

# Testing: A Roadmap

Mary Jean Harrold  
College of Computing  
Georgia Institute of Technology  
801 Atlantic Drive  
Atlanta, GA 30332-0280  
harrold@cc.gatech.edu

## ABSTRACT

Testing is an important process that is performed to support quality assurance. Testing activities support quality assurance by gathering information about the nature of the software being studied. These activities consist of designing test cases, executing the software with those test cases, and examining the results produced by those executions. Studies indicate that more than fifty percent of the cost of software development is devoted to testing, with the percentage for testing critical software being even higher. As software becomes more pervasive and is used more often to perform critical tasks, it will be required to be of higher quality. Unless we can find ways to perform more effective testing for less cost, the percentage of development costs devoted to testing will increase significantly. This report briefly assesses the state of the art in software testing, outlines some future directions in software testing, and gives some pointers to software testing resources.

## 1 INTRODUCTION

A report by the Workshop on Strategic Directions in Software Quality posits that software quality will become the dominate success criterion in the software industry [30]. If this occurs, the practitioner's use of processes that support software quality assurance will become increasingly important. One process that is performed to support quality assurance is *testing*. Testing activities support quality assurance by executing the software being studied to gather information about the nature of that software. The software is executed with input data, or *test cases*, and the output data is observed. The output data produced by the execution of the program with a particular test case provides a specification of the actual program behavior [30]. Studies indicate that testing consumes more than 50% of the cost of software development. This percentage is even higher for critical software, such as that used for avion-

ics systems. As software becomes more pervasive and is used more often to perform critical tasks, it will be required to be of higher quality. Unless we can find ways to perform more effective testing for less cost, the percentage of development costs devoted to testing will increase significantly.

Because testing requires the execution of the software, it is often referred to as *dynamic analysis*. Forms of verification that do not require execution of the software are referred to as *static analyses*. As a form of verification, testing has several advantages over static techniques. One advantage of testing is the relative ease with which many of the testing activities can be performed. Test-case requirements can be generated from various forms of the software, such as its implementation. Often, these test-requirements can be generated automatically. Software can be instrumented so that when it is executed, it reports information about the executions with the test cases. This information can be used to measure how well the test cases satisfy the test-case requirements. Output from the executions can be compared with expected results to identify those test cases on which the software failed. A second advantage of testing is that the software being developed can be executed in its expected environment. The results of these executions with the test cases provide confidence that the software will perform as intended. A third advantage of testing is that much of the process can be automated. With this automation, the test cases can be used for retesting as the software evolves.

Although, as a form of verification, testing has a number of advantages, it also has a number of limitations. Testing cannot show the absence of faults — it can show only their presence. Additionally, testing cannot show that the software has certain qualities. Moreover, test execution results for specific test cases cannot usually be generalized. Despite these limitations, testing is widely used in practice to provide confidence in the quality of software. However, the emergence of new technologies, such as component-based systems and product families, and the increased emphasis on software quality, highlight the need for improved testing methodologies.

