Adequate Testing of Component-Based Software

David S. Rosenblum

Technical Report 97-34
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

11 August 1997

Adequate Testing of Component-Based Software

David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
+1 714.824.6534
dsr@ics.uci.edu
http://www.ics.uci.edu/~dsr/

ABSTRACT
People have long advocated a component-based approach to software construction as a way of simplifying the design and maintenance of large software systems, increasing the opportunities for reuse, and increasing software development productivity. Although the technology for constructing component-based software is relatively advanced, we lack a sufficient theoretical basis for testing component-based software. This paper initiates the development of such a theory. The main result is a formal definition of the concept *C-adequate-for-*$\mathcal{P}$ for adequate unit testing of a component and the concept *C-adequate-on-M* for adequate integration testing of a component-based system. The paper uses these concepts to discuss practical considerations in adequate testing of component-based software.

Keywords
Component-based software engineering, integration testing, subdomain-based testing, test adequacy criterion, unit testing

INTRODUCTION
People have long advocated a component-based approach to software construction as a way of simplifying the design and maintenance of large software systems, increasing the opportunities for reuse of software assets, and increasing the overall productivity of software development. Recent technical advances have finally enabled component-based software engineering to become a widespread reality.

Reusable components exist in a multitude of forms. Older forms of components include logically-encapsulated code fragments such as *programming libraries*, *class libraries*, *reusable modules* (e.g., Ada packages), and *legacy systems* that have been "wrapped" for use as a component within some other application. Newer forms of components include user-level *plug-ins* and *add-ins*, downloadable components such as Java *applets*, and domain-specific *frameworks* such as the Microsoft Foundation Classes for building user interface software. And a variety of *infrastructure* technologies now exist for composing applications from multiple, heterogeneous component parts, including CORBA [13], ActiveX [2] and JavaBeans [8], as well as *compound document* technologies such as OLE [1] and OpenDoc [4].

Given the increasing dominance of component-based software engineering, it is important to begin studying the problem of testing component-based software. Yet although the technology for constructing component-based software is relatively advanced, we lack a sufficient theoretical basis for the testing of component-based software. This paper initiates the development of such a theory.

The problem of testing component-based software is complicated by a number of characteristics of component-based software engineering. *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 hardware platforms, meaning that testing a component-based system may require a testing method that works for all possible languages and platforms. Enterprise heterogeneity refers to the fact that different 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 deployed, brought off-line, and then re-deployed in new versions, all in a manner that is transparent to applications that are using the components.

As a first step in developing a foundation for testing component-based software, this paper defines a formal model of test adequacy for component-based software. 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 component under test. Previous

definitions of adequacy criteria have defined adequate testing of a component independently of any larger system that uses the component. This perhaps may be due perhaps 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 all by the same organization that built the software in the first place. However, a test suite 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 the larger system.

Consider the simple example of the *statement coverage* criterion, which requires a test suite 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.

The model defined in this paper is developed in terms of a program $\mathcal{P}$ that contains a single component of interest M.[1] The model can be generalized for programs containing multiple components. The model can be applied in two different ways to the problem of testing component-based software. In one way, the problem is viewed as one of adequate *unit testing* of a component that is to be used (perhaps off-the-shelf) within a larger system. In the other way, the problem is viewed as one of adequate *integration testing* of the application containing the component.

The paper begins with a discussion of previous efforts to formalize the notion of test adequacy. The paper then defines a formal model of component-based software. This model is then used to formally define a notion of test adequacy for component-based systems. The presentation of the formal model is followed by a discussion of some of the practical considerations involved in ensuring adequate testing of component-based systems. The paper concludes with a discussion of future work.

BACKGROUND

Weyuker defined a set of ten axioms that formalize much of the intuition underlying the idea of test adequacy, as well as its less obvious ramifications [14]. Eight of the axioms apply in a straightforward way to adequate testing of all software, component-based or otherwise. The remaining two axioms apply specifically to component-based software. They are the axioms of Antidecomposition and Anticomposition, which Weyuker stated as follows:

> *Axiom of Antidecomposition:* There exists a program P and component Q such that T is adequate for P, T' is the set of vectors of values that variables can assume on entrance to Q for some t of T, and T' is not adequate for Q.
>
> *Axiom of Anticomposition:* There exist programs P and Q such that T is adequate for P and $P(T)$ is adequate for Q, but T is not adequate for $P;Q$ [i.e., to the sequential composition of P with Q] [14].

