Simulation-Based Test Adequacy Criteria for Distributed Systems[*]

Matthew J. Rutherford[†] Antonio Carzaniga[†,‡] Alexander L. Wolf[†,‡]

[†]Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430

[‡]Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland

ABSTRACT

Developers of distributed systems routinely construct discrete-event simulations to help understand and evaluate the behavior of inter-component protocols. Simulations are abstract models of systems and their environments, capturing basic algorithmic functionality at the same time as they focus attention on properties critical to distribution, including topology, timing, bandwidth, and overall scalability. We 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 rich new basis for defining and applying system-level test adequacy criteria. We describe a framework for evaluating distributed system test adequacy criteria, and demonstrate our approach on simulations and implementations of three distributed systems, including DNS, the Domain Name System.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

Distributed systems, discrete-event simulation, test adequacy criteria, fault-based analysis

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 [31], and improve distributed algorithms [7]. They are appealing to developers because of their inherent efficiency and scalability. 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 purposes to which they can be put. In particular, we ask whether they can increase the rigor with which distributed systems are tested, and make the following hypothesis: *Simulations can be used within a specification-based testing regime to help developers of distributed systems define and apply effective system-level test suites.* The intuition leading to this hypothesis 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.

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

Test suites are typically constructed with respect to adequacy criteria. Adequacy criteria are used as a means of organizing the testing activity, serving both as stopping conditions on testing and as measures of progress toward that

[*]Research supported in part by the National Science Foundation and Army Research Office under agreement numbers ANI-0240412 and DAAD19-01-1-0484.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM 1-59593-468-5/06/0011 ...$5.00.

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

goal. To date, adequacy criteria have been studied in the context of sequential (e.g., Frankl and Weyuker [13]) and concurrent (e.g., Carver and Tai [5]) systems, but not in the context of distributed systems.

In this paper 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 [3], 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 are led to 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. We argue that a test suite with a higher level of simulation-code coverage, under a valid adequacy criterion, will have a greater effectiveness at causing such failures in the implementation.

Notice, too, that we do not assume the simulation to be correct. As in other specification-based testing or analysis approaches, the specification is correct only by definition and, therefore, test failures merely establish non-conformance of the implementation and specification.

The primary conceptual contribution of this paper is the notion that simulations can be used to define and evaluate adequacy criteria for system-level testing of distributed systems. We give baseline experimental evidence that valid adequacy criteria do indeed exist. Yet it is highly probable that criteria will differ in their effectiveness 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 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 second contribution of this paper, in which we demonstrate 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.

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 34 releases of MaraDNS, which consists of between 15,000 and 24,000 lines of code, depending on the version.

In our studies we used the comprehensive experimentation and analysis method introduced by Frankl and Weiss [12]. 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 described by Briand et al. [4], 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 approach performs significantly better than a random selection process for test suites. Moreover, we are able to show that we can successfully establish an effectiveness ranking among adequate test suites, as well as among the 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 the details of our approach to establishing simulation-based test adequacy criteria. In discussing this, 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. SIMULATIONS AND 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.

Consider a simple client/server system designed to operate over a network with unreliable communication. A simulation of this system might consist 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

We refer to 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, uses 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 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.

This use of simulation, as with all adequacy-based testing techniques, requires the developer to choose a particular criterion, and to select test cases that comprise only a single adequate test suite. Making each of these decisions exposes the developer to risk. First, there is little empirical or analytical data that the developer can use for guidance in selecting an adequacy criterion that is likely to be effective for their particular system. Therefore, they run the risk of selecting a criterion that happens to be less effective than another candidate criterion; we refer to this as *inter-criterion risk.* Second, there is often significant variability in effectiveness of test suites adequate with respect to a particular criterion, exposing the developer to the risk of selecting an ineffective, adequate test suite; we refer to this as *intra-criterion risk.*

Therefore, the second idea is to provide the developer with a general ranking mechanism to: (1) guide the selection of the most effective criterion for the system at hand and (2) 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 *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 incorrect from correct specifications are said to cover that particular fault class. The underlying assumption of this kind of fault-based analysis, known as the *coupling effect* [8], 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 that can be applied to the specification. Simulations are typically coded in imperative programming languages and so well suited to the code-mutation operators developed in the context of mutation testing [8]. 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 specifications. Each individual test case is then applied to each mutant in turn. That is, for each test case, we run a simulation using each mutated version of the simulation code. 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 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 with simulation-based testing 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”.

