A strategy to improve component testability without source code

Camila Ribeiro Rocha, Eliane Martins

Institute of Computing (IC)
Campinas State University (UNICAMP)
P.O. Box 6176 – Campinas – SP ZC: 13083-970
{camila.rocha, eliane}@ic.unicamp.br

Abstract. A software component must be tested every time it is reused, to guarantee the quality of both the component itself and the system in which it is to be integrated. To reduce testing costs, we propose a model to build highly testable components by embedding testing and monitoring mechanisms inside them. The approach is useful to component developers, who can use these built-in test capabilities in the testing phase, as well as to component users, who can verify whether the component fulfills its contract. Our approach proposes the insertion of built-in mechanisms directly into intermediate code, allowing its use even in COTS components.

Keywords: testability – design for testability – OO components – UML

1 Introduction

Component-based software development is a current trend in the construction of new systems. The main motivation is the possibility to significantly reduce development costs and time yet satisfying quality requirements.

A software component can be defined as a "unit of composition with contractually specified interfaces and explicit dependencies. A software component can be deployed independently and is subject to composition by third parties" [Sz98]. A software component may be: in-house software designed for one project and then used in others; software created to be part of an in-house repository; third party software which might be either open-source software or commercial-off-the-shelf (COTS) components.

For component-based development to be successful, verification and validation activities should be carried out during component development and development of systems where they are used [Ad99]. In particular, components should be tested many times: individually and also each time it is integrated into a new system [We98].

Testing components present, however, additional difficulties with respect to testing in-house software. These difficulties are mainly due to lack of information [BG03]: component providers (who develop the component) do not know in advance all different contexts in which the component will be used; component users (who develop a system using the component), on the other hand, typically have only provided interface access. This usually happens with COTS components whose

source code or other information that could help testing is not available. Even when source code is available, as open-source components, users might not have either the required documentation nor the expertise that would help them test and debug.

Component providers can facilitate matters to the users by enhancing component testability. Design for testability (DFT) techniques have long been used by integrated circuit (IC) engineers to reduce test cost and effort yet achieving the required quality level. The built-in test (BIT) approach, which is a DFT technique, introduces standard test circuits and pins to allow a card or an IC to be put in test mode, to transmit test inputs and to capture the outputs. An extension to that approach consists of integrating test inputs generation mechanisms inside the IC, which renders the use of an external tester unnecessary. This concept is known as built-in self-test, or BIST, for short. Similar approaches have been proposed to enhance software testability. A discussion about component testing and the difficulties to accomplish this activity, as well as the approaches used to improve software components testability, are the subject of Section 2.

This text presents an approach for components testability improvement, making them testable. A testable component contains, in addition to its implementation, information that supports testing activities. Our concern in this work is OO components, defined as sets of tightly coupled classes. This work extends the one in [MTY01], where a component is considered a unique class. It is also an extension of the work in [UM01], in which built-in capabilities are inserted in intermediate code, thus allowing its use when no source code is available. A testable component, in our approach, contains a test and a tracking interface to allow users to access test facilities embedded into the component. These facilities include self-checking capabilities, state and error tracking functions as well as a test specification from which test cases can be automatically derived. Section 3 shows design aspects and Section 4 presents the architecture of a testable component.

Our aim is unit testing: to determine whether a component conforms to its usage contract. This is the first step users should perform in order to guarantee that the component fulfils its contract in the new context of execution. Possible misbehavior can be detected earlier and countermeasures can be taken by users, such as creating a wrapper or a protector to avoid that the identified misbehavior jeopardizes application's quality.

Section 5 contains implementation issues, showing the steps performed in building a testable component. Finally, Section 6 concludes this paper, presenting current status and future works.

2. Component testing

Reusable components are increasingly being used nowadays, specially in critical applications. It is expected that their use may shorten development time and ease maintenance, yet guaranteeing a required quality level. To achieve good quality, systems must be built upon high quality components that interoperate safely [We98]. Since testing is the most commonly used quality assurance technique, components should be extensively tested, both individually and once integrated.

Component-based systems, as well as any other system, undergo at least three test phases: unit (to check one individual component), integration (to check whether the components fit together) and system testing (to check whether the system fulfills its requirements). Furthermore, regression testing is needed each time there is a modification.