While undeniably true as stated, these axioms have limited utility as a basis for developing a notion of test adequacy for component-based software. First, the model of components embodied in these axioms is one of sequential composition of program statements, which does not correspond well to a modular or object-oriented style of composition. Second, the statement of the axioms in existential terms means that they describe the worst case rather than the general case. Third, the notion of adequacy underlying the axioms is 100% satisfaction of the test requirements induced by a test adequacy criterion, yet testers rarely achieve 100% adequacy. Hence, a formalization of test adequacy is needed that better addresses the realities of testing component-based software.

Perry and Kaiser extended Weyuker's work to demonstrate how features of the object-oriented paradigm (particularly class inheritance, method overriding, and multiple inheritance) complicate the problem of adequately testing object-oriented programs [9]. While employing a more useful notion of component, their attention is confined primarily to the problem of testing individual classes rather testing compositions of class instances.

This paper develops a general formal model of test adequacy for component-based software. As described in detail in the following sections, the model is defined in terms of *subdomain-based test adequacy criteria*, as defined by Frankl and Weyuker [6]. Before defining the model of test adequacy, it is first necessary to define a formal model of component-based software.

A FORMAL MODEL OF COMPONENT-BASED SOFTWARE

The notion of *component* as used in this paper corresponds to a general object-oriented notion of a component. In particular, a component M encapsulates some state and provides a well-defined interface that strictly governs access to the state by other parts of a system containing the component. The interface is usually a set of *methods* or operations that can be applied to the component; *attributes* or data members of an interface can easily be represented with pairs of *get* and *set* methods. Without loss of generality, M is viewed in this paper as declaring in its

[1] As is traditional in formal treatments of test adequacy, the letter C will be used to refer to test adequacy criteria. Therefore, the letter M will be used to refer to components since they are somewhat synonymous with modules.

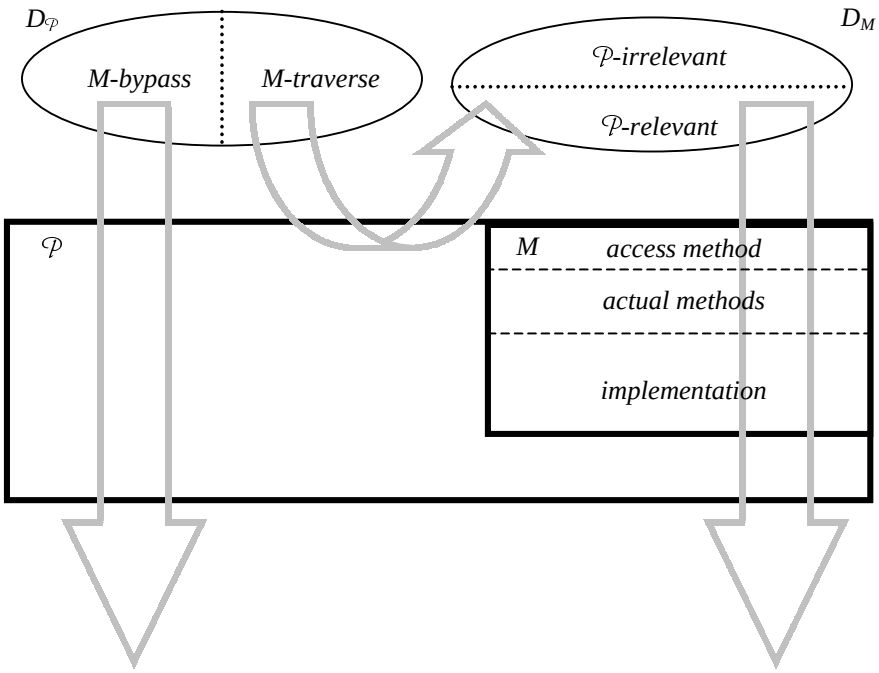

Fig. 1 . A Model of Component-Based Software.

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.

