

Architecture-Based Regression Testing of Evolving Systems

Mary Jean Harrold

Computer and Information Science
The Ohio State University
395 Drees Lab, 2015 Neil Avenue
Columbus, OH 43210-1227 USA
+1 614 292 2568
harrold@cis.ohio-state.edu

ABSTRACT

Researchers have begun to use formal architectural specification as a basis on which to develop testing techniques. These techniques promise to improve the development process by uncovering defects early. However, few of these techniques consider the ways in which software architecture can be used to facilitate (or misused to hinder) effective regression testing and analysis of evolving systems. Despite efforts to reduce its cost, regression testing remains one of the most expensive activities performed during a software system's lifetime – studies indicate that it can consume up to one-third of the cost of the software. Because of the expense of regression testing and analysis of evolving software, efforts to improve the testing process should focus on techniques and tools to reduce the cost of these activities – the emerging formal notations for software architecture specification can provide a basis on which effective regression testing and analysis techniques can be developed. This paper considers some potential areas for research in using software architecture specification for effective regression testing and analysis. Because of the high proportion of a software system's lifetime that is spent in regression testing and analysis, the use of software architecture for regression testing activities has the potential for a bigger impact on the cost of software than those techniques that focus only on development testing.

Keywords

Software architecture, regression testing, regression testability

1 INTRODUCTION

The increased size and complexity of software systems has led to the emergence of the discipline of software architecture. Software architecture, according to Garlan and Shaw [23], involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. Software architecture styles define families of systems in terms of patterns of struc-

tural organization. In a pipe-and-filter style, for example, each component has a set of inputs, from which it reads data, and a set of outputs, on which it produces data, and the components are arranged sequentially with the output of one component being the input to another component. In an object-oriented organization, in contrast, components are the objects or instances of abstract data types that interact through their primitive operations.

To be useful for the types of analyses that are required for software development activities, software architectures should be specified formally. A number of techniques have been developed using one of the general categories of approaches to formal architectural specification [24]: module interconnection languages, process algebras, and event languages (e.g., [1, 9, 12, 13, 14, 15, 25]).

Recently, researchers have begun to use these formal architectural specification as a basis on which to develop architectural 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 [6], Bertolino and Inverardi consider the ways in which the architectural specification can be used in integration and unit testing [4], and Richardson and Wolf consider the way in which architecture-based coverage criteria can be defined [18]. As a further example, Richardson, Stafford, and Wolf present a formal approach to architecture-based software testing [17], which is based on the Chemical Abstract Machine (CHAM) model for architecture specification [9]. They outline several areas for testing and analysis of software architectures including architecture-based coverage criteria, architectural testability, and architecture slicing. 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.

Although these techniques and tools promise to improve the development process by uncovering defects early, few of them consider the ways in which software architecture can be used to facilitate effective regression test-

ing and analysis of evolving systems.¹ 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. During development, regression testing is used to test families of similar products, safety-critical software, and software that is developed under constant evolution as the market or technology changes; during maintenance, regression testing is used to test new or modified components of the system. Despite efforts to reduce its cost, regression testing remains 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 system [3, 10, 22].

Because of the expense of regression testing and analysis of evolving software, efforts to improve the testing process should focus on techniques and tools to reduce the cost of these activities – the emerging formal notations for software architecture specification can provide a basis on which effective regression testing and analysis techniques can be developed. This paper considers some potential areas for research that use software architecture specification for effective regression testing and analysis. Because of the high proportion of a software system’s lifetime that is spent in regression testing and analysis, the use of software architecture for regression testing activities has the potential for a bigger impact on the cost of software than those techniques that focus only on development testing.

2 REGRESSION TESTING

Research on regression testing spans a wide variety of topics, including test environments and automation, capture-playback mechanisms, test-suite management, and program-size reduction. Recent research has focused on selective-retest techniques (e.g., [5, 7, 21]) and regression testability (e.g., [8, 10, 20]).

Selective-retest techniques can reduce the cost of regression testing by reusing existing tests, and identifying portions of the modified program or its specification that should be tested. For example, a selective retest technique may select all tests from an existing test suite that execute new or modified code. Selective retest techniques differ from a *retest-all* approach, which either runs all tests in the existing test suite or reanalyzes and retests the entire modified program. Leung and White [11] show that a selective-retest technique is more economical than the retest-all technique only if the cost of selecting a reduced subset of tests to run is less than

¹One exception is the work by Stafford, Richardson, and Wolf [24] that defines an architecture-based “slicing” technique called chaining, which can be used for impact analysis of evolving systems.

the cost of running the tests that the selective-retest technique omits.