2.1. Difficulties in testing components

There are several technical difficulties that remain to be solved in component-based development. These difficulties can be attributed mainly to two main causes [BG03]: heterogeneity and lack of information. Heterogeneity stems from various reasons: (i) the same specification can have different implementations for the same component; (ii) different components can be implemented in different programming languages and for different platforms (database technology, operating system, runtime environment, hardware); (iii) different components can be provided by different suppliers, possibly competing with each other.

A consequence of this heterogeneity for testing is that it is hard to build test suites or test tools that can operate on a large variety of programming languages and platforms. Test drivers and stubs cannot be built in the ad-hoc and project specific way, as they could not effectively cope with different components and their customizable functions [Bo01].

Lack of information, on the other hand, is due to the limited exchange of information among component providers and component users [BG03]. Component providers do not know in advance the different contexts in which the component will be used, so they need to test it in a context-independent way. Nevertheless, explicit or implicit assumptions are made by the providers to circumvent this lack of knowledge about usage contexts. These assumptions, specially the implicit ones, might lead the users to the architectural mismatch problem [GAO95]. Mismatches can occur due to incompatibility of the data exchanged among components or due to problems in the dynamic interaction among them. Consequently users must retest a component each time it is used in a new context, as it may function well in some of them but not in others. Integration testing by component users is absolutely necessary to reveal dynamic interaction conflicts.

Component users, on the other hand, generally do not have the information needed (input domain, specification, source code) for adequately test a component. Lack of source code makes it difficult to statically analyze the code to obtain data-flow or control-flow dependencies, or to employ white-box coverage criteria for testing. After testing, debugging is also complicated without source code information. Even in the case it is available, users may have neither the time nor the expertise necessary to understand it well enough to perform testing and debugging tasks.

2.2. Component's testability

Given the need of extensive testing, building components with good testability is very important because test tasks are simplified and test costs are reduced without jeopardizing system quality.

As pointed out by different authors [Bi94, Pr97, Ga00], testable software possesses a set of characteristics such as controllability (how easy it is to control a software in its inputs/outputs and behavior?), observability (how easy it is to observe a software during testing?), traceability (how easy it is to track the status of component attributes and component behavior?), understandability (how much information is provided and how well is it presented?), among others.

In the context of component-based development there are various ways to enhance testability. Two alternatives can be considered [Ga02]: creating new test suite technologies and embedding test facilities inside components. The first category consists of enabling components users to build test suites for any component and perform tests. Such techniques are called as "plug-in-and-test". Examples of such approaches can be found in [Vo97] and [MK96].

The second category refers to approaches that are aimed at augmenting a component with test facilities such as test suites (or capabilities to generate them), runtime checkers, monitoring capabilities. These capabilities are known as built-in tests (BIT), and a standard interface is needed for the user to access such extra information. In this text we focus these approaches, considering a testable component as one that contains built-in testing capabilities.

2.6. Building testable components

Design for testability techniques and the self-testing concept have been used in hardware for a long time; however, only recently this subject is gaining more attention of software community.

Some authors have proposed the use of these hardware approaches to improve software testability. A pioneering work is the one presented in [Ho89] which augments a module's interface with testing features – access programs, corresponding to the test access ports (TAP) used in BIT approaches in hardware. Other features are assertion using, automate test driver generation from a test specification and insertion of testing features by compiler directives. Binder adapted this approach to the OO context, proposing the construction of self-testing classes [Bi94]. A self-testing class is composed by the class (or component) under test (CUT) augmented with BIT capabilities and a test specification used for test generation and as test oracle.

Since then, various approaches have been proposed to construct testable components. For the sake of conciseness we present here only those that are closer to our work and relevant in the paper's context. A more detailed overview of several approaches can be seen in [BG03] and [Bo01].

Voas et al. present a tool, ASSERT++ [VSS98], to help users to place assertions into programs to improve OO software testability. The tool supports assertion insertion and suggests points where the assertions might be inserted, according to testability scores obtained. Their work is complementary to ours, in that we are not

concerned with the best place to introduce assertions. Moreover, they are only concerned with improving controllability and observability; test case generation and other built-in facilities, such as reporter methods, were not considered.

