

An Approach To Analyzing and Testing Component-Based Systems

Mary Jean Harrold, Donglin Liang, Saurabh Sinha

Computer and Information Science

Ohio State University

395 Dreese Lab, 2015 Neil Avenue

Columbus, OH 43210-1227

{harrold,dliang,sinha}@cis.ohio-state.edu

ABSTRACT

Software testing and maintenance account for as much as two-thirds of the cost of software production. Program analysis techniques offer the potential to automate testing and maintenance tasks, and thereby reduce the cost of these tasks. An emerging paradigm of software development that promises to enhance software productivity and quality composes software from independently-developed components. The nature of component-based software, however, introduces new problems for applying program analysis techniques to testing and maintenance of such software: the providers of software components develop and test the components independently of the applications that use the components; the users of software components analyze and test their applications without access to source code of the components used in those applications. This paper describes issues and challenges in applying analysis and testing techniques to component-based software and presents an approach to analyzing and testing component-based systems.

Keywords

Component-based system, software components, program analysis, software testing

1 INTRODUCTION

Software testing and maintenance account for as much as two-thirds of the cost of software production [22]. Software tools that use program-analysis techniques promise to automate testing and maintenance tasks, and thereby reduce the cost of these tasks and improve software quality. For example, data-flow testing techniques use data-flow information (e.g., [1, 11, 21]) to evaluate the adequacy of test suites and to assist in test development (e.g., [6, 10]). Data-flow information is also useful for validating properties of concurrent programs (e.g., [4]). Regression testing techniques

use data-flow or control-dependence information to determine the retesting required after changes are made to a program (e.g., [9, 24]). Techniques for debugging, program understanding, and impact analysis use data-flow and control-dependence information to reveal dependencies in software (e.g., [8, 13, 26]).

One class of software on which efficient and effective program analysis techniques and program-analysis-based testing and maintenance tools will be useful is *component-based software*. A component-based system is composed primarily of *components*: modules that encapsulate both data and functionality and are configurable through parameters at run-time [16]. Component-based systems form a significant class of existing software. Given the increasing incidence of component-based systems, it is appropriate to consider techniques for ensuring their quality. We require efficient, effective ways to test and maintain these component-based systems.

To describe our approach to analysis and testing of component-based systems, we consider a component-based system that consists of three parts: the user application, the components, and the infrastructure that provides communication channels between the user application and the components. The user application communicates with the components through the interfaces. The communication infrastructure maps the interfaces used by the user application to the interfaces of the components. A component provides multiple sets of functionalities and each set of the functionalities can be accessed by a *configuration*.

The issues that arise in the analysis and 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 analysis and testing issues that are of interest to the provider of software components. The *component-user perspective*, in contrast, addresses analysis and testing issues that concern the user of software components. The component provider views the components independently of the context in which the components are used. The provider must therefore, ef-

fectively test all configurations of the components in a context-independent manner. The component user, in contrast, 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. 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.

We need to test, maintain, and ensure the quality of components individually, and we need to test, maintain, and ensure the quality of systems that use components. We require analysis techniques that operate efficiently both on components and on systems that are constructed from them. We need tools that let us track the impact of changes in components on systems that use them. We need tools that help us retest systems that use modified components.

In this paper, we describe analysis and testing issues that arise for the component providers and for the component users. We present techniques that fall into two categories: (1) techniques that component providers use to analyze and test the individual components in a context-independent manner and gather information that will assist in analysis and testing of systems that use the components, and (2) techniques that component users require to analyze and test a system that uses these components. Our approach thus separates the analysis and testing of component-based systems (component-user perspective), and the analysis and testing of components (component-provider perspective).

2 ISSUES IN THE ANALYSIS AND TESTING OF COMPONENT-BASED SYSTEMS

In this section, we describe some issues that must be addressed to effectively analyze and test component-based systems. We also discuss issues that arise in the analysis and testing of individual components.