Note of course that *M* can be viewed as a program for the purpose of testing its own constituent subcomponents according to the notions of test adequacy defined in this paper. Indeed, *M* might be some large legacy system that has been adapted for use within $\mathcal{P}$.

Fig. 1 represents pictorially the model of component-based software used in this paper. The figure illustrates a program $\mathcal{P}$ containing an constituent component *M*. Let $D_{\mathcal{P}}$ be the input domain of $\mathcal{P}$, 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:

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

$$M\text{-}bypass(D_{\mathcal{P}}) = D_{\mathcal{P}} - M\text{-}traverse(D_{\mathcal{P}})$$

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

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

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*.

A FORMAL DEFINITION OF COMPONENT-BASED TEST ADEQUACY

A test set $T_{\mathcal{P}}$ is said to be *C-adequate* if $T_{\mathcal{P}}$ satisfies the test requirements induced by a test adequacy criterion *C* on

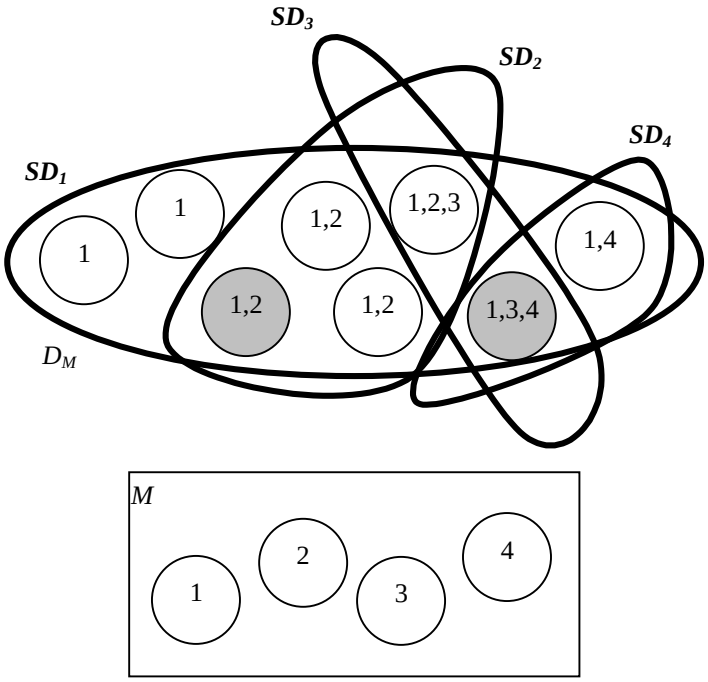

Fig. 2. Graphical depiction of a subdomain-based test adequacy criterion.

program $\mathcal{P}$.[2] Similarly, a test set T_M is *C-adequate* if T_M satisfies the test requirements induced by *C* on component *M*.

The model of test adequacy for component-based software developed below is defined in terms of applicable subdomain-based test adequacy criteria as defined by Frankl and Weyuker [6]. In particular, a test adequacy criterion *C* is *subdomain-based* if, for program $\mathcal{P}$ (component *M*), there is a nonempty multiset $\mathcal{SD}_C(D_\mathcal{P})$ ($\mathcal{SD}_C(D_M)$) of subdomains of $D_\mathcal{P}$ (D_M), such that *C* requires the selection of one test case from each subdomain in $\mathcal{SD}_C(D_\mathcal{P})$ ($\mathcal{SD}_C(D_M)$). Furthermore, *C* is *applicable* if the empty subdomain is not an element of $\mathcal{SD}_C(D_\mathcal{P})$ ($\mathcal{SD}_C(D_M)$) [6]. Thus, adequacy is defined for subdomain-based criteria as follows:

Definition (C-adequate): Given an applicable subdomain-based criterion *C*, a program $\mathcal{P}$, and a component *M*,

1. A test set $T_\mathcal{P}$ for $\mathcal{P}$ is *C-adequate* if $T_\mathcal{P}$ contains at least one test case from each subdomain in $\mathcal{SD}_C(D_\mathcal{P})$.
2. A test set T_M for *M* is *C-adequate* if T_M contains at least one test case from each subdomain in $\mathcal{SD}_C(D_M)$.

This definition captures the traditional notion of test adequacy, and it makes no distinction between a program and a component. In the remainder of the paper, criteria will be assumed to be applicable and subdomain-based.

