Merging components and testing tools: The Self-Testing COTS Components (STECC) Strategy*

Sami Beydeda, Volker Gruhn
University of Leipzig
Department of Computer Science
Applied Telematics / e-Business
Klostergasse 3
04109 Leipzig, Germany
{sami.beydeda, volker.gruhn}@informatik.uni-leipzig.de

Abstract

Development of a software system from existing components can surely have various benefits, but can also entail a series of problems. One type of problems is caused by a limited exchange of information between the developer and user of a component. A limited exchange and thereby a lack of information can have various consequences, among them the requirement to test a component prior to its integration into a software system. A lack of information cannot only make test prior to integration necessary, it can also complicate this tasks.

This paper proposes a new strategy to testing components and making components testable. The basic idea of the strategy is to merge components and testing tools in order to make components capable of testing their own methods. Such components allow their thorough testing without disclosing detailed information, such as source code. This strategy thereby fulfills the needs of both the developer and user of a component.

1 Introduction

A major trend of recent years in software engineering is that of component-based development. The underlying idea of component-based development is to construct a software system from existing components instead of programming the software system from scratch.

Quality assurance, including testing, conducted in development and use of a component can be considered according to [13, 12] from two distinct perspectives. These perspectives are those of the *component provider* and *component user*. The component provider corresponds to the role of the developer of a component and the component user to that of a client of the component provider, thus to that of the developer of a system using the component.

The use of components in the development of software systems can surely have several benefits, but can also introduce new problems. Such problems concern, for instance, testing of components. The component provider and component user need to exchange various types of information during the development of the component itself and also during the development of a system using the component. However, exchange of such information can be limited due to various reasons and both the component provider and component user can face a lack of information. Such a lack of information might cause various problems which in turn might require that tests have also to be carried out by the component user. This contradicts to the believe that a component thoroughly tested by the component provider does not need to be retested by the component user. Such a lack of information might not only obligate the component user to test a component, it might also complicate component user's tests. An important example for this is a lack of source code for test case generation purposes.

A number of approaches have been proposed in the literature which aim at either avoiding a lack of information or allowing testing in spite of such a lack. These approaches, however, suffer from certain limitations and we believe that a novel type of approaches is required to address the testing of component appropriately. This paper summarizes our results. Sec. 2 describes the limited exchanged of information among the component provider and component user, and the potential problems due to a lack of information. Sec. 3 explains approaches proposed to tackle the problems in testing components together with their limitations. Sec. 4 intro-

*The chair for Applied Telematics / e-Business is endowed by Deutsche Telekom AG.

duces the *self-testing COTS components* (STECC) method, a novel strategy for testing components and making components testable. Sec. 5 finally gives our conclusions and shows the directions of our current and future research.

2 Exchange of information in development of and with components

2.1 Information relevant for development of components and component-based systems

The component provider and component user generally need to exchange information during the various phases of developing the component and a component-based system. The development of a component, if the component is itself developed non-component-based and thus is not itself a component-based system, usually consists of the typical phases of software development. Software development usually includes the phases of requirements analysis and definition, system and software design, implementation and unit testing, integration and system testing, and operation and maintenance, if it is conducted according to the waterfall model or a derivative of it [19]. The single phases might be named differently depending on the actual software process model, which, however, does not affect the following explanations. During some of these phases, the component provider needs to exchange information with the component user. Such phase are, for instance:

> *Requirements analysis and definition.* The phase of requirements analysis and definition obviously necessitates information concerning the capabilities and conditions the component needs to satisfy according to the component user's expectations.
>
> *Operation and maintenance.* The phase of operation and maintenance usually requires on the one hand information for the operation the component by the component user, i.e. assembly of the component with others, and on the other hand information for the maintenance of the component by the component provider.

Even though a waterfall model-based software process has been assumed so far, similar patterns of information exchange can also be identified for other software process models and the results obtained also apply for them.

