Evaluating Test Suites and Adequacy Criteria Using Simulation-Based Models of Distributed Systems

Matthew J. Rutherford, *Member*, *IEEE Computer Society*, Antonio Carzaniga, and Alexander L. Wolf, *Member*, *IEEE Computer Society*

Abstract—Test adequacy criteria provide the engineer with guidance on how to populate test suites. While adequacy criteria have long been a focus of research, existing testing methods do not address many of the fundamental characteristics of distributed systems, such as distribution topology, communication failure, and timing. Furthermore, they do not provide the engineer with a means to evaluate the relative effectiveness of different criteria nor the relative effectiveness of adequate test suites satisfying a given criterion. This paper makes three contributions to the development and use of test adequacy criteria for distributed systems: 1) a testing method based on discrete-event simulations, 2) a fault-based analysis technique for evaluating test suites and adequacy criteria, and 3) a series of case studies that validate the method and technique. The testing method uses a discrete-event simulation as an operational specification of a system in which the behavioral effects of distribution are explicitly represented. Adequacy criteria and test cases are then defined in terms of this simulation-based specification. The fault-based analysis involves mutation of the simulation-based specification to provide a foil against which test suites and the criteria that formed them can be evaluated. Three distributed systems were used to validate the method and technique, including the Domain Name System (DNS).

Index Terms—Distributed systems, discrete-event simulation, test adequacy criteria, fault-based analysis.

1 Introduction

Consider entry #851 in the Squid Web cache bug database:[1]

> *DNS retransmits too often. The DNS retransmissions does* [sic.] *not decay by time as documented but is* [sic.] *always at their shortest interval.*

Fig. 1 presents a high-level view of the components involved in this scenario. As a Web proxy, Squid sits between a browser program and a server program. The browser sends HTTP requests to the Squid program, which sends them on to the proper Web server if the response has not previously been cached. Integration with the Domain Name System (DNS) is crucial for the operation of Squid; it can be configured to use a built-in DNS resolver or to access DNS through an external program. When using the internal resolver, Squid is supposed to double the interval between repeat DNS requests, but this bug report indicates that the time between them erroneously remains constant.

1. http://www.squid-cache.org/bugs/show_bug.cgi?id=851.

- *M.J. Rutherford is with the Department of Computer Science, University of Denver, 2260 S. Gaylord St., Denver, CO 80208. E-mail: mjr@cs.du.edu.*
- *A. Carzaniga is with the Faculty of Informatics, University of Lugano, Via Giuseppe Buffi 13, CH-6904 Lugano, Switzerland. E-mail: antonio.carzaniga@unisi.ch.*
- *A.L. Wolf is with the Department of Computing, 180 Queen's Gate, Imperial College London, London, SW7 2AZ, UK. E-mail: a.wolf@imperial.ac.uk.*

Manuscript received 3 June 2007; accepted 14 Dec. 2007; published online 15 May 2008.
Recommended for acceptance by M. Young and P. Devanbu.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2007-06-0177.
Digital Object Identifier no. 10.1109/TSE.2008.33.

Why was this failure not caught during testing? More specifically, what testing method would have led the engineer to create and run a test case that could reveal this error? Consider the sequence of steps needed to reproduce the Squid failure:

1. Configure the scenario:
   - The browser is configured to use a Squid instance.
   - Squid is configured to use its internal DNS resolver.
2. Cause the browser to issue an HTTP request that requires a DNS lookup.
3. Cause the DNS request packet to be dropped or ignored.
4. Wait long enough to observe that the time between DNS requests does not increase.

This bug and the steps needed to reproduce it demonstrate why distributed systems are so difficult to test: There are multiple interacting components (the browser, the server, and Squid), each individually configured and then combined into an overall system topology (the browser making requests to the server through Squid); there are interactions with external systems (DNS); the network must be controlled (packets dropped); and the behavior is time dependent (the retransmission interval must be measured). Aside from being difficult, distributed systems can be especially costly to test, if for no other reason than the significantly greater space of behaviors they exhibit compared to other kinds of systems.

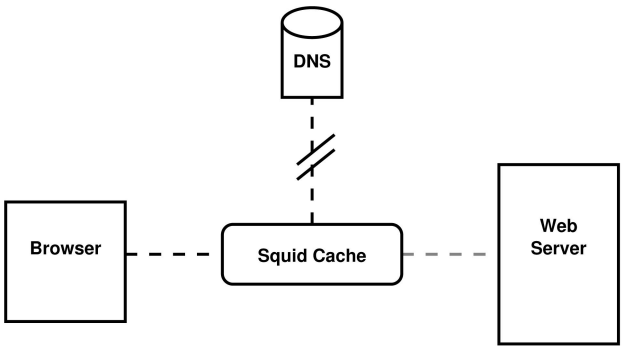

Fig. 1. Squid bug #851 configuration.

Our aim is to develop a rigorous testing method for distributed systems. As a first step, this paper looks at *test 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 goal. In particular, test suites (i.e., prescribed sets of test cases) are constructed with respect to such criteria. To date, adequacy criteria have been studied in the context of sequential (e.g., Frankl and Weyuker [20]) and concurrent (e.g., Carver and Tai [6]) systems, but not in the context of distributed systems.

Historically, the development of adequacy criteria has been an evolutionary process: A new criterion of some sort is proposed and then evaluated against prior criteria to understand its relative effectiveness at causing fault-revealing failures in an implementation. Our contribution in this work is not to propose specific criteria, but rather to provide a framework in which criteria can be used and evaluated. The most basic capability, of course, is to determine whether a given test suite is adequate with respect to a given criterion. But, it is well known (e.g., from the work of Frankl and Weiss [18]) that otherwise adequate test suites constructed using the same criterion might exhibit relative differences in their effectiveness. Moreover, the relative effectiveness of test adequacy criteria themselves are not universal. Instead, they are dependent upon the peculiarities of the system under test. In other words, for system S_1, criterion C_1 might be better at guiding the selection of effective test suites than criterion C_2, but might be worse than C_2 for system S_2.

This uncertainty places the engineer at risk. One strategy would be for the engineer to execute multiple adequate test suites corresponding to one or more adequacy criteria. But this ignores the cost involved, which, as we point out above, is particularly great for distributed systems. What the engineer needs, therefore, is a means to *predict the effectiveness* of candidate test suites and the adequacy criteria from which they are formed *without having to execute the test suites* on the system.

Test adequacy criteria are usually defined with respect to a model of the system under test. Common models in use today include some form of flow or dependence graph derived from the implementation or an abstract specification of behavior, such as a finite-state machine (FSM). To capture the critical characteristics of distributed systems, we require a richer model, one that can be used to specify not just the basic behavior of the system, but its behavior in the context of an operational environment. We propose *discrete-event simulation* for this purpose.[2] Specifically, we propose using simulation as the model of both the intended behavior of a distributed system and the ways in which the operational environment affects that behavior. In the case of Squid, for example, the engineer would develop a simulation of the three main functional components and the network that connects them; inclusion of the network allows the engineer to explicitly model such things as packet drops and time delays.

The intuition that led us to a simulation-based model arises from the following observations:

- Simulations are already commonly used to understand and evaluate the functionality and performance of complex distributed systems. For example, they are used to understand network protocols [1], tune distributed systems [39], [55], and improve distributed algorithms [9]. They can abstract away irrelevant details of the implementation of a distributed system, as well as irrelevant details of the operational environment, yet still provide a faithful model of the expected behavior of the system in its environment. They are appealing to engineers 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.
- Simulations embody abstractions for the underlying mechanisms and environmental conditions that affect the distribution properties of systems. In addition to modeling 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, failures, and bandwidths.
- Recent simulation frameworks encourage simulations to be written in one or another common imperative programming language, such as C++ or Java.[3] 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.