Given P a procedure or program, P' a modified version of P , S and S' the specifications for P and P' , respectively, and T a test suite created to test P , a typical selective-retest technique proceeds as follows:

1. Select $T' \subseteq T$, a set of tests to execute on P' .
2. Test P' with T' , establishing P' ’s correctness with respect to T' .
3. If necessary, create T'' , a set of new functional or structural tests for P' .
4. Test P' with T'' , establishing P' ’s correctness with respect to T'' .
5. Create T''' , a new test suite and test history for P' , from T , T' , and T'' .

In performing these steps, a selective-retest technique addresses several problems. Step 1 involves the regression test selection problem: the problem of selecting a subset T' of T with which to test P' . Step 3 addresses the coverage-identification problem: the problem of identifying portions of P' or S' that require additional testing. Steps 2 and 4 address the test suite execution problem: the problem of efficiently executing tests and checking test results for correctness. Step 5 addresses the test suite maintenance problem: the problem of updating and storing test information. Although each of these problems is important for regression testing and analysis, this paper considers only issues in the use of software architecture as a basis for regression-test-selection (Step 1) and coverage-identification (Step 3). Techniques can be developed that rely on analysis of the formal specifications of the architectures for P and P' to select tests and identify requirements for coverage.

Regression testability refers to the property of a program, modification, or test suite that lets it be effectively and efficiently regression tested. Leung and White [10] classify a program as regression testable if most single statement modifications to the program entail rerunning a small proportion of the current test suite. Extending Leung and White’s work, Rosenblum and Weyuker [20] consider a program and test suite, and present a formal model of the cost-effectiveness of regression-test-selection techniques. Harrold, et al. [8] performed additional experiments with Rosenblum and Weyuker’s model, and incorporated, in that model, information about the location of the modifications. In addition to providing techniques for analyzing existing programs, test suites, and modifications for regression testability, these classifications and models may facilitate the development of requirements for regression testable software and test suites. To date, these techniques have been applied only to the source-code repre-

sentation of the software. However, techniques can be developed, based on the formal specification of the software architecture, and used to classify and assess the regression testability of the system.

3 ARCHITECTURE-BASED REGRESSION TESTING

This section describes some potential areas for research in the use of software architectures to reduce the cost of regression testing and analysis. In particular, the discussion focuses on selective-retest techniques and regression testability.

3.1 Selective Retest

Steps one and three of a typical selective retest technique (given in Section 2) involve the regression-test-selection problem and the coverage identification problem, respectively. Research can investigate ways in which software architecture can be used to solve these problems.

3.2 Regression Test Selection

One method for reducing the cost of regression testing is to save the test suites that are developed for a product, and reuse them to revalidate the product after it is modified. The retest-all strategy reruns all such tests, but this approach may consume excessive time and resources. Regression test selection techniques, in contrast, attempt to reduce the cost of regression testing by selecting a subset of the test suite that was used during development and using that subset to test the modified program.

In previous work, we developed and described a new regression-test-selection technique [21]. We proved that, under certain well-defined conditions, our test selection algorithms exclude no tests (from the original test suite) that if executed would reveal faults in the modified program. Under these conditions our algorithms are *safe*, and their fault-detection abilities are equivalent to those of the retest-all approach; other researchers have also presented safe approaches to regression test selection (e.g., [2, 5]).

We initially applied our regression-test-selection technique to a source-code representation of the software. Under certain conditions, however, our techniques can be applied to other formal representations of the software; for example, in preliminary work, we have successfully applied our technique to a formal specification of the software. Our techniques, however, can be applied to various levels of the software if we can (1) represent the software as a “graph”, (2) associate tests with “edges” in this graph, and (3) determine whether two “nodes” in the graph differ. When this regression-test-selection technique is applied to the source code, for ex-

ample, the graph representation is a control flow graph, the association between tests and edges is determined by instrumenting and executing the program to produce an edge profile, and the difference between nodes is accomplished by comparing the text associated with the nodes in the graph.