Information flow between component provider and component user does not only occur during the development of a component. Information needs often also to be exchanged between the two roles during the development of a component-based system using the component. The explanations in the following assume a concrete process model for component-based development as described in [19]. However, the explanations are also valid for other process models for component-based development, since the phases in which information is exchanged between the two roles usually have their counterparts also in those models. The process model for reuse-oriented software development using components includes six phases as described in [19]. These phases are requirements specification, component analysis, requirements modification, system design with reuse, development and integration, and system validation. During some of these phases, a bidirectional information flow between the component provider and component user can be observed. Examples of these phases are:

> *Component analysis.* The component analysis phase necessitates information supporting identification of components available from the various sources, their analysis with respect to certain criteria, and finally selection of the component being most suitable for the component-based system to be developed.
>
> *Development and integration.* The phase of development and integration can also require technical information which the component user needs to obtain from the component provider during the phase. Such technical information might concern the interfaces of the component, the required middleware etc.
>
> *System validation.* The phase of system validation is also a phase in which information often needs to be exchanged between the two roles. Such an exchange of information might concern white-box test cases generated by the component provider, meta-information supporting the component user in testing etc.

The above list of phases in component-based development requiring exchange of information between the component provider and component user is not necessarily comprehensive. However, the aim of the above list is only to show that interaction between the two roles takes place throughout the lifecycles of components and component-based systems, and the flow of information is not merely one way [17, 18].

2.2 Factors affecting exchange of information

Various factors impact the exchange of information between the component provider and component user. The information requested by one role and delivered by the other can differ in various aspects, if it is delivered at all. It can differ syntactically insofar that it is, for instance, delivered in the wrong representation and it can also differ semantically in that it, for instance, is not in the abstraction level needed. The differences might be due to various factors and one of these factors is the organizational relation between the two roles.

A component can be distinguished with respect to the organizational relational between the component provider and component user according to [6] in the categories *independent commercial item*, *special version of commercial item*, *component produced by contract*, *existing component from external sources* and *component produced in-house*. Information exchange among the component provider and component user depends, as one factor, on the category into which the component is classified regarding to the organizational relation between the two roles.

As the one extreme, the component can be a commercial item. The quality of information exchange between component provider and component user is in comparison to the other cases often the worst. There are various reasons for this, such as the fact that the component provider might not know the component user due to an anonymous market. In such a case, the component provider can base own development tasks on assumptions and can deliver only information to the component user supposed to be needed. Furthermore, the component might be used by several component users and the component provider might decide to only consider the needs of the majority of them. The specific needs of a single component user might than be ignored. Finally, the component provider might not disclose detailed technical information even if needed by the component user to avoid that another component provider receives this information. The component provider might decide to only make the information available which respects intellectual property and retains competition advantages.

As the other extreme, the component can be produced in-house. The quality of information exchange between component provider and component user is in comparison to the other cases often the best. One of the reasons for this can be, for instance, the fact that the component is developed in the same project in which it is assembled. The exchange of information in both directions, from the component provider to the component user and the reverse direction, can take place without any differences in the requested and delivered information. Furthermore, the component provider and component user are roles, they can also be played by the same person, if the component is used in the same project in which it is developed. Information would even not be necessary to be exchanged.

2.3 Problems due to a lack of information

2.3.1 Context-dependent development of a component

One type of information required for the development of a component is that indicating the application environment in which it will later be used. Such information, however, might not be available so that the component provider might develop the component on the basis of assumptions concerning the application environment. The component is then explicitly designed and developed for the needs of the assumed application environment, which, however, might not be the one in which the it will be actually used. Even if the component is not tailored to a certain application environment but constructed for the broader market, the component provider might unconsciously assume a certain application environment and its development might again become context-dependent. A consequence of context-dependent development of a component can be that testing is also conducted context-dependently. A component might work well in a certain application environment and can exhibit failures in another [24, 22].

One of the reasons for context-dependent development of a component is often the component provider's lack of information concerning the possible application environments in which the component might be used later. Tests conducted by the component provider might also be context-dependent and a change of application environment, which might be due to reuse of the component, generally requires additional tests in order to give sufficient confidence that the component will behave as intended also in the new application environment. Additional tests are required even if often contrary claimed that components frequently reused need less testing, e.g. [20]. Moreover, a component reused in a new application environment needs to be tested irrespective of its source. A component produced in-house does not necessarily need less testing for reuse purposes than a component being an independent commercial item [24].

2.3.2 Insufficient documentation of a component

Development of a component-based system generally requires detailed documentations of the components which are to be assembled. Such documentations are usually delivered together with the respective components and each of them needs to include three types of information related to the corresponding component:

Functionality. The specification of the component functionality gives a description of the functions of that component, i.e. its objectives and characteristics actions, to support an user in solving a problem or achieving an objective.

