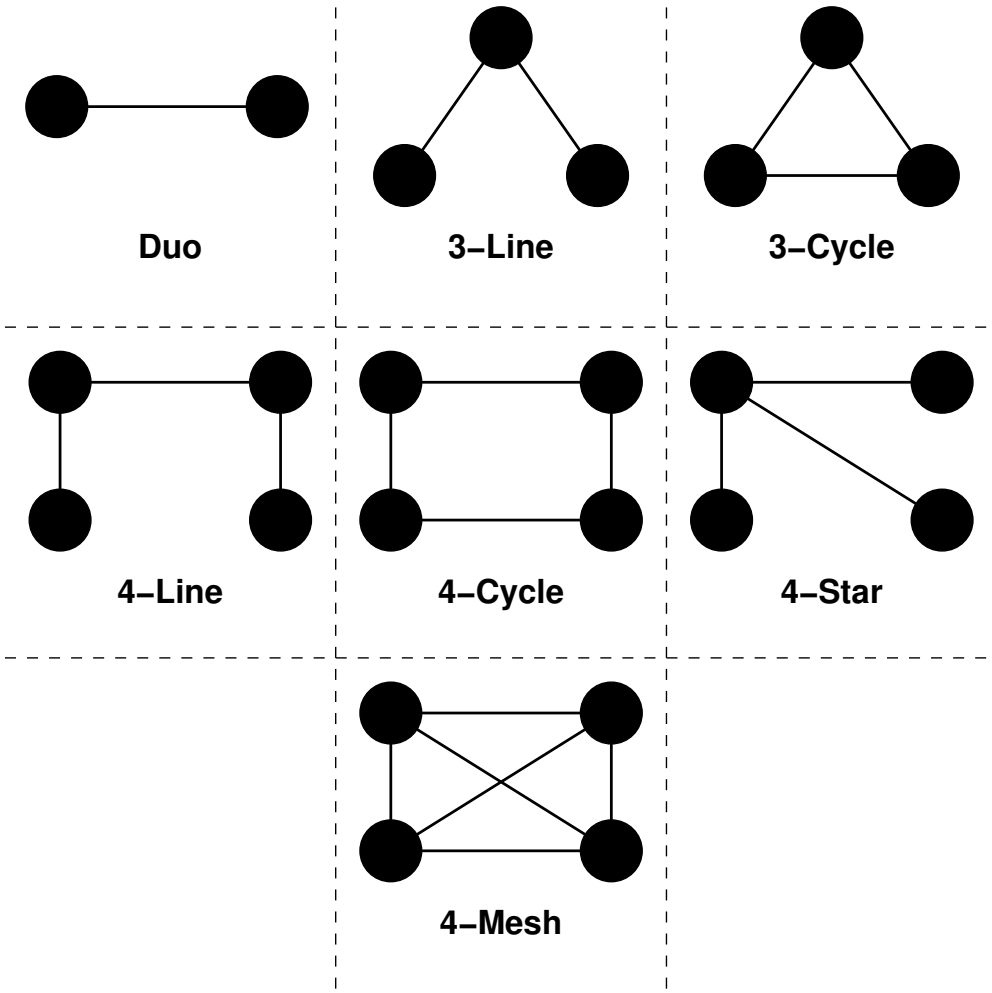

Figure 1: LSR Topologies

2. **message count**: the number of messages to be sent, a value in the range $[0, 2n]$, where n is the number of routers in the topology; and

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 member 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 publication is also correct.

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

3.2.2 Implementations

Students were again provided with an application framework within which they implemented the algorithm. On average, students implemented the LSR logic within 120 lines of Java and the framework itself is about 500 lines of code. In the course, this framework merely simulated the network interaction. For our experiments, we reimplemented the framework to run over the network. This change required some of the supporting classes written by students to be altered to implement the `java.io.Serializable` interface. This 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 Experiments

We now describe our experiments. First we discuss the criteria and metrics used in the experiments. Then we describe the preparation step in which we gather the raw data. Finally, we describe the two experiments we conducted to evaluate the fault-based analysis described in Section 2.2.