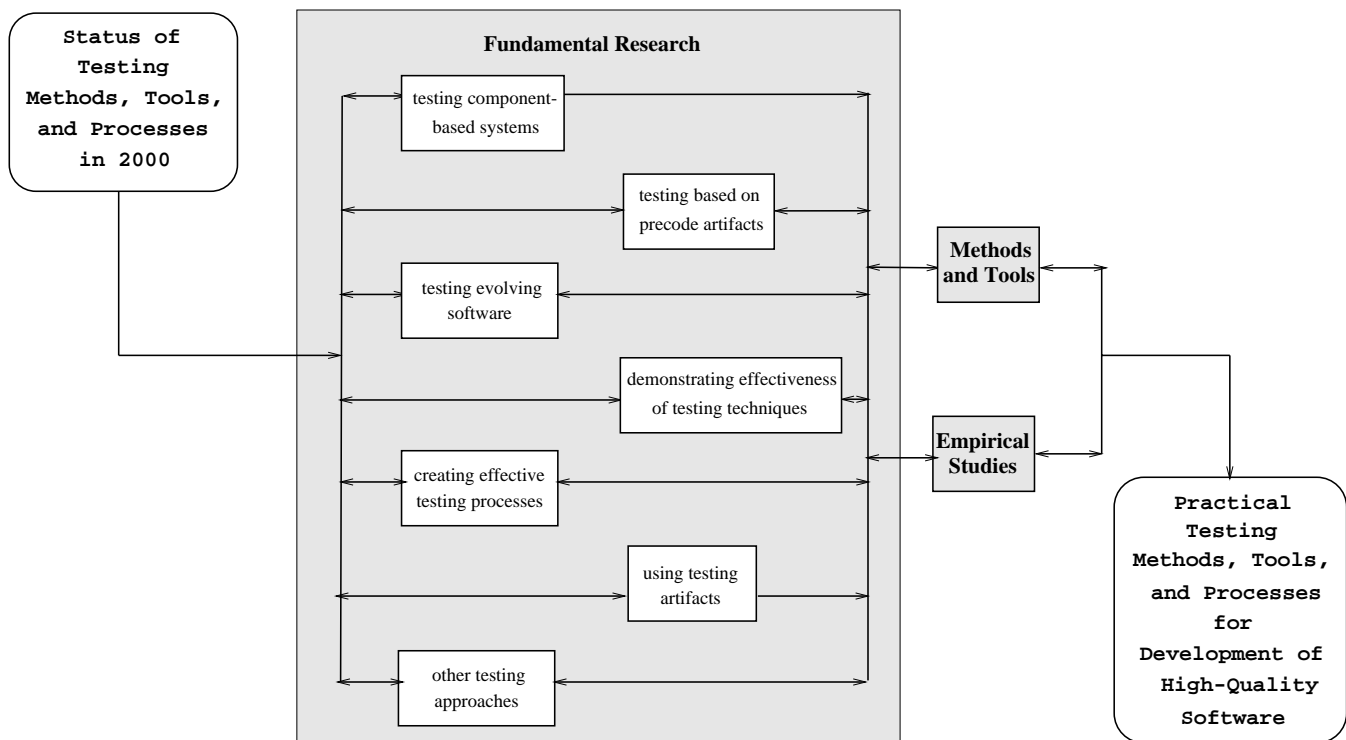


Figure 1: Software Testing Roadmap.

This report focuses on testing techniques. *Reasoning and Analysis: A Roadmap* by Jackson, another report in this series, provides additional information about static-analysis techniques. Instead of presenting a comprehensive overview of the state of the art or state of the practice in software testing, the report presents information about the current state only for those approaches that are encountered on the road to our destination of providing practical testing methods, tools, and processes that will help software engineers develop high-quality software. The next section outlines some of future directions in testing. The report closes by giving summary remarks and providing pointers to resources for more information.

## 2 ROADMAP FOR THE FUTURE

Testing is one of the oldest forms of verification. Thus, there are numerous testing techniques that have been developed and used by developers to help them increase their confidence that the software has various qualities. The ultimate goal of software testing is to help engineers construct systems with high quality. Testing is thus used by developers of all types of systems. As technology improves, it has become possible to apply testing techniques to larger systems. However, widespread use of systematic testing techniques is not common in industry. For example, although a number of code-based testing techniques have been developed for unit testing, studies show that even the weakest forms of

these techniques are not being employed by many practitioners. For another example, although retesting after software modifications (regression testing) can be automated, studies show that many practitioners still perform this task manually.

Figure 1 shows a roadmap for testing that leads to the destination: providing practical testing methods, tools, and processes that can help software engineers develop high-quality software. Progress toward this destination requires fundamental research, creation of new methods and tools, and performance of empirical studies to facilitate transfer of the technology to industry. As the arrows in the figure show, areas may be revisited on the way to the destination. For example, after performing empirical studies using a prototype tool that implements algorithms for testing component-based software, both the research and the method and tool development may be revisited.

### Fundamental Research

Research in many areas of testing has provided advances that hold promise for helping us reach the goal of providing practical tools that can help software engineers develop high-quality software. Additional work, however, needs to be done in a number of related areas, as illustrated in Figure 1. For example, in providing techniques for testing evolving software, we may incorporate techniques for architecture-based testing or techniques

that combine static analysis with testing.

### *Testing Component-Based Systems*

The increased size and complexity of software systems has led to the current focus on developing distributed applications that are constructed from component-based systems. A component-based system is composed primarily of components: modules that encapsulate both data and functionality and are configurable through parameters at run-time [25]. Given the increasing incidence of component-based systems, we require efficient, effective ways to test these systems.

The issues that arise in the testing of component-based systems can be viewed from two perspectives: the component-provider perspective and the component-user perspective. The component-provider perspective addresses testing issues that are of interest to the provider of software components. The component provider views the components independently of the context in which the components are used. The provider must therefore, effectively test all configurations of the components in a context-independent manner. The component-user perspective, in contrast, addresses testing issues that concern the user of software components. The component user (or application developer) views the components as context-dependent units because the component user's application provides the context in which the components are used. The component user is thus concerned with only those configurations or aspects of the behavior of the components that are relevant to the component user's application.