Quality. The specification of component quality can address, for instance, of quality assurance, particularly including testing techniques, applied, metrics used to measure quality characteristics and their values.

Technical requirements. The specification of the technical requirements of a component needs to address issues such as the resources required, the architectural style assumed, the middleware used.

Documentation delivered together with a component and supposed to include specifications of the above outlined aspects might, however, be insufficient for development of a component-based system. The various types of information provided by the documentation can deviate from those expected syntactically as well as semantically, and it can even be incomplete. This problem can be viewed from two different perspectives. On the one hand, it can be considered as a problem due to a lack of information. The component provider might be suffering from a lack of information and might therefore not provide the information as documentation actually needed by the component user. On the other hand, it can be considered as a reification of a lack of information. Instead assuming the component provider as suffering from a lack of information while developing the component and assembling its documentation, the component user is assumed as suffering from such a lack while developing a component-based system using the component. Insufficient documentation is according to the latter perspective not the effect of a lack of information but its reification. However, the subtle differences of these perspective are not further explored.

Both prototyping and familiarization require that the component under consideration is executed, which is also the main characteristic of testing. In fact, both can be considered as testing, if the term of testing is defined more generally without assuming that testing is a quality assurance action. The objectives of both are not necessarily related to quality assurance, but are in principle to obtain information which is not delivered as part of the documentation. Furthermore, components delivered with insufficient documentation might also required testing in its original sense, particularly if the documentation does not include information concerning quality assurance conducted. Even if the documentation includes such information, quality assurance conducted might not be sufficient for the application environment in which the component will be used. In such cases, the component usually needs to be retested also by the component user, since the component user is from the viewpoint of the end-user responsible for the quality of the component-based system and the component user's reputation depends on its quality [24].

3 Existing approaches

3.1 Overview of the approaches discussed

The approaches considered in the following are solely those which take into account a lack of information, even if not explicitly mentioned, and which can be applied by the component user. Approaches have been considered in particular which aim at tackling the problems caused by such a lack. These approaches do not address the cause but rather tackle the potential difficulties which might be encountered when testing components.

The discussion does not consider approaches to testing components which are not concerned by a lack of information. Such approaches are those which do not require source code access or information not available to the component user, such as black-box approaches. The discussion also does not consider approaches to testing components which either are explicitly intended to be used by the component provider or require information without giving a solution to tackle a possible lack of that information to the component user.

Further note that the approaches discussed are not necessarily described entirely. Only those aspects of an approach are described which are relevant in this context and other aspects, which obviously might be important in other discussions, are omitted for the sake of brevity.

A more comprehensive overview of approaches to testing COTS components can be found in [4, 5].

3.2 Built-in testing approaches

A component can contain test cases or can possess facilities capable of generating test cases which can be accessed by the component user or which the component can use to test itself and its own methods. The corresponding capabilities allowing this are called built-in testing capabilities, which are one type of the approaches addressing the effects of a lack of information. The component user thus does not need to generate test cases and difficulties which the component user would otherwise face thus can in principle not complicate the component user's test.

A built-in test approach can be found in [23]. A component can operate according to this approach in two modes, namely in a *normal mode* and a *maintenance mode*. In the normal mode, the built-in test capabilities are transparent to the component user and the component does not differ from other, non-built-in testing enabled components. In the maintenance mode, however, the component user can test the component with the help of its built-in testing features. The component user can invoke the respective methods of the component, which execute the test, evaluate autonomously its results, and output a test summary. The authors describe a generic technical framework for enhancing a component with built-in tests. One of the few assumptions is that the component is implemented as a class. Under this assumption, it is suggested to implement built-in testing by additional methods which either contain the test cases to be used in hard-wired form or are capable of generating them. The integral benefit of such an implementation is that the methods for built-in testing can be passed to subclasses by inheritance.