Using our method, adequacy criteria are defined in terms of the simulation-based model. 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 FSM specifications in protocol testing [4], [38], where adequacy is established by measuring the extent to which a test suite

2. Below, when we use the term "simulation," we are referring exclusively to "discrete-event simulation."

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

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 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. We demonstrate below that a test suite with a higher level of simulation-code coverage, under a valid adequacy criterion, will have a greater effectiveness at causing fault-revealing failures in the implementation.

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, as mentioned above, 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 a given system. The result is a second contribution of this paper in which we demonstrate a method for predicting 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 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 [18]. 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. [5] in which different testing strategies are emulated once the failure data for each test case has been collected.[4]

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 engineer 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 establish simulation-based test adequacy criteria. In discussing this, we take the perspective of the engineer of a distributed system and consider several ways in which 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. Typically, events represent the arrival of a network message at one of the processes. 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 which 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

4. Briand et al. use the term "simulation" in their paper to describe the method of evaluating different techniques; here, we use "emulation" to avoid confusion with our use of the term "simulation."

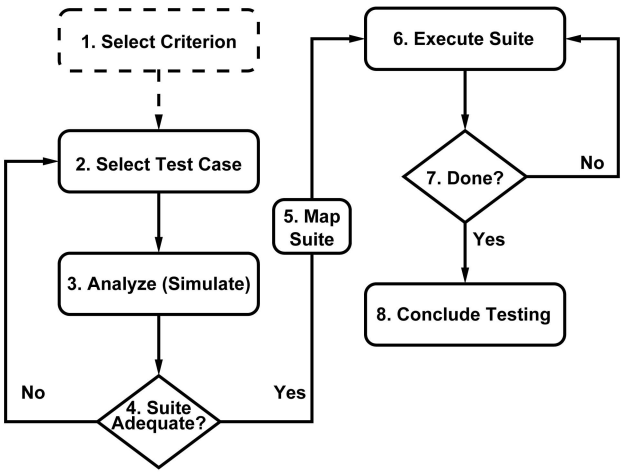

Fig. 2. The specification-based testing process.

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 fail to propagate events with a certain probability. A given simulation 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, both 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

Fig. 2 depicts a simple and generic specification-based testing process. As a first step, the engineer selects a particular adequacy criterion to organize the rest of the process; we defer discussion of this difficult decision. For now, assume that a criterion is being used and the engineer must select a test suite that will satisfy it. A test suite is composed of test cases, each one consisting of an input vector which includes direct inputs to the system, representing functional parameters, as well as inputs to the environment, representing environmental conditions. In the figure, actions 2 through 4 are repeated until the engineer determines that the suite is adequate. Once a suite has been selected, the engineer uses it to test the implementation by first mapping input vectors into the implementation domain (step 5) and then executing each test case on the implementation (step 6). The testing activity is completed by reviewing the test results (step 8).

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 (step 3). The engineer does this by running simulations, using the parameters given by the test cases in the suite, and collecting coverage data on the simulation code. The failure-inducing test case associated with Squid bug #851, for example, would involve the appropriate DNS configuration, browser request, packet drops, and time delays. Ideally, this test case would be found in the test suite. Note, however, that the process by which individual test cases are created or generated is outside the scope of this paper. Similarly, we do not propose or discuss any specific strategy by which the engineer 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 nonenvironmental processes (`Client` and `Server` in the example above) or a system-specific criterion might require that each event type be scheduled at least once during a simulation run. Once a criterion is defined, the engineer 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 engineer 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 engineer to risk. First, there is little empirical or analytical data that an engineer 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 *intercriterion risk*. Second, there is often significant variability in the effectiveness of test suites adequate with respect to a particular criterion, exposing the engineer to the risk of selecting a less-effective, adequate test suite; we refer to this as *intracriterion risk*.

Therefore, the second idea is to provide the engineer with a general 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 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 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* [10], 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 thus are well suited to the code-mutation operators developed in the context of mutation testing [10]. 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 nondeterministic. 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 using 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 and select a single test suite that is adequate with respect to it. In this scenario, the engineer is exposed to both types of risk, intercriterion and intracriterion, 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 engineer has somehow chosen a particular adequacy criterion, as before, but here they want to reduce the risk of picking an adequate but less-effective test suite. Thus, they select multiple adequate test suites, use fault-based analysis to relate the suites by mutant score, and apply the highest scoring 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, intracriterion risk is reduced.

Ranking. In this scenario, the engineer is looking to rationally decide between multiple criteria. So, the engineer 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 intercriterion risk. At this point, the engineer 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 engineer 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 [57]. 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 well-known distributed algorithms and systems. The first system, GBN, is the "go-back-n" transport protocol, which provides reliable data transfer over an unreliable communication channel. The second, LSR, is a "link-state routing" scheme in which each router broadcasts its adjacent connections to every other router in the network. In this way, all routers assemble the same global view of the network and can then use Dijkstra's algorithm to compute routes to each destination. The third, and most significant, is a recursive, caching DNS resolver.

For each system, we created simulations and experimented with available implementations. To create the simulations, we followed the specifications given in the Kurose and Ross networking textbook [37] as well as the relevant Internet RFCs (the mechanism by which standard Internet protocols are published).

Implementation of the first two systems was 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 as source descriptions. For the DNS resolver, we used historical releases of MaraDNS,[5] 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.[6] In the course of this work, we developed a thin layer over the discrete-event core that

5. http://maradns.org.
6. http://www.dcs.ed.ac.uk/home/hase/simjava/.

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. The sender and receiver maintain a sequence number that they use to mark and check data packets and acknowledgments so as to ensure that all packets are received in the proper order. In addition to that, the sender 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 noncomment, nonblank 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* is the number of bytes to be transferred, sampled uniformly in [1, 335,760].
2. *DropProb* is the probability of a packet being dropped by the network, distributed uniformly in [0.0, 0.2).
3. *DupProb* is the probability of a nondropped packet being duplicated, distributed uniformly in [0.0, 0.2).

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.

Experimental universe. For GBN, the universe of test cases consists of 2,500 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 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 [41]. This implementation uses the basic sockets facilities provided by the `java.net` package and consists of 24 classes containing 565 lines of code.

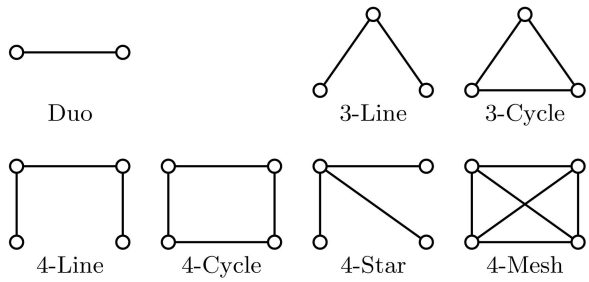

Fig. 3. LSR topologies.

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

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.

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

1. *Topology* is an integer in the range [1, 7] representing a particular arrangement of routers and costs (Fig. 3 shows each possibility graphically).
2. *MessageCount* is 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.

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, *MessageCount* instances of the `Client` process are created and scheduled for execution at regular intervals. Clients send a message, consisting of a short text string, from a source router to a destination router, each chosen randomly from the range $[1, n]$ (local messages are also allowed). As each message is propagated by a router, a per-message path-cost variable is updated with the costs of the links traversed by the message. When a router receives

a message 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 messages, and that the path cost for each is correct.

Experimental universe. The universe of test cases for LSR consists of 7,000 parameter tuples, 1,000 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 a reliable network interaction. For our experiments, we reimplemented the framework to run over an 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 components of the Internet. It provides 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 two and five 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 nondropped 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.

Experimental universe. The universe for DNS consists of 2,000 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. 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*[7] as a stub resolver implementation and the *tinydns* server included with version 1.0.5 of *djbdns*[8] 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

To validate our ideas, we follow the experimental method introduced by Frankl and Weiss [18]. At a high level, this involves sampling a large universe of test cases to randomly construct test suites that are adequate with respect to different criteria. Statistical inference is then used to test hypotheses about the relative fault-detecting ability of competing criteria and their relative cost. We evaluate the different criteria by employing a technique described by Briand et al. [5] in which different testing strategies are emulated once the failure data for the universe of test cases has been collected.

