BISTFaSC: An Approach To Embed Structural Testing Facilities Into Software Components

Marcelo Medeiros Eler
School of Arts, Sciences and Humanities
University of Sao Paulo
Sao Paulo – SP
marceloeler@usp.br

Paulo Cesar Masiero
Institute of Mathematics and Computer Science
University of Sao Paulo
Sao Carlos – SP
masiero@icmc.usp.br

***Abstract*—Component-based applications can be composed by in-house or COTS (Commercial off-the-shelf) components. In many situations, reused components should be tested before their integration into an operational environment. Testing components is not an easy task because they are usually provided as black boxes and have low testability. Built-in Testing (BIT) is an approach devised to improve component testability by embedding testing facilities into software components usually to support specification-based testing. Such components are called testable components. There are situations, however, in which combining specification and program-based testing is desirable. This paper proposes a BIT technique designed to introduce testing facilities into software components at the provider side to support structural testing at the user side, even when the source code is unavailable. An implementation to generate testable components written in Java is also presented. The approach was firstly evaluated by an exploratory study conducted to transform COTS components into testable components.**

I. Introduction

Component-Based Software Development (CBSE) is a reuse-based approach that defines techniques to build systems by putting existing software components together. According to Szyperski, software components are units of composition with contractually specified interfaces and context dependencies [1]. A software component implements specific functionalities that may be shared by several applications. The functionalities provided by a component can be only used via operations exposed by its interfaces. Component-based applications can use in-house or COTS (Commercial off-the-shelf) components.

CBSE brings many benefits to software development and maintenance [2]. Components are assumed to reach a high level of quality assurance in a short period of time. Due to market pressure, applications composed by such components are expected to inherit this high level of quality [3]. Experience showed, however, that this assumption is not necessarily true in practice [4], [3], [5].

Beydeda and Weyuker state that the component provider might not be able to anticipate all possible application context and technical environment in which the component might be used [3], [4]. Thus, the quality assurance conducted by the component provider might not be effective. This scenario gives to component users the responsibility of testing the reused components before gluing them together.

Testing COTS, however, is not an easy task, because they present low testability. Testability has been defined as the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [6]. Components present low testability because they are usually provided as a black-box and the source code is seldom available to conduct program-based testing [3]. Component users are thus forced to use only specification-based techniques. Moreover, component users also suffer from lack of information because relevant documentation and data to derive test cases and plans might not be available.

Several Metadata and Built-In Testing (BIT) approaches have been proposed to mitigate the problem of lack of information and low testability. Metadata are intended to provide component users with relevant information to conduct and to evaluate testing activities. These information may be test scripts, the inner structure of the component or invocation sequence constraints, for example [7].

BIT approaches improve component testability by adding testing facilities at the provider side to support testing activities at the user side. Such facilities are usually operations to control and observe a state machine, to evaluate invariants, to validate contracts or sequence constraints, and to generate or execute test cases automatically [8], [9], [10], [11], [12], [13].

Metadata and BIT approaches indeed contributed to improve components testability, especially to support specification-based testing. However, there could be situations in which black-box testing of components is not deemed sufficient and combining implementation and specification-based testing techniques is desirable. In fact, these two techniques are meant to find different types of failures and their combined application may provide higher confidence [14].

The main purpose of this paper is to present an overview of the approach called BISTFaSC (Built-In Structural Testing Facilities for Software Components) that was designed to improve components testability by embedding testing facilities into software components to support program-based testing techniques. Components with testing facilities are called testable components, as in traditional BIT techniques. They have probes inserted by instrumentation to record information about their execution (paths and data exercised). Tester components are associated with testable components to define the boundaries of a test session and to generate coverage analysis based on the information collected during a test session.

BISTFaSC is a generic approach that can be applied to different technologies and platforms since it only defines guidelines to transform software components into testable components. However, to validate our approach, we also present an implementation to generate testable components written in Java. This implementation is used to validate the feasibility of the approach by means of an exploratory study. The exploratory study is used to present how testable components can be used at the user side during testing activities.