A built-in testing approach is also proposed in [15, 21,

8, 2, 3]. Even though this approach is called by its authors a self-testing approach, it is referred to for the sake of consistency as a built-in testing approach. The features characterized by the authors as self-testing significantly differ from those characterized as self-testing in the context of the STECC strategy and resemble those of the above approach. The approach and that explained share several properties. Besides various modes of operation, a component is assumed to be implemented using object-oriented languages, Java in particular. Built-in testing is implemented by additional methods. Each component method testable by built-in testing capabilities possesses a testing method as counterpart which invokes it with predefined arguments. An oracle is implemented by the means of component invariant, method pre- and postcondition. Invariants, pre- and postconditions are determined based on the specification of the component and are embedded by the component provider in the source code of the component. The functionality necessary to validate them and other functionality, such as that necessary for tracing and reporting purposes, is implemented by a framework, which technically requires that the component, or more clearly the main class of the component, implements a certain interface. Similar to the above approach, the built-in testing capability can be passed to subclasses by inheritance. The authors propose to measure test completion by the means of fault injection, which is, however, not feasible in this context, since this requires source code access, which the component user does not have. The component user therefore has to assume that the built-in test are sufficient.

Another built-in test approach, the *component+* approach, can be found in [14, 1]. A shortcoming of the last built-in testing approach is that test cases or a description of their generation need to be stored within the component. This can increase the resource consumption of the component, which, particularly taking into account that the built-in testing capabilities of a component is often required only once for deployment, can be an obstacle for its use. To avoid this shortcoming, the authors define an architecture consisting of three types of components, namely *BIT components*, *testers*, and *handlers*. The BIT components are the built-in testing enabled components. These components implement certain mandatory interfaces. Testers are components which access to the built-in testing capabilities of BIT components through the corresponding interfaces and which contain the test cases in a certain form. In the above approach, a built-in testing enabled component also encompasses the functionality of the testers. Here, however, they are separated with the benefit that they can be developed and maintained independently, and that they do not increase resource requirements of BIT components in the operational environment. Finally, handlers are components in this architecture which do not contribute to testing, but can be required, for instance, to ensure recovery mechanisms in the case of failures.

The built-in testing approaches presented do not restrict the tests which can be conducted insofar that they are not constrained to black-box testing. Built-in testing approaches which are constrained to black-box testing, such as those in [9, 10, 11] and [16], are not discussed, since black-box testing does not necessarily require provisions by the component provider. Assuming a specification is given, the component user can obtain the appropriate test cases and test the component in principle without the component provider's support. Black-box built-in testing capabilities undoubtedly have the potential of simplifying component user's tests by improving component testability, but the corresponding tasks can usually also be accomplished by the component user.

3.3 Limitations of existing approaches

The built-in testing approaches in [23], [15, 21, 8, 2, 3] and [14, 1] aim at tackling difficulties in testing components caused by such a lack, difficulties in test case generation in particular. They can simplify component user's test insofar that the component user might not need to generate test cases, but they might be in some cases not appropriate. The reasons include:

> Firstly, the built-in testing approaches explained are static in that the component user cannot influence the test cases employed in testing. A component which is built-in testing enabled according to one of the approaches explained either contains a predetermined set of test cases or the generation, even if conducted on-demand during runtime, solely depends on parameters which the component user cannot influence. Specifically, the component user cannot specify the adequacy criterion to be used for test case generation. However, the component user might wish to test all components to be assembled with respect to an unique adequacy criterion. Built-in testing approaches, at least those described, do not allow this.

> Secondly, built-in testing approaches using a predefined test case set generally require considerable storage. Specifically, large components with high inherent complexity might require a large set of test cases for their testing. A large set of test cases obviously requires a substantial amount of storage which, however, can be difficult to provide taking into account the storage required in addition for execution of large components. This is also the case if test cases are stored separately from the component, such as proposed by component+ approach.

Existing approaches generally do not ensure that tests are conducted as required by the component user. They have the benefit that the component user does not need to generate test cases, since test cases are provided by a built-in testing enabled component itself, and thus does not need access to detailed information. However, test cases provided by such a component might not be adequate with respect to requirements of the component user.

4 STECC strategy in component testing

4.1 Underlying idea

The existing approaches do not address the needs of the component user appropriately as shown by the previous discussion. The main drawback of the approaches of the second category is their static nature. They are inflexible insofar that test cases provided by a built-in testing enabled component are static, even if they are not specified by a concrete set but by a generation procedure. In both forms, the component provider thus needs to make some assumptions concerning the requirements of the component user, which again might be wrong or inaccurate. The strategy proposed obviates such assumptions. The underlying idea of the strategy proposed is to augment a component with functionality of analysis and testing tools. A component augmented accordingly is capable of conducting some or all activities of the component user's testing processes. The strategy is thereby called the *self-testing COTS components* (STECC) strategy [4].