We conducted three separate experiments, aimed at evaluating the usage scenarios described in Section 2.3. In

7. http://www.dnsjava.org/.
8. http://cr.yp.to/djbdns.html.

the first usage scenario, code-based or environmental adequacy criteria are instantiated against a simulation to select an adequate test suite. There is an almost infinite variety of specialized adequacy criteria that could be defined against a simulation, but we believe the typical engineer interested in this *conventional* scenario would be most interested in applying a simple, general-purpose simulation-code-coverage criterion. Thus, in Experiment 1, we compare the cost and effectiveness of three white-box adequacy criteria to randomly selected test suites of varying size.

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

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

4.1 Experimental Method

We primarily compare adequacy criteria by measuring the effectiveness of adequate test suites. The effectiveness E of a test suite, measured with respect to a set of implementations, is the proportion of implementations found to be faulty by the test suite; a test suite that fails on 6 out of 10 subject implementations has $E = 0.6$. Others define and measure effectiveness on a per-implementation basis as the proportion of adequate test suites that fail [18]. 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.

For cost C, we use total execution time, including both analysis time (c_a) and test execution time (c_e). As we assume the existence of a suitable simulation, we do not consider the cost of developing this. In the context of different testing processes, a specification-based test suite may be executed multiple times (e.g., by different engineers during QA, as regression tests, or as part of a regularly scheduled automated test strategy). To account for this, we introduce a process factor F, which represents the number of times a selected test suite is executed. Clearly, since the analysis cost is amortized over F test executions, even a very expensive analysis technique may become cost-effective with high-enough values of F. In the cost-effectiveness analysis presented below, we use a value of $F = 10$, which we believe is quite conservative for a system-level, specification-based testing technique. The overall cost for a particular test suite is given by

$$C = c_a + F \times c_e. \quad (1)$$

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

Of course, random and input-partitioning criteria have $c_a = 0$ since they do not require any analysis. For white-box criteria, c_a consists of the total execution time of the simulations of all test cases considered plus the time needed to compute aggregated coverage for each of the intermediate suites examined. This is the time spent to assemble an adequate suite in the process of Fig. 2. For example, suppose we consider five test cases, but discard the second and third because they do not improve coverage. The analysis cost is the time needed to run five simulations plus the time taken to determine coverage of the suites assembled at each iteration, which correspond to the combinations of tests {1, 2}, {1, 3}, {1, 4}, and {1, 4, 5}. Note that this puts white-box criteria at a disadvantage since it assumes that the engineer has no intuition about the relation between input values and coverage, which is not usually the case in practice.

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

$$H_0 : \bar{E}_A \leq \bar{E}_B$$

and an alternative hypothesis

$$H_a : \bar{E}_A > \bar{E}_B,$$

where $\bar{E}_A$ and $\bar{E}_B$ are the average effectiveness values for criteria A and B, respectively.

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 [12]. Therefore, we compute the z value for this hypothesis,

$$z = \frac{\bar{E}_A - \bar{E}_B}{\sigma_{E_A}/\sqrt{n}},$$

and use the *p-value* formula for high-tailed hypothesis tests (i.e., when the rejection region consists of high z values),

$$p = 1 - \Phi(z),$$

where 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 criterion A is more effective than criterion B.

TABLE 1
Implementation Failure Rates: (a) GBN, (b) LSR, (c) DNS

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

(a)

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

(b)

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

(c)

4.1.1 Preparation

Following the terminology of Briand et al. [5], we first *prepare* our data by executing and analyzing all test cases individually. This entails the management of three different executions: 1) simulations, 2) implementations, and 3) mutants.

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

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

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

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

As the goal of these experiments is to differentiate between testing techniques, we eliminate implementations with high failure rates since a test suite of any reasonable size will always detect failures, regardless of the technique used to select its cases. We do this by initially executing 100 randomly selected test cases against each implementation and eliminating from further consideration those with failure rates higher than 20 percent. The same approach is used by others in empirical studies [16] to focus attention on faults that are hard to detect by eliminating implementations that would be found by virtually any test suite. After executing all test cases, we also excluded correct implementations (i.e., those without failures) from further analysis.

Table 1 lists the 28 remaining faulty, nonpathological implementations along with the percentage of test cases that fail on each. The GBN and LSR implementations with names beginning with "stud" are student implementations. For GBN, the two beginning with "mut" are mutants of our own Java implementation.[9] For DNS, the names correspond to the version numbers of the releases. The absolute failure rates are determined by the nature of the generated test cases, the automated oracles, and the quality of the software being tested. For DNS, the test cases explore aspects of the DNS RFCs that were not addressed fully in the MaraDNS implementations and, therefore, cause a high degree of failures.

Mutants. Mutants are generated from the simulation code by applying all conventional mutation operators provided by the MuJava tool [41]. We target classes encoding the core algorithms and logic, but exclude classes used as messages in the system. When executing the simulation runs, we use Java's class loader path to ensure that the compiled mutants are loaded instead of the original, correct version.

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

TABLE 2
Generated Mutants

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

Table 2 contains the details of mutants generated for each system. The second column, labeled N_M, is the total number of mutants generated by MuJava. The third and fourth columns, labeled N_e and N_p, are the number of equivalent and pathological mutants. The fifth column, N_u, contains the number of mutants for each system used

9. We wanted to avoid using implementation mutants as subjects since we use the same operators to mutate simulation code (although the code is entirely different). However, as a last resort, we generated mutants of our GBN implementation after determining that there were not enough student implementations with appropriate failure rates.

TABLE 3
Adequacy Targets

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

during analysis. We attribute the comparatively large N_M for GBN to the amount of integer arithmetic used in this algorithm. The most striking aspect of these data is the number of pathological mutants we were able to eliminate. For GBN and LSR, these mutants account for 70 percent of the total; for DNS, 42 percent of the mutants were pathological. By eliminating these early in the execution process, we were able to realize significant savings in terms of execution time. Interestingly, there were no equivalent mutants for DNS. We confirmed this by examining the differences between each mutant and the original and determined that all mutants were in fact semantically different.

4.1.2 Testing Techniques

The white-box criteria used in our experiments are the well-known *all-blocks*, *all-branches*, and *all-uses* coverages [20] 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.

During simulation execution, coverage data are collected and stored for each test case. In order to determine target values for each criterion (100 percent was typically not possible due to infeasible paths), we created a test suite out of 50 test cases and verified that any uncovered elements are truly infeasible. We did this iteratively during the generation of test cases to ensure that our random generator was not missing subtle cases. Table 3 contains the target adequacy values used in the experiments.

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

1. Initialize coverage value to zero,
2. randomly select a candidate test case from the universe.
3. Determine the aggregate coverage value for the suite, including the previously selected members plus the candidate.
4. If the computed coverage value improves with the addition of the candidate, add it to the suite, otherwise discard the candidate and try again.
5. Stop when the coverage value meets the target value.