Wang et al. describe how to build testable OO software by embedding test cases into source code [WKW99, Wa97]. These test cases aimed at covering implementation-based criteria applied to the methods of a class. They can be inherited, as they are implemented as methods, allowing tests to be reused by derived classes.

Le Traon et al. also present self-testable OO components which embed their specification (documentation, methods signature and invariant properties) and test cases [TDJ97]. Test cases are generated manually and embedded into the component. Assertions are used as an oracle, but manually generated oracles can also be used. An interesting point is that a test quality estimate can be associated to each self-test, either to guide the choice of a component, or to help reaching a test adequacy criteria when generating test cases. Similar to their approach, we also use assertions to build self-checking capabilities inside a component.

The component test bench approach (CTB) [Bu00] also enhances a component with a test specification. This specification contains a description of the component implementation (programming language, platform), its implemented interfaces and the test sets appropriate for each interface. A test specification is composed of test operations, which define the necessary steps to test a specific method. It is described in XML and stored in descriptor files, which are updated with test results as CTB runs so they may also be used for regression testing. Test operations can be executed either in a specific runtime environment, IRTS (Instrumented Runtime System), provided with the CTB tool, or in a standard runtime environment. IRTS has symbolic execution capabilities to help users parameterize test cases or to obtain conditions not yet exercised. A test pattern verifier module is also available to allow the generation of test drivers from the test specification. This module, written in Java, can be packed with the component, together with the test specification.

In the three previous approaches, test cases generated during development are embedded into the component, while our approach embeds the component specification from which test cases are generated. An advantage of these approaches is that users test effort is reduced. However, as pointed out in 2.4, component providers' test suites might not be adequate for users.

The Self-Testing COTS Component (STECC) strategy aims at both reducing users effort on testing and allowing them to create tests that are more suitable to their requirements [BG03a]. Instead of test suites, components are augmented with analysis and testing functions. Component providers do not need to disclose detailed information to users: the test tool may access the necessary information from inside the component. Component users can use the embedded analysis and test functions to test a component according to adequacy criteria that are most interesting to them. Test drivers are automatically generated by the embedded test tools. Test cases do not necessarily include expected results: this depends on whether the component specification is available in a form which can be automatically processed or not.

Gao et al. [Ga02] propose the testable bean concept, a component architecture with well-defined built-in test interfaces to enhance the component testability and support

external interactions for software testing. There are also built-in tracking interfaces which provide tracking mechanisms for monitoring various component behaviors in a systematic manner. Testing and tracking code are separated from normal functional parts of the component as in our work, but there is no automatic generation of test cases.

Our work extends previous ones developed by our group. In Martins et al. [MTY01] we presented an approach for building self-testable classes. This work was extended to consider components that can contain more than one class [UM01]. In these works the main focus was test case generation from a specification embedded in the component. Here our main concern is to define what resources a testable component must provide to support testing, which can be easily used by component providers and component users, with minimum programming overhead. We still consider specification-based test case generation, but the embedded resources should also be used with other types of testing.

3. Design issues

The use of additional information to components is not new. Besides the approaches to enhance component testability presented in Section 2, we can also mention other approaches that are not specifically for testing purposes. For example, a JavaBean component can be associated with a BeanInfo Object, which contains information such as the component name, a textual description of its functions and textual description of its properties [Su04]. These are aimed at helping to customize the component within an application.

In our approach, the additional information provides resources to support testing activities. Design of the architecture aims at achieving the following properties: *ease of use*, so that a testable component could be provided and used with minimum programming effort; *separation of concerns*, providing independence between the component under test (CUT) and the instrumentation features necessary for testing and tracking, as well as between the instrumentation mechanisms with respect to each other; *extensibility*, so that new capabilities can be added without further difficulties; *portability*, among various applications and platforms; *reduced intrusiveness*, such that the instrumentation inserted for testing purposes does not impact too much in storage and behavior of the component; and *source code independence*, which allows the instrumentation code to be added or removed without the need to re-compile the component and also to deal with COTS components.

A testable component is augmented with built-in testing and tracking capabilities, which allow control of pre-test state and observation of post-test state. A test and tracking interface is implemented to access embedded test facilities.