4.1 Adequacy Criteria

We consider white-box and input-partitioning criteria in our experiments. The white-box criteria are the well-known *all-blocks*, *all-branches*, and *all-uses* coverages [14] 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. We create adequate suites for these criteria by randomly selecting test cases from the universe and including them if they improve the coverage value.

Input-partitioning criteria are defined with respect to the input domain of the simulations we defined for each system. Once partitions are selected, we grow suites 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.

The input-partitioning criteria we devised for GBN and LSR are:

- **GBN IP-1**: This criterion simply divides value ranges of each GBN 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 "harder".

- **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 1 are represented. The intuition behind this is that 4-router topologies are more complicated and therefore will make better test scenarios.

4.2 Metrics

We compare adequacy criteria based on their effectiveness (E). This is measured as the average proportion of faults found by adequate test suites. Others have instead defined and measured effectiveness on a per-subject basis as the proportion of adequate test suites that fail. Our metric 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 to be three times as effective as a suite that finds just one.

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-value 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, when determining 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 of statistics [10] and assume that the distribution of the normalized form of our test statistic (E) approximates a normal distribution. According to Devore [10], we can use this theorem comfortably with sample sizes larger than 30. 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, we reject H_0 and conclude that $A > B$.

4.3 Preparation

As preparation for our experiments, we simulate each test case using the correct simulation code and all of its mutants. We also execute each test case against each implementation under test (IUT). As described in Section 3, the universes for GBN and LSR contain 2500 and 7000 test cases respectively.

We generate mutants of the simulation code by using all conventional mutation operators provided by the MuJava tool [21]. This resulted in 488 mutants for GBN, and 59 for LSR. We attribute the disparity in the number of mutants to the amount of integer arithmetic used in the GBN algorithm.

We simulate test cases against all mutants of the simulation code. This step is quite expensive, and we mitigate this by aggressively excluding mutants with high failure rates. We identify these "pathological" mutants by initially simulating a small sample of test cases against all mutants. Following this, 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. The same approach is used by others in empirical studies [12] to focus attention on faults that are hard to detect. After this initial sampling, we also examined the code of mutants with zero kills and eliminated any ones semantically equivalent to the original code.

We also simulate each test case against the "golden" specification and collect coverage data.

Finally, we execute each test case against all implementations under test. For GBN, we started with 19 student implementations and 192 non-equivalent mutants of a Java implementation;[2] for LSR we had 16 student implementations.

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

Table 1: Implementation Failure Rates

After executing an initial sampling of test cases for each IUT, we eliminated those having failure rates higher than 20%. After executing all test cases, we also exclude correct implementations (i.e., those without failures), leaving the set of subjects shown in Table 1.

Effectiveness values are obtained by simulating each criterion 200 times. Table 2 contains the average size (n) and average effectiveness of each strategy for GBN and LSR. These data show that the mix of input-partitioning and white-box criteria is quite natural.

4.4 Experiment 1

One of the claims we make in Section 2 is that using a fault-based analysis we can improve the effectiveness of criteria. To justify this claim, we perform the following experiment for each criterion:

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.

[2]On principle, we wanted to avoid using mutants as IUT, 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.

		n	$\bar{E}$
GBN	all-blocks	3.02	0.19
	all-branches	4.12	0.22
	IP-1	6.92	0.29
	all-uses	4.68	0.35
	IP-2	11.35	0.38
LSR	all-blocks	1.60	0.46
	all-uses	2.92	0.61
	all-branches	3.05	0.64
	Quartet	8.15	0.84
	Pair-50	17.08	0.89

Table 2: Criteria Effectiveness

We performed this process 100 times each with values of M ranging from two to eight and determined the smallest M value at which the fault-based technique is statistically better than the baseline effectiveness of the criterion (with $\alpha = 0.05$).

		min-M
GBN	all-blocks	2
	all-branches	2
	IP-1	2
	all-uses	–
	IP-2	3
LSR	all-blocks	2
	all-uses	2
	all-branches	3
	Quartet	3
	Pair-50	3

Table 3: Experiment 1 Data

Table 3 shows the results of these experiments. For nearly all of the criteria, the minimum multiplier M is 3 or less, meaning that the tester starts to see *significant* improvement of effectiveness with only 2 additional adequate suites. We did find, however, for *all-uses* with GBN that there was no improvement, even after analyzing seven additional adequate suites.