One area for research concerns applying the regression test selection technique to the formal representation of the software. Richardson, Stafford, and Wolf have developed a set of architecture-based coverage criteria. Their criteria are based on the CHAM architecture model, and provide a type of “structural” coverage of the architecture. Their criteria include the all-processing-elements criterion, which requires that all processing elements are executed, and the all-connecting-elements criterion, which requires that all communication channels and connections on them are exercised. By considering the processing elements as “nodes” and the communication channels as “edges” in a graph, we may be able to apply our regression-test-selection technique to the formal representation of the architecture.

Test-selection algorithms can be used in two ways. First, these algorithms can determine the retesting that is required after changes are made to the software’s architecture. Applying the test selection at the architectural level may be more efficient than at the source-code level. Second, the test selection can be used to assess the retesting that will be required for different modifications to an architecture. This information can help with decisions among several potential architectural changes.

3.3 Coverage Identification

Our regression-test-selection techniques, described above, identify those tests in an existing test suite that must be rerun after changes are made to the software because they execute new or modified code. However, after running these selected tests, there may be parts of the modified program that may remain untested. Thus, after selecting a safe test set T' to run on a modified program, an alternate version of our algorithm finds definition-use pairs² that should be retested, to support data-flow testing criteria [16]; the algorithm uses forward-slicing techniques to identify the affected definition-use pairs. Using this approach, we need not consider each modification individually. Instead multiple modifications are handled in a single pass over the graphs for the original and modified programs; similar approaches could support statement or branch testing coverage criteria.

²A *definition-use pair* is a pair of statements ($S1, S2$), such that $S1$ defines some variable v , $S2$ uses v , and there is a path in the program from $S1$ to $S2$ along which v is not redefined.

Stafford, Richardson, and Wolf [24] describe a type of architecture dependence analysis called *chaining*, which can be used to provide “architectural slices”. Chaining defines links that connect elements of the architectural specification that are directly related, which produces a chain of dependencies that can be followed during analysis. Stafford et al. suggest using chaining to reduce the portions of the architecture that must be examined. For example, chaining could be used to determine the impact of a proposed change to the architecture.

Future research can investigate the incorporation of techniques such as chaining, which represents dependencies among architectural processing elements, with a regression-test-selection technique, which has been applied to the architectural specification. After the technique finds those tests to rerun because of changes in the specification, it can identify those structures of the architecture that must be retested to provide the desired coverage.

3.4 Regression Testability

Eickelmann and Richardson [6] present issues involved in using software architecture analysis to determine the testability of a software system – the system’s ability to support certain testing strategies or detection of specific fault types. Leung and White also define the notion of testability, and define it in terms of the paths in a program. They define a *testing number*, which is the set of paths affected by a single instruction modification. Because this set of paths can be determined by static analysis of the program, assuming that all instructions are equally likely to be modified, this testing number can be determined statically. This testing number gives a way to classify and compare programs with respect to testability: a program is testable if its testing number is small; a program P is said to be more testable than program Q if P’s testing number is less than Q’s testing number. For a single modification, a more testable program requires an analysis of fewer paths on average than a less testable program.

Using the notion of “paths” in the specification of the architecture, as defined by Richardson, Stafford, and Wolf [17], the testing number for a system can be determined from its architectural specification. Using this testing number, software architectures can be classified and compared, and the results of these comparisons can be used to guide the selection of testable architectures.

Leung and White also discuss the *regression testability* of a software system – a system is regression testable if most single statement modifications will entail rerunning a small proportion of the current test suite [10]. Under this definition, regression testability is a function of both the design of the program and the test suite. Rosenblum and Weyuker presented a model for pre-

dicting the cost-effectiveness of regression-test-selection techniques in terms of the coverage provided by a particular test suite [19]. Under their approach, both the program and the test suite are used for prediction.

To consider regression testability, a *regression number*, which is the average number of affected test cases in the test suite that are affected by any modification to a single instruction, is computed. This regression number is computed using information about the test-suite coverage of the program. If a program has a regression number that is close to 1, any statement modification will affect most of the test cases. Using an approach that considers the location of the change in the model [8], the regression number can be computed for specific statements or modules in the program instead of for the entire program.