There are different approaches to develop built-in testing and tracking capabilities, according to Gao et al. [GZS00]. One is the framework-based approach, where a trace library is used to add tracking and testing code into components. Besides the high programming overhead, this approach requires that component source code is available. Another approach is based in automatic code insertion, in which the instrumentation is introduced in the component source code. This can be

accomplished either by an automatic tool (e.g, by extending a parser, like OpenC++ 2.0 [Ch97]) or by using aspect-oriented programming [Ki01]. Although this approach reduces the programming overhead, it still assumes component source code availability. The third is the automatic component wrapping approach, where a component is wrapped with testing and tracking code as a black-box. This approach has low programming overhead and the component source code is not required. However, it is not suitable to support error and state tracking (see Section 4.1 for details), as they highly depend on detailed information about a component internal logic as well as on the business rules of the application.

We used the second approach, which is based on code insertion, since error and state tracking is an important feature for us. To avoid source code dependency, built-in testing and tracking code is inserted by manipulating the intermediate code (bytecode, for Java language, for example).

BIT capabilities also include runtime checking mechanisms, which are responsible for: (i) checking conditions on inputs provided by component's clients (precondition), (ii) checking conditions on the outputs delivered by a component to its clients (postcondition), (iii) checking conditions that must be true at all times (invariants) and (iv) produce an action when such conditions are violated. The design-by-contract approach developed in OO domain [Me92] can be applied. A contract here is considered as the specification of rights and obligations between a component and its clients. The process by which this contract is obtained from requirements is not our concern in this text.

A component specification model can also be made available for test case generation purposes. This specification is not part of built-in capabilities; it can be deployed with the component or can be made available at the component provider site. Aspects of test case generation are not addressed in this text. Our main concern here is the testable component architecture.

4. Testable component architecture

The proposed architecture is shown in Figure 1. The testable component is composed by the component under test (CUT), a Tracker component, which is responsible for the tracking mechanisms, and a Tester component, which is responsible for the assertions inclusion and the specification retrieving. With this proposal, we achieved the *separation of concerns* and *extensibility* properties since the testing and tracking codes are separated from the CUT's code.

The Tester and Tracker components will be detailed in the following subsections.

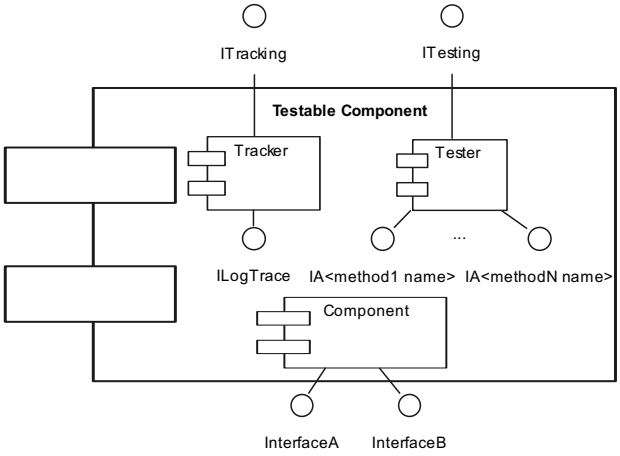

Figure 1: Testable component architecture

4.1. Tracker Component

The Tracker component provides methods to include tracking mechanisms into the CUT. The kinds of traces available are [GZS00]: *Operational trace*, which records the interactions between component public operations; *Error trace*, which records exceptions generated by the component; and *State trace*, which traces the public attributes state. The event trace is not available because it is directly related to GUI components, but the proposed architecture can be extended to support it. The performance trace is not available either, because the presence of both tracking and testing mechanisms can influence performance testing results.

The ITracking interface is public and provides methods to include tracking code into the CUT byte codes. The tracking code consists of calls to the methods of the interface ILogTrace, which has package visibility and provides methods to register the traces into a log file.

The ITracking interface methods are:

- `operationalTrace(method: String, type: char)`: enables the operational tracking for `method`. The parameter `type` specifies which calls will be registered: the calls to the method, the calls from the method, or both.
- `stateTrace(attribute: String[])`: enables the state tracking for the public attributes which names as passed in `attribute`.
- `errorTrace(exception: String)`: enables the error tracking for `exception` in public methods where it is raised.

It is worth noting that only public elements of a component are traceable, so users have no access to non-disclosed information.

