Challenges in Exploiting Architectural Models for Software Testing

David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
+1 949.824.6534
dsr@ics.uci.edu
http://www.ics.uci.edu/~dsr/

ABSTRACT

Software architectural modeling offers a natural framework for designing and analyzing modern large-scale software systems and for composing systems from reusable off-the-shelf components. However, the nature of component-based software presents particularly unique challenges for testing component-based systems. To date there have been relatively few attempts to establish a sound theoretical basis for testing component-based software.

This paper discusses challenges in exploiting architectural models for software testing. The discussion is framed in terms of the author's recent work on defining a formal model of test adequacy for component-based software, and how this model can be enhanced to exploit formal architectural models.

Keywords

ADLs, architectural modeling, component-based software, integration testing, software testing, subdomain-based testing, test adequacy criterion, unit testing.

1 INTRODUCTION

Software architectural modeling offers a natural framework for designing and analyzing modern large-scale software systems and for composing systems from reusable off-the-shelf components [9,10].   However, the nature of component-based software presents particularly unique challenges for testing component-based systems.   In particular, while the technology for constructing component-based software is relatively advanced, and while the architecture research community has produced a number of powerful formal notations and analysis techniques for architectural modeling, there have been relatively few attempts to establish a sound theoretical basis for testing component-based software (e.g., see [3,11,12]).

An architectural model can be used in a variety of ways to aid the testing of a component-based system:

- The model itself can be tested directly, prior to the selection of components and the development of the implementation.   This requires that the model be expressed in an *architecture description language* (ADL) having a simulation or execution semantics [6]. Rapide is an example of such a language [4].

- The model can be used to guide integration testing of the implemented system. In particular, the structure of the model can be used to guide the order in which components are assembled and tested, and the specifications of the model elements can be used as test oracles.

- The model can be used to guide selective regression testing of the system as it evolves in maintenance [13].

The software testing literature offers a variety of techniques that can be applied or adapted in a reasonably straightforward way to these kinds of testing (e.g., by defining architecture-oriented structural coverage criteria, defining the architectural equivalent of top-down or bottom-up integration testing, etc.). Yet there is very little in the testing literature that addresses the unique testing challenges posed by component-based software.

Distributed component-based systems of course exhibit all of the well-known problems that make testing "traditional" distributed and concurrent software difficult. But testing of component-based software (distributed or otherwise) is further complicated by *technological heterogeneity* and *enterprise heterogeneity* of the components used to build systems. Technological heterogeneity refers to the fact that different components can be programmed in different programming languages and for different operating system and hardware platforms, meaning that testing a component-based system may require a testing method that operates on a large number of languages and platforms. Enterprise heterogeneity refers to the fact that off-the-shelf components can be provided by different, possibly competing suppliers, meaning that no one supplier has complete control over or complete access to the development artifacts associated with each component for purposes of testing. And in the most extreme situations of *dynamic evolution*, components can be replaced within, added to, and deleted from a running system, potentially forgoing a traditional period of testing prior to deployment of the new configuration [5,8].

Thus, in this author's view, it is these problematic characteristics of component-based software that raise the
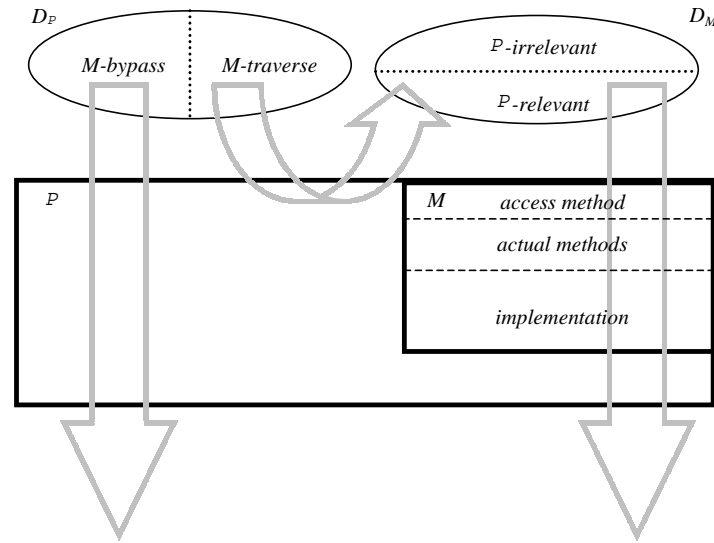
Fig. 1 . A Model of Component-Based Software.

most interesting and important challenges in exploiting architectural models for software testing.