An important issue for the component user concerns the problems resulting from using commercial off-the-shelf software (COTS). Although there are no regulations imposed on developers of COTS, many critical applications are requiring the use of these systems to standardize development and reduce costs [26]. Another factor that distinguishes the issues that are pertinent in the two perspectives is the availability of the source code of the components: the component providers have access to the source code, whereas the component users typically do not.

Researchers have extended existing testing techniques for use by component providers. For example, Doong and Frankl describe techniques based on algebraic specifications [13], Murphy et al. describe their experiences with cluster and class testing [28], and Kung et al. present techniques based on object states [22]. Other researchers have extended code-based approaches for use by component providers for testing individual components. For example, Harrold and Rothermel present a method for using data-flow testing that computes definition-use pairs for use in class testing [20]. Buy et al. present a similar approach that uses symbolic

evaluation to generate sequences of method calls that will execute the definition-use pairs [6].

Researchers have considered ways that component users can test systems that are constructed from components. Rosenblum proposed a theory for test-adequacy of component-based software [38]. His work extends Weyuker's set of axioms that formalize the notion of test adequacy [45], and provides a way to test the component from each subdomain in the program that uses it. Devanbu and Stubblebine present an approach that uses cryptographic techniques to help component users verify coverage of components without requiring the component developer to disclose intellectual property [12].

With additional research in these areas, we can expect techniques and tools that will help component users test their applications more efficiently and effectively. We need to understand and develop effective techniques for testing various aspects of the components, including security, dependability, and safety; these qualities are especially important given the explosion of web-based systems. These techniques can provide information about the results of the testing that will increase the confidence of developers who use the components in their applications.

We need to identify the types of testing information about a component that a developer needs for testing applications that use the component. For example, a developer may want to measure coverage of the parts of the component that her application uses. To do this, the component would need to be able to react to inputs provided by the application, and record the coverage provided by those inputs. For another example, a component user may want to test only the integration of the component with her application. To do this, the component user would need to be able to identify couplings between her application and the component.

We need to develop techniques for representing and computing the types of testing information that a component user needs. Existing component standards, such as COM and JavaBeans, supply information about a component that is packaged with the component. Likewise, standards for representing testing information about a component, along with efficient techniques for computing and storing this information, could be developed. For example, coverage information for use in code-based testing or coupling information for use in integration testing could be stored with the component or techniques for generating the information could be developed by the component provider and made accessible through the component interface.

Finally, we need to develop techniques that use the information provided with the component for testing the application. These techniques will enable the compo-

nent user to effectively and efficiently test her application with the component.

#### *Testing Based On Precode Artifacts*

Testing techniques can be based on precode artifacts, such as design, requirements, and architecture specifications. In the past, many of these techniques have been based on informal specifications. However, more formal approaches are being used for these specifications. Techniques that use these precode specifications for tasks such as test-case planning and development can help improve the overall testing process. This section discusses the use of software architectures for testing.

The increased size and complexity of software systems has also led to the emergence of the discipline of software architecture. Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns [43]. Software architecture styles define families of systems in terms of patterns of structural organization. Given the increasing size and complexity of software systems, techniques are needed to evaluate the qualities of systems early in their development. Through its abstractions, software architecture provides a promising way to manage large systems.

The emerging formal notations for software architecture specification can provide a basis on which effective techniques for testing and analysis can be developed. Recently, researchers have begun to investigate ways to use these formal architectural specifications as a basis on which to develop architectural-based analysis and testing techniques. For example, Eickelmann and Richardson consider the ways in which architectural specification can be used to assess the testability of a software system [14]; Bertolino et al. consider the ways in which the architectural specification can be used in integration and unit testing [5]; Harrold presents some approaches for using software architecture specification for effective regression testing [17]; and Richardson, Stafford, and Wolf present a comprehensive architecture-based approach to testing that includes architecture-based coverage criteria, architectural testability, and architecture slicing [35]. These architecture-based testing techniques and tools can facilitate dynamic analysis, and thus, detection of errors, much earlier in the development process than is currently possible.

To expedite research in this area, in 1998, the Italian National Research Council (CNR) and the U. S. National Science Foundation (NSF) sponsored the Workshop on the Role of Software Architecture in Testing and Analysis [36]. This workshop brought together researchers in software architecture, testing, and analysis to discuss research directions. A re-

port on the results of this workshop can be found at <http://www.ics.uci.edu/~djr/rosatea>.