This paper is organized as follows. Section II presents basic concepts of built-in and structural testing. Section III introduces the main concepts of BISTFaSC and Section IV shows its Java implementation. Section V presents an exploratory study conducted to validate and to understand the main concepts of the approach. Section VI discusses the related work and Section VII provides some concluding remarks and future directions.

II. Background

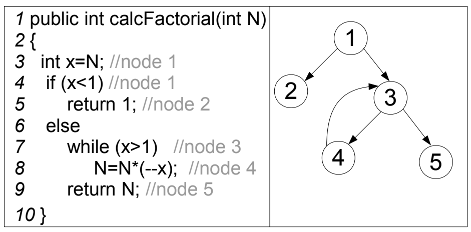
A. Structural Testing

Testing is the process of executing a program with the intent of finding faults [14]. Structural testing focuses on testing the structure of a program. Test cases are generated to exercise the internal logic considering instructions, paths and data. Test data is derived from the implementation according to criteria used to determine whether the program under test is completely tested [15], [14]. Three well known structural criteria are the following: all-nodes, all-edges and all-uses.

The all-nodes and all-edges criteria consider the execution control of the program and they are known as control-flow criteria [16]. It is common to adopt a model called Control-Flow Graph (CFG) to represent the inner structure of the program to support the analysis of control-flow criteria. In this particular graph, each node represents a block of instructions without flow deviation and each edge represents a possible transition from a block to another. The all-nodes criterion requires that every node of the CFG be executed at least once, while the all-edges criterion requires the execution of every edge at least once.

The all-uses criterion takes information about the program data flow. Rapps and Weyuker [17] proposed an extension of the CFG called Def-Use Graph (DUG) to add information related to variable usage. The classical all-uses criterion requires that every definition of a data object and its associated use be executed at least once.

Structural testing criteria are used to derive test requirements that should be met by test cases execution. Examples of test requirements is presented in Figure 1, which shows an example of a Java method and its CFG associated. The structural test requirements of the method `calcFactorial` are presented in Table I. The all-nodes requirements define which blocks of instructions should be executed by the test cases. The all-edges requirements define which transitions from a node to another should be exercised at least once. In the all-uses requirements, (x,4,(3,5)) means that the variable `x` is defined in node 4 and used in a decision that takes the control flow from node 3 to node 5. The requirement (x,1,4) means that variable `x` is defined in node 1 and it is used in a computation in node 4. Test cases should be created to exercise all possible definition and use pairs.



```
public int calcFactorial(int N)
{
  int x=N; //node 1
  if (x<1) //node 1
     return 1; //node 2
  else
     while (x>1)  //node 3
          N=N*(--x);  //node 4
     return N; //node 5
}
```

Fig. 1. Source code and the CFG of the operation `calcFactorial`

TABLE I. Test requirements of `calcFactorial`

Criterion	Test requirements
All-nodes	1, 2, 3, 4, 5
All-edges	(1,2), (1,3), (3,4), (3,5), (4,3)
All-uses	(N,1,5), (N,1,4), (N,4,5), (x,4,(3,5)), (x,4,(3,4)), (x,1,4), (x,1,(3,4)), (x,1,(3,5)), (x,1,(1,2)), (x,1,(1,3))

After executing the test cases, a coverage analysis is performed to measure how many test requirements were satisfied, which indicates how much of the structure of the program was actually exercised during the test session.

B. Built-In Testing (BIT)

According to Harrold et. al [7], the lack of information regarding COTS brings many problems to the validation, to the maintenance and to the evolution of component-based applications. BIT is one of the approaches that stands out from the literature to handle the issue of lack of control and information in component testing.

BIT is an approach created to improve the testability of software components based on the self-testing and defect-detection concepts of electronic components. The general idea is to introduce functionalities into the component to provide its users with better control and observation of its internal state [10], [11], [12]. A component developed under the BIT concepts can also contain test cases or the capability to generate test cases. Such components are commonly called testable components.

Components without testing facilities are called regular components in the remainder of this paper. Interfaces and operations of regular components are called, respectively, regular interfaces and regular operations. When a regular component becomes a testable component, it has a regular interface with regular operations as well a testing interface with operations to support testing activities.

