Black- and White-Box Self-testing COTS Components[*]

Sami Beydeda and Volker Gruhn
University of Leipzig
Chair of Applied Telematics / e-Business
Klostergasse 3
04109 Leipzig, Germany
{beydeda,gruhn}@ebus.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, i.e. the developer of a component-based system. A limited exchange of information cannot only require the testing by the user but it can also complicate this tasks, since vital artifacts, source code in particular, might not be available. Self-testing components can be one response in such situation. This paper describes an enhancement of the Self-Testing COTS Components (STECC) Method so that an appropriately enabled component is not only capable of white-box testing its methods but also capable of black-box testing.

1 Introduction

Quality assurance, including testing, conducted in development and use of a component can be considered according to [12, 11] 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 user has often to test a component, particularly a third-party component, prior to its integration into the system to be developed. The various reasons obligating the component's testing by the component user are outlined in [7] with an overview of existing approaches to testing components.

In this paper, we describe an enhancement of the *Self-Testing COTS Components (STECC) Method* [4, 6]. The main idea of the STECC method is to augment a component with self-testability, so that the component user can test it thoroughly without necessitating the component provider to disclose certain information. In particular, a STECC self-testing component allows white-box tests without access to the component's source code. Source code information is processed within the component in an encapsulated manner not visible to the component user.

The enhancement of the STECC method addresses the need that the component user often not only needs to white-box test the component, but also black-box test according to the component's specification. For this purpose, the internal model encapsulated in a STECC self-testing component, which is particularly used for test case generation, has been augmented to also embrace information extracted from its specification. We, however, have not developed a new model, but rather use one which reached a certain maturity in testing classes, the *Class Implementation Specification Graph (CSIG)* [8]. Note that in the following a component is assumed to be implemented as a class, such as components according to the Enterprise JavaBeans Specification [9]. A positive side effect of CSIGs is that they do not only allow an integrated black- and white-box testing, the total number of test cases required for black- and white-box testing can be less than in the case when both tasks are carried out separately [8].

2 Self-Testing COTS Components Strategy

The component provider and component user generally need to exchange information during the various phases of developing the component and a component-based system [6]. Various factors, however, impact the exchange of

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

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.

A lack of information might require the testing of a component by its user prior to its integration in a system, and might significantly complicate this task at the same time. The component user might not possess the information required for this task. Theoretically, the component user can test a component by making certain assumptions and approximating the information required. Such assumptions, however, are often too imprecise to be useful. For instance, control-dependence information can be approximated in safe-critical application contexts by conservatively assuming that every component raises an exception, which is obviously too imprecise and entails a higher testing effort than necessary [12, 11].

Even though often claimed, source code as one type of information often required for testing purposes is not required by itself for testing purposes. It often acts as the source for obtaining other information, such as that concerning control-dependence. Instead making source code available to allow the generation of such information, the information required can also be directly delivered to the component user, obviating source code access. This type of information is often referred to as *meta-information* [17]. Even though the information required might already be available from own testing activities, the component provider might nevertheless not deliver this information to the component user. One reason may be that detailed information, including parts of the source code, can be deduced from it depending on the granularity of the meta-information. Therefore, there is a natural boundary limiting the level of detail of the information deliverable to the user. For some application contexts, however, the level of detail might be insufficient and the component user might not be able to test the component according to certain quality requirements.

The underlying strategy of the method proposed differs from those discussed thus far. Instead of providing the component user with information required for testing, component user tests are supported by the component explicitly. The underlying strategy of the method is to augment a component with functionality specific to testing tools. A component possessing such functionality is capable of testing its own methods by conducting some or all activities of the component user's testing processes, it is thus *self-testing*. The method is thereby called the *Self-Testing COTS Components* (STECC) method. Self-testability does not obviate the generation of detailed technical information. In fact, this information is generated by the component itself during runtime and is internally used in an encapsulated manner. The information generated is transparent to the component user and can thus be more detailed than in the case above. Consequently, tests carried out by the component user through the self-testing capability can thereby be more thorough as in the case of meta-information. Self-testability allows the component user to conduct tests and does not require the component provider to disclose source code or other detailed technical information. It thereby meets the demands of both parties. The STECC method does not only benefit the user of a component in that the user can test a component as required. It can also benefit its provider, as self-testability provided by an appropriately augmented component can be an advantage in competition.