We also tested the opposite hypotheses, which is that the effectiveness drops when increasing the number of suites considered, and this never occurred. Thus, we have shown that a tester sees significant improvement at relatively small multipliers without any downside for nearly all criteria considered.

4.5 Experiment 2

The second experiment we conducted was aimed at evaluating our ability to use fault-based analysis to determine the relative effectiveness of candidate adequacy criteria.

During our preparation phase, we created 200 adequate suites for each criterion and computed their effectiveness values. These data are summarized in Table 2. For this experiment, we compute the mutant scores of the same suites.

Using these data we perform hypothesis testing on each pair of criteria to determine the ranking of criteria both with the actual effectiveness values, and with the mutant scores.

Figures 2 and 3 depict all statistically significant relationships with *p-values* less than 0.05. For example, Figure 3 indicates that Pair-50 has a consistently higher mutant score than Quartet, which is itself higher

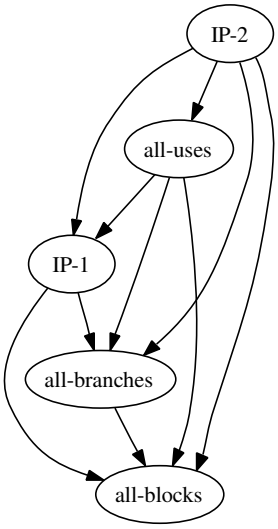

Figure 2: GBN Criterion Relationships

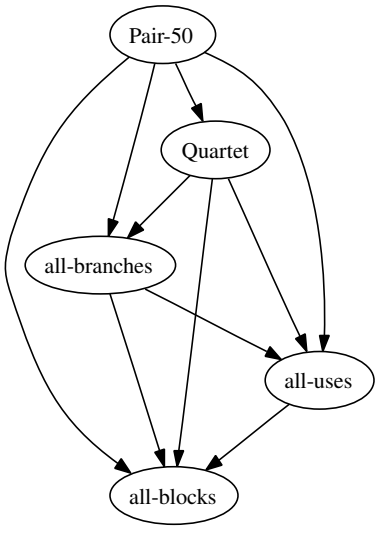

Figure 3: LSR Criterion Relationships

than all-branches and all-uses, etc. Our experiments show that the same exact relations between criteria hold when the criteria are applied to the implementations. Since the graphs are identical, we do not report them in the paper. Instead, we report the *p-value* computed for each relation using the mutant score and the effectiveness, respectively.

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: GBN Criteria *p-values*

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 5: LSR Criteria *p-values*

Tables 4 and 5 show the computed *p-values* for GBN and LSR, respectively. The hypothesized relation between the criteria are listed in the first column. For each relation, the second column reports the *p-values* for the mutant score (simulation) while the third column reports the *p-values* for the effectiveness (implementation). Notice that the correspondence is virtually exact with *p-values* all lower than 0.02.

4.6 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, 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 first author who had no contact with the students or the materials they were presented with before receiving the implementations that make up the bulk of the empirical subjects.

Finally, as the scope of our empirical study is limited to 17 implementations of two 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 in particular our fault-based analyses, at this point we 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.

5 Related Work

In this section we discuss the relation of our work to other research efforts. First, we summarize existing specification-based testing techniques with an emphasis on those that are applicable to distributed systems. Next, we place our work in the context of existing fault-based techniques.