4.2. Tester Component

The Tester component provides the interface ITesting, which is public and responsible for self-checking capabilities included into the CUT. It also offers specification retrieval service to the users, if test cases are automatically generated.

The Tester also offers the IA<method1 name>… IA<methodN name> interfaces, which have package visibility and are accessed by the CUT after the instrumentation for assertion checking. Each interface IA<method name> is related to a CUT's public method. E.g. if the CUT has a pop method in the interface Queue, the Tester component will have an interface called IAQueue_pop, which handles assertion verification.

The Tester structure is shown in Figure 2. To simplify only one IA<method name> interface is presented. The symbol "+" means the method is public, so visible outside the testable component.

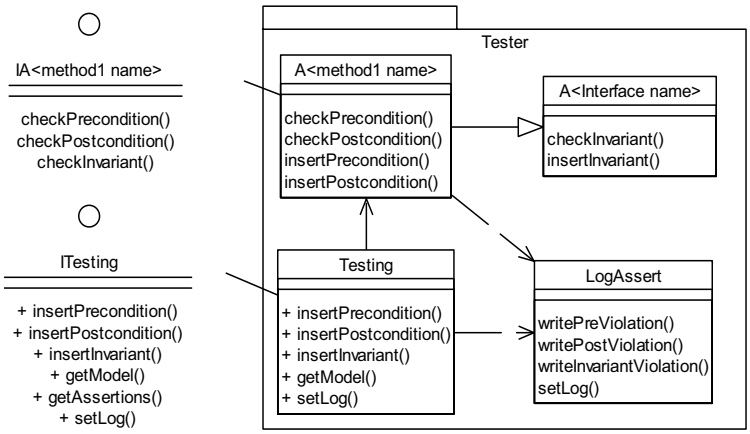

Figure 2: Component Tester architecture

The ITesting interface is implemented by the class Testing, following the Facade pattern [Ga95]. The methods `insertPrecondition(method: String, continue: boolean)`, `insertPostcondition(method: String, continue: boolean)` and `insertInvariant(method: String, continue: boolean)` are delegated to the A<method name> class corresponding to the parameter `method`. The assertion code is included in CUT's `method`, and consists of a call to corresponding method A<method name>.check<Assertion>. If an assertion violation happens, the log registration is made by the class LogAssert.

The `continue` parameter indicates whether the CUT execution should continue after the assertion violation or not. This option is useful to test exception handlers, and that is the reason the assertion verification had to be implemented as a method call rather than using the `assert` Java library.

Figure 3 ilustrates an assertion inclusion, verification and violation example (without continuing the execution). The client chooses to include postcondition

verification for `Queue.pop` method. The CUT is instrumented to include a call to the `IAQueue_pop.checkPostcondition()` in the end of `Queue.pop`. If the postcondition is violated, the violation is registered in the log file by `LogAssert.writePostViolation()`.

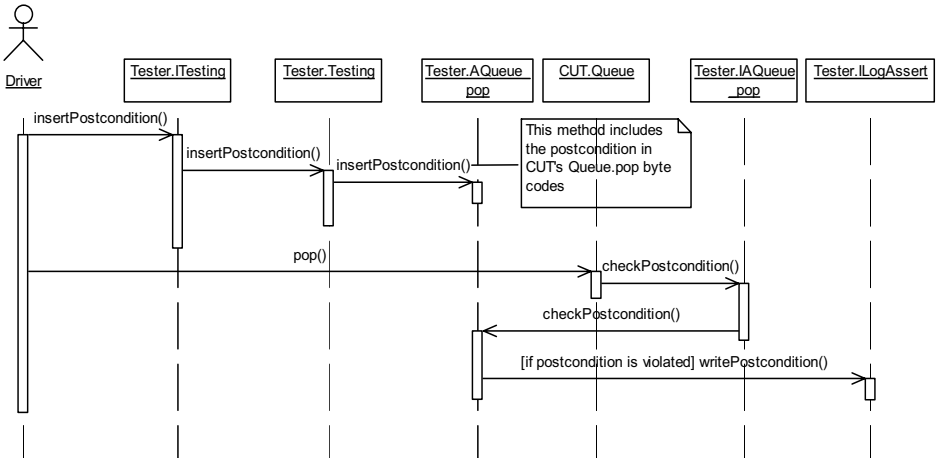

Figure 3: Example of assertion inclusion and violation

Component Specification

The component specification is retrieved by the methods getModel() and getAssertions(). These methods return files that contain the components behavior model and the component's public methods assertions.