From a technical point of view, a STECC self-testing component maintains a model of its own and generates test cases with regards to an adequacy criterion specified by the tester, who can particularly be the component user. The STECC framework, which implements the various relevant algorithms, determines the paths to be traversed according to the specified criterion and generates the necessary test cases as possible. The test case generation algorithm employed for this purpose is the *Binary Search-based Test Case Generation (BINTEST) Algorithm* [5]. The internal model used by the component is a control flow graph. It can be replaced by another control flow graph as long as its syntactical representation does not change. This is exactly the enhancement of the STECC method described in this paper. CSIGs are control flow graphs which also embrace specification information and thus allow generation of black-box test cases.

3 Class Implementation Specification Graphs

3.1 Motivation

Analysis and testing tasks are usually conducted using a model of the program under consideration which abstracts from certain aspects and focuses on others assumed to be more significant. Typical examples of such models are control flow graphs or finite state machines. Models used in analysis and testing are often constructed on the basis of the implementation, such as control flow graphs, or the specification, such as finite state machines, of the program under consideration, they seldom cover both. However, we often need to analyze and test a program according to both information sources. In the case of class-level analysis and testing, one answer to this need are *Class Specification Implementation Graphs (CSIGs)* [8].

The distinguishing feature of CSIGs from existing class models is that they combine the specification and implementation of a class. Each method is represented by two control flow graphs in possibly different abstraction levels, i.e. control flow as specified and control flow as implemented. We refer to the former as the *specification view* and the latter as the *implementation view* of a method. Therefore, this model is called the *class specification implementation graph* (CSIG) of a class to emphasize the combination of the two different views. Although the method views can differ in abstraction level, the difference does not affect the integration, as the integration is carried out at control flow graph level. As control flow graphs are used to model specification and implementation, structural techniques, such as the BINTEST algorithm, can be used for test case generation. An important feature of a CSIG is that generated test cases can cover both specification and implementation.

3.2 A demonstrative example

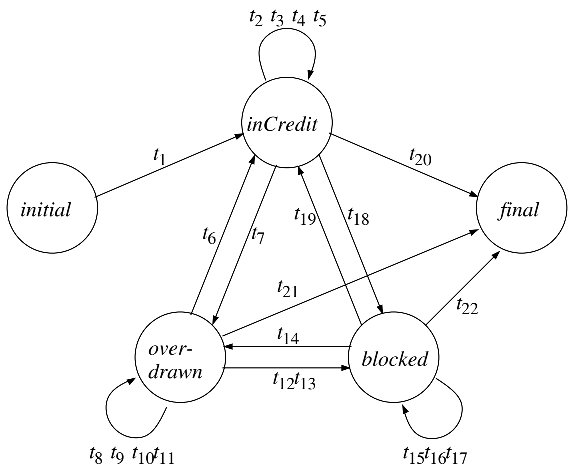
The example consists of a component, called `account`, which simulates a bank account. This component provides the appropriate methods for making bank account deposits (`deposit()`) and withdrawals (`withdraw()`). Furthermore, it provides methods for paying interest (`payInterest()`) and for printing bank statements (`printBankStatement()`).

Figure 1 shows the specification of component `account` in form of a *class state machine* (CSM) [14]. In this figure each state of component `account` is represented by a circle, while each transition is depicted by an arrow leading from its source state to its target state. These transitions are formally specified through 5-tuples $(source, target, event, guard, action)$ below this figure. A transition consists – besides a $source$ and a $target$ state – of an $event$ causing the transition, a predicate $guard$ which has to be fulfilled before the transition can occur, and an $action$ defining operations on the attributes during the transition. There are also two special circles labeled $initial$ and $final$. These two circles represent the state of a component before its instantiation and after its destruction, respectively. Thus, they represent states in which the attributes and their values are not defined, meaning that these two states are not concrete states of a component instance. For the sake of brevity, below the CSM in figure 1 only the transitions with event type `deposit()` are given.