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

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

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

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

TABLE 4
White-Box Effectiveness and Size

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

4.2 Experiment 1: Effectiveness

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

Table 4 contains the initial data obtained from analyzing adequate suites for each criterion. In Table 4, the second column contains the average size of the adequate suites, measured in test cases, and the third column contains the average effectiveness of the same suites. The differences in effectiveness shown in these tables are all statistically significant, except for the difference between *all-branches* and *all-uses* for DNS. Therefore, for GBN, the criteria ranking is the canonical ranking one would expect from the literature:

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

For LSR, the ranking is

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

Finally, for DNS, the ranking is

$$all\text{-}blocks < all\text{-}branches,$$
$$all\text{-}uses < all\text{-}branches.$$

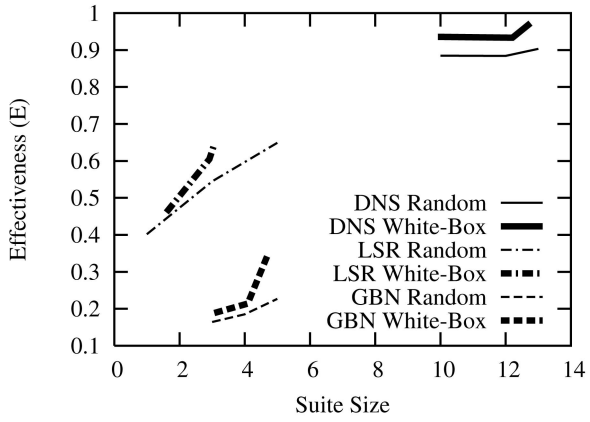

Fig. 4. Absolute white-box effectiveness.

We believe that *all-uses* is not significantly more effective than *all-branches* for LSR and DNS because the simulations of these two systems make extensive use of collections such as lists and maps to store data, resulting in an imprecise data-flow analysis. These inconsistent relationships are not due to a failing in specification-based testing, but rather are fundamental to the use of adequacy criteria. Experimental results reported by others support this observation: *all-uses* is shown to be better than *all-branches* on average, but not universally [18].

To evaluate the absolute effectiveness of the white-box techniques, we generate random suites with sizes corresponding to the average sizes of white-box simulation-adequate suites. Fig. 4 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 simulation-adequate suites compared to random suites of the same size is statistically significant.

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

Table 5 contains the results of this analysis for GBN. In this table, the columns represent random suites of the labeled size. The rows represent either the cost, labeled C, or effectiveness, labeled E for each criterion. The cells contain either "$-$," "$=$," or "$+$," which are interpreted generally as "worse than," "equivalent to," and "better than," respectively. So, for cost rows, cells having a "$-$" symbol indicate that the white-box technique is worse (i.e., more expensive) than the random suite of that size. Similarly, in effectiveness rows, cells marked with a "$+$" symbol indicate that the white-box technique is significantly better than random suites. Finally, cells marked with "$=$" symbols indicate that there is no statistically significant relationship between the two criteria, in either direction.

TABLE 5
GBN White-Box Cost Effectiveness

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

In Table 5, there are some general trends that help understand what is being displayed. First, each of the white-box criteria is worse in terms of cost than small random suites. Recall from Section 4.1 that white-box techniques have an additional cost component due to analysis C_a. This means that, all else being equal, white-box techniques do worse in terms of cost than random suites of the same size. Moving to the right in the cost rows, there is eventually a point at which the white-box technique breaks even or is better than random suites. Intuitively, this happens when the random suites become sufficiently large that their execution cost overcomes the analysis cost incurred by the white-box technique.

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

The data shown in Fig. 4 already establish that white-box techniques are inherently more effective than random suites of the same size. Here, we are interested in determining if the cost of analysis needed to obtain the white-box suite is so large that it would be more cost-effective to simply use randomly selected suites. In other words, we are trying to determine if the engineer would have a more effective test suite had they simply used the analysis time expended in selecting white-box suites to instead execute more tests.

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

The same data are shown for LSR in Table 6. These data show that *all-blocks* has equivalent cost effectiveness, *all-branches* is more cost-effective than random, and *all-uses* is less cost-effective. Again, we believe that *all-uses* is not effective for LSR because of the extensive use of collections.

TABLE 6
LSR White-Box Cost Effectiveness

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

TABLE 7
DNS White-Box Cost Effectiveness

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

Similarly, the cost-effectiveness data for DNS are listed in Table 7. Here, *all-blocks* and *all-branches* are significantly more cost-effective and *all-uses* is more cost-effective. This is interesting because, in terms of absolute effectiveness, *all-blocks* and *all-uses* are statistically indistinguishable, but, when the analysis cost is considered, *all-blocks* is shown to be better.

4.3 Experiment 2: Boosting Criteria

Experiment 2 is aimed at determining if the fault-based analysis technique can be used to improve the effectiveness of white-box and black-box criteria. This experiment corresponds to the *boosting* usage scenario in which an engineer selects more than one adequate suite during analysis and then uses the mutant score to decide which one to use on the implementation. There are two costs associated with this usage; the first comes from having to select multiple adequate test suites and the second is the cost of performing the fault-based analysis by executing each test case on all specification mutants. Thus, the main goal of this experiment is to determine the smallest number of additional suites at which the engineer sees a statistically significant improvement in effectiveness.

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

The black-box criteria are as follows:

- **GBN IP-1**: This criterion simply divides value ranges of each parameter into four equally sized partitions. A suite is adequate for this criterion when each partition is represented by at least one test case value.
- **GBN IP-2**: This criterion divides the NumBytes value range into seven equally sized partitions. For DropProb and DupProb, the value range [0, 0.1) is split in half, while the range [0.1, 0.2) is split into four bins, thereby placing an emphasis on higher drop and duplication rates. Again, a test suite is adequate for this criterion when each partition is accounted for. The underlying intuition here is that higher drop and duplication rates make a test exercise more of the retransmission and window manipulation code.
- **LSR pair-50**: This criterion ensures that paths between pairs of routers are well represented by a test suite. A test suite is adequate for this criterion if all topologies are represented and 50 percent of the possible router pairs are accounted for.
- **LSR quartet**: This criterion simply requires that all 4-router topologies in Fig. 3 be represented. The intuition behind this is that 4-router topologies are more complicated and, therefore, will make better test scenarios.
- **DNS all-resource-types**: This criterion requires that a query be made for each of the core DNS resource types. Intuitively, this exercises the breadth of the user inputs.
- **Typical usage**: This criterion attempts to exercise the DNS features that are used most often. Thus, it requires a valid query, a name-error query, and a no-data query to be issued for A records (which map names to addresses) and MX records (which map domain names to mail exchanges). These two record types are those most often used and, so, this criterion targets this narrow usage pattern.

For this experiment, we use the 200 adequate white-box suites that were created for Experiment 1 and create 200 new suites for each of the black-box techniques. To underscore the inherent variability in the effectiveness of adequate test suites, we show the probability distributions for each of the criteria in Figs. 5, 6, and 7. Each of these plots has effectiveness on the x-axis and probability on the y-axis. So, the height of each bar in the plots represents the probability that an adequate test suite has the corresponding effectiveness. A vertical dashed line shows the average effectiveness. For instance, consider the plot shown in Fig. 5a, the distribution for *all-blocks* for GBN. The bar at 0.25 on the x-axis has a height of 0.25, indicating that, for this criterion, there is a 25 percent chance that an adequate suite will have an effectiveness score of 0.25. Since the average effectiveness of *all-blocks* for GBN is only 0.19, selecting a suite by chance that actually had 0.25 effectiveness would result in a significant benefit for the engineer. This is exactly the kind of improvement that we are trying to achieve by using fault-based analysis to predict where candidate test suites fall in the effectiveness distribution.

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

1. Choose M suites at random from the set of suites adequate for the criterion in question.
2. Select the suite with highest mutant score and determine its effectiveness.