2 A FORMAL DEFINITION OF COMPONENT-BASED TEST ADEQUACY

Any attempt to develop a foundation for testing component-based software must begin by establishing an appropriate formal model of test adequacy. A *test adequacy criterion* is a systematic criterion that is used to determine whether a test suite provides an adequate amount of testing for a program under test. Previous definitions of adequacy criteria have defined adequate testing of a program independently of any larger system that uses the program as a component. This perhaps may be due to the traditional view of software as a monolithic code base that can be put through several phases of testing prior to its deployment, and by the same organization that built the software in the first place. However, a test set that satisfies a criterion in the traditional sense might not satisfy the criterion if it were interpreted with respect to the subset of the component's functionality that is used by a larger system.

Consider the simple example of the *statement coverage* criterion, which requires a test set to exercise each statement in the component under test at least once. There may be a large number of elements in the component's input domain that could be chosen to cover a particular statement. However, the element that is ultimately chosen may not be a member of that subset of the input domain that is utilized by the larger program using the component. Hence, while according to traditional notions of test adequacy the test case could serve as a member of an adequate test set for the component, from the perspective of the larger program using the component, the test set would be inadequate.

In a recent paper, the author developed a formal model of component-based software and a formal definition of component-based test adequacy [12]. This work attempts to capture in a formal way the need to consider the context in which a component will be used in order to judge whether or not the component, and the system using the component, has been adequately tested.

Fig. 1 represents pictorially the formal model of component-based software. The figure illustrates a program P containing a constituent component M. In general, P may contain several such components M, and P may itself be a component within some larger system. As shown in the figure, M is viewed as declaring in its interface a single *access method* that handles the invocation of the *actual methods* of M. For each parameter of an actual method of M, there is a corresponding parameter of the same type and mode in the access method. The access method includes an additional parameter used to identify the actual method that is to be invoked. The input domain of M is then the input domain of its access method, which is the union of the input domains of the actual methods of M, but with each element extended with the appropriate method identifier.

Let D_P be the input domain of P, and let D_M be the input domain of the access method of M. As shown in the figure, there are four important subsets of these input domains, which are defined formally as follows:

Definitions:

$$M\text{-}traverse(D_P) \quad = \\ \{\, d \in D_P \mid \text{execution of } P \text{ on input } d \text{ traverses } M \,\}$$

$$M\text{-}bypass(D_P) \quad = \quad D_P - M\text{-}traverse(D_P)$$

$\mathcal{P}\text{-relevant}(D_M) \quad = \\ \quad \{ d \in D_M \mid \exists\, d' \in M\text{-traverse}(D_\mathcal{P}) \bullet \text{execution of} \\ \quad\quad \mathcal{P} \text{on input } d' \text{ traverses } M \text{ with input } d \}$

$\mathcal{P}\text{-irrelevant}(D_M) \quad = \quad D_M - \mathcal{P}\text{-relevant}(D_M)$

The phrase "execution of $\mathcal{P}$ traverses M" is taken to mean that the execution of $\mathcal{P}$ includes at least one invocation of M's access method. The *M-traverse* subset of $D_\mathcal{P}$ is then the set of all inputs of $\mathcal{P}$ that cause the execution of $\mathcal{P}$ to traverse M. The $\mathcal{P}$*-relevant* subset of D_M is the set of all inputs of M's access method that $\mathcal{P}$ uses for its traversals of M. The *M-bypass* subset of $D_\mathcal{P}$ is the set of all inputs of $\mathcal{P}$ that cause the execution of $\mathcal{P}$ to "bypass" or avoid traversing M. Finally, the $\mathcal{P}$*-irrelevant* subset of D_M is the set of all inputs of M's access method that $\mathcal{P}$ never uses for its traversals of M.[1]

The formal definition of test adequacy for component-based software is developed in terms of applicable subdomain-based test adequacy criteria, as defined by Frankl and Weyuker [2]. In particular, a test adequacy criterion *C* is *subdomain-based* if there is a nonempty multiset $\mathcal{SD}_C(D)$ of subdomains of *D* (the input domain of the program under test), such that *C* requires the selection of one test case from each subdomain in $\mathcal{SD}_C(D)$.[2] Furthermore, *C* is *applicable* if the empty subdomain is not an element of $\mathcal{SD}_C(D)$ [2]. Thus, a test set is *C-adequate* if and only if it contains at least one test case from each subdomain in $\mathcal{SD}_C(D)$. Since testers rarely satisfy 100% of the test requirements induced by a test adequacy criterion, it also makes sense to say that a test set is *n% C-adequate* if it contains at least one test case from *n* percent of the subdomains in $\mathcal{SD}_C(D)$. These definitions capture the traditional notion of test adequacy, and they make no distinction between a program and a component.