Additional research in this area promises to provide significant savings in software testing. We need to develop techniques that can be used with the architectural specification for test-case development. These techniques can provide test-cases requirements for testing various aspects of the architecture. This approach will let various aspects of the system be tested early in development. These techniques can also provide functional test-case requirements that can be used to develop test cases for use in testing the implementation. These techniques will facilitate the systematic development of test cases early in the development process. Finally, these techniques can provide ways for test cases to be generated automatically. These techniques would enable efficient generation of test cases at an early stage of the software development.

We also need to develop techniques that can be used to evaluate software architectures for their testability. With this information, developers can consider alternative designs and select the one that suits their testability requirements. We can also apply this architecture-based testability information for identification of designs that will facilitate more efficient testing of modified versions of the software or various members of product families. Most of the testing effort is aimed at verification of software that has been modified or of derivative versions of the software. Thus, improved management of retesting will reduce the cost of maintaining the software.

#### *Testing Evolving Software*

Regression testing, which attempts to validate modified software and ensure that no new errors are introduced into previously tested code, is used extensively during software development and maintenance. Regression testing is used to test safety-critical software that must be retested often, to test software that is being developed under constant evolution as the market or technology changes, to test new or modified components of a system, and to test new members in a family of similar products. Despite efforts to reduce its cost, regression testing remain one of the most expensive activities performed during a software system's lifetime: studies indicate that regression testing can account for as much as one-third of the total cost of a software systems [24].

Because regression testing is expensive, but important, researchers have focused on ways to make it more efficient and effective. Research on regression testing spans a wide variety of topics. Chen et al. [7], Ostrand and Weyuker [31], and Rothermel and Harrold [40] developed techniques that, given an existing test suite and information about a previous testing, select a subset of

the test suite for use in testing the modified software.<sup>1</sup> Harrold, Gupta, and Soffa [18] and Wong et al. [46] present techniques to help manage the growth in size of a test suite. Leung and White [24] and Rosenblum and Weyuker [37] presented techniques to assess regression testability. These techniques permit estimation, prior to regression test selection, of the number of tests that will be selected by a method. Other techniques, such as that developed by Stafford, Richardson, and Wolf evaluate the difficulty of regression testing on precode artifacts [44].

Because most software development involves applying modifications to existing software, additional research that provides effective techniques for testing the modified software can significantly reduce software development costs. We need to develop techniques that can be applied to various representations of the software, such as its requirements or architecture, to assist in selective retest of the software. These techniques will let us identify existing test cases that can be used to retest the software. These techniques will also let us identify those parts of the modified software for which new test cases are required.

We also need to develop techniques to assist in managing the test suites that we use to test the software. Effective techniques that can reduce the size of a test suite while still maintaining the desired level of coverage of the code or requirements will help reduce testing costs. Techniques that let us identify test cases that, because of modifications, are no longer needed will also help to reduce the cost of testing. Because the testing may be performed often, there may not be time to run the entire test suite. Thus, we need techniques that will let us prioritize test cases to maximize (or minimize) some aspect of the test cases such as coverage, cost, or running time. These techniques can help testers find faults early in the testing process.

Finally, we need to develop techniques that will let us assess the testability of both software and test suites. Techniques that will let us assess the testability of the software using pre-code artifacts promise to provide the most significant savings. For example, using the software architecture may let us evaluate alternative designs and select those that facilitate efficient retesting of the software. These techniques can be applied to evolving software and product families to help identify the most efficient designs. Techniques that will let us assess the testability of a test suite will also provide savings. For example, a test suite that contains test cases that validate individual requirements may be more efficient for use in regression testing than one in which a single test cases validates many requirements.

<sup>1</sup>Rothermel and Harrold present comprehensive comparison of regression-test selection techniques [39].

### *Demonstrating Effectiveness Of Testing Techniques*

Because testing is one of the oldest forms of verification, numerous testing techniques that have been developed and used to help developers increase their confidence that the software has various qualities. Most of these techniques focus on selection of the test cases. Good-enough and Gerhart suggested how to evaluate criteria for determining adequacy of test suites, and they focused on how to select test cases that inspire confidence. [16]

Since then, many techniques for selection of test cases have been developed. Some testing techniques select test cases that are based on the software's intended behavior without regard to the software's implementation and others guide the selection of test cases that are based on the code.

There have been some studies that demonstrate the effectiveness of certain test-selection criteria in revealing faults. However, there are many areas for additional research. We need to identify classes of faults for which particular criteria are effective. To date, a number of test-selection criteria been developed that target particular types of faults. Several researchers, including Rapps and Weyuker [34] and Laski and Korel [23], developed testing criteria that focus test selection on the data-flow in a program. For critical safety applications, it is estimated that over half of the executable statements involve complex boolean expressions. To test these expressions, Chilenski and Miller developed a criterion, modified condition/decision coverage, that specifically concentrates the testing on these types of statements [8].