A possible implementation of method `deposit()` is:

```
public void deposit(double amount) {
  balance += amount;
  t[idx++] = new transaction("Deposit ",
                    amount, balance);
}
```

With `transaction` implementing a financial transaction stored in an array `t` for later generation of bank statements.



inCredit := *balance>=0*
overdrawn := *balance<0 && balance>=limit*
blocked := *balance<limit*

t_2 = $(inCredit, inCredit,$ `deposit(amount)`, `true`, `balance += amount;`$)$
t_7 = $(overdrawn, inCredit,$ `deposit(amount)`, `balance+amount >= 0`, `balance += amount;`$)$
t_8 = $(overdrawn, overdrawn,$ `deposit(amount)`, `balance+amount < 0`, `balance += amount;`$)$
t_{14} = $(blocked, overdrawn,$ `deposit(amount)`, `limit <= balance+amount && balance+amount < 0`, `balance += amount;`$)$
t_{15} = $(blocked, blocked,$ `deposit(amount)`, `balance+amount < limit`, `balance += amount;`$)$
t_{19} = $(blocked, inCredit,$ `deposit(amount)`, `balance+amount >= 0`, `balance += amount;`$)$

Figure 1. Specification of component account by a class state machine

3.3 CSIG constituents

Figure 2 shows the CSIG of component `account`. Each method of a component is represented by two control flow graphs in its CSIG. One of them is a control flow graph generated on the basis of the method specification (*method specification graph*), whereas the other is a control flow graph determined using the method implementation (*method implementation graph*). In figure 2, method specification graphs are drawn light gray whereas method implementation graphs are drawn dark gray. For convenience, the two control flow graphs are called *method graphs*, if they do not have to be distinguished. Thus, the CSIG of a component shows each method from two different perspectives, namely what the method should do and what the method actually does.

The two method graphs of each method are embedded within a frame structure called a *class control flow graph frame* (CCFG frame). Generally, a component cannot be tested without a test driver, which creates an instance of the