To demonstrate how a criterion induces a set of subdomains, Fig. 2 illustrates a subdomain-based criterion *C* that is defined in terms of code entities within *M*.[3] The numbered circles in *M* represent these entities. The numbered circles in D_M represent the elements of *M*'s input domain, with the numbers indicating which entities an element exercises. The regions labeled SD_i represent the subdomains induced by *C*, with SD_i being the set of inputs that exercise entity *i*. The shaded circles represent an example of a test set that is adequate according to this code-based criterion. Note that subdomains typically overlap, as do the test cases in their coverage of subdomains.

As was observed previously, testers rarely satisfy 100% of the test requirements induced by a test adequacy criterion.

---

[2] Test adequacy is usually defined in terms of program/specification pairs. However, the formal model developed in this paper does not rely on the availability of a specification for a program or component. Hence, in the discussion that follows, and in the description of Frankl and Weyuker's model of subdomain-based test adequacy criteria, the specification is ignored.

[3] An example of such a criterion is *statement coverage*.

In terms of subdomain-based criteria, they typically select test cases from fewer than 100% of the subdomains induced by the criterion they are using. It is therefore useful to extend the definition of adequacy in a way that accounts for the percentage of subdomains a test set covers:[4]

Definition (n% C-adequate): Given criterion *C*, program $\mathcal{P}$, and component *M*,

1. A test set $T_{\mathcal{P}}$ for $\mathcal{P}$ is *n% C-adequate* if $T_{\mathcal{P}}$ contains at least one test case from *n* percent of the subdomains in $\mathcal{SD}_C(D_{\mathcal{P}})$.
2. A test set T_M for *M* is *n% C-adequate* if T_M contains at least one test case from *n* percent of the subdomains in $\mathcal{SD}_C(D_M)$.

Corollary 1: A test set is *C-adequate* if and only if it is *100% C-adequate*.

The previous section formally partitions $D_{\mathcal{P}}$ with D_M according to $\mathcal{P}$'s traversal of *M*. These partitions carry over to the subdomains induced by an applicable subdomain-based criterion *C* as follows:

Definitions:

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

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

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

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

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. Thus, there is a pairwise correspondence between each subdomain in $\mathcal{SD}_C(\textit{M-traverse}(D_{\mathcal{P}}))$ and a subdomain in $\mathcal{SD}_C(D_{\mathcal{P}})$, and between each subdomain in $\mathcal{SD}_C(\textit{M-bypass}(D_{\mathcal{P}}))$ and a subdomain in $\mathcal{SD}_C(D_{\mathcal{P}})$. In addition, each subdomain induced by criterion *C* on component *M* is partitioned into its $\mathcal{P}$-*relevant* subset and its $\mathcal{P}$-*irrelevant* subsets, with a similar pairwise correspondence established with subdomains in $\mathcal{SD}_C(D_M)$. Note also 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 $\mathcal{SD}_C(\mathcal{P}\textit{-relevant}(D_M))$.

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 $\mathcal{SD}_C(\mathcal{P}\textit{-relevant}(D_M))$.

Note that although it is $\mathcal{P}$ that is being tested in integration testing, the criterion *C* must 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 $T_{\mathcal{P}}$.

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

Definition (n% C-adequate-for-$\mathcal{P}$*):* A test set T_M is *n% C-adequate-for-*$\mathcal{P}$ if it contains at least one test case each from *n* percent of the subdomains in $\mathcal{SD}_C(\mathcal{P}\textit{-relevant}(D_M))$.

Corollary 2: A test set T_M is *C-adequate-for-*$\mathcal{P}$ if and only if it is *100% C-adequate-for-*$\mathcal{P}$.

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

Corollary 3: A test set $T_{\mathcal{P}}$ is *C-adequate-on-M* if and only

[4] Of course, low percentages of coverage can hardly be considered "adequate testing" in any qualitative sense of the term, especially since experimental studies have shown that a test set must achieve coverage in the range of 90% or greater in order to be truly effective at detecting faults [7]. However, the definitions given here simply formalize standard characterizations of coverage that are used in testing practice.