Rothermel et al. developed testing techniques based on existing code-based techniques to test form-based based visual programming languages, which include commercial spreadsheets [42]. The method is validation-driven and incremental. Recent studies indicate that, given the interface, users untrained in testing techniques can effectively test their programs.

We need to perform additional research that provides analytical, statistical, or empirical evidence of the effectiveness of the test-selection criteria in revealing faults. We also need to understand the classes of faults for which the criteria are useful. Finally, we need to determine the interaction among the various test-selection criteria and find ways to combine them to perform more effective testing.

Even for test-selection criteria that have been shown to be effective, there may be no efficient technique for providing coverage according to the criteria. For example, although mutation analysis [10] has been shown to be effective adequacy criterion, researchers have yet to find an efficient way to perform the analysis. Given effective

testing criteria, we need to develop ways to perform the testing efficiently. We also need to investigate techniques that approximate complete satisfaction of the adequacy criterion but are still sufficiently effective. For example, consider performing data-flow testing on programs that contain pointer variables. Testing that considers all data-flow relationships involving pointer variables may be too expensive to perform. However, some preliminary results suggest that the test suite obtained without considering these pointer relationships may provide sufficient coverage. Additional research can determine if such approximations of complete coverage suffices for data-flow and other testing criteria.

#### *Establishing Effective Processes For Testing*

An important aspect of testing is the process that we use for planning and implementing it. Beizer (and many other authors) describes a process for testing [4]. These techniques typically consist of construction of a test plan during the requirements gathering phase and implementation of the test plan after the implementation phase. To develop its software, Microsoft Inc uses a different model, which (1) frequently synchronizes what people are doing and (2) periodically stabilizes the product in increments as a project proceeds. These activities are done continually throughout the project. An important part of the model builds and tests a version of the software each night [9]. Richardson et al. advocate the idea of a perpetual testing process.<sup>2</sup> Their perpetual testing project is building the foundation for treating analysis and testing as on-going activities to improve quality. Perpetual testing is necessarily incremental and is performed in response to, or in anticipation of, changes in software artifacts or associated information.

A process for regression testing is implicit in selective regression testing techniques [7, 31, 40, 46]. For these techniques to be employed, testing must be performed on one version of the software, and testing artifacts, such as input-output pairs and coverage information, must be gathered. These artifacts are used by the techniques to select test cases for use in testing the next version of the software. Onoma et al. [29] present an explicit process for regression testing that integrates many key testing techniques into the development and maintenance of evolving software. This process considers all aspects of development and maintenance.

Additional research can validate these existing models. For example, does a nightly build and test, such as that performed by Microsoft, reduce the amount of testing that is required later? For another example, how often do testing artifacts need to be compute for effective regression-test selection? Additional research can also develop new process models for testing and vali-

date these models.

Although testing is important for assessing software qualities, it has several limitations: it can show only the presence of faults, not their absence; it cannot show that the software possesses certain qualities; and the results obtained from the testing often cannot be generalized. A process for developing high-quality software, however, could combine testing with other quality tools. Osterweil et al. [30] suggested that various quality techniques and tools could be integrated to provide value considerably beyond what the separate technologies can provide.

We need to understand the way in which these various testing and analysis techniques are related, and develop process models that incorporate them. A process that combines static analysis techniques with testing has the potential to improve quality and reduce costs.

#### *Using Testing Artifacts*

The process of testing produces many artifacts. Artifacts from the testing include the execution traces of the software's execution with test cases. These execution traces may include information about which statements were executed, which paths in the program was executed, or which values particular variables got during the execution. Artifacts from the testing also include results of the test-case execution, such as whether a test passed or failed. These artifacts can be stored for use in retesting the software after it is modified.

Given the magnitude and complexity of these artifacts, they can also be useful for other testing and software engineering tasks. Researchers have begun to investigate new ways to use these artifacts. Many techniques have been developed that use execution traces. Pan, DeMillo, and Spafford present a technique that uses dynamic program slices,<sup>3</sup> which are derived from execution traces, along with the pass/fail results for the executions, to localize faulty code [32]. They apply a number of heuristics, which consider various combinations of the intersections and unions of the dynamic slices for the subset of the test suite that passed and the subset of the test suite that failed. In empirical studies on small subjects, the results of applying the heuristics helped to localize the faulty code.