Based on the concepts of BIT, an European group called *Component+* designed a testing architecture composed by three components [8], [9], [10], [11], [12]:

- Testable Component: it is the component under test which incorporates testing facilities.
- Tester Component: implements or generates test cases to test the regular operations of the testable component.

- Handler Component: it is used to throw and handle exceptions. This component is especially important in fault-tolerant systems.

The testable components of the *Component+* architecture have testing interfaces, whose operations control a state machine to support model based testing. A generic example of a testable component is presented in Figure 2. Component users can set component testers to testable components. Component testers execute test cases against testable components, evaluate autonomously its results and output a test summary [18].

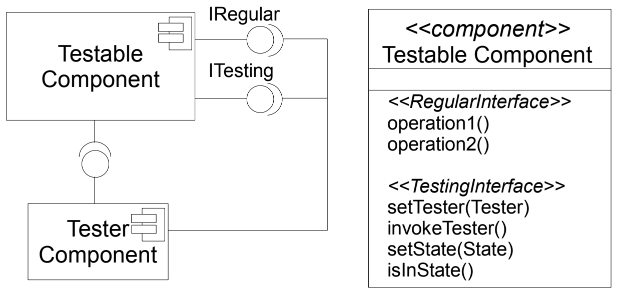


Fig. 2. A *Component+* testable component.

A testable component can operate in two modes: in normal and in maintenance mode. The testing capabilities of the testable component are turned off in the normal mode and they are turned on in the maintenance mode.

Atkinson and Gross [9] proposed a BIT method integrated with the KobrA approach [19] to validate contracts between components and their users during deployment time. Lima et. al [20] developed a model-driven platform to generate testers according to this method. Brenner et. al [21] developed an environment in which tasks can be set to activate tester components in many situations to perform testing activities during runtime. This environment can also react according to the test results. For example, the system can be shut down and components can be replaced.

There are situations in which COTS have no BIT capabilities. Barbier et. al [22] created a library called BIT/J and a set of tools to allow users to introduce testing facilities into COTS developed in Java. BIT/J generates the testable and the tester components automatically. The testable component code must be changed to manually include states and possible transitions. The tester component code must also be changed to include test cases. Bruel et. al [23] proposed an evolution to the BIT/J library using aspect oriented programming.

In summary, BIT approaches focus on providing support to specification-based testing, which is a natural alternative given the black box nature of software components. The approach presented in this paper, on the other hand, proposes facilities to support structural testing.

III. BISTFaSC: Built-In Structural Testing Facilities for Software Components

Components are usually provided as a black box and source code is seldom available to users who cannot conduct program-based testing [3]. The BISTFaSC approach was devised to improve the testability of Component-Based Systems by adding testing facilities into software components at the provider side to support structural testing at the user side, but without revealing the source code of the component. Figure 3 shows an illustration of the approach.

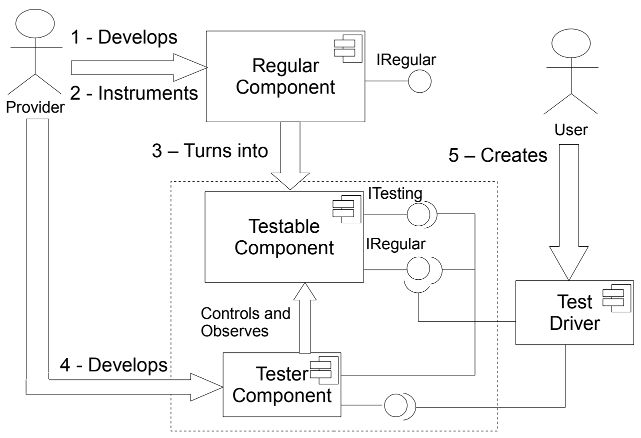


Fig. 3. An illustration of the BISTFaSC approach

The BISTFaSC approach was designed to be used by component providers since they should be interested in providing components with high testability to their users. Testability is an important quality indicator since its measurement leads to the prospect of facilitating and improving a test process [24], [25]. Then, providing components with high testability can represent an advantage in competition [3].