The behavior model is a UML Activity diagram, which represents the call dependencies among the CUT's public methods. In other words, it represents the component usage workflow. This diagram will be used to automatically generate test drivers for the component by a tool which is being developed and will be based in ConCAT [MTY01]. Because it is a specification based test, the generation will be independent from the component's source code. Besides the tool is not dependent on a specific component implementation: the models only depend on the public interfaces provided by the CUT, following the *portability* property.

The assertions file contains public methods' invariants, pre and postconditions written in OCL (Object Constraint Language). It will be used to automatically generate the Tester component.

5 Implementation issues

The first studies used the Java language to implement the CUT as well as the Tester and Tracker components. To achieve the *source code independence* property,

the instrumentation was implemented using a byte code manipulation library, BCEL (Byte Code Engineering Library) [Da01].

The BCEL API is a toolkit for the static analysis and dynamic creation or transformation of Java class files. The API contains methods that retrieve method instructions and allow their manipulation. The main problem in using BCEL is implementation complexity, because the developer has to understand the Java byte codes structure.

5.1. Instrumentation code

The BCEL API was used to include tracking and testing code into CUT's public methods instructions, both for tracking and assertion checking. The code included were basically method calls, following the *ease of use* property.

Figure 4 and Figure 5 present which code was included into the CUT's method Queue.pop for the tracking and testing instrumentation respectively. A queue implementation is used as an example, and the code included is described in Java syntax starting with %.

The operational trace code (Figure 4 (a)) is included in the beginning of the method. It consists of a method call: `insertOperationalTrace(type: char, call: String, parameters: String)`. The parameter `type` is 'i' if the method is being called and 'o' if the method is calling other public method; `call` is the method name; and `parameters` represents a string with the methods parameters and their values.

The error code (Figure 4 (b)) is included in the beginning of the catch block. The method `insertErrorTrace(String exception, String thr, String message)` is called. Its parameters means: `exception`, the exception raised, `thr`, the method which thrown it and `message`, information about the exception handling.

The precondition code (Figure 5 (a)) is simply a method call in the beginning of the method code. For the method `Queue.pop` postcondition (Figure 5 (b)) it was necessary to keep the initial vector length to compare it with the final one and check if its size has decreased. This checking is made by `IAQueue_pop.checkPostcondition()` method, based on the parameter values. Each `checkPrecondition` and `checkPostcondition` methods will have different parameters according to their checking needs.

The postcondition checking happens right before the end of method, and it is not performed if an exception is raised. The invariant is checked in the beginning and in the end of the method, even if an exception is raised, to check component's consistency. That is why its implementation needs a try-catch block (Figure 5 (c)).

As the complete source code can not be shown here, we have made some measurements to demonstrate the programming overhead in testable component creation.

```
%  Tracker.ILogTrace tracker = new LogTrace();
%  tracker.insertOperationalTrace('i', "Queue.pop()", "");
   index--;
   Object data = vector[index];
   vector[index] = null;
   return data;
```

(a) Operational trace code

```
   try{
    index--;
    …
    return data;
   }
%  catch (ArrayIndexOutOfBoundsException e) {
%   Tracker.ILogTrace log = new Tracker.LogTrace();
%   log.insertErrorTrace("ArrayIndexOutOfBoundsException",
%        "Queue.pop", "Exception handled by Queue.pop: " +
%                     "Previous queue size kept.");
    index++;
    return null;
   }
```

(b) Error trace code

Figure 4: Code necessary for tracking

```
%  Tester.IAQueue_pop tester = new Tester.AQueue_pop();
%  tester.checkPreCondition(index);
   index--;
   …
   return data;
```

(a) Precondition code

```
%  int size = index;
%  index--;
   …
%  Tester.IAQueue_pop tester = new Tester.AQueue_pop();
%  tester.checkPostcondition (index, size);
   return data;
```

(b) Postcondition code

```
%  Tester.IAQueue_pop tester = new Tester.AQueue_pop();
%  tester.checkInvariant(index);
%  try{
     index--;
     …
     return data;
%  }
%  finally {
%    tester.checkInvariant(index);
%  }
```

(c) Invariant code

Figure 5: Code necessary for assertion checking

In our queue example, the CUT has 31 lines of Java code, two private attributes, one constructor and two methods (push and pop without exception handler). The Tester component has 470 lines of Java code, in which 125 concern the set up required by BCEL to instrument each method. The Tracker has 180 lines of Java code, in which 50 corresponds to the BCEL set up.