Ernst et al. present another technique that uses the execution traces [15]. Their approach uses the execution traces that contain values, at each program point, for each variable under consideration. The goal of their approach is to identify program invariants. After repeated execution of the program with many test cases, the approach provides a list of likely invariants in the program.

<sup>3</sup>A *dynamic program slice* for a program point, a variable, and a test case is the set of all statements in the program that affected (either directly or indirectly) the value of the variable at the program point when the program is run with the test case.

<sup>2</sup>More information can be found at the Perpetual Testing home page: <http://www.ics.uci.edu/~djr/edcs/PerpTest.html>.

Their empirical results show that this approach can be quite successful in identifying these invariants.

Other researchers have developed techniques that use coverage information for software engineering tasks. Rosenblum and Weyuker [37] present a technique that uses coverage information to predict the magnitude of regression testing. Their technique predicts, on average, the percentage of the test suite that must be retested after changes are made to a program. Later work by Harrold et al. provided additional evaluation of the work, and presented an improved model of prediction [19]. Rothmel and Harrold present techniques based on coverage information to select test cases from a test suite for use in regression testing [40]. Several researchers have used testing artifacts for test-suite minimization and prioritization [18, 41, 46]. Ball presented a technique that performs concept analysis on coverage information to compute relationships among executed entities in the program. Comparing these dynamic relationships with their static counterparts can help testers uncover properties of their test suite [2].

Other researchers have provided visualization techniques for testing artifacts. For example, Ball and Eick present a system for visualizing information, including testing information such as coverage, for large programs [3], and Telcordia Technologies has several tools that combine analysis and visualization of testing artifacts to help software maintainers [21].

Although there have been some successes in using testing artifacts for software engineering tasks, this research is in its infancy. Additional research can verify that existing techniques provide useful information for software engineers. For example, we need to determine whether the heuristics developed by Pan et al. help to localize faulty code when there are many faults or interacting faults in a program. These results can provide a starting point for additional research.

Additional research in this area can also provide new techniques that use testing artifacts for software engineering tasks. We need to identify the types of information that software engineers and managers require at various phases of the software's development. We also need techniques that will find important relationships that exist in the software. Techniques such as data mining may help with this task. Given these types of information, we need to develop techniques to present the information in a useful way. Techniques for effective visualization of the testing information can provide effective tools for software engineers.

#### *Other Testing Techniques*

In addition to the areas for fundamental research discussed in the preceding sections, there are many other areas in which techniques could help us reach our des-

tinuation. This section briefly presents a few of them.

Generating test data (inputs for test cases) is often a labor-intensive process. To date, a number of techniques have been presented that generate test data automatically. Most of these techniques, however, are applicable for unit testing, and may not scale to large systems. We need to develop automatic or semi-automatic test-data generation techniques that testers can use for large systems. These data could be generated using precode representations or using the code itself.

Many testing techniques require some type of static analysis information. For example, data-flow analysis is useful for data-flow testing of software units and for integration testing when these units are combined. However, existing techniques for computing precise data-flow information are prohibitively expensive. We need to develop scalable data-flow analysis techniques that can be used to compute the required information.

Current techniques for measuring adequacy for rigorous testing criteria, such as data-flow, require expensive instrumentation. If we expect to use these more rigorous criteria, we need efficient instrumenting and recording techniques. We need to develop efficient techniques for instrumenting and recording coverage.

#### **Methods and Tools**

Ultimately, we want to develop efficient methods and tools that can be used by practitioners to test their software. Pfleeger presented reasons why software engineering technology requires, on average 18 years to be transferred into practice [33]. Researchers must work with industry to reduce this time for technology transfer. She also presented a comprehensive approach to effecting that transfer. One important aspect of this approach for technology transfer is the development of methods and tools that can be used in industrial settings to demonstrate the effectiveness of the techniques we create. We must develop methods and tools that implement the techniques and that can be used to demonstrate their effectiveness.

To accomplish this, an important criterion is that these methods and tools be scalable to large systems. Industrial systems are large and complex and the methods and tools must function on these systems. Scalable tools will provide useful information in an efficient way. Researchers often demonstrate the effectiveness of their techniques using tools that function on contrived or toy systems. Thus, the results of their experimentation with these tools may not scale to large industrial systems. We need to develop robust prototypes, identify the context in which they can function, and use them to perform experiments to demonstrate the techniques.

In developing these tools, we need to consider compu-

tational tradeoffs. For example, we need to consider precision versus efficiency of the computation, and we need to consider storing information versus computing it on demand as needed. Murphy and Notkin [27] and Atkinson and Griswold [1] provide further discussion of some of these tradeoffs.