2.3 Usage Scenarios

We propose to use fault-based analysis of the simulation code and simulation-based adequacy criteria, individually or in combination, to support the identification of effective test suites. We describe this approach through 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 developer is exposed to both types of risk, inter-criterion and intra-criterion, 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 developer 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 suite to the implementation. This usage is more costly than the conventional usage, since multiple adequate suites must be selected, and each selected test suite must undergo a fault-based analysis. On the other hand, intra-criterion risk is reduced.

Ranking. In this scenario, the developer is looking to rationally decide between multiple criteria. So, the developer creates many adequate test suites for each candidate criterion, and uses fault-based analysis to determine which criterion is likely to be the most effective, thereby reducing inter-criterion risk. At this point, the developer simply uses boosting on the adequate test suites already created for the highest-ranking criterion.

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 and to inform the developer 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 [33]. Specifically, deciding whether a test suite is adequate with respect to a given criterion requires only one execution of the simulation for each test case. Fault-based analysis is more expensive, as it requires multiple simulation executions (one for each mutant) for each test case in a test suite.

3. SUBJECT SYSTEMS

Our study 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 [17] as source descriptions. For the DNS resolver we used historical releases of MaraDNS,[2] an open-source resolver that implements the core DNS functionality as described by RFCs 1034 and 1035.

To simulate each system, we used the *simjava* discrete-event simulation engine.[3] 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.

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

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.

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.

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 in $[1, 335760]$.

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 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 parameter tuples.

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

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

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

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 faulty versions using the MuJava automatic mutation tool [18]. 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.

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.

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.

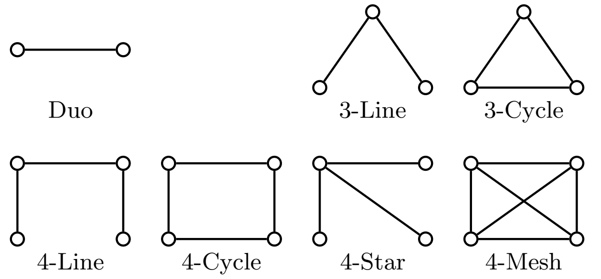


Figure 1: LSR Topologies

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 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.

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 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.

3.3 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 "root" 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.

Specification

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 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 provided 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.

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. Again, the source code for the entire MaraDNS package ranges from roughly 15KLOC to 24KLOC, depending on the release.

During test executions we used version 2.0.1 of *dnsjava*[4] as a stub resolver implementation, and the *tinydns* server included with version 1.0.5 of *djbdns*[5] as a server implementation; in this paper we assume they 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. EXPERIMENTS

We now present the experiments that validate our approach. We first describe the white-box specification-code-coverage criteria and metrics, and then describe a preparation step in which we gather raw data.

The white-box criteria used in our experiments are the well-known *all-blocks*, *all-branches*, and *all-uses* coverages [13] 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.

We compare adequacy criteria based on their effectiveness, E. This is measured as the average proportion of faults found by adequate test suites. Others define and measure effectiveness on a per-implementation 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, 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 [9]. 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 $A > B$.

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, LSR and DNS contain 2500, 7000, and 2000 test cases, respectively.

We generate mutants of the simulation code by using all conventional mutation operators provided by the MuJava tool [18]. This resulted in 488 mutants for GBN, 59 for LSR, and 230 for DNS. We attribute the comparatively large number of GBN mutants to the amount of integer arithmetic used in this 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.

[4] `http://www.dnsjava.org/`

[5] `http://cr.yp.to/djbdns.html`

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 [11] 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" (i.e., non-mutated) specification and collect coverage data.