The STECC strategy meets the demands of both the component provider and component user. It has two main benefits:

Firstly, the component provider does not need to disclose detailed information. This does not mean that such information is not processed during tests conducted by the component user. Such information is either available to the component in an encapsulated form or is generated on-demand by it during testing. In both cases, the corresponding information is not accessible to the component user, but is nevertheless processed by the component. As a consequence, the information processed can be very fine-grained. For instance, source code, which the component provider would not disclose to the component user, can be packaged in a certain form into the component and can be used for test case generation purposes. Even if the test case generated are returned to the component user, source code still remains hidden.

Secondly, the component user can parameterize tests as required. A component augmented according to the STECC strategy possesses functionality of an analysis and testing tool and provides the component user the full functionality of such tools. As an example, the component user does not need to test a component according to the adequacy criterion anticipated by the component provider. The component user can generate test cases exactly as in the same case as having access to e.g. the source code of the component and using a separate analysis and testing tool for test case generation.

4.2 Impact on the component user's testing processes

The STECC strategy, if considered for testing purposes by the component user, impacts several activities of a component user's testing process. In particular, the single activities of a typical testing process are impacted as follows:

Test plan definition. Some of the decisions made during definition of test plans are addressed by conventions of the STECC strategy and its actual implementation. Such decisions concern, for instance, the target component model and framework. The actual implementation might assume, for instance, the Enterprise JavaBeans component model and framework [7]. Another decision can concerns the technique used for analysis and testing purposes, such as the test case generation technique. Related to test case generation, the actual implementation of the STECC strategy might also prescribe a certain type of completion criterion used to measure testing progress.

Test case generation. Generation of test cases is the integral constituent of self-testability as assumed by the STECC strategy. Test case generation needs to be entirely conducted by the self-testing component due to a lack of its source code and necessary white-box information to the component user, who therefore cannot carry out this task. Various types of test case generation techniques can be embedded in the actual implementation of the STECC strategy. Test cases as generated in this context do not necessarily include expected results. This depends on the fact whether the specification of the component is available in a form in which it can automatically processed.

Test driver and stub generation. The component user does not need to generate test drivers for component method testing. The actual implementation of the STECC strategy usually includes the necessary provisions to execute the methods of the component considered. Stubs, however, might be necessary if the method to be tested needs to invoke those of absent components. A component can often be embedded in a wide

variety of application contexts and the specific application context can therefore often not be anticipated. The component user needs either to provide the stubs or to embed the component in the target application context.

Test execution. The execution of the methods under consideration with generated test cases can also be conducted by the implementation of the STECC strategy. As one possibility for test execution, a dynamic technique can be used for test case generation purposes, which iteratively approaches to appropriate test cases and successively executes the method to be tested for this purpose. As another possibility for test execution, test cases generated can be stored and executed in a separate testing phase.

Test evaluation. The evaluation of tests needs either to be addressed by the component user or by the implementation of the STECC strategy. The reason for this is mainly the fact whether the specification or expected results are available to the implementation. In the case in which the specification or expected results are available, this task can be conducted by the implementation of the STECC strategy. Otherwise, expected results have to be determined and compared with those observed during and after test execution by the tester, i.e. component user.

5 Conclusions and future research

One of the factors complicating development of components and component-based system is a limited exchange of information between the component provider and component user. Both roles can face a lack of information relevant for their development tasks. A lack of information can have manifold consequences. A consequence is, for instance, the requirement to test a component prior to its integration into a system, even it has been tested by the component provider. Furthermore, a lack of information can not only make tests mandatory, it can even complicate testing.

This paper has introduces the self-testing COTS components (STECC) strategy. The basic idea of this strategy is to augment a component with functionality similar to that of analysis and testing tools. The idea of this strategy is allow the component user to test the component with respect to information which is not directly accessible to the component user. The information is either generated by the component itself on-demand or is encapsulated in it. In both cases, the information is transparent to the component user and is processed the component itself for testing purposes without disclosing it.