Future research can investigate the way in which software-architecture coverage criteria, such as those defined by Richardson, Stafford, and Wolf [17], can be used in conjunction with regression-testing prediction models to predict the cost-effectiveness of regression-test-selection for the given test suite. This approach requires that a set of tests have been developed for the system and used during development testing of the system. These techniques can help design software and test suites on which efficient regression testing can be performed.

The ability to consider regression testability early in the development process has the potential to provide significant savings in the cost of development and maintenance of the software.

4 Conclusions

This paper has considered two broad areas of regression testing in which techniques that apply to the source code of the program have been developed: selective retest and regression testability. After giving an overview of these research results, the paper presented some areas for future research in which a formal specification of the software architecture, instead of the source code, is used as a basis for selective retest and regression testability. Architecture-based testing should focus on the development of techniques that can help reduce the cost of regression testing and analysis. Because of the high cost of regression testing and analysis, these techniques have the potential to significantly reduce the cost of software development and maintenance. Future research will investigate architecture-based approaches to regression testing.

5 Acknowledgement

Thanks to Gregg Rothermel for his comments on a draft of this paper.

REFERENCES

- [1] R. Allen and D. Garlan. A formal basis for architectural connection. *IEEE Trans. on Softw. Eng.*, 6(3):213–249, July 1997.
- [2] T. Ball. On the limit of control-flow analysis for regression testing. In *Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis*, Mar. 1998.
- [3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.
- [4] A. Bertolino and P. Inverardi. Architecture-based software testing. In *Proc. of Int'l Softw. Arch. Workshop*, pages 62–64, October 1996.
- [5] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *Proc. of Int'l Conf. on Softw. Eng.*, pages 211–222, May 1994.
- [6] N. Eickelmann and D. Richardson. What makes one software architecture more testable than another? In *Proc. of Int'l Softw. Arch. Workshop*, pages 65–67, October 1996.
- [7] M. Harrold and M. Soffa. Interprocedural data flow testing. In *Proc. of the Third Symp. on Softw. Testing, Analysis, and Verification*, pages 158–167, Dec. 1989.
- [8] M. J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. Technical Report OSU-CISRC-2/98-TR06, Ohio State Univer., Feb. 1998.
- [9] P. Inverardi and A. Wolf. Formal specification and analysis of software architecture using the Chemical Abstract Machine Model. *IEEE Trans. on Softw. Eng.*, 21(4):373–386, Apr. 1995.
- [10] H. Leung and L. White. Insights Into Regression Testing. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 60–69, Oct. 1989.
- [11] H. Leung and L. White. A cost model to compare regression test strategies. In *Proc. of Conf. on Softw. Maint.*, pages 201–208, Oct. 1991.
- [12] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Trans. on Softw. Eng.*, 21(4), Apr. 1995.
- [13] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proc. of the 6th Europe. Softw. Eng. Conf./5th ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, pages 60–76, Sept. 1997.
- [14] D. Perry and A. Wolf. Foundations for the study of software architecture.
- [15] M. Pezze and M. Young. Constructing multi-formalism state-space analysis tools: Using rules to specify dynamic semantics of models. May 1997.
- [16] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Softw. Eng.*, 11(4):367–375, Apr. 1985.
- [17] D. Richardson, J. Stafford, and A. Wolf. A formal approach to architecture-based software testing. Technical report, University of California, Irvine, 1998.
- [18] D. Richardson and A. Wolf. Software testing at the architectural level. In *Proc. of Int'l Softw. Arch. Workshop*, pages 6–70, October 1996.
- [19] D. S. Rosenblum and E. J. Weyuker. Predicting the cost-effectiveness of regression testing strategies. In *Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, Oct. 1996.
- [20] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Trans. Softw. Eng.*, 23(3):146–156, Mar. 1997.
- [21] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Meth.*, pages 173–210, Apr. 1997.
- [22] S. Schach. *Software Engineering*. Aksen Associates, Boston, MA, 1992.
- [23] M. Shaw and D. Garlan. *Software Architecture Perspectives on an Emerging Discipline*. Prentice Hall, New Jersey, 1996.
- [24] J. Stafford, D. Richardson, and A. Wolf. Chaining: A software architecture dependence analysis technique. Technical Report CU-CS-845-97, University of Colorado, Sept. 1997.
- [25] A. Wolf, L. Clarke, and J. Wileden. The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Trans. on Softw. Eng.*, 15(3):250–263, March 1989.