The component-user perspective

Component users develop component-based systems by integrating their applications with independently-developed components. There are many challenges to adapting traditional program analysis and testing techniques for analyzing, testing, and maintaining such systems.

First, the source code of the components is typically not available to the component users. Traditional program analysis techniques, such as alias analysis [15, 18] and control-dependence computation [2, 5, 8], and testing

techniques, such as data-flow testing [6, 10], require access to the source code of the system being analyzed or tested. Because the source code of the components is unavailable, traditional program analysis and testing techniques cannot be applied to component-based systems. One way to perform the analyses without the source code is to make conservative approximations about the analysis relations that hold in the components and about analysis relations that are caused in the user application by the code in the components. Such approximations, however, can cause the analysis results to be too imprecise to be useful. For example, suppose that a component user wants to perform control-dependence analysis on the component-based system. Without access to the source code or information computed by the component provider, to compute safe control-dependence information, the component user must assume that every component raises an exception, and incorporate that information into its analysis.

Second, in a component-based system, even if the source code is available, the components and the user application can be implemented in different languages. An analysis tool that is based on the implementation language of the user application would fail to analyze the components.

Third, a component often provides more functionality than that is used by a particular user application. Without identifying the portion of functionality that is used by a particular user application, a testing tool or an analysis tool can report imprecise results. For example, structural testing criteria [20] measure the adequacy of a test suite by examining the extent to which that test suite covers various structural elements of a program. In measuring the adequacy of a test suite developed for a component-based system, structural elements that comprise unused parts of components embedded in that system must be excluded from consideration. Failure to do so would cause a testing tool to report low coverage for that test suite even if the test suit exhaustively tests the system with respect to the coverage criteria [23].

The component-provider perspective

Component providers develop and test software components independently of the applications that use the components. Unlike the component users, the component providers have access to the source code of the components. Therefore, testing a component is similar to traditional unit testing. However, traditional unit-testing criteria, such as statement and branch testing, may not be sufficient for testing a component because of the weak fault-detection capabilities of those criteria. Fixing a fault that is revealed in a component by the component users would typically entail a much higher cost than fixing a similar fault detected during integra-

tion testing of a non-component-based system because of the potential use of such a component by many user applications.

The component provider must address two issues. First, the component provider must effectively test the components as context-independent units of software. Effective and adequate testing of software components independent of the contexts of their use increases the component users' confidence in the quality of those components, and reduces the burden of the users in testing those components. The adequacy criterion for unit testing of components described in Reference [23] tests the components within the context of a user application. The criterion therefore is more relevant for component users rather than for component providers.

Second, the component provider must support better testing and analysis of the user applications so that the faults related to components can be easier revealed before the user applications are released. This can improve the quality of a component-based system and reduce the cost of testing and maintenance of the system.

3 OUR APPROACH

In this section, we first provide an overview of our approach to analyzing and testing component-based systems. We then illustrate the approach with three examples.

To analyze and test a component-based system, we use an approach that separates the analysis and testing of the user application from the analysis and testing of the the components. In this approach, the component provider tests the components, and, using analysis techniques, gathers *summary information* that facilitates further analysis and testing of those components by component users without requiring access to the source code of the components. The component provider makes the summary information available with the component. The component user integrates the components with the user application, and queries the summary information to drive the analysis and testing of the integrated system. The summary information obviates the need to access the components' source code.

To facilitate testing and dynamic analysis of a component-based system in which a tool desires to compute information about a component under a specific execution, our approach divides the input space of each operation into a set of subdomains and associates the summary information with each subdomain. To compute the information about a component in an execution, the tool first maps the inputs of an operation into a subdomain, and then uses the summary information associated with that subdomain. This method obviates the need to instrument the component in testing and

dynamic analyses.¹ This method also can improve the precision of program analyses when a user application does not use all the functionalities provided by the component. For example, we can divide the input space of an operation according to the configurations. In this case, a tool uses only the summary information about the functionalities used by the user application.

