State of the art in testing components[*]

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

Abstract

The use of components in development of complex software systems can surely have various benefits. Their testing, however, is still one of the open issues in software engineering. Both the developer of a component and the developer of a system using components often face the problem that information vital for certain development tasks is not available. Such a lack of information has various consequences to both. One of the important consequences is that it might not only obligate the developer of a system to test the components used, it might also complicate these tests. This article gives an overview of component testing approaches that explicitly respect a lack of information in development.

1 Introduction

Quality assurance, including testing, conducted in development and use of a component can be considered according to [13] 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 difficulties 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. Limited exchange of information among the component provider and component user is to our opinion the main reason why testing of components is a problem of its own and needs to be considered as such.

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. Sec. 2 describes the limited exchanged of information among the component provider and component user, and the potential problems due to a lack of such information. Sec. 3 explains approaches proposed to tackle the problems in testing components. Sec. 4 finally gives our conclusions.

2 Problems due to a lack of information

2.1 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

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

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 [25, 23].

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. [21]. 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 [25].

2.2 Insufficient documentation of a component

Development of a component-based system generally requires detailed documentations of the components which are to be assembled. The corresponding documentation 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.

In [18, 19], the authors propose a process model for COTS-based development which includes a specific activity to tackle problems due to insufficient documentation. The process model encompasses an activity called COTS components familiarization in which components selected before are actually used to gain a better understanding of the functionalities available, their quality, and architectural assumptions. Importance of such an activity depends on the quality of the component documentation and decreases with an increase of that.

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 principally 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 [25].

2.3 Component user's dependence on the component provider

Context-dependent development and insufficient documentation of a component are two problems resulting by a lack of information which often obligate the component user to test a component before its use. The component user can encounter after the tests are finished and a failure is revealed another problem also due to a lack of information. The problem which the component user can encounter is that of dependence on the component provider. The fault causing the failure often cannot be removed by the component user, since the component user might not have the software artifacts required for isolating and removing the fault. Such artifacts include documentation, test plans and source code of the component, which is usually the case for COTS components. Even if the artifacts required are available to the component user, debugging might be significantly difficult or even impossible due to missing expertise. Missing expertise and insight of the component user might entail significant debugging costs which can even offset the benefits gained by using the component. The component user thereby has often to rely on the component provider for maintenance and support, which the component user, however, might not be able to influence, which gives an uncertainty for the future.

The problem of dependence on the component provider can even aggravate if the component is not maintained as demanded by the component user, or if even the component provider decides to cease support and maintenance or goes bankrupt [25, 23]. The possible financial effects of such an event is shown in [23] on a simple example. It has been suggested to create escrow agreements and protective licensing options for the relevant artifacts of a component to avoid the problems in the above case. Even if the component provider accepts such an agreement, the problems due to missing expertise can still hinder the component user from carrying out the corresponding tasks.

Difficulties for the component user by a dependence on the component provider are not necessarily restricted to maintenance and support. Generally, several of the decisions taken by the component provider during the lifecycle of the component also impact its use as part in a component-based system. Other problems which can occur due to the dependence on the component provider can be found in [18, 19, 23].

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. The approaches considered can be classified in two categories, which are:

> Firstly, approaches which aim at avoiding a lack of information. These approaches address the cause of such a lack so that difficulties in testing components are not be entailed.
>
> Secondly, approaches 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.

Note that the approaches considered do not include those which do not respect a lack of information and 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.

3.2 Approaches addressing the cause of a lack of information

3.2.1 Component meta-data approach

An approach which can be used to avoid a lack of information can be found in [20]. The underlying idea of this approach is to avoid such a lack and thus difficulties in testing components by augmenting a component with the information, which the component user might require for analysis and testing tasks, or capabilities for its generation. The authors suggest to technically enhance a component so that it can process information requests and deliver the information required.

The information which the component user can retrieve from a component enhanced as suggested is represented in form of *meta-data*. It generally concerns aspects of the component relevant for analysis and testing tasks, but it is not respecified and the meta-data format can be extended if necessary. The meta-data available with a component might differ from that available with others and might also change due. Flexibility in providing the types of meta-data required is ensured with an open format which can be extended if necessary. The component provider can choose the information to be presented according to the needs of a particular component user or a group of component users. It can in principle be any artifact of the component's development. The three basic properties of meta-data is that it originates from the component provider, is packaged in a standard way with the component, and is processable by tools.