We perform this procedure 100 times each with values of M starting at 2 and stopping at a value at which the fault-based

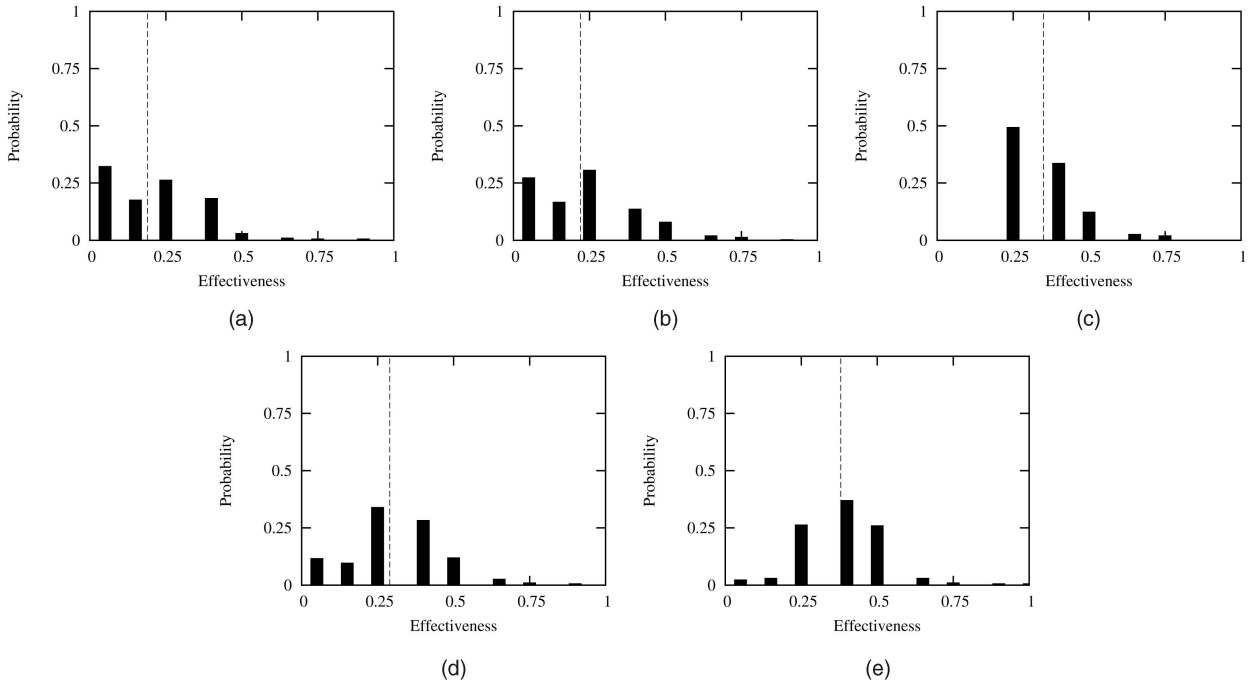

Fig. 5. GBN criteria effectiveness distributions. (a) All-blocks. (b) All-branches. (c) All-uses. (d) IP-1. (e) IP-2.

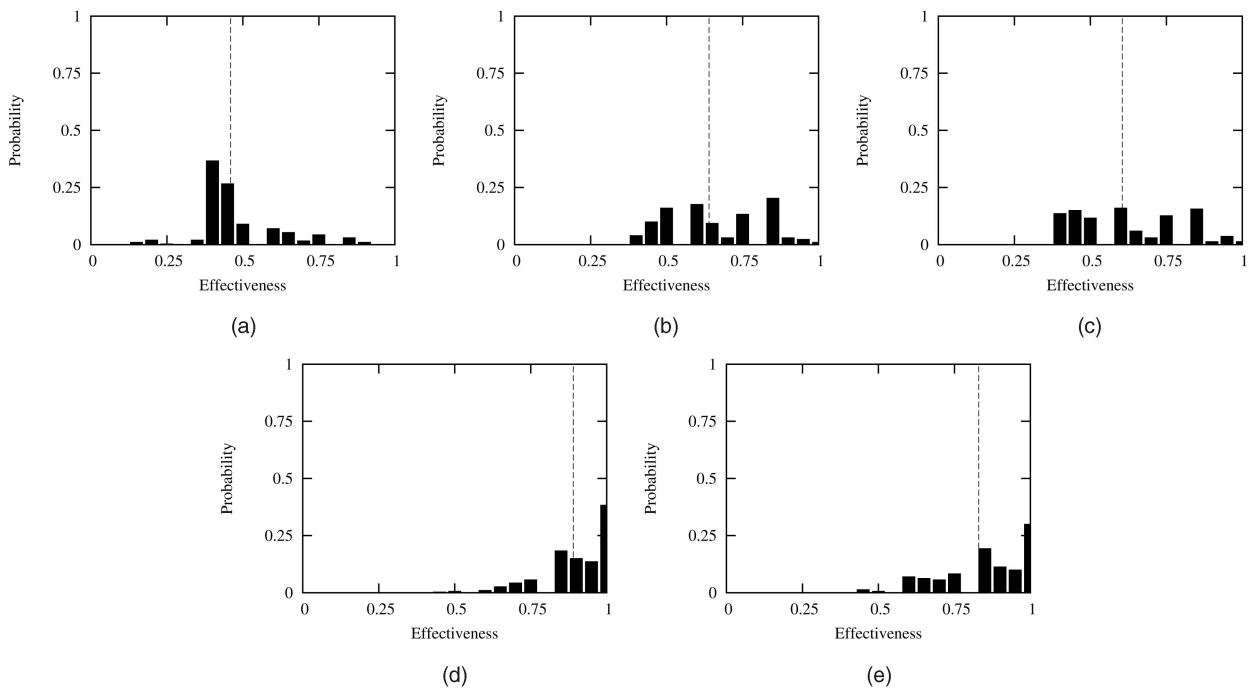

Fig. 6. LSR criteria effectiveness distributions. (a) All-blocks. (b) All-branches. (c) All-uses. (d) IP-1. (e) IP-2.

technique is statistically better than the baseline effectiveness of the criterion (with $\alpha = 0.05$).

Tables 8, 9, and 10 show the results of these experiments. In these tables, the second column contains the number of adequate suites needed to achieve a significant boost in the effectiveness, the third column contains the baseline effectiveness, and the fourth column the boosted effectiveness. The *min* M values reported in the second column of these tables are the total number of adequate suites needed, including the suite needed for the conventional usage scenario.

For GBN, shown in Table 8, three of the five criteria needed only one additional adequate suite to improve the effectiveness and one, *IP-2*, needed two additional suites. For *all-uses*, we examined up to seven additional adequate suites without achieving a significant boost in effectiveness.

The data shown in Tables 9 and 10 for LSR and DNS are very similar to this. For many of the criteria, the minimum multiplier M is 3 or less, meaning that the engineer first sees significant improvement in effectiveness with only two additional adequate suites. For DNS, *all-blocks* is harder to boost than the others, requiring seven additional test suites