The analysis information stored within each component can be varied according to the level of analysis and testing that is desired by the component users. For example, to support alias analysis of the user application, a component can store alias pairs that hold in all contexts of the component; this information can subsequently be used to generate context-sensitive alias pairs for that component [3, 12]. As another example, to support program slicing of the user application, the component can store summary information about data dependences of the parameters of the component's operations [13] and control-dependence side-effects caused by operations of the component [8, 25]. As a final example, to support structural testing of the user application, the component can store information that facilitates the generation of the testing requirements. Through the stored analysis information, a component can support, for the component user, a wide range of analysis and testing techniques; minimally, the stored information should support a basic set of analysis and testing techniques.

The summary information provided with a component should be represented in a standard notation that is independent of the language in which the component is implemented. The component must provide suitable query facilities — for example, methods or operations — to retrieve the summary information.

We next illustrate our approach by describing three types of summary information that can be provided with components, and facilitate the analysis and testing of component-based systems.

Program Slicing

Program slicing [26] identifies statements in a program that may affect the value of a variable v at a specific program point p ($\langle p, v \rangle$ is called the *slicing criterion*). The set of statements identified by program slicing is a *slice* with respect to $\langle p, v \rangle$. Program slicing is used to support software engineering tasks, such as program understanding, debugging, and impact analysis.

¹One approach to providing testing information with software components uses retrospectors [19]. A *retrospector* records the testing history of a component through instrumentation of the source code of the component, and recommends appropriate testing requirements for that component to the component user. Our approach to providing summary information with components avoids code instrumentation, and provides a wider variety of information that facilitates the application of several analysis and testing techniques to component-based systems.

Program slicing techniques can be classified as *static* or *dynamic*. Static program slicing computes *program dependence* [5] among the program statements by examining the paths in the control-flow graph and computes a transitive dependence closure over the statements. Dynamic program slicing [14] computes only the program dependence that occurs in an execution of a program given a specific input by examining the execution trace. Thus, the slicing criterion for the dynamic slicing also includes the input to the program. Dynamic program slicing can compute, at a higher cost, more precise slices than static program slicing.

Summary information about a component allows a static program slicer to compute more precise slices for the user application in a component-based system. Summary information of a component is represented by the dependence of some abstract variables. When a static slicer encounters a call to an operation of the component, it maps the real input and real output onto the abstract variables and uses the dependence among the abstract variables to approximate the dependence among the real input and the real output. For a component that has no state, summary information provides the dependence between the abstract input variables and the abstract output variables of each operation. For a component that has a state, summary information also provides a set of abstract state variables whose values can define the states of the component, and the dependence between the abstract state variables and the abstract input variables and the abstract output variables. The levels of the abstraction of the input, output, and state variables reflect the levels of the precision of the summary information. At the lowest level, there can be only one input variable, one output variable, and one state variable, and the state variable depends on the input variable and the output variable depends on both the input and the state variables. At a higher level, there are multiple input variables, output variables, and state variables so that the dependence can be represented at a higher precision level. With the summary information, the techniques used in [17] to handle objects can be adapted to handle components in the user applications.

To support more precise dynamic slicing of the component-based system, the dependence in the summary information of a component is defined under the context of the subdomains of the input variables of an operation and the state variables of a component. When a dynamic slicer encounters a call to an operation of the component, it maps the values of the real input and the real state of the component to the subdomains of the input and state variables and then, it uses subdomains to search for the appropriate dependences. The subdomains can be divided at different precision levels. At

the lowest level, each input variable and each state variable can have only one subdomain. In this case, the summary information is the same as the summary information used in static slicing. At the highest level of precision, the subdomains are computed with symbolic execution of the operations.

Control-Dependence Analysis

Control-dependence analysis [2, 5] determines, for each program statement, the predicates that control the execution of that statement. Control-dependence information is required for analyses, such as slicing, that are used for software engineering tools, such as debuggers, impact analyzers, and regression testers.