5.1 Specification-Based Testing

The work of Richardson, O'Malley, and Tittle [29] 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 [26], 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 [6, 7] on using a function-level assertion-based language to guide testing. Offutt and Liu [24] describe the generation of test data from a higher-level, object-oriented specification notation. Offutt et al. [25] describe the use of generic state-based specifications (e.g., UML and statecharts) for system-level testing. Harder et al. [15] 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 [4] and Lai [19] 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 [27]. A number of researchers have studied the use of software architectures for testing. Richardson and Wolf [31] propose several architecture-based adequacy criteria based on the Chemical Abstract Machine model. Rice and Seidman [28] describe the ASDL architecture language and its toolset, and discuss its use in guiding integration testing. Jin and Offutt [16] define five general architecture-based testing criteria and applied them to the Wright ADL. Muccini et al. [22] 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 Fault-Based Testing

In fault-based testing, models of likely or potential faults are used to guide the testing process. The best-known fault-based testing technique is probably mutation testing [9]. In mutation testing, the engineer applies mutation operators [23] 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 [9]. 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. In our work we use a standard set of mutation operators for Java as implemented by the MuJava tool [21]. However, we do not use the generated mutants for mutation testing, but rather we use them to measure other adequacy criteria.

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 [11] and Estelle specifications [32]. 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 [2] 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 [30, 36]. Recently, a fault-class hierarchy has been determined analytically and used to explain some of the earlier experimental results [17, 34]. We are targeting a more expressive specification method whose fault classes (mutation operators) are not amenable to a general analytical comparison.

6 Conclusion

The work described in this paper makes two main contributions to the field of testing. First, we identify the potential for using discrete-event simulations in the specification-based testing of distributed systems and propose a concrete process for doing so. Second, we leverage the executable nature of these specifications in a novel fault-based analysis method and identify several ways in which the method can be useful to developers of distributed systems. Our approach is validated by an initial empirical study of two distributed systems.

In the future, we plan to continue our work with simulation-based testing by estimating test execution time using the virtual time derived from the simulations. This should provide a useful measure of cost, which can be factored into the prediction of effectiveness. We also plan to investigate ways in which the

simulations can be used as advanced oracles. Finally, we will be looking into ways in which the fault-based analysis method can be used to determine relationships between regions of the input space and effectiveness, leading to new kinds of adequacy criteria for testing distributed systems.

Acknowledgments

The work reported here was supported in part by the National Science Foundation and Army Research Office under agreement numbers ANI-0240412 and DAAD19-01-1-0484.

References

[1] Mark Allman and Aaron Falk. On the effective evaluation of tcp. *SIGCOMM Comput. Commun. Rev.*, 29(5):59–70, 1999.

[2] 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(4):275–299, 2001.

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

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

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

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

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

[8] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[9] R. A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[10] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole, 4th edition, 1995.

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

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

[13] Phyllis Frankl and Stewart Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[14] Phyllis Frankl and Elaine Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[15] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 60–71. IEEE Computer Society, 2003.

[16] Zhenyi Jin and Jeff Offutt. Deriving tests from software architectures. In *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*, pages 308–313, November 2001.

[17] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, 1999.

[18] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Pearson Benjamin Cummings, 2004.

[19] R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62(1):21–46, 2002.

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

[21] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[22] H. Muccini, A. Bertolino, and P. Inverardi. Using software architecture for code testing. *IEEE Transactions on Software Engineering*, 30(3):160–171, March 2004.

[23] 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(2):99–118, 1996.

[24] A. Jefferson Offutt and Shaoying Liu. Generating test data from SOFL specifications. *Journal of Systems and Software*, 49(1):49–62, 1999.

[25] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Journal of Software Testing, Verification and Reliability*, 13(1):25–53, March 2003.

[26] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[27] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[28] Michael D. Rice and Stephen B. Seidman. An approach to architectural analysis and testing. In *Proceedings of the 3rd International Workshop on Software Architecture*, pages 121–123. ACM Press, 1998.

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

[30] 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(6):533–553, 1993.

[31] Debra J. Richardson and Alexander L. Wolf. Software testing at the architectural level. In *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*, pages 68–71. ACM Press, 1996.

[32] 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(4):285–301, 1999.

[33] 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(1):17–32, 2003.

[34] Tatsuhiro Tsuchiya and Tohru Kikuno. On fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 11(1):58–62, 2002.

[35] Yanyan Wang, Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. Automating experimentation on distributed testbeds. In *Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 164–173, Long Beach, CA, November 2005.

[36] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, 1994.