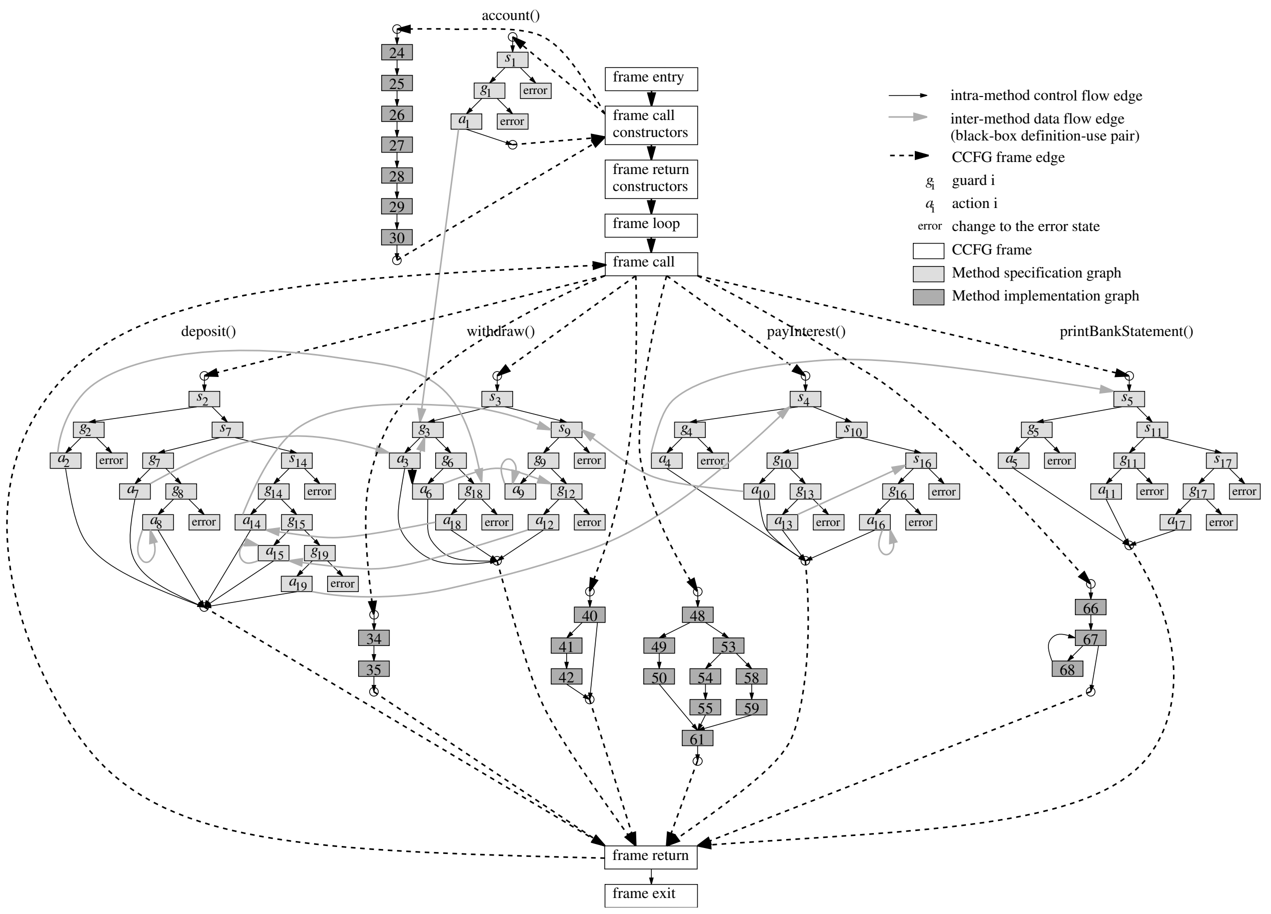

account()
frame entry
frame call constructors
frame return constructors
frame loop
frame call
deposit()
withdraw()
payInterest()
printBankStatement()
frame return
frame exit
intra-method control flow edge
inter-method data flow edge (black-box definition-use pair)
CCFG frame edge
g_i guard i
a_i action i
error change to the error state
CCFG frame
Method specification graph
Method implementation graph

Figure 2. Class specification implementation graph of component account

component, invokes the corresponding methods in a particular order, and finally deletes the instance. A CCFG frame represents an abstract test driver fulfilling this task[1]. In figure 2, the CCFG frame nodes are drawn without shading.

Three types of edges can be distinguished within a CSIG:

1. *Intra-method control and data flow edges*
   Intra-method control and data flow edges depict control and data dependencies within a single method graph. For instance, an intra-method data flow edge connects a node representing a definition of a local variable with another node representing a use in the same method (as a simple example of a def-use pair). In figure 2, these edges are drawn as solid arrows.

2. *Inter-method control and data flow edges*
   Edges of this type model control and data flow between two method specification graphs and two method implementation graphs, respectively. Assume that G_1 and G_2 are the method implementation graphs of methods M_1 and M_2, respectively. Then, an invocation of method M_2 within the implementation of method M_1 is modeled by an inter-method control flow edge leading from the corresponding node in G_1 to the entry node of G_2. In figure 2, these edges are shown as gray arrows. For the sake of simplicity, this type of edges is only given for method specification graphs.

3. *CCFG frame edges*
   The third type of edges in a CSIG consists of nodes, which either connect two CCFG frame nodes or the CCFG frame with entry and exit nodes of method graphs. In figure 2, this type of edges is shown as dotted arrows.

Method implementation graphs are generated on the basis of the implementations of the respective methods. Control flow graph generation is, for instance, described in [1]. The generation of method specification graphs is conducted on the basis of *method prototypes*, which are constructed us-

[1]A CCFG frame is a part of a CCFG suggested by Harrold et al. [13] for class-level data flow testing. As we only need the frame structure as an abstract test driver, we do not introduce CCFGs in this paper.

ing the finite state machine specification of the component.

For the construction of method prototypes, a prototype is generated for each transition $t = (source, target, event, guard, action)$ in the form of a nested `if-then-else` construct:

```
if (source)
  if (guard)
    action;
  else throw new ErrorStateException();
else throw new ErrorStateException();
```

`source` refers to the predicate of the source state. For instance, the predicate of state $inCredit$ is defined as $balance \geq 0$.

After generating these prototypes, those having the same event type are combined. For instance, transitions t_2, t_7, t_8, t_{14}, t_{15} and t_{19} share the event `deposit()`. Their prototypes can be merged to the following method prototype:

```
    deposit(double amount) {
 s2   if (balance >= 0)
 g2     if (true)
 a2       balance += amount;
        else throw new ErrorStateException();
      else
 s7     if (balance < 0 && balance >= limit)
 g7       if (balance + amount >= 0)
 a7         balance += amount;
          else
 g8         if (balance + amount < 0)
 a8           balance += amount;
            else throw new ErrorStateException();
        else
s14       if (balance < limit)
g14         if (limit <= balance + amount
                && balance + amount < 0)
a14           balance += amount;
            else
g15           if (balance + amount < limit)
a15             balance += amount;
              else
g19             if (balance + amount <= 0)
a19               balance += amount;
                else throw new ErrorStateException();
          else throw new ErrorStateException();
    }
```

Generation of control flow graphs for method prototypes can again be carried out as described in [1]. The process of embedding the various control flow graphs into a CCFG frame is explained in [13].