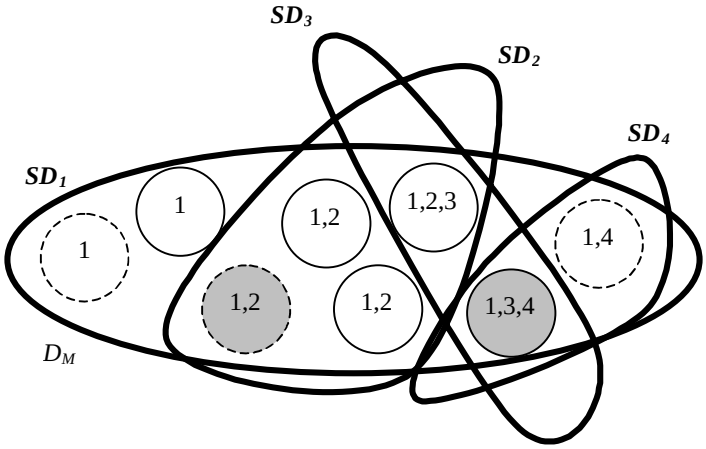

Fig. 3. Subdomain relationships for proof of Theorem 1.

if it is *100% C-adequate-on-M*.

There are a number of interesting consequences of these definitions that merit discussion. These consequences are stated as theorems, which are proven by appealing to hypothesized subdomain relationships that could be generated easily by programs and test sets specifically synthesized to exhibit such relationships.

The first two theorems concern the relationship between *C-adequate* and *C-adequate-for-*$\mathcal{P}$. In particular, a test set T_M that is *C-adequate* might not be *C-adequate-for-*$\mathcal{P}$, and vice versa. Theorem 1 is roughly analogous to Weyuker's Axiom of Anticomposition.

Theorem 1: There exist a program $\mathcal{P}$, a component M of $\mathcal{P}$, a criterion C, and a test set T_M for M such that T_M is *C-adequate* but not *C-adequate-for-*$\mathcal{P}$.

Proof: Consider Fig. 3, which depicts the input domain D_M and subdomains $\mathcal{SD}_C(D_M)$ appearing in Fig. 2. Suppose that T_M contains the test cases whose circles are shaded light gray. Suppose that the elements of D_M whose circles are dashed are the $\mathcal{P}$*-relevant* elements of D_M. Then T_M is *C-adequate* since it contains one test case drawn from each of the subdomains induced by C. However, SD_4 contains only one $\mathcal{P}$*-relevant* element, and that element is not a member of T_M. Hence, T_M is not *C-adequate-for-*$\mathcal{P}$.

Corollary 4: There exist a program $\mathcal{P}$, a component M of $\mathcal{P}$, a criterion C, and a test set T_M for M such that T_M is n_1*% C-adequate*, T_M is n_2*% C-adequate-for-*$\mathcal{P}$, and $n_1 > n_2$.

Theorem 2: There exist a program $\mathcal{P}$, a component M of $\mathcal{P}$, a criterion C, and a test set T_M for M such that T_M is *C-adequate-for-*$\mathcal{P}$ but not *C-adequate*.

Proof: Consider Fig. 4, which depicts the input domain D_M and subdomains $\mathcal{SD}_C(D_M)$ appearing in Fig. 2. Suppose that T_M contains the test cases whose circles are shaded light gray. Suppose that the elements of D_M whose circles are dashed are the $\mathcal{P}$*-relevant* elements of D_M. Then T_M is *C-adequate-for-*$\mathcal{P}$ since it contains one test case drawn from each of the $\mathcal{P}$*–relevant* subdomains induced by C. However, T_M contains no element from SD_3. Hence, T_M is not *C-adequate.*

Corollary 5: There exist a program $\mathcal{P}$, a component M of $\mathcal{P}$, a criterion C, and a test set T_M for M such that T_M is n_1*% C-adequate*, T_M is n_2*% C-adequate-for-*$\mathcal{P}$, and $n_1 < n_2$.