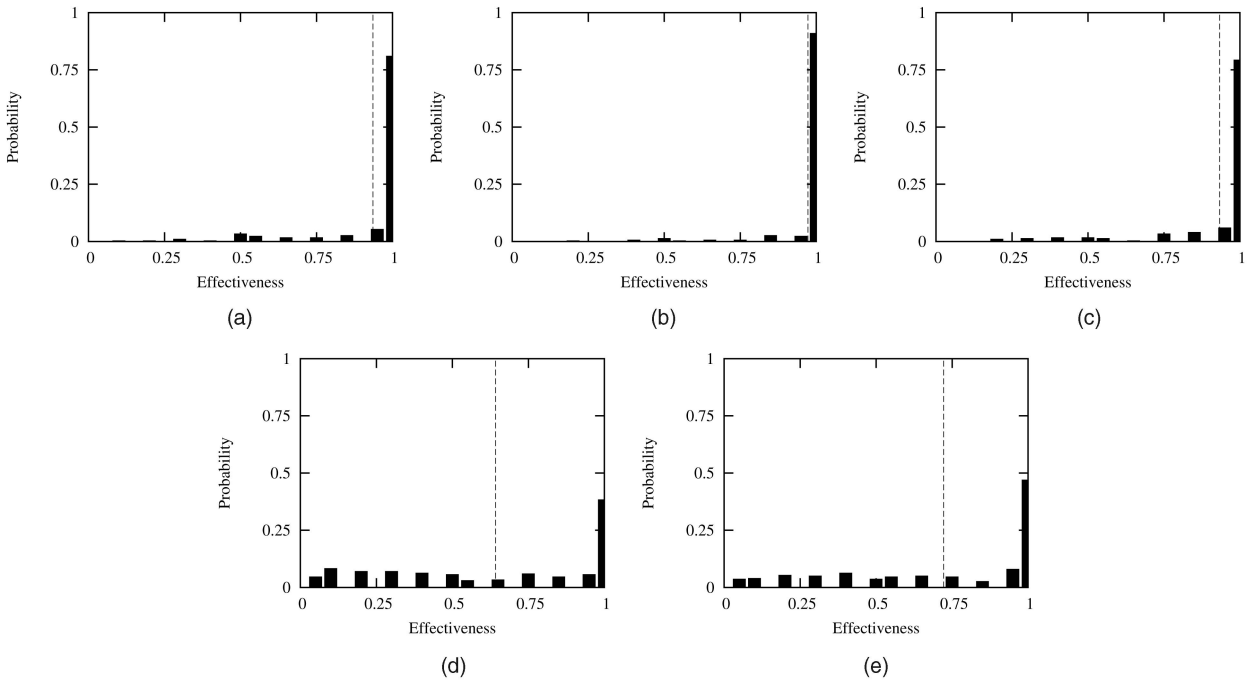

Fig. 7. DNS criteria effectiveness distributions. (a) All-blocks. (b) All-branches. (c) All-uses. (d) Resources. (e) TypicalUsage.

before a significant improvement is achieved. Also, for DNS, we were unable to boost *all-branches*, even trying up to seven additional suites. For DNS, *all-branches* is the most effective criterion we evaluated, with an effectiveness of 0.98. Referring to the distribution of adequate suites shown in Fig. 7b, it is understandable that this criterion would be difficult to boost since virtually all adequate test suites are in the rightmost bin.

TABLE 8
Boosting GBN Criteria

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

TABLE 9
Boosting LSR Criteria

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

TABLE 10
Boosting DNS Criteria

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

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

An important aspect of this result is that the technique works for both black-box and white-box adequacy criteria. If an engineer prefers to not 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.

4.4 Experiment 3: Ranking Criteria

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

From the previous two experiments, we have, for each system, 200 adequate suites for each of five different criteria. Fig. 8 depicts all statistically significant relationships with $\alpha = 0.05$. For example, Fig. 8a indicates that *all-uses* has a significantly higher effectiveness than *IP-1*, which is itself higher than *all-branches* and *all-blocks*, etc. This figure underscores the difficulty practitioners face when choosing between criteria: There is no rational way, aside from our *ranking* technique, for an engineer to know where *IP-1* and *IP-2* fall with respect to the white-box criteria.

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

Our experiments show that the fault-based analysis predicts the actual relationships almost perfectly. Since the predicted relationship graphs are virtually identical to those shown in Fig. 8, we do not report them here. Instead,

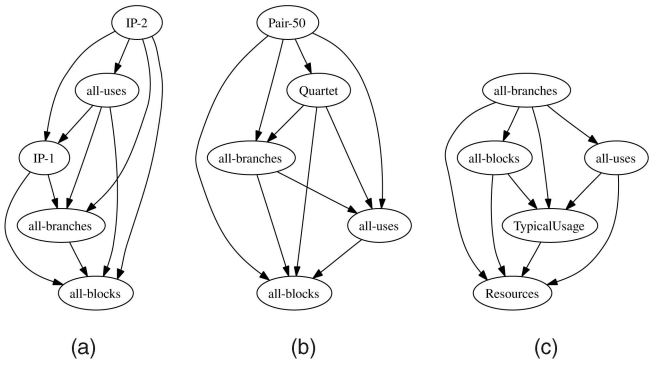

Fig. 8. Criteria relationships. (a) GBN. (b) LSR. (c) DNS.

Tables 11, 12, and 13 show the computed *p-values* for the relationships. For instance, each row in Table 11 corresponds to an edge in Fig. 8a.

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

For DNS, the relationship between these two criteria represents the only situation in which the predicted relationship does not match the actual relationship; this is highlighted in the bottom row of Table 13. In this situation, the fault-based analysis predicts, with a strong level of confidence, that *all-uses* will be more effective than *all-blocks*. However, this relationship is not borne out by the effectiveness data from the implementations. So, in statistical terms, the fault-based technique would have caused the engineer to favor a criterion that was not actually more effective than another one considered. However, we should point out that, in real terms, the average effectiveness of the two criteria are practically identical, as shown in Table 4.

TABLE 11
GBN Criteria *p-Values*

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 12
LSR 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 13
DNS Criteria *p-Values*

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

4.5 Summary of Results

The results of the three experiments presented above are compelling and provide strong validation for the ideas we present in this paper.

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

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

Finally, we show through Experiment 3 that, for the collection of adequacy criteria used here, we can predict the actual effectiveness relationships very accurately through analysis and execution of the simulations.

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

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, 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. Also, our cost analysis does not take

into consideration the effort required to create test cases and map them from the simulation domain to the implementation domain. We were unable to measure this directly since our test cases were automatically generated. The test-case development costs are highly dependent on both the system being developed and the development processes and environment in place. Suffice to say that, as with all testing techniques, the decision to use our technique must be made in the context of a comparison of the entire breadth of costs that apply to its usage and to the potential cost of software failure.

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

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

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. Following this, we summarize the prior empirical studies of test adequacy criteria. Finally, we describe other testing techniques targeted specifically at distributed systems.

5.1 Specification-Based Testing

The work of Richardson et al. [51] 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 [48], 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 engineers 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 et al. [7], [8] propose a function-level, assertion-based language to guide testing. Offutt and Liu [45] describe the generation of test data from a higher level, object-oriented specification notation. Offutt et al. [46] describe the use of generic state-based specifications (e.g., UML and statecharts) for system-level testing. Harder et al. [29] 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 FSM specification. Bochmann and Petrenko [4] and Lai [38] 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 [49]. A number of researchers have studied the use of software architectures for testing. Richardson and Wolf [53] propose several architecture-based adequacy criteria based on the Chemical Abstract Machine model. Rice and Seidman [50] describe the ASDL architecture language and its toolset and discuss its use in guiding integration testing. Jin and Offutt [34] define five general architecture-based testing criteria and apply them to the Wright ADL. Muccini et al. [42] describe a comprehensive study of software architectures for implementation testing. They propose a technique that relies on Labeled Transition System (LTS) specifications of dynamic behavior. Their method derives simpler, abstract LTSs (ALTSs) from a 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 [10]. In mutation testing, the engineer applies mutation operators [44] 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 nonequivalent mutants.

Mutation testing is based on two complementary theories [10]. 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 [41]. 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 [54] and statecharts [15]). 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 be checked 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 [52], [58]. Recently, a fault-class hierarchy has been determined analytically and used to explain some of the earlier experimental results [36], [56]. We are targeting a more expressive specification method whose fault classes (mutation operators) are not amenable to a general analytical comparison.

5.3 Empirical Studies

The comparison of competing testing strategies has been the focus of research for decades. Theoretical studies using simulation-based [14], [28], [43] and analytical [21], [22], [30], [59] methods have been conducted. Definitive results for general relationships among testing strategies are not generally available and the focus of more recent work has been empirical evaluations.

An experimentation and analysis method was presented by Frankl and Weiss [17], [18]. This method relies on the generation of a large universe of test cases from which many adequate suites can be selected by the criteria being evaluated. For each criterion, the proportion of fault-detecting suites is computed and statistical inferences are used to test hypotheses about the relative effectiveness of criteria. Frankl, Weiss, and others have used this method to compare *all-uses* and *all-edges* [16], [17], [18] and *all-uses* and *mutation* testing [19]. Hutchins et al. [33] used a similar empirical framework to compare *all-edges* and *all-DUs*. Recently, Briand et al. [5] extended the basic experimental approach by introducing a step in which different testing strategies are emulated after the initial data are collected. In this study, we employ the Frankl and Weiss experimental method with the test strategy emulation introduced by Briand et al.