Important in this context is that the component provider can also augment the component with information supporting testing activities of the component user. An example of such an information can be a control flow graph represented in an abstract level. Thus, information sources, such as the source code of the component necessary for generating a control flow graph, do not need to be disclosed to the component user. The information which can be obtained from such a source can be directly delivered to the component user, who then does not require its source.

The meta-data approach can be considered as a generalization of introspection mechanisms available in most component models. However, they differ in some aspects. These aspects are:

> Firstly, introspection often only provide syntactical information. The typical information which can be obtained by introspection is, for instance, signatures of the public methods of a component. In contrast, the meta-data approach can also cover semantic information.
>
> Secondly, the information available through introspection is also often prespecified by the corresponding

component model. The meta-data approach does not have such a restriction.

A similarity to the introspection mechanism is that only an unidirectional information flow is supported. The meta-data approach solely supports the component provider in offering information to the component user.

A possible implementation of the meta-data framework is extending the component considered with two methods which provide the appropriate functionality to query the types of the meta-data available and retrieve a specific type of meta-data, respectively. Thus, the component provider is generally in charge of augmenting a component so that meta-data can be obtained, if necessary. The meta-data itself, however, does not necessarily need to be packaged to the component. For flexibility reasons, it can also be generated on-demand or stored remotely. The ultimate decision depends on factors such as the complexity in computing the meta-data, amount of the meta-data, and context dependence of the meta-data. Furthermore, a component user might not need all meta-data available, but can only be interested in a specific type of meta-data. Packaging all meta-data available to the component can increase its resource requirements.

3.2.2 Retro-components approach

Another approach which can be used to avoid a lack of information can be found in [16]. The strategy employed by this approach to avoid such a lack and thus difficulties in testing components is twofold. It consists of augmenting a component with firstly information concerning tests conducted by the component provider and indications for tests to be conducted by the component user, and secondly capabilities to gather information during tests of the component user. The authors suggest to technically enhance a component so that the component user can query the information provided by the component provider and collect relevant information during own testing activities.

A component enhanced in the suggested form, called *retro-component*, supports exchange of two types of information, which are the following:

> Firstly, the information exchanged can be static. A retro-component can offer information to the component user describing, for instance, the component provider's assumptions for tests conducted, adequacy criteria used and test histories. Static information can also embrace indications for further tests in form of recommendations for the component user.

> Secondly, the information can also be dynamic. A retro-component is capable of, for instance, computing adequacy of a test case set and collecting information describing its use. Computation of adequacy can specifically be conducted according to a white-box criterion, which obviates source code access to the component user for such a task.

Information gathered during test and use of a component can be delivered back to the component provider, which can be valuable for perpetual testing and further development. Thus, the retro-component approach supports to some extend, as the only approach discussed in this context, a bidirectional information flow.

Retrospection as provided by retro-component is similar to introspection widely found in component models. Both retrospection and introspection are mechanisms of exchanging information between the component provider and component user. However, two significant differences can be identified. These differences are:

> Firstly, introspection is static, whereas retrospection is dynamic. Retro-component can autonomously gather information during testing and operation of a component. Both the component provider and component user can benefit from this. The component provider can gain a better insight in the use of the component and the component user can carry out tasks which otherwise would require access to information not available.

> Secondly, introspection only facilitates an unidirectional information flow, retrospection a bidirectional information flow. A retro-component possesses the capability of gathering information during its test and operation at component user site, which can be delivered to the component provider. Such a flow of information does not occur in introspection.

From a technical point-of-view, the component provider is in charge of enhancing a component with retrospection. The component provider can use for this task predefined retrospection facilities, which provide a default retrospection behavior. The component is assumed in this context to be implemented using object-oriented languages. The predefined facilities consist of certain framework of classes, which can be integrated with the component under consideration by inheritance and implementing certain interfaces. The default implementation can operate in distinct modes, including the modes of test-time and run-time. Retrospection can be used during testing, i.e. in test-time mode, and during operation, i.e. in run-time mode, to access static information and to gather dynamic information. The information accessible can specifically be processed by external tools, such as test case generators.

3.2.3 Reflective wrapper approach

The approach described in [10] can also contribute to avoiding a lack of information and difficulties in component testing. The approach proposed mainly facilitates exchange of information concerning, besides the component's specification, the quality assurance actions conducted by the component provider. This approach differs from those presented at a technical level. The author suggests to support information flow from the component provider to the component user with the help of a certain technical architecture using wrappers.