A component can affect the control-dependence relations that hold in the user application that uses the component because errors may arise in the computation performed by the component. The underlying reason of such an error can be either within the component, or propagated from outside the component through the parameters of the component. If the cause of the error is within the component, the component can take steps to recover from the error. If, however, the error is propagated from outside the component, the component cannot reasonably be expected to correct the error because appropriate error recovery depends on the context of the user of the component. Different users may choose to recover from the same error in different ways. As a context-independent unit, the component signals the error condition, and leaves the context-dependent error recovery to the user.

Exception-handling constructs provide a mechanism for raising exceptions and a facility for designating protected code by attaching exception handlers to blocks of code. A component therefore, raises exceptions corresponding to errors that it cannot recover from, and user application provides handlers for those exceptions.

Summary information about exceptions that can be raised by a component enables several analysis and testing techniques to be performed on a component-based system. Such summary information can be provided at various levels of precision. Each of these levels, at a successively higher cost, records more information and therefore, enables more analysis and testing techniques to be applicable to the component-based system.

At the most basic level, the summary information simply lists the exceptions that can be raised by a component. Such a facility is available in several languages, such as Java and C++, in which a method can declare the exceptions that it can raise. Summary information about potential exceptions has some utility in that if a component raises no exceptions, the control-dependence computation can proceed in the user application without considering possible effects of that component. If

a component raises exceptions, however, the control-dependence computation must consider the effects of that component on control-dependence relations in the user application. In such cases, summary information that only lists the potential exceptions does not facilitate correct control-dependence computation because it does not provide information about variables that determine whether the exceptions are raised.

At a higher level, the summary information can associate with each exception those input parameters and state variables of the component that determine whether that exception is raised. The increased precision in the summary information enables control-dependence computation, and allows the computation of slices that correctly account for side-effects on control dependences caused by the components.

A more precise level of summary information stores the constraints or conditions on the input parameters and state variables that cause various exceptions to be raised. Such analysis information supports control-dependence and slice computation, and also allows the generation of appropriate testing requirements for testing the behavior of exception-handling constructs in the component-based system.

A component can thus choose to represent summary information related to control-dependence computation at various levels of precision depending on the analysis and testing requirements of the system in which the component is embedded.

Data-Flow Testing

Data-flow testing (e.g., [6, 10]) uses data-flow information to guide the selection of test cases and to measure test-suite coverage for a program. In data-flow testing, an assignment to a variable in a program is tested by executing subpaths from the assignment (*definition*) to points where the variable is used (*use*). Test-data adequacy criteria require that test cases in the test suite exercise certain definition-use relationships in the program. Data-flow testing has been described for single procedures or functions [6], for interacting procedures [10], and for object-oriented classes [7].

Component providers can use data-flow testing to provide adequate coverage of their components and to provide summary information for use by component users. Component providers can use the approach to data-flow testing of classes [7] to test their components in a context-independent manner. This approach provides a technique for performing analysis and computing testing requirements for individual methods and interacting methods within the class; the approach can also be applied to interacting classes.

Component providers can use data-flow analysis to pro-

vide summary information that component users will use to test their software. At a minimum, this summary information will provide two types of information. First, the summary information will consist of a set of uses of input variables that can be reached on entry to the module. These uses should be tested with definitions in the component user's software that reach the interface to the module. Second, the summary information will consist of a set of definitions that reach the end of the module. These definitions should be tested with the uses in the component user's software that can be reached after the component executes.

The levels of abstraction of the sets of definitions and uses provide different levels of testing information. At the lowest level, the component is assumed to use and define every input variable. At this level, it may be possible to infer subdomains of the input variables from the component's specification. At subsequently higher levels, more precise information can be provided. At the highest level of precision, the summary information will contain each use and each definition of input variables, an association between uses and definitions and the subdomain to which they belong, and the constraints on inputs associated with the subdomains. The component will store each subdomain along with its set of uses and definitions for use in testing the component-based system.