Finally, we execute each test case against all IUTs. For GBN, we started with 19 student implementations and 192 non-equivalent mutants of a Java implementation.[6] For LSR we had 16 student implementations, and for DNS we used the 34 public releases.

GBN		LSR		DNS	
IUT	**%**	**IUT**	**%**	**IUT**	**%**
stud07	0.28	stud16	5.15	1.0.29	9.15
stud06	0.32	stud03	5.21	1.1.59	15.25
stud08	0.40	stud08	7.70	1.1.91	17.30
stud15	3.16	stud01	7.94	1.2.03.3	17.45
stud09	3.40	stud06	8.02	1.1.60	17.55
stud18	4.72	stud14	8.48	1.2.01	18.00
mutROR15	16.96	stud15	12.17	1.2.03.5	18.20
mutLOI35	17.00	stud09	12.42	1.2.07.1	18.40
				1.2.00	18.45
				1.1.61	18.70
				1.2.03.4	18.90

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 excluded correct implementations (i.e., those without failures), leaving the set of subjects shown in Table 1.

4.1 Test Suite Adequacy and Effectiveness

The first experiment we describe is aimed at verifying our most basic claim that test suites adequate with respect to white-box simulation-code-coverage criteria are effective at testing implementations.

Experiment 1

In this experiment we create many adequate suites for each criterion out of the universe of test cases, and determine effectiveness against the IUTs in Table 1. After doing this for the white-box techniques, we generate random test suites from the universe at the sizes corresponding to the size of the white-box suites.

Figure 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.

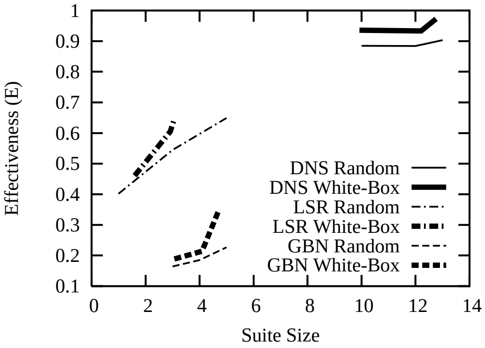


Figure 2: Results for Experiment 1

The effectiveness relationships between white-box criteria uncovered during this experiment are interesting. Based on the classic literature, the expected relationship is *all-uses* > *all-branches* > *all-blocks*. However, this is only supported experimentally for GBN. For LSR, *all-branches* > *all-uses*, while for DNS *all-branches* is more effective than both *all-branches* and *all-blocks*, which have have statistically equivalent effectiveness. We do not believe that these inconsistent relationships are due to a failing in simulation-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 [12].

4.2 Improving and Ranking Criteria

The next two experiments are aimed at validating our claims about the fault-based analysis techniques. For these experiments, 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 (referred to below as *IP-1*, *IP-2*, *Quartet*, *Pair-50*, *Resources*, and *TypicalUsage*) are provided elsewhere [29]. Here it suffices to say that they derive their test suites from analyses of the functional and distribution input spaces, rather than the code of the simulations. Nevertheless, the presence of the simulation code is critical to the technique, as we explain below.

Experiment 2

One of the claims we make in Section 2 is that by using a fault-based analysis of simulation code in the *boosting* usage, we can reduce the risk of selecting an ineffective, adequate test suite. If test suites with higher effectiveness are preferentially selected, then the net effect should be that test suites selected using this method should be statistically more effective than the baseline population of adequate test suites. To justify this claim, we perform the following experiment for each criterion:

[6] On principle we avoid using mutants as IUTs, 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.

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 performed this process 100 times each with values of M ranging from 2 to 8 and determined the smallest M value at which the fault-based technique is statistically better than the baseline effectiveness of the criterion (with $\alpha = 0.05$).

Table 2 shows the results of these experiments. These tables show, for each criterion, the minimum M value, the baseline effectiveness, labeled E, and the boosted effectiveness, labeled $E+$. For virtually all of the criteria applied to GBN and LSR, the minimum multiplier M is 3 or less, meaning that the developer sees *significant* improvement of effectiveness with only 1 or 2 additional adequate suites. We did find, however, that for *all-uses* with GBN there was no improvement, even after analyzing seven additional adequate suites.