An efficient approach for development of methods and tools is to provide ways to automatically create them; a similar approach is used to automatically generate compilers. One example of such an approach is the Genoa framework for generating source code analysis tools [11]. Genoa is retargetable to different parsers; parse tree data structures built by such parsers are used in the analysis. This approach could be used to automatically generate specialized testing analysis or tools.

After demonstrating, with the prototype methods and tools, that the techniques can be effective in practice, we must work to develop methods and tools that are attractive to practitioners. The methods and tools should be easy to use and their output should be presented in a clear and understandable way. The methods and tools should also be easy to learn. Finally, as much as possible, testing tools should be automated and require minimal involvement by the software engineers.

### Empirical Studies

Closely associated with the development of methods and tools is the performance of empirical studies. Using the methods and tools, these studies will help to demonstrate the scalability and usefulness of the techniques in practice. These studies will also provide feedback that will help guide fundamental research and tool development. Both the transfer of scalable techniques into practice, and the creation of such techniques, require significant empirical studies.

There is much evidence of the growing emphasis on experimentation. In addition to analytical evaluation of scalability and usefulness of software engineering techniques, more and more papers in proceedings and journals reporting these techniques also report the results of empirical studies that attempt to demonstrate these qualities. Moreover, a new international journal, *Empirical Software Engineering*,<sup>4</sup> provides a forum for reporting on the methods and results of various types of empirical studies along with descriptions of infrastructures for supporting such experimentation. Finally, funding agencies, such as National Science Foundation, are supporting a number of large projects for work in experimental systems.

Efforts to empirically evaluate testing techniques faces a number of obstacles. One obstacle, which was discussed in the preceding section, is the difficulty of acquiring

sufficiently robust implementations of those techniques. A second obstacle to significant experimentation with is the difficulty of obtaining sufficient experimental subjects. The subjects for testing experimentation include both software and test suites. Practitioners are reluctant, however, to release this types of experimental subject.

We need to design controlled experiments to demonstrate our techniques. We also need to collect sets of experimental subjects, and, if possible, make them available to researchers. We also need to perform experimentation with industrial partners. Testing techniques can be implemented in the industry environment, and industrial subjects can be used for experimentation. If these subjects cannot be make available publicly, we may be able to create sanitized information that would reveal no proprietary information but would still be useful for experimentation.

### 3 CONCLUDING REMARKS

Historically, testing has been widely used as a way to help engineers develop high-quality systems. However, pressure to produce higher-quality software at lower cost is increasing. Existing techniques are not sufficient for this purpose. To make progress in providing efficient testing techniques that can help engineers produce higher-quality software, there are a number of challenging areas for research, including fundamental research, development of effective methods and tools, and performance of empirical studies that will help to transfer the technology to industry. This paper has presented an overview of some of those areas.

Other reports in this series, such as *Metrics: A Roadmap* by Fenton, *Reliability and Dependability: A Roadmap* by Littlewood and Strigini, *Software Engineering for Safety: A Roadmap* by Lutz, *Software Engineering for Security: A Roadmap* by Devanbu, and *Reasoning and Analysis: A Roadmap* by Jackson provide additional information about verification. Several recent workshops, including the Workshop on Strategic Directions (June 1996) sponsored by Association of Computing Machinery (ACM), National Science Foundation (NSF), and Computing Research Association (CRA), International Workshop on the Role of Software Architecture in Testing and Analysis (July 1998), sponsored by the Italian National Research Council (CRA) and the U.S. National Science Foundation (NSF), and the International Conference on Software Engineering Workshop on Testing Distributed Component-Based Systems (May 1999), have addressed specific testing issues.

A number of web sites contain links to a wealth of information about testing, including papers, reports, books, conferences, journals, projects, tools, educational re-

<sup>4</sup>More information can be found at the journal home page: [http://kapis.www.wkap.nl/aims\\_scope.htm/1382-3256](http://kapis.www.wkap.nl/aims_scope.htm/1382-3256).



sources, and people. Some examples of these sites are Middle Tennessee State's STORM Software Testing Online Resources at <http://www.mtsu.edu/~storm/>, Reliable Software Technology's Software Assurance Hotlist at <http://www.rstcorp.com/hotlist/>, and Software Research Institute's Software Quality Hotlist at <http://www.soft.com/Institute/HotList/index.html>. Online forums include the net newsgroup comp.software.testing.

#### 4 Acknowledgments

The author is supported by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University and by a grant from Boeing Commercial Airplanes.