Corollary 6: Given a program $\mathcal{P}$, a component M of $\mathcal{P}$, a criterion C, and a test set T_M for M,

$$((\forall\ t \in T_M \ \ \exists\ D \in \mathcal{SD}_C(\mathcal{P}\text{-relevant}(D_M)) \bullet\ t \in D)$$
$$\text{and}\ \ (|\,\mathcal{SD}_C(\mathcal{P}\text{-relevant}(D_M))\,| \ < \ |\,\mathcal{SD}_C((D_M))\,|))$$
$$\rightarrow\ \ \neg\ C\text{-adequate}(T_M)$$

Corollary 6 captures the situation when there is a subdomain in $\mathcal{SD}_C((D_M))$ that has no $\mathcal{P}$*-relevant* members and no test cases in T_M. In this situation, T_M is not *C-adequate*.

The next two theorems concern the relationship between *C-adequate* and *C-adequate-on M*. In particular, a test set $T_{\mathcal{P}}$ that is *C-adequate* might not be *C-adequate-on-M*, and vice versa. As mentioned above, the criterion used to judge the adequacy of testing with M can be different from the criterion used to design the test set for $\mathcal{P}$. Hence, the theorems are stated for two different criteria $C\mathcal{P}$ (used to test $\mathcal{P}$) and CM (used to test M). This implies that there need not be a relationship between the subdomains in $\mathcal{SD}_{C\mathcal{P}}(M\text{-traverse}(D_{\mathcal{P}}))$ and the subdomains in $\mathcal{SD}_{CM}(\mathcal{P}\text{-relevant}(D_M))$. Theorem 3 is roughly analogous to

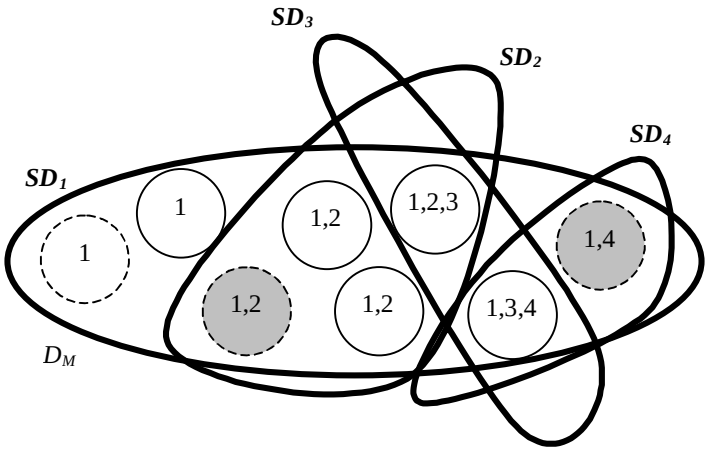

Fig. 4 . Subdomain relationships for proof of Theorem 2.

Weyuker's Axiom of Antidecomposition.

Theorem 3: There exist a program $\mathcal{P}$, a component *M* of $\mathcal{P}$, criteria $C\mathcal{P}$ and *CM*, and a test set $T_{\mathcal{P}}$ for $\mathcal{P}$ such that $T_{\mathcal{P}}$ is $C\mathcal{P}$*-adequate* but not *CM-adequate-on-M*.

Proof: Suppose there is more than one subdomain in $\mathcal{SD}_{C\mathcal{P}}(D_{\mathcal{P}})$ and that $T_{\mathcal{P}}$ contains at least one element from each of those subdomains. $T_{\mathcal{P}}$ is thus $C\mathcal{P}$*-adequate*. Suppose there is more than one subdomain in $\mathcal{SD}_{CM}(\mathcal{P}\text{-relevant}(D_M))$, and suppose that the test cases of $T_{\mathcal{P}}$ traverse *M* through only one of those subdomains. Then $T_{\mathcal{P}}$ is not *CM-adequate-on-M*.

Corollary 7: There exist a program $\mathcal{P}$, a component *M* of $\mathcal{P}$, criteria $C\mathcal{P}$ and *CM*, and a test set $T_{\mathcal{P}}$ for $\mathcal{P}$ such that $T_{\mathcal{P}}$ is n_1*% C*$\mathcal{P}$*-adequate*, $T_{\mathcal{P}}$ is n_2*% CM-adequate-on-M*, and $n_1 > n_2$.

Corollary 8: A test set $T_{\mathcal{P}}$ is not *CM-adequate-on-M* if $T_{\mathcal{P}}$ only contains elements from $M\text{-bypass}(D_{\mathcal{P}})$.