The CUT changes were measured from the increase of its size. For operational and error trace observation the pop method had an exception handler as presented in Figure 5 (b). The CUT's .class file had 798 bytes. When it were included operational trace in pop and push methods, and error trace in pop method CUT's size increased to 1,19 kbytes.

For assertion measurements, the pop method didn't have the exception handler. So its initial size was 660 bytes. With preconditions, postconditions and invariant checking in pop method only, its size increased to 927 bytes. It is worth noting that this increase on component size is constant, since the code used in each method is the same whichever the component. So, for bigger components, the overhead will be less significant.

Measurements of performance impact are also envisaged in future work.

5.2. Test driver example

This section brings an example of a test driver that uses the testing mechanisms. The test case is to pop an element from the queue when it is empty. The CUT was instrumented to include Queue.pop methods' postcondition (the queue size must decrease in one element) and invariant (the queue index must be between zero and ten elements). The code necessary to apply this test case is shown in Figure 6. It is worth noting that the CUT's testability has increased as the test driver code is simpler than in a regular component.

```
public class Driver {

  public static void main (String args[]) {
       Tester.ITesting tester = new Testing();
   //Queue.pop Postcondition enabled
       tester.insertPostcondition ("pop", false);
   //Queue.pop Invariant enabled
       tester.insertInvariant ("pop", false);
       tester.setLog("Empty_queue.log");
       Component.Queue queue = new Component.queue();
       queue.pop();
  }
}
```

Figure 6: Class driver's source code

Figure 7 shows the generated log file. As the tracking mechanisms were not enabled, just the invariant violation was registered. If there were other test cases, their assertion violations and tracking messages would be registered in the same file.

```
Invariant violation: Method Queue.pop
Vector index = -1
```

Figure 7: Generated log file

6 Conclusions and future works

This article proposed a testable component architecture to facilitate testing realization. The testability is increased by the inclusion of self-checking capabilities and tracking mechanisms into the component under test, which can be accomplished even when the source code is not available.

The major advantage is helping component users to test in an efficient way, guaranteeing the component's quality in the new enviroment. An advantage for the component providers is that they can offer a testable component without revealing its source code.

The major disadvantage is the amount of extra information, which is proportional to number of methods in the component's interface.

Up to now we have defined the testable component architecture and a test interface which minimizes programming overhead and also satisfies other design constraints such as ease of use, separation of concerns, extensibility, portability, reduced intrusiveness and source code independence. As a proof of concept we implemented the proposed architecture in a simple component, using Java language and BCEL library to manipulate byte code. Using in a real application is also under way, as well as performance impact measurements.

Next activities are the elaboration of a systematic approach to build testable components to be embedded in a component-based development process. Also, the development of test automation tools to create the components Tester and Tracker based on the assertions written in OCL is being envisaged. The automation of test case generation based on the behavior model is under development.

References

[Ad99] Addy, E.: Verification and Validation in Software Product Line Engineering. PhD Dissertation. College of Engineering and Mineral Resources at West Virginia University, 1999.

[BG03] Beydeda, S.; Gruhn, V.: State of the Art in Testing Components. In: 3rd International Conference on Quality Software, Dallas, 2003.

[BG03a] Beydeda, S.; Gruhn, V.: Merging components and testing tools – The self-testing COTS components (STECC) strategy. In: Proc. EUROMICRO Conference Component-based Software Engineering Track, 2003.

[Bi94] Binder, R.: Design for Testability in Object-Oriented Systems. In:

Communications of ACM, 37(9), Sept/Oct 1994; pp 87-101.

[Bo01] Bhor, A.: Software Component Testing Strategies. Technical Report UCI-ICS-02-06, University of California. Irvine, 2001.

[Bu00] Bundell, G. et. al.: A Software Component Verification Tool. In: Proc. International Conference on Software Methods and Tools, Wollongong, 2000.