#### REFERENCES

- [1] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, March 1996.
- [2] T. Ball. The concept of dynamic analysis. In *Proceedings of the Joint Seventh European Software Engineering Conference (ESEC) and Seventh ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, September 1999.
- [3] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, April 1996.
- [4] B. Beizer. *Black-Box Testing*. John Wiley and Sons, New York, NY, 1995.
- [5] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An approach to integration testing based on architectural descriptions. In *Proceedings of the IEEE ICECCS-97*.
- [6] U. Buy, A. Orso, and Pezzè. Issues in testing distributed component-based systems. In *Proceedings of the First International Workshop on Testing Distributed Component-Based Systems*, May 1999.
- [7] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. Test-Tube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.
- [8] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):191–200.
- [9] M. A. Cusamano and R. Selby. How microsoft builds software. *Communications of the ACM*, 40(6):53–61, June 1997.
- [10] R. A. DeMillo. Test Adequacy and Program Mutation. In *Eleventh*, pages 355–356, Pittsburgh, PA, May 16–18 1989. IEEE Computer Society Press.
- [11] P. Devanbu. GENOA - A customizable, front-end retargetable source code analysis framework. *ACM Transactions on Software Engineering and Methodology*, 9(2), April 1999.
- [12] P. Devanbu and S. Stubblebine. Cryptographic verification of test coverage claims. *IEEE Transactions on Software Engineering*, to appear.
- [13] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.
- [14] N. S. Eickelmann and D. J. Richardson. What makes one software architecture more testable than another? In *Proceedings of the International Software Architecture Symposium*, October 1996.
- [15] M. D. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, May 1999.
- [16] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions of Software Engineering*, pages 156–173, June 1975.
- [17] M. J. Harrold. Architecture-based regression testing of evolving systems. In *International Workshop on the Role of Software Architecture in Testing and Analysis*, July 1998.
- [18] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. on Softw. Eng. and Methodology*, 2(3):270–285, July 1993.
- [19] M. J. Harrold, G. Rosenblum, D. Rothermel, and E. J. Weyuker. Empirical Studies of a Prediction Model for Regression Test Selection. *IEEE Transactions on Software Engineering*, to appear.
- [20] M. J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, December 1994.
- [21] J. R. Horgan. Mining system tests to aid software maintenance. The Telcordia Software Visualization and Analysis Research Team, Telcordia Technologies.

- [22] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On object state testing. In *Proc. of COMPSAC'94*, 1994.
- [23] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–54, May 1983.
- [24] H. K. N. Leung and L. White. Insights Into Regression Testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, October 1989.
- [25] T. Lewis. The next 10,000<sub>2</sub> years, part II. *IEEE Computer*, pages 78–86, May 1996.
- [26] G. McGraw and J. Viega. Why COTS software increases security risks. In *Proceedings of the First International Workshop on Testing Distributed Component-Based Systems*, May 1999.
- [27] G. Murphy and D. Notkin. Lightweight source model extraction. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 116–127, October 1995.
- [28] G. Murphy, P. Townsend, and P. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39–47, 1994.
- [29] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Comm. of the ACM*, 41(5):81–86, May 1988.
- [30] L. J. Osterweil ET AL. Strategic directions in software quality. *ACM Computing Surveys*, (4):738–750, December 1996.
- [31] T.J. Ostrand and E.J. Weyuker. Using dataflow analysis for regression testing. In *Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–247. Lawrence and Craig, September 1988.
- [32] H. Pan, R. DeMillo, and E. H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of COMPSAC '97*, August 1997.
- [33] S. L. Pfleeger. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software*, 47(2–3):111–124, July 1999.
- [34] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.
- [35] D. Richardson, J. Stafford, and A. Wolf. A formal approach to architecture-based testing. Technical report, University of California, Irvine, 1998.
- [36] D. J. Richardson, P. Inverardi, and A. Bertolino. In *Proceedings of the CNR-NSF International Workshop on the Role of Software Architecture in Testing and Analysis*, July 1998.
- [37] D. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):146–156, March 1997.
- [38] D. S. Rosenblum. Adequate testing of component-based software. Technical Report Technical Report UCI-ICS-97-34, August 1997.
- [39] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), August 1996.
- [40] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.
- [41] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault-detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, November 1998.
- [42] G. Rothermel, L. Li, C. DuPuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 20th International Conference on Software Engineering*, pages 198–207, April 1998.
- [43] M. Shaw and D. Garlan. *Software Architecture Perspectives on an Emerging Discipline*. Prentice Hall, New Jersey, 1996.
- [44] J. Stafford, D. J. Richardson, and A. L. Wolf. Chaining: A dependence analysis technique for software architecture. September 1997.
- [45] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(12):1128–1138, December 1986.
- [46] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.