In order to define test adequacy for component-based software, it is necessary to first partition the subdomains induced by an applicable subdomain-based criterion *C* according to the partitioning of $D_\mathcal{P}$ and D_M:

Definitions:

$\mathcal{SD}_C(M\text{-traverse}(D_\mathcal{P})) \quad = \\ \quad \{ D \subseteq M\text{-traverse}(D_\mathcal{P}) \mid \exists\, D' \in \mathcal{SD}_C(D_\mathcal{P}) \bullet D \subseteq D' \\ \quad\quad \text{and } D'-D \subseteq M\text{-bypass}(D_\mathcal{P}) \text{ and } D \neq \varnothing \}$

---

[1] Note that this model fails to account for the possibility of non-determinism in the execution of $\mathcal{P}$ or M.

[2] An example of such a criterion is *statement coverage*, which induces one subdomain for each executable statement in a program, with each subdomain containing exactly those inputs that cover its associated statement.

$\mathcal{SD}_C(M\text{-bypass}(D_\mathcal{P})) \quad = \\ \quad \{ D \subseteq M\text{-bypass}(D_\mathcal{P}) \mid \exists\, D' \in \mathcal{SD}_C(D_\mathcal{P}) \bullet D \subseteq D' \\ \quad\quad \text{and } D'-D \subseteq M\text{-traverse}(D_\mathcal{P}) \text{ and } D \neq \varnothing \}$

$\mathcal{SD}_C(\mathcal{P}\text{-relevant}(D_M)) \quad = \\ \quad \{ D \subseteq \mathcal{P}\text{-relevant}(D_M) \mid \exists\, D' \in \mathcal{SD}_C(D_M) \bullet D \subseteq D' \\ \quad\quad \text{and } D'-D \subseteq \mathcal{P}\text{-irrelevant}(D_M) \text{ and } D \neq \varnothing \}$

$\mathcal{SD}_C(\mathcal{P}\text{-irrelevant}(D_M)) \quad = \\ \quad \{ D \subseteq \mathcal{P}\text{-irrelevant}(D_M) \mid \exists\, D' \in \mathcal{SD}_C(D_M) \bullet D \subseteq \\ \quad D' \text{ and } D'-D \subseteq \mathcal{P}\text{-relevant}(D_M) \text{ and } D \neq \varnothing \}$

Note that according to these definitions, each subdomain induced by criterion *C* on program $\mathcal{P}$ is partitioned into its *M-traverse* subset and its *M-bypass* subset. Note that the definitions discard empty subdomains, in order to retain the applicability of *C*.

Given the above definitions, adequate testing of component-based software can now be formally defined. First, the concept *C-adequate-for-$\mathcal{P}$* is defined to characterize adequate *unit testing* of *M*:

Definition (C-adequate-for-$\mathcal{P}$): A test set T_M is *C-adequate-for-$\mathcal{P}$* if it contains at least one test case from each subdomain in $\mathcal{SD}_C(\mathcal{P}\text{-relevant}(D_M))$.

Second, the concept *C-adequate-on-M* is defined to characterize adequate *integration testing* of $\mathcal{P}$ with respect to its usage of *M*:

Definition (C-adequate-on-M): A test set $T_\mathcal{P}$ is *C-adequate-on-M* if it traverses *M* with at least one element from each subdomain in $\mathcal{SD}_C(\mathcal{P}\text{-relevant}(D_M))$.

These definitions can be extended as before to accommodate a notion of percentage of adequacy.

Note that although it is $\mathcal{P}$ that is being tested in integration testing, these definitions require the criterion *C* to be chosen and then evaluated *in terms of M* in order to ensure adequate testing of the relationship between $\mathcal{P}$ and *M*. For example, *C* could be a criterion that requires each of the actual methods of *M* to be exercised at least once. This is a reasonable requirement for adequate integration testing of $\mathcal{P}$, and the definition of *C-adequate-on-M* ensures that the criterion would be interpreted only with respect to the methods of *M* that $\mathcal{P}$ invokes anywhere in its source code. Of course, *C* need not be the same criterion as the one used to design $T_\mathcal{P}$ in the first place; it merely imposes a requirement on the testing achieved by $T_\mathcal{P}$.

There are a number of additional interesting consequences of these definitions. For instance, a test set T_M that is *C-adequate* might not be *C-adequate-for-$\mathcal{P}$*, and vice versa. Furthermore, a test set $T_\mathcal{P}$ that is *C-adequate* might not be *C-adequate-on-M*, and vice versa. And similar statements can be made with respect to percentage of adequacy.

3 ADEQUATE TESTING AND SOFTWARE ARCHITECTURES

The formal model presented above provides an initial foundation for studying and evaluating test adequacy for component-based systems. However, there are two important issues that merit consideration. One issue is the practical applicability of the model. It is one thing to argue that components must be tested with respect to the context in which they will be used. It is another thing to determine how this will be accomplished, especially in the presence of off-the-shelf components, whose important attributes needed for evaluation of test adequacy criteria (such as input domain, specification, implementation structure, etc.) may be difficult or impossible to ascertain.