Information flow is, similar to the meta-data approach, solely facilitated in only one direction, namely from the component provider to the component user. The component provider can package according to this approach information which is to be made available to the component user into the component which the component user can retrieve either in human-readable form or in a form processable by tools, such as browsers. The data representing the information packaged is also referred to in the context of this approach as *meta-data*, since it describes the component and how it behaves rather the information processed by it.

The main type of information to be made available through the corresponding mechanism is according to the author the specification of the component. This possibility is, however, not further elucidated in the following. The specification of a component is assumed to be available to the component user in a certain form, as some kind of description of the component's behavior has to be provided to the component user. Consideration is restricted to approaches permitting access to information which would otherwise not available at all. The author also mentions another type of information which can be exchanged through the proposed mechanism being more interesting in this context. This type of information is that concerning quality assurance conducted by the component provider. The component provider can augment the component with information indicating the specific actions used, which can be valuable to the component user in order to avoid repeating the actions already conducted and to assess the suitability of the component within the intended application environment.

The technical implementation of the reflective wrapper approach significantly differs from those of the two approaches explained. As its name suggests, the implementation is based on the concept of wrappers. A wrapper encapsulates the target component but is transparent to clients of it at the same time. It conforms to the specification of the target component by implementing the same interfaces and delegating, possibly after certain computations, clients' requests to the component. The functionality related to the reflection mechanism is implemented by additional methods by the wrapper. These methods return the requested information in form of meta-data, if it is available. The benefits of using wrappers is obvious, they can added and removed by the component user without access to the source code of the wrapped component. This is an important features, as the services provided by a wrapper are usually only required during development. Increased resource requirements can be avoided by removing wrappers after development has been finished. Similar to the two previous approach, the component provider is also here responsible of enhancing the component with the necessary capabilities. This is one of the similarities of the presented approaches to avoiding a lack of information.

3.2.4 Component test bench approach

The approach described in [6] can also contribute to avoiding a lack of information and thereby the difficulties in testing components. Similar to those already described, this approach also gives a possibility of augmenting a component with additional information which can be used for analysis and testing purposes. A difference to the other approaches is, however, that the information provided is constrained to a specific type. It is constrained to test specifications.

A test specification, as defined by the authors, describes implementations of a component, interfaces provided by each implementation, and concrete sets of test cases appropriate to an interface. The authors do not assume a specific component model so that a component might be implemented in different programming languages. Implementations in different programming languages might even coexist with each other and a specific operation can have multiple implementations. Furthermore, test specifications also support components offering several interfaces. A component can offer several interfaces, for instance, in order to provide several views to its operations depending on the application environment or for the purpose of compatibility to previous versions. A set of test cases is associated with each interface of a component. An element of such a set is called by the authors a *test operation*. A test operation defines the necessary steps for testing a specific method in one of the interfaces of a component.

Test specifications are formulated in the context of the approach proposed in XML. The benefits gained by using XML is that such specifications can be automatically processed by third-party tools once the syntax and semantics are known. Third-party tools can be applied to read, interpret and modify test specifications.

The concrete test inputs and expected test output packaged in a test operation can be determined as usual in testing. Depending on the intended type of testing, test inputs can be determined using black- and also white-box techniques. A test operation does not only include the arguments of the corresponding method, it also encompasses other provisions necessary, for instance, to enter the component in a necessary state. The arguments and the provisions necessary for a test can be defined in various ways. A test specification can be defined by

- using a regular text editor to assemble XML descriptors,
- using a XML editor which can offer more support than a text editor,
- using a graphical interface with e.g. input boxes for each element,
- using a data flow visual editor as suggested by the authors,
- using a program fragment consisting of a subset of Java.

The approach also encompasses certain tools which can be used to automate definition of test specifications and execution of tests. These tools, however, are not described here, since the majority of these tools, except a test pattern verifier, are intended to be used by the component provider. The test pattern verifier is a stand-alone module which the component user can apply to test the component with the test cases specified.

3.3 Approaches addressing the effects of a lack of information

3.3.1 Built-in test 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 [24]. 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, 22, 7, 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 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 [8, 9, 10] and [17], 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 principally 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.

Note that approaches supporting and facilitating information exchange between the component provider and component user, such as the component meta-data, retro-components, reflective wrapper and component test bench approaches, can also form a basis for built-in testing. Depending on the type of information, it is in principle possible to automatically generate test cases using such information and conduct tests. A respective application of the reflective wrapper approach is given in [8, 9, 10].