In the STECC method as initially designed, a component encapsulates an ordinary control flow graph modeling source code information. Tests as conducted by a STECC self-testing component were therefore solely white-box oriented. An enhancement of the STECC method to also cover black-box tests can be achieved by using CSIGs instead of ordinary control flow graphs. CSIGs also model specification information and tests conducted are thus also black-box oriented. Specifically, the total number of test cases required can even be less than in the case when black-box and white-box testing separately. A suitable test suite reduction technique is described in [8].

4 Related work

The STECC approach can be compared to built-in testing approaches in the literature. A number of built-in testing approaches have been proposed in the literature, e.g. [19], [16, 18, 10, 3] and [15, 2]. Similar as the STECC approach, built-in testing approaches aim at tackling difficulties in testing components caused by a lack of information, difficulties in test case generation in particular. The STECC approach has the same objective and the approaches can thus be directly compared to it. A comparison of them highlights several differences.

Firstly, the built-in testing approaches 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 these approaches 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. However, the component user might wish to test all components to be assembled with respect to an unique adequacy criterion. Built-in testing approaches usually do not allow this. The STECC approach does not have such a restriction. Adequacy criteria, even though constrained to control flow criteria, can be freely specified.

Secondly, built-in testing approaches using a predefined test case set generally require more storage than the STECC approach. 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. In contrast, the STECC strategy does not require predetermined test cases and does also not store the generated test case.

Thirdly, built-in testing approaches using a predefined test case set generally require less computation time at component user site. In such a case, the computations for test case generation were already conducted by the component provider and obviously do not have to be repeated by the component user, who thus can save resources, particularly computation time, during testing. Savings in computation time are even magnified if the component user needs to frequently conduct tests, for instance, due to volatility of the technical environment of the component. Storage and computation time consumption of a built-in testing enable component obviously depends on the implementation of the corresponding capabilities and the component provider needs to decide between the two forms of implementation, predefined test case set or generation on-demand, carefully in order to ensure a reasonable trade-off.

Fourthly, none of the existing built-in testing approaches, at least those known to the authors, are capable of providing or generating test cases for both black- and white-box testing. This is to our opinion the most significant difference.

5 Conclusions

The STECC strategy addresses the needs of both the component provider and component user. A situation particularly encountered in the case of commercial components, thus COTS components, is that the component provider might not wish to disclose information, particularly source code, which the component user might require for testing purposes. Our research started with the observation that existing approaches do not appropriately tackle such a situation.

The STECC strategy is a response to such situations. It allows the component user to test the component and to ensure suitability of the component to the target application context regarding its quality without requiring the component provider to publish specific information. It thus meets the demands of both parties. The STECC strategy can lead to a win-win situation insofar that both the component provider and component user can benefit from it. The benefit of the component user is obvious. The STECC strategy, or more clearly self-testability of a component, can be a valuable factor in competition. This potential benefit of the component provider from the STECC strategy becomes more obvious taking into account the specific type of components which are the most appropriate candidates for STECC self-testability, COTS components.

We have shown that an enhancement of the STECC method is easily possible using CSIGs. CSIGs represent both specification and implementation information and tests conducted are thus black- and white-box oriented. Specifically, the total number of test cases required can even be less than in the case when black-box and white-box testing separately.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison Wesley, 1988.

[2] 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.

[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*. ISBN 3-89975-462-X, Martin Meidenbauer Verlag, München, 2004.

[5] S. Beydeda and V. Gruhn. BINTEST - binary search-based test case generation. In *Computer Software and Applications Conference (COMPSAC)*, pages 28–33. IEEE Computer Society Press, 2003.

[6] 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 (EUROMICRO)*, pages 107–114. IEEE Computer Society Press, 2003.

[7] S. Beydeda and V. Gruhn. State of the art in testing components. In *International Conference on Quality Software (QSIC)*, pages 146–153. IEEE Computer Society Press, 2003.

[8] S. Beydeda, V. Gruhn, and M. Stachorski. A graphical representation of classes for integrated black- and white-box testing. In *International Conference on Software Maintenance (ICSM)*, pages 706–715. IEEE Computer Society Press, 2001.

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

[10] 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.

[11] 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.

[12] 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.

[13] M. J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (New Orleans, USA)*, volume 19 of *ACM SIGSOFT Software Engineering Notes*, pages 154–163. ACM Press, 1994.

[14] H. S. Hong, Y. R. Kwon, and S. D. Cha. Testing of object-oriented programs based on finite state machines. In *Second Asia-Pacific Software Engineering Conference (Brisbane, Australia)*, pages 234–241. IEEE Computer Society Press, 1995.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.