Andrews et al. [3] have also studied the use of automatically generated mutants as subjects in empirical studies. They used as a basis some of the canonical examples from the testing literature for which there were known naturally occurring faults. They hand-seeded faults and automatically generated mutants. The goal of their study was to determine which method of introducing faults is more representative of naturally occurring faults: hand seeding or mutation. They concluded that using mutation operators results in subject systems that are more representative since hand-seeded faults are often more subtle and harder to uncover than natural faults. This is certainly a useful tool for researchers since it provides the ability to automatically generate a large number of potentially faulty implementations.

5.4 Testing Distributed Systems

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

Several authors describe their experience and highlight the issues related to testing distributed systems [13], [25], [40]. In particular, Ghosh and Mathur [25] list a number of differentiating characteristics of distributed software systems that are representative of those mentioned by other authors: large scale, heterogeneity, difficult monitoring and control, nondeterminism, concurrency, scalability, performance, and partial failure.

Tool support for monitoring and controlling distributed tests has received considerable attention. Some tools primarily support functional testing [23], [24], [31], [32]; others target performance testing [11], [27]. While this is important work, it addresses only the accidental issues associated with distributed system testing and does not address the more fundamental questions having to do with what and how best to test a system.

Several studies concentrate on distributed component-based systems. Gosh and Mathur [26] present an adequacy criterion targeted at covering CORBA component interfaces. Krishnamurthy and Sivilotti [35] also specifically target CORBA components and present a method for specifying and testing progress properties of components. Williams and Probert [60], [61] apply techniques of pairwise interaction testing to component-based systems. By its nature, work targeting distributed component-based systems is restricted to a limited level of distribution since much of the complexity of developing these systems is handled by the container infrastructure.

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 engineers of distributed systems. Our approach is validated by an initial empirical study of three distributed systems.

One disappointing aspect of existing software testing research is the lack of penetration of the advanced techniques proposed by researchers into industry [47]. We believe that the development of techniques like our fault-based analysis method that allow engineers to predict the relative efficacy of testing procedures for their specific system will help improve the adoption of advanced testing techniques by reducing the risk of using them.

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

This research was supported in part by the US National Science Foundation, US Army Research Office, and European Commission under agreement numbers ANI-0240412, DAAD19-01-1-0484, and IST-026955. A previous version of this paper appeared in the *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, Oregon, November 2006.

References

[1] M. Allman and A. Falk, "On the Effective Evaluation of TCP," *SIGCOMM Computer Comm. Rev.,* vol. 29, no. 5, pp. 59-70, 1999.

[2] P.E. Ammann and P.E. Black, "A Specification-Based Coverage Metric to Evaluate Test Sets," *Int'l J. Reliability, Quality and Safety Eng.,* vol. 8, no. 4, pp. 275-299, 2001.

[3] J. Andrews, L. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proc. 27th Int'l Conf. Software Eng.,* pp. 402-411, May 2005.

[4] G. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 109-124, 1994.

[5] L.C. Briand, Y. Labiche, and Y. Wang, "Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts," *Proc. 26th Int'l Conf. Software Eng.,* pp. 86-95, 2004.

[6] R.H. Carver and K.-C. Tai, "Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs," *IEEE Trans. Software Eng.,* vol. 24, no. 6, pp. 471-490, June 1998.

[7] J. Chang and D.J. Richardson, "Structural Specification-Based Testing: Automated Support and Experimental Evaluation," *Proc. Seventh European Software Eng. Conf./Seventh ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 285-302, 1999.

[8] J. Chang, D.J. Richardson, and S. Sankar, "Structural Specification-Based Testing with ADL," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 62-70, 1996.

[9] I. Clarke, S.G. Miller, T.W. Hong, O. Sandberg, and B. Wiley, "Protecting Free Expression Online with Freenet," *IEEE Internet Computing,* vol. 6, no. 1, pp. 40-49, Jan./Feb. 2002.

[10] R.A. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, no. 4, pp. 34-41, Apr. 1978.

[11] G. Denaro, A. Polini, and W. Emmerich, "Early Performance Testing of Distributed Software Applications," *Proc. Fourth Int'l Workshop Software and Performance),* pp. 94-103, 2004.

[12] J.L. Devore, *Probability and Statistics for Engineering and the Sciences.* Brooks/Cole, 1995.

[13] E.J. Dowling, "Testing Distributed Ada Programs," *Proc. Conf. Tri-Ada '89,* pp. 517-527, 1989.

[14] J.W. Duran and S. Ntafos, "A Report on Random Testing," *Proc. Fifth Int'l Conf. Software Eng.,* pp. 179-183, 1981.

[15] S.C.P.F. Fabbri, J.C. Maldonado, T. Sugeta, and P.C. Masiero, "Mutation Testing Applied to Validate Specifications Based on Statecharts," *Proc. 10th Int'l Symp. Software Reliability Eng.,* pp. 210-221, 1999.

[16] P. Frankl and O. Iakounenko, "Further Empirical Studies of Test Effectiveness," *Proc. Sixth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 153-162, 1998.

[17] P. Frankl and S. Weiss, "An Experimental Comparison of the Effectiveness of the All-Uses and All-Edges Adequacy Criteria," *Proc. Symp. Testing, Analysis, and Verification,* pp. 154-164, 1991.

[18] P. Frankl and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Trans. Software Eng.,* vol. 19, no. 8, pp. 774-787, Aug. 1993.

[19] P. Frankl, S. Weiss, and C. Hu, "All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness," *J. Systems and Software,* vol. 38, no. 3, pp. 235-253, 1997.

[20] P. Frankl and E. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Eng.,* vol. 14, no. 10, pp. 1483-1498, Oct. 1988.

[21] P. Frankl and E. Weyuker, "An Analytical Comparison of the Fault-Detecting Ability of Data Flow Testing Techniques," *Proc. 15th Int'l Conf. Software Eng.,* pp. 415-424, 1993.

[22] P. Frankl and E. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods," *IEEE Trans. Software Eng.,* vol. 19, no. 3, pp. 202-213, Mar. 1993.

[23] S. Ghosh, N. Bawa, G. Craig, and K. Kalgaonkar, "A Test Management and Software Visualization Framework for Heterogeneous Distributed Applications," *Proc. Sixth IEEE Int'l Symp. High-Assurance Systems Eng.,* pp. 106-116, 2001.

[24] S. Ghosh, N. Bawa, S. Goel, and Y.R. Reddy, "Validating Run-Time Interactions in Distributed Java Applications," *Proc. Eighth IEEE Int'l Conf. Eng. Complex Computer Systems,* pp. 7-16, 2002.

[25] S. Ghosh and A.P. Mathur, "Issues in Testing Distributed Component-Based Systems," *Proc. First Int'l ICSE Workshop Testing Distributed Component Based Systems,* citeseer.ist.psu.edu/ghosh99issues.html, May 1999.

[26] S. Ghosh and A.P. Mathur, "Interface Mutation," *J. Software Testing, Verification, and Reliability,* vol. 11, no. 4, pp. 227-247, Dec. 2001.

[27] J. Grundy, Y. Cai, and A. Liu, "SoftArch/MTE: Generating Distributed System Test-Beds from High-Level Software Architecture Descriptions," *Automated Software Eng.,* vol. 12, no. 1, pp. 5-39, 2005.

[28] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Software Eng.,* vol. 16, no. 12, pp. 1402-1411, Dec. 1990.

[29] M. Harder, J. Mellen, and M.D. Ernst, "Improving Test Suites via Operational Abstraction," *Proc. 25th Int'l Conf. Software Eng.,* pp. 60-71, 2003.