A second issue is to determine how the model relates to, and can be adapted to, software architectural modeling. One approach is to view formal architectural models as inducing definitions of input domains for architectural elements, and then applying the model to these induced input domains. However, this approach must take into consideration at least three important attributes of architectural models and the ADLs used to specify them.

First, many ADLs support specification of other kinds of architectural elements in addition to components, such as connectors and configurations (see Medvidovic et al. for a complete discussion of ADL features and capabilities [6,7]). The formal model presented above must be enhanced to take these additional kinds of elements into account. In some ADLs these elements can possibly be treated as components for the purpose of testing. For instance, the connectors in Wright encapsulate behavior and provide a static, finite collection of interface elements [1]. However, in other ADLs such elements differ substantially from components. For instance, a connector in C2 does not have an interface per se, but instead is *contextually reflective* of the interfaces of the components that it connects [15]. Furthermore, this set of components can vary dynamically [8].

Second, many ADLs distinguish between the *conceptual architecture* that is formally modeled in an ADL and the *implementation architecture* of the actual system, and different ADLs impose different requirements on the relationship between the two. At a minimum, this distinction means that the input domains of the conceptual architecture may differ from those of the implementation architecture. In particular, the input domain of a component in the conceptual architecture can be a (proper) subset of the input domain of the implementation-level component or components that implement the conceptual component. This is to be expected especially in situations where an off-the-shelf component provides more functionality than is needed by a system that uses it. Thus, the formal model of test adequacy developed above must be enhanced to account for the additional complexities introduced by domain relationships between conceptual and implementation architectures.

Third, ADLs support explicit representation of a rich variety of behavioral relationships and other dependencies between architectural elements [14]. These relationships do not always strictly conform to a caller/callee style of component relationships as depicted in Fig. 1, and it may not be possible to characterize them fully and precisely in terms of the input domains of related elements. Thus, the formal model of test adequacy developed above must be enhanced to account for the richness of inter-element relationships.

With these enhancements in place, the formal model of test adequacy can be used in conjunction with formal architectural models to guide testing of software in a manner that is truly adequate. A key challenge in incorporating such enhancements is to address the broad range of semantic models and modeling and analysis concerns of the many ADLs that have been defined.

4 CONCLUSION

This paper has discussed challenges in exploiting architectural models for software testing, with the discussion framed in terms of the author's recent work on defining a formal model of test adequacy for component-based software. An explicit architectural viewpoint in software engineering offers the promise of dramatically improving—and in the process altering— the way software is developed [10]. While these changes will not obviate the need for testing, one can at least attempt to find ways of exploiting formal architectural models for the purpose of testing. While architectural models offer a rich source of information to support testing, any attempt to exploit architectural models for testing must be cognizant of the unique characteristics of the new kinds of systems that an architectural viewpoint engenders.

REFERENCES

1. R. Allen and D. Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 213–249, 1997.

2. P.G. Frankl and E.J. Weyuker, "A Formal Analysis of the Fault-Detecting Ability of Testing Methods", *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 202–213, 1993.

3. D. Hamlet, "Software Component Dependability—a Subdomain-based Theory", RST Corporation, Technical Report RSTR-96-999-01, September 1996.

4. D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language", *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, 1995.

5. J. Magee and J. Kramer, "Dynamic Structure in Software Architectures", *Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*, San Francisco, CA, pp. 3–14, 1996.

6. N. Medvidovic and D.S. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages", *Proc. USENIX Conference on Domain Specific Languages*, Santa Barbara, CA, pp. 199–212, 1997.

7. N. Medvidovic and R.N. Taylor, "A Framework for Classifying and Comparing Architecture Description Languages", *Proc. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 60–76, 1997.

8. P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution", *Proc. 20th International Conference on Software Engineering*, Kyoto, Japan, 1998.

9. P. Oreizy, N. Medvidovic, R.N. Taylor, and D.S. Rosenblum, "Software Architecture and Component Technologies: Bridging the Gap", Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA January 1998.

10. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

11. D.J. Richardson and A.L. Wolf, "Software Testing at the Architectural Level", *Proc. Second International Software Architecture Workshop*, San Francisco, CA, pp. 68–71, 1996.

12. D.S. Rosenblum, "Adequate Testing of Component-Based Software", Department of Information and Computer Science, University of California, Irvine, Irvine, CA, Technical Report 97-34, August 1997.

13. G. Rothermel and M.J. Harrold, "Analyzing Regression Test Selection Techniques", *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

14. J.A. Stafford, D.J. Richardson, and A.L. Wolf, "Chaining: A Software Architecture Dependence Analysis Technique", Department of Computer Science, University of Colorado, Boulder, CO, Technical Report CU-CS-845-97, September 1997.

15. R.N. Taylor, N. Medvidovic, K.M. Anderson, J. E. James Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow, "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.