[Ch97] Chiba, S. OpenC++ 2.5 Reference Manual. Institute of Information Science and Eletronics. University of Tsuukuba. 1997-99

[Da01] Dahm, M.: Byte Code Engineering with the BCEL API. Technical Report B-17-98, Freie Universität at Berlin, Institut für Informatik, 2001.

[Ga00] Gao, J.: Component Testability and Component Testing Challenges. URL: www.sei.cmu.edu/cbs/cbse2000/papers/18/18.pdf. Acessed in May 20, 2004.

[Ga02] Gao, J. et. al.: On building testable software components. In: Lecture Notes in Computer Science, V. 2255, Springer Verlag, 2002; pp. 108-121.

[GAO95] Garlan, D.; Allen, R.; Ockerbloom, J.: Architectural Mismatch – Why Reuse is so Hard. In: IEEE Software, 12(6), Nov 1995; pp. 17-26.

[GZS00] Gao, J.; Zhu, E.; Shim, S.: Monitoring Software Components and Component-Based Software. In: IEEE Computer Software and Application Conference (COMPSAC), Taipei, 2000.

[HMF92] Harrold, M.; McGregor, J.; Fitzpatrick, K. Incremental Testing of Object-Oriented Class Structures. In Proc. 14th. IEEE International Conference on Software Engineering (ICSE-14), 1992, pp 68-80.

[Ho89] Hoffman, D.: Hardware Testing and Software Ics. In: Proc. Northwest Software Quality Conference, Portland, 1989; pp. 234-244.

[Ki01] Kiczales, G. et. al. An Overview of AspectJ. In: Lecture Notes in Computer Science, V. 2072, Springer Verlag, 2001; pp. 327-353.

[Me92] Meyer, B.: Applying Design by Contract. In: IEEE Computer, 25(10), Oct 1992; pp. 40-51.

[MK96] McGregor, J.; Kare, A.: Parallel Architecture for Componet Testing. In: Proc. 9th International Quality Week, 1996.

[MTY01] Martins, E.; Toyota, C.; Yanagawa, R; Constructing Self-Testable Software Components. In: Proc. IEEE/IFIP Dependable Systems and Networks (DSN) Conference. Gothemburg, 2001.

[Pr97] Pressman, R.: Software Engineering – a Practitioner's Approach. McGraw-Hill, 4th Edition, 1997.

[Su04] Sun Microsystems, Inc.: BeanInfo (Java 2 Plataform SE v1.5.0). 2004. URL: http://java.sun.com/j2se/1.5.0/docs/api/java/beans/BeanInfo.html. Accessed in May 28, 2004.

[Sz98] Szyperski, C.: Component Software – Beyond OO Programming. Addison-Wesley, 1998.

[TDJ97] Traon, Y.; Deveaux, D.; Jézéquel, J.: Self-testable Components – from Pragmatic Tests to Design-for-Testability Methodology. In: Proc. 1st. Workshop on Component-Based Systems, Switzerland, 1997.

[UM01] Ukuma, L.; Martins, E.: Development of Self-Testing Software Components". In: Proc. 2nd IEEE Latin-American Test Workshop (LATW), Cancun, 2001; pp. 52-55.

[Vo97] Voas, J.: A Defensive Approach to Certifying COTS Software. Technical Report, Reliable Software Technologies Corporation, 1997.

[VSS98] Voas, J.; Schmid, M.; Schatz, M.: A Testability based Assertion Placement Tool for Object-Oriented Software. Technical Report NIST CGR 98-735, Information Technology Laboratory, 1998. URL: http://www.rstcorp.com

[Wa97] Wang Y. et. al.: On Testable Object-Oriented Programming. In: ACM Software Engineering Notes, 22(4), July 1997; pp. 84-90.

[We98] Weyuker, E.: Testing Component-Based Software – a Cautionary Tale. In: IEEE Software, 15(5), Sept/Oct 1998; pp. 54-59.

[WKW99] Wang, Y.; King, G.; Wickburg, H.: A Method for Built-in Tests in Component-based Software Maintenance. In: Proc. 3rd European Conference on Software Maintenance and Reengeneering (CSMR), Holland, 1999; pp. 186-189.