For DNS, shown in Table 2(c), aside from *all-uses*, the other criteria require more candidate test suites to boost. Both *Resources* and *TypicalUsage* require three additional candidate test suites before significant improvement is realized, *all-blocks* requires seven additional test suites, and *all-branches* could not be boosted, even when considering eight suites.

While the boosting usage, particularly for DNS, can require significant effort to gain effectiveness, it appears to come with no downside in terms of effectiveness. We evaluated this by considering the opposite hypotheses, which test whether the effectiveness drops when increasing the number of suites considered. This never occurred. The data show that we can gain a significant improvement at relatively small multipliers without any downside for nearly all criteria considered.

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.

Experiment 3

Next we validate our ability to use fault-based analysis of simulation code 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. For this experiment, we compute mutant scores of the same suites, and then perform hypothesis testing on each pair of criteria to determine rankings based on implementation effectiveness values and mutant scores. By doing this we are able to compare the relations predicted under the fault-based analysis with the actual empirical relationships that emerge from running the tests.

Figure 3 depicts all statistically significant relationships with *p-values* less than 0.05. (Note that although all pairwise relationships are shown, the relationships are indeed transitive.) For example, Figure 3b indicates that *Pair-50* has a consistently higher effectiveness than *Quartet*, which is itself higher than *all-branches* and *all-uses*, etc.

Our experiments show that virtually the same exact relationships between criteria hold when using mutant scores derived from the simulations. Since the graphs are identical, we do not report them here. Instead, Table 3 shows the computed *p-values* for relationships between the criteria applied to GBN. Each row in this table corresponds to an edge in Figure 3a.

The hypothesized relationship between the criteria are listed in the first column. For each relationship, 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.

The results for LSR and DNS are similar, so we omit them due to space considerations. In particular, for LSR the fault-based analysis accurately predicts the reversal of the relation between *all-uses* and *all-branches*.

For DNS, the fault-based analysis faithfully predicts all the relationships found when testing implementations. Fault-based analysis also predicts a relationship, *all-uses* > *all-blocks*, that is not supported by the implementation experiments. Therefore, a developer might incorrectly decide to use *all-uses* instead of *all-blocks*. While statistically this would be an incorrect choice, in real terms there is very little difference between the two criteria. Referring to the effectiveness values in Table 2(c), both *all-uses* and *all-blocks* have an effectiveness slightly greater than 0.93.

4.3 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 first 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 ex-

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 3: GBN Criteria *p-values*

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

(a) GBN

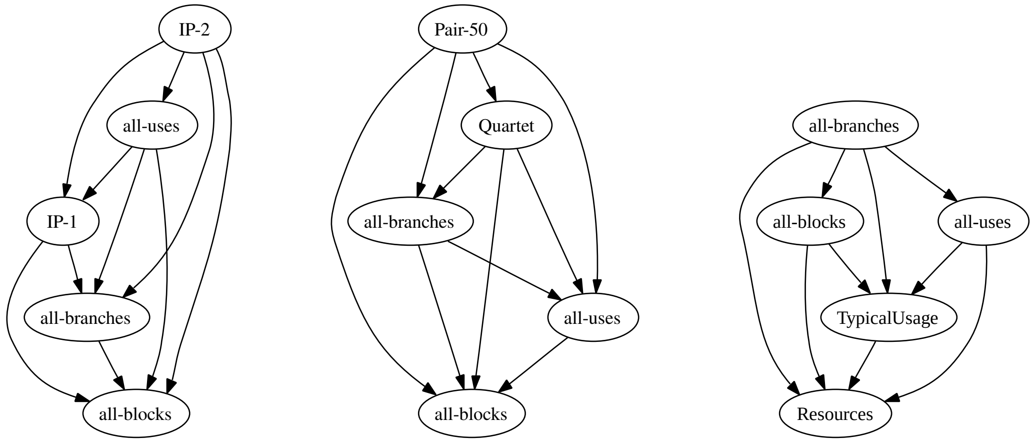
Criterion	min M	E	$E+$
all-blocks	2	0.46	0.51
all-branches	3	0.64	0.69
all-uses	2	0.61	0.68
Quartet	3	0.84	0.87
Pair-50	3	0.89	0.92