Theorem 4: There exist a program $\mathcal{P}$, a component *M* of $\mathcal{P}$, criteria $C\mathcal{P}$ and *CM*, and a test set $T_{\mathcal{P}}$ for $\mathcal{P}$ such that $T_{\mathcal{P}}$ is *CM-adequate-on-M* but not $C\mathcal{P}$*-adequate*.

Proof: Suppose that the number of subdomains in $\mathcal{SD}_{CM}(\mathcal{P}\text{-relevant}(D_M))$ is much smaller than the number of subdomains in $\mathcal{SD}_{C\mathcal{P}}((D_{\mathcal{P}}))$. Suppose that the number of test cases in $T_{\mathcal{P}}$ is exactly the number of subdomains in $\mathcal{SD}_{CM}(\mathcal{P}\text{-relevant}(D_M))$ and that each test case traverses *M* through a different subdomain. Then $T_{\mathcal{P}}$ is *CM-adequate-on-M* but not $C\mathcal{P}$*-adequate*.

Corollary 9: There exist a program $\mathcal{P}$, a component *M* of $\mathcal{P}$, criteria $C\mathcal{P}$ and *CM*, and a test set $T_{\mathcal{P}}$ for $\mathcal{P}$ such that T_M is n_1*% C*$\mathcal{P}$*-adequate*, T_M is n_2*% CM-adequate-on-M*, and $n_1 < n_2$.

APPLICATIONS OF THE MODEL

The formal model presented in the previous sections provides a foundation for studying and evaluating test adequacy for component-based systems. An important issue is the practical applicability of the model. It is one thing to argue that components must be tested with respect to the context of the larger systems in which they will be used. It is another thing to determine how this will be accomplished, especially in light of the fact that the component developer may be different from the system developer and that the two developments may proceed at widely separate points in time.

As mentioned in the introduction, the testing of component-based software can be viewed as both a unit testing problem for component *M*, and an integration testing problem for program $\mathcal{P}$ containing *M*. The unit-testing viewpoint requires the developer of *M* to test *M* with criterion *C* and to carry out the testing with a test set that is *C-adequate-for-*$\mathcal{P}$. The integration-testing viewpoint requires the developer of $\mathcal{P}$ to test $\mathcal{P}$ with a test set that is *C-adequate-on-M*. Both of these requirements are reasonable (and some would say barely sufficient) ways of systematically testing the quality of *M* and $\mathcal{P}$. If the test adequacy criteria being used are code coverage criteria, then satisfaction of these requirements (or the percentage of coverage achieved) can be checked easily and automatically with the aid of test coverage analysis tools.

However, there is a problem with the unit-testing viewpoint. If *M* is an off-the-shelf component produced by a supplier other than the developer of $\mathcal{P}$, a practical question arises as to how adequacy of the testing of *M* is to be assured and/or determined. Even with a way of

determining the adequacy of the testing of *M* (e.g., using a technique such as the one described by Devanbu and Stubblebine [3]), as *M* is used within more and more programs, the unit-testing viewpoint could require more and more tests to be run on *M*.

Another approach to studying the applicability of the model is to consider the different scenarios that will be encountered in the process of testing component-based software. These scenarios involve both the testing of individual components as well as the testing of the systems that are composed from multiple components. The scenarios below are described in terms of actions that need to be performed by "the developer" or "the tester" of a component; of course, this developer (tester) may actually be a large team of developers (testers) within a software development organization.

Component testing prior to deployment: The developer of a new component must thoroughly test the component prior to deploying it within a larger system. This scenario is complicated by the fact that the developer will not be able to predict all of the ways in which the component will be used by other components. This complication is captured in the definition of *C-adequate-for-*$\mathcal{P}$.

Testing an integrated system: A developer desiring to create a new system by integrating a collection of components in a particular way must test the new configuration prior to deployment. This scenario is complicated by the fact that it may not be convenient to subject some of the components to a testing process, since the components may only be available as black-box binaries and/or may be in use within other deployed systems. The test requirements for this scenario are captured in the definition of *C-adequate-on-M*.