3.3.2 Testable beans approach

A lack of information can complicate testing of components in various aspects and several approaches have been proposed to tackle such difficulties. One of these approaches is that proposed in [11]. The potential difficulties are addressed by this approach by improving component testability. Component testability depends besides other factors on the ability of a component to support test execution and test observation, and it can thus be increased by augmenting a component with capabilities in order to support these tasks. A component augmented with such capabilities is called in this context *testable bean*, which indicates, at least technical, the target component model and framework of the approach, the Enterprise JavaBeans component model and framework.

One of the difficulties in testing a component, particularly of those being independent commercial items, is that the component user has generally very limited possibilities of observing execution of tests. A component usually does not possess the necessary capabilities to allow the component user this and such capabilities cannot be added to the component by the component user, since this requires source code and other detailed information. The suitable provision thus need to be taken by the component provider and a component needs to offer the corresponding capabilities by itself for testability reasons. The testable beans approach is a possible answer to this requirement.

A component, in this context an EJB component, needs to satisfy certain requirements and to possess certain features to become a testable bean. These requirements and features are:

> Firstly, a testable bean is deployable and executable. A testable bean can be used exactly in the same way as a regular component and do not require specific provisions for its operation.
>
> Secondly, a testable bean is traceable. A testable bean possesses certain capabilities which permit the component user observation of its behavior during a test, which would be encapsulated without such capabilities.
>
> Thirdly, a testable bean implements the *test interface*. A testable bean implements a consistent and well-defined interface which allows access to the capabilities supporting its testing.
>
> Fourthly, a testable bean includes the necessary provisions to interact with external testing tools. The approach suggests to functionally separate business logic implemented from the testing-specific logic at an architectural level.

From a technical point-of-view, the test interface is maybe the most obvious difference between a testable bean and a regular component to the component user. The test interface declares three methods. One of which initializes a test given the class and method to be tested and the test case, another method executes the test as initialized, and the last method finally evaluates the test. The methods declared by the test interface and implemented by a testable bean can be used in two possible ways. These possibilities are:

> Firstly, the test interface can be used by other tools. For instance, it can be used by tools in the environment in which the testable bean is embedded. In [11, 12], an environment is described containing two tools, called *test agent* and *tracking agent*. The first triggers the tests, whereas the second mainly has the purpose of monitoring.
>
> Secondly, the test interface can also be used by the testable bean itself. A testable bean can contain built-in tests scripts which access the methods declared in the test interface as necessary. These test scripts, possibly embracing test cases, can initialize tests, execute them, and evaluate the results autonomously.

4 Conclusions

Research in testing components is still an open problem. We still do not have appropriate methods, techniques and tools supporting both the component provider and component user in testing a component. The overview given in this article has shown the existing approaches in this area. This article focused on those approaches which are applicable by the component user. In [4, 5], limitations of the presented approaches are outlined and a novel strategy is proposed which does not suffer from similar limitations. The reader might also interested in reader them. 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.stecc.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. Merging components and testing tools: The self-testing COTS components (STECC) strategy. In *EUROMICRO Conference Component-based Software Engineering Track*. IEEE Computer Society Press, 2003.

[6] G. A. Bundell, G. Lee, J. Morris, K. Parker, and P. Lam. A software component verification tool. In *International Conference on Software Methods and Tools (SMT)*, pages 137–146. IEEE Computer Society Press, 2000.

[7] 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.

[8] 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.

[9] 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.

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

[11] J. Gao, K. Gupta, S. Gupta, and S. Shim. On building testable software components. In *COTS-Based Software Systems (ICCBCC)*, volume 2255 of *LNCS*, pages 108–121. Springer Verlag, 2002.

[12] J. Gao, E. Y. Zhu, and S. Shim. Monitoring software components and component-based software. In *Computer Software and Applications Conference (COMPSAC)*, pages 403–412. IEEE Computer Society Press, 2000.

[13] 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.

[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] C. Liu and D. Richardson. Software components with retrospectors. In *International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 63–68, 1998.

[17] 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.

[18] 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.

[19] 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.

[20] A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *International Workshop on Engineering Distributed Objects (EDO)*, volume 1999 of *LNCS*, pages 129–144. Springer Verlag, 2000.

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

[22] 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.

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

[24] 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.

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