(b) LSR

Criterion	min M	E	$E+$
all-blocks	8	0.936	0.955
all-branches	–	0.973	0.973
all-uses	2	0.933	0.968
Resources	4	0.642	0.69
TypicalUsage	4	0.722	0.823

(c) DNS

Table 2: Results for Experiment 2



(a) GBN (b) LSR (c) DNS

Figure 3: Criteria Relationships

ternally 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.

5. RELATED WORK

We now discuss related research efforts. We summarize existing specification-based testing techniques, with an emphasis on those that are applicable to distributed systems. We then discuss existing fault-based techniques.

5.1 Specification-Based Testing

The work of Richardson, O'Malley, and Tittle [26] 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 [23], 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 developers 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. Chang and Richardson [6] propose a function-level, assertion-based language to guide testing. Offutt and Liu [21] describe the generation of test data from a higher-level, object-oriented specification notation. Offutt et al. [22] describe the use of generic state-based specifications (e.g., UML and statecharts) for system-level testing. Harder et al. [14] 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. Bochmann and Petrenko [3] describe 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.

Software architectures have been studied as a means to describe and understand large, complex systems [24]. A number of researchers have studied the use of software architectures for testing. Richardson and Wolf [28] propose several architecture-based adequacy criteria based on the Chemical Abstract Machine model. Rice and Seidman [25] describe the ASDL architecture language and its toolset, and discuss its use in guiding integration testing. Jin and Offutt [15] define five general architecture-based testing criteria and applied them to the Wright ADL. Muccini et al. [19] 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 LTSs (ALTSs) 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 ALTSs, 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. Simulations are encoded in languages 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 [8]. In mutation testing, the developer applies mutation operators [20] 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 [8]. 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 [18]. 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 specification formalisms (e.g. Estelle [30] and statecharts [10]). This work differs from ours in that their goal is specification testing, while ours is specification-based implementation testing. The mutation operators 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 adequacy 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 [27, 34]. Recently, a fault-class hierarchy has been determined analytically and used to explain some of the earlier experimental results [16, 32]. 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 here 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 three 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.

7. REFERENCES

[1] M. Allman and A. Falk. On the effective evaluation of TCP. *SIGCOMM Computer Communications Review*, 29(5):59–70, 1999.

[2] P. E. Ammann and P. 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] G. Bochmann and A. 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, 1994.

[4] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statecharts. In *Proceedings of the 26th International Conference on Software Engineering*, pages 86–95, 2004.

[5] R. H. Carver and K.-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24(6):471–490, 1998.

[6] J. Chang and D. 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, 1999.

[7] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

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

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

[10] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, page 210, Washington, DC, USA, 1999.

[11] P. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 153–162, New York, NY, USA, 1998.

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

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

[14] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, 2003.

[15] Z. Jin and J. Offutt. Deriving tests from software architectures. In *Proceedings of the 12th IEEE International Symposium on Software Reliability Engineering*, pages 308–313, Nov. 2001.

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

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

[18] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

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

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996.

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

[22] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Journal of Software Testing, Verification and Reliability*, 13(1):25–53, Mar. 2003.

[23] 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.

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

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

[26] 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, 1989.

[27] D. Richardson and M. 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.

[28] D. J. Richardson and A. 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, 1996.

[29] M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Simulation-based testing of distributed systems. Technical Report CU-CS-1004-06, University of Colorado at Boulder, Boulder, CO, Jan. 2006.

[30] S. D. R. S. D. Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. L. D. Souza. Mutation testing applied to Estelle specifications. *Software Quality Control*, 8(4):285–301, 1999.

[31] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

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

[33] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. 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, Nov. 2005.

[34] 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.