Testing in the presence of system evolution: A tester may desire to regression test a component or system for which he or she is responsible whenever another developer has installed a new version of some other constituent component of the system. This scenario is complicated by the fact that the replacement can happen dynamically, without the knowledge of the affected tester, meaning that a previous round of adequate testing may no longer be adequate, from either the unit-testing viewpoint or the integration-testing viewpoint.

Testing in the presence of new features: A tester may desire to regression test a component or system for which he or she is responsible whenever another developer has changed or enhanced the functionality of some other constituent component of the system. This scenario is a variant of the previous scenario, with the additional complication that the affected tester may have no knowledge about the changes in functionality to expect.

Non-functional testing: A tester may desire to perform various kinds of non-functional testing on a system, such as performance testing, stress testing and load testing. This scenario is complicated by the fact that such testing can interfere with the ongoing use of the system's constituent components by other applications.

CONCLUSION

This paper has described a formal model of test adequacy for component-based software. The model is captured in the definitions of *C-adequate-for-*$\mathcal{P}$ and *C-adequate-on-M*. The model applies to unit testing of individual components and to integration-testing of compositions of components.

The work described in this paper is part of a broader study of the problem of validating distributed component-based software. Many of the characteristics of distributed component-based software complicate and even limit the ways in which they can be tested prior to deployment.

The notion of test adequacy can be viewed ultimately as a tool that attempts to produce a high degree of fault detection in a component. Furthermore, high levels of adequacy or coverage may go a long way towards identifying faults in the component, independently of the extent to which testing is carried out in the context of larger programs using the component. This possibility has been born out in experimental studies of coverage criteria, such as the study of Hutchins et al. [7]. However, more extensive empirical studies are needed that are targeted to evaluating the ways in which relationship between a component and the programs that use it affect the fault detecting ability of a test adequacy criterion.

There are a number of directions for future work on component-based software testing. For instance, it would be fruitful to study how a component-based viewpoint affects the different relationships between test adequacy criteria that have been studied in the testing literature, including the traditional *subsumption* relation as well as the different relations studied by Frankl and Weyuker [6].

Additionally, the model defined in this paper is relevant to *architectural-level testing* of a software system, in which a formal architecture description of the system is used to guide integration testing of the system [10]. The model is relevant as well to *selective regression testing*, whereby test cases are selected from a regression test suite for the system according to the changes that are made to create each new version of the system [11,12]. In both cases, a component-based orientation would be helpful for ensuring the adequacy of the testing that is carried out.

Finally, the model can be used to study the relationship between component-based test adequacy and the reliability of component-based software. A good starting point for such a study would be the reliability framework described by Frankl et al. [5].

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973. This effort was also sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air

Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

REFERENCES

[1] K. Brockschmidt, *Inside OLE*. Redmond, WA: Microsoft Press, 1995.

[2] D. Chappell, *Understanding ActiveX and OLE*. Redmond, WA: Microsoft Press, 1996.

[3] P. Devanbu and S.G. Stubblebine, "Cryptographic Verification of Test Coverage Claims", *Proc. Sixth European Software Engineering Conference/Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, 1997.

[4] J. Feiler and A. Meadow, *Essential OpenDoc*. Reading, MA: Addison-Wesley, 1996.

[5] P. Frankl, D. Hamlet, B. Littlewood, and L. Strigini, "Choosing a Testing Method to Deliver Reliability", *Proc. 19th International Conference on Software Engineering*, Boston, MA, pp. 68–78, 1997.

[6] 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.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria", *Proc. 16th International Conference on Software Engineering*, Sorrento, Italy, pp. 191–200, 1994.

[8] JavaSoft, "JavaBeans 1.0 API Specification", Sun Microsystems, Inc., Mountain View, CA version 1.00-A, December 4 1996.

[9] D.E. Perry and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming", *Journal of Object-Oriented Programming*, vol. 2, no. 5, pp. 13–19, 1990.

[10] 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.

[11] D.S. Rosenblum and E.J. Weyuker, "Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies", *IEEE Transactions on Software Engineering*, vol. 23, no. 3, pp. 146–156, 1997.

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

[13] J. Siegel, *CORBA Fundamentals and Programming*. New York, NY: Wiley, 1996.

[14] E.J. Weyuker, "Axiomatizing Software Test Data Adequacy", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, pp. 1128–1138, 1986.