The component will also possess a method for mapping, at runtime, the input values into the subdomain to which they belong. The method can then report the uses and definitions that are associated with this subdomain as being covered by the input.

A typical scenario for testing the component-based system could proceed in several steps. First, the tester could execute the component-based system with an initial test suite. During these executions, when an input is passed to the component, the component maps the input to a subdomain, reads the uses and definitions covered by that input from the summary information associated with the subdomain, and records the coverage. After each execution, the component returns the coverage measurement.² Next, the tester assesses the coverage and, if the desired coverage is not achieved, requests information about input constraints that are associated with uncovered subdomains. Finally, the tester can use the constraints to generate test cases and execute the system with those test cases to cover the uncovered uses and definitions.

4 SUMMARY AND FUTURE WORK

In this paper, we have identified issues that arise in the

²In this description, we are measuring data-flow coverage but other coverage criteria, such as statement coverage, could also be used.

analysis and testing of component-based systems. We described the issues from the perspectives of the component provider and the component user. We presented an approach that addresses these issues and provides a framework for analyzing and testing component-based systems.

The techniques described in the paper are based on a preliminary examination of the problems that arise in the application of analysis and testing techniques to component-based systems, and have to be further investigated in future work. We need to identify a minimal set of analysis and testing techniques that must be supported by the summary information provided in the components. The various levels of support for analysis and testing techniques could potentially be organized into a hierarchical structure that would provide guidance to component providers in attaining a certain level of support, and in progressing from a lower level of support to a higher one.

We need to formally define the language-independent notation that is used to record summary information, and mechanisms that allow the retrieval of that information. Our approach is based on an abstract model of component-based systems, and therefore, should be applicable to existing component models such as CORBA and COM; in future work, we will investigate how our approach can be adapted to these existing component models. We will also investigate new techniques and adaptations to existing techniques that may be required to effectively analyze and test component-based systems.

ACKNOWLEDGMENTS

This work was supported in part by NSF NYI Award CCR-9696157 to Ohio State University, ESS Award CCR-9707792 to Ohio State University.

REFERENCES

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [2] G. Bilardi and K. Pingali. A framework for generalized control dependence. In *Proc. of SIGPLAN'96 Conf. on Prog. Lang. Design and Implem.*, pages 291–300, May 1996.
- [3] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of 26th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 133–146, January 1999.
- [4] M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT Symposium on foundations of Software Engineering*, pages 62–75, December 1994.
- [5] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [6] P. E. Frankl and E. J. Weyuker. An applicable family of data flow criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
- [7] 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.
- [8] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis*, pages 11–20, March 1998.
- [9] M. J. Harrold and M. L. Soffa. An incremental data flow testing tool. In *Proceedings of the Sixth International Conference on Testing Computer Software*, May 1989.
- [10] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the ACM Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.
- [11] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.
- [12] M.J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7):442–460, July 1996.
- [13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
- [14] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–63, October 1988.
- [15] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.
- [16] T. Lewis. The next 10,000₂ years, part II. *IEEE Computer*, pages 78–86, May 1996.
- [17] D. Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the international conference on software maintenance*, pages 358–368, November 1998.
- [18] D. Liang and M. J. Harrold. Context-sensitive, procedure-specific points-to analysis. Technical Report OSU-CISRC-3/99-TR05, The Ohio State University, March 1999.
- [19] C. Liu and D. Richardson. Software components with retrospectors. In *Proc. of International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 63–68, June 1998.
- [20] S. Ntafos. A comparison of some structural testing strategies. *IEEE Transaction on Software Engineering*, 14(6):868–874, June 1988.

- [21] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.
- [22] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, NY, 1987.
- [23] D. S. Rosenblum. Adequate testing of component-based software. Technical Report UCI-ICS-97-34, University of California, Irvine, August 1997.
- [24] G. Rothermel and M. J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance '93*, pages 358–367, September 1993.
- [25] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proc. of the Int'l Conf. on Softw. Eng.*, May 1999. (to appear).
- [26] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.