[30] R.M. Hierons, "Comparing Test Sets and Criteria in the Presence of Test Hypotheses and Fault Domains," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 4, pp. 427-448, 2002.

[31] A. Hubbard, C.M. Woodside, and C. Schramm, "DECALS: Distributed Experiment Control and Logging System," *Proc. Conf. Centre for Advanced Studies on Collaborative Research,* p. 32, 1995.

[32] D. Hughes, P. Greenwood, and G. Coulson, "A Framework for Testing Distributed Systems," *Proc. Fourth IEEE Int'l Conf. Peer-to-Peer Computing,* pp. 262-263, 2004.

[33] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng.,* pp. 191-200, 1994.

[34] Z. Jin and J. Offutt, "Deriving Tests from Software Architectures," *Proc. 12th Int'l Symp. Software Reliability Eng.*, pp. 308-313, Nov. 2001.
[35] P. Krishnamurthy and P.A.G. Sivilotti, "The Specification and Testing of Quantified Progress Properties in Distributed Systems," *Proc. 23rd Int'l Conf. Software Eng.*, pp. 201-210, 2001.
[36] D.R. Kuhn, "Fault Classes and Error Detection Capability of Specification-Based Testing," *ACM Trans. Software Eng. and Methodology*, vol. 8, no. 4, pp. 411-424, 1999.
[37] J.F. Kurose and K.W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet.* Pearson Benjamin Cummings, 2004.
[38] R. Lai, "A Survey of Communication Protocol Testing," *J. Systems and Software*, vol. 62, no. 1, pp. 21-46, 2002.
[39] C. Liu and P. Cao, "Maintaining Strong Cache Consistency in the World-Wide Web," *Proc. 17th Int'l Conf. Distributed Computing Systems*, pp. 12-21, 1997.
[40] B. Long and P. Strooper, "A Case Study in Testing Distributed Systems," *Proc. Third Int'l Symp. Distributed Objects and Applications*, pp. 20-29, 2001.
[41] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: An Automated Class Mutation System," *J. Software Testing, Verification, and Reliability*, vol. 15, no. 2, pp. 97-133, 2005.
[42] H. Muccini, A. Bertolino, and P. Inverardi, "Using Software Architecture for Code Testing," *IEEE Trans. Software Eng.*, vol. 30, no. 3, pp. 160-171, Mar. 2004.
[43] S. Ntafos, "On Random and Partition Testing," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis*, pp. 42-48, 1998.
[44] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, pp. 99-118, 1996.
[45] A.J. Offutt and S. Liu, "Generating Test Data from SOFL Specifications," *J. Systems and Software*, vol. 49, no. 1, pp. 49-62, 1999.
[46] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating Test Data from State-Based Specifications," *J. Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25-53, Mar. 2003.
[47] L. Osterweil, "Strategic Directions in Software Quality," *ACM Computing Surveys*, vol. 28, no. 4, pp. 738-750, 1996.
[48] T.J. Ostrand and M.J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Comm. ACM*, vol. 31, no. 6, pp. 676-686, 1988.
[49] D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Eng. Notes*, vol. 17, no. 4, pp. 40-52, 1992.
[50] M.D. Rice and S.B. Seidman, "An Approach to Architectural Analysis and Testing," *Proc. Third Int'l Software Architecture Workshop*, pp. 121-123, 1998.
[51] D. Richardson, O. O'Malley, and C. Tittle, "Approaches to Specification-Based Testing," *Proc. ACM SIGSOFT '89: Third Symp. Software Testing, Analysis, and Verification*, pp. 86-96, 1989.
[52] D. Richardson and M. Thompson, "An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection," *IEEE Trans. Software Eng.*, vol. 19, no. 6, pp. 533-553, June 1993.
[53] D.J. Richardson and A.L. Wolf, "Software Testing at the Architectural Level," *Proc. Joint Second Int'l Software Architecture Workshop and Int'l Workshop Multiple Perspectives in Software Development*, pp. 68-71, 1996.
[54] 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*, vol. 8, no. 4, pp. 285-301, 1999.
[55] 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 Trans. Networking*, vol. 11, no. 1, pp. 17-32, 2003.
[56] T. Tsuchiya and T. Kikuno, "On Fault Classes and Error Detection Capability of Specification-Based Testing," *ACM Trans. Software Eng. and Methodology*, vol. 11, no. 1, pp. 58-62, 2002.
[57] Y. Wang, M.J. Rutherford, A. Carzaniga, and A.L. Wolf, "Automating Experimentation on Distributed Testbeds," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 164-173, Nov. 2005.
[58] E. Weyuker, T. Goradia, and A. Singh, "Automatically Generating Test Data from a Boolean Specification," *IEEE Trans. Software Eng.*, vol. 20, no. 5, pp. 353-363, May 1994.
[59] E.J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Trans. Software Eng.*, vol. 17, no. 7, pp. 703-711, July 1991.
[60] A.W. Williams and R.L. Probert, "A Practical Strategy for Testing Pair-Wise Coverage of Network Interfaces," *Proc. Seventh Int'l Symp. Software Reliability Eng.*, pp. 246-254, 1996.
[61] A.W. Williams and R.L. Probert, "A Measure for Component Interaction Test Coverage," *Proc. ACS/IEEE Int'l Conf. Computer Systems and Applications*, pp. 304-311, 2001.

Matthew J. Rutherford received the BSE degree in civil engineering from Princeton University in 1996 and the MS and PhD degrees in computer science from the University of Colorado, Boulder, in 2001 and 2006, respectively. He is an assistant professor in the Department of Computer Science at the University of Denver, Colorado. He is also the founder and the president of Praxes Inc., a software research and development company based in Denver. He was previously a postdoctoral researcher at the University of Colorado and the chief engineer at Chronos Software Inc. He has published in the areas of software testing, configuration management, model-driven development, testing and validation of distributed systems, and content-based networking. He is a member of the IEEE Computer Society.

Antonio Carzaniga received the Laurea degree in electronic engineering and the PhD degree in computer science from the Politecnico di Milano, Italy, in 1994 and 1999, respectively. He is an assistant professor on the Faculty of Informatics at the University of Lugano, Switzerland. From 2002 to 2007, he was an assistant research professor in the Department of Computer Science at the University of Colorado, Boulder. He has conducted research and published in the areas of software process, mobile code, distributed configuration management, software deployment, testing and validation of distributed systems, distributed publish/subscribe middleware, and networking. His primary research interests include the design and realization of advanced communication systems (e.g., content-based networking) with particular emphasis on both the algorithmic aspects of routing protocols, and the engineering methods that make those systems and protocols usable, scalable, robust, and secure.

Alexander L. Wolf received the BA degree in geology and computer science from Queens College of the City University of New York and the MS and PhD degrees from the University of Massachusetts, Amherst. He is a professor in the Department of Computing at Imperial College London. He was previously a professor at the University of Colorado, Boulder and the University of Lugano, Switzerland, and a member of the technical staff at AT&T Bell Laboratories, Murray Hill, New Jersey. His research interests are directed toward the discovery of principles and development of technologies to support the engineering of large, complex software systems. He has published in the areas of software engineering, distributed systems, networking, security, and database management. He is a member of the ACM Council, the governing authority of the 65,000-member professional association, and serves on the editorial board of the *IEEE Transactions on Software Engineering*. He previously served as a vice chair and then the chair of the ACM Special Interest Group in Software Engineering (SIGSOFT) and on the editorial boards of the ACM journal *Transactions on Software Engineering and Methodology* (TOSEM) and the Wiley journal *Software Process—Improvement and Practice* (SPIP). He has chaired or cochaired a number of international program committees, including the International Conference on Software Engineering (ICSE) in 2000. He is a fellow of the ACM, a member of the IEEE Computer Society, and a holder of a UK Royal Society-Wolfson Research Merit Award.