Our research in the future will focus on developing a method giving guidance to both the component provider and component user. The method will guide the component provider in augmenting a component. For this purpose, a framework will be developed implementing the necessary functionality of analysis and testing tools. The component user will be guided by the method in carrying out tests.

We would like to invite the reader to participate in an open discussion started to gain a consensus concerning the problems and open issues in testing components. The contributions received so far can be found at `http://www.lpz-ebusiness.de` and new contributions can be made by email to `sami.beydeda@informatik.uni-leipzig.de`.

References

[1] C. Atkinson and H.-G. Groß. Built-in contract testing in model-driven, component-based development. In *ICSR Workshop on Component-Based Development Processes*, 2002.

[2] B. Baudry, V. L. Hanh, J.-M. Jezequel, and Y. L. Traon. Trustable components: Yet another mutation-based approach. In *Symposium on Mutation Testing (Mutation)*, pages 69–76, 2000.

[3] B. Baudry, V. L. Hanh, J.-M. Jezequel, and Y. L. Traon. Trustable components: Yet another mutation-based approach. In W. E. Wong, editor, *Mutation testing for the new century*, pages 47–54. Kluwer Academic Publishers, 2001.

[4] S. Beydeda. *The Self-Testing COTS Components (STECC) Method*. PhD thesis, Universität Leipzig, Fakultät für Mathematik und Informatik, 2003.

[5] S. Beydeda and V. Gruhn. State of the art in testing components. In *International Conference on Quality Software (QSIC)*. IEEE Computer Society Press, 2003.

[6] D. Carney and F. Long. What do you mean by COTS? – finally, a useful answer. *IEEE Software*, 17(2), 2000.

[7] L. G. DeMichiel. Enterprise javabeans specification, version 2.1. Technical report, Sun Microsystems, 2002.

[8] D. Deveaux, P. Frison, and J.-M. Jezequel. Increase software trustability with self-testable classes in java. In *Australian Software Engineering Conference (ASWEC)*, pages 3–11. IEEE Computer Society Press, 2001.

[9] S. Edwards. A framework for practical, automated black-box testing of component-based software. In *International ICSE Workshop on Automated Program Analysis, Testing and Verification*, 2000.

[10] S. H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97–111, 2001.

[11] S. H. Edwards. Toward reflective metadata wrappers for formally specified software components. In *OOPSLA Workshop Specification and Verification of Component-Based Systems (SAVCBS)*, 2001.

[12] M. J. Harrold. Testing: A roadmap. In *The Future of Software Engineering (special volume of the proceedings of the International Conference on Software Engineering (ICSE))*, pages 63–72. ACM Press, 2000.

[13] M. J. Harrold, D. Liang, and S. Sinha. An approach to analyzing and testing component-based systems. In *International ICSE Workshop Testing Distributed Component-Based Systems*, 1999.

[14] J. Hörnstein and H. Edler. Test reuse in cbse using built-in tests. In *Workshop on Component-based Software Engineering, Composing systems from components*, 2002.

[15] J.-M. Jezequel, D. Deveaux, and Y. L. Traon. Reliable objects: Lightweight testing for oo languages. *IEEE Software*, 18(4):76–83, 2001.

[16] E. Martins, C. M. Toyota, and R. L. Yanagawa. Constructing self-testable software components. In *International Conference on Dependable Systems and Networks (DSN)*, pages 151–160. IEEE Computer Society Press, 2001.

[17] M. Morisio, C. Seaman, A. Parra, V. Basili, S. Kraft, and S. Condon. Investigating and improving a COTS-based software development process. In *International Conference on Software Engineering (ICSE)*, pages 32–41. ACM Press, 2000.

[18] M. Morisio, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon. COTS-based software development: Processes and open issues. *The Journal of Systems and Software*, 61(3):189–199, 2002.

[19] I. Sommerville. *Software Engineering*. Addison-Wesley, sixth edition, 2001.

[20] C. Szyperski. *Component Software Beyond Object Oriented Programming*. Addison-Wesley, 1998.

[21] Y. L. Traon, D. Deveaux, and J.-M. Jezequel. Self-testable components: from pragmatic tests to design-to-testability methodology. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 96–107. IEEE Computer Society Press, 1999.

[22] J. Voas. COTS software: The economical choice? *IEEE Software*, 15(2):16–19, 1998.

[23] Y. Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 186–189. IEEE Computer Society Press, 1999.

[24] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.