In BISTFaSC, component providers develop regular components and include structural testing facilities by instrumentation. Components with structural testing facilities are called testable components to comply with the BIT approaches found in the literature. The providers also develop tester components to control and observe testable components, according to the recommendations proposed by the *Component+* architecture [9], [10], [11], [12]. Both testable and tester components are packed and made available to external users.

Component users purchase a component with testing capabilities and develop a test driver to execute test cases against the testable component. The test driver uses the tester component to put the testable component in testing mode (as the maintenance mode in *Component+*) and execute the test cases to exercise the regular operations of the testable component. Then, the tester component is used again to put the testable component in regular mode and generate coverage analysis.

BISTFaSC is a generic approach and may be possibly used to create testable components with any implementation technology and platform. This approach only provides guidelines to help providers creating testable components and users to use the available structural testing facilities. These guidelines are presented in details as follows.

A. Guidelines To Create Testable Components At The Provider Side

1) Development of the Regular Component: this stage represents the regular component engineering activities conducted to develop components. Regular components, in this paper, are components that do not have any intended testing facilities to support testing activities at the user side. Component providers employ specific programming tools and languages to develop their regular components for specific target platforms or frameworks. They can also develop test cases to perform quality assurance activities.

2) Instrumentation of the Regular Component: the purpose of this activity is to modify the regular component to give it the capability to support the structural testing at the user side. This modification process is called instrumentation. Instrumented regular components are called testable components. Instrumentation is a technique in which probes are inserted into all the component's code. Probes are instructions placed in specific locations of the code to log execution data.

The implementation of the probes depends on which information must be collected during the component execution. The information to be collected depends on the structural testing criteria supported by the testable component. If the testable component provides coverage analysis only for the all-nodes and the all-edges criteria (control flow), for example, the probes must log information about execution paths. If the testable component also provides coverage analysis for the all-uses criterion (data-flow), for example, the probes must also log data related to variable definition and usage. The data collected by the probes must be stored somewhere. A database or an XML file could be used, for example.

The instrumentation process must also collect and store the test requirements of the component according to the criteria employed to implement the probes. The test requirements could be sent to a database or written in an XML file, for example. A standard format to express the test requirements of the component and the data collected from its execution must be defined. If, for example, the test requirement for the deviation flow from Node 12 to Node 17 is expressed as `Node 12 -> Node 17`, the probes must register this information using the same pattern when the deviation flow goes from Node 12 to Node 17 during the component execution. This is important because the coverage analysis will use the log generated by the probes to define which test requirements were satisfied and which were not.

Probes are instructions that record data into files or databases (I/O operations), which may slow down the performance of the component. To avoid the overhead that may be brought by probes execution, BISTFaSC defines that testable components should operate in two modes: in regular and in testing mode. The probes should be turned off when the testable component is in regular mode and should be turned on when the testable component is in testing mode. In general, testable components operate in testing mode only when they are being executed in the context of a test session.

The instrumentation process can be manually done by the providers or fully automated by a tool. Performing this process manually, however, brings many effort to providers and it is also error prone. Then, BISTFaSC recommends that this process is performed by a tool that should be implemented according to the target implementation.

3) Development Of The Tester Component: the instrumentation process collect test requirements and insert probes into regular components that are transformed into testable components. This process, however, only prepares the testable component to generate data to support structural testing. The objective of this activity is to develop the tester component that controls and observes the testable component.

The tester component interface must expose operations to define the boundaries of a test session (control) and to generate a coverage analysis report based on the information collected from a test session (observe). The boundaries of a test session can be defined by operations developed to start and to finish a test session. The testable component starts to operate in testing mode when the operation of the tester component to start a test session is called. Consequently, the probes of the testable components are turned on at this point and start to log execution data. The testable component must return to normal mode and the probes must be turned off when the operation of the tester component to finish the test session is invoked.

The tester component also defines an operation to report the coverage analysis of a test session execution. When this operation is requested, the tester component use the log generated by the probes and the test requirements of the testable components to calculate the coverage measure. The coverage report may be presented in many ways. BISTFaSC suggest four coverage analysis profiles:

- Operations: presents the coverage for all operation of each class of the component (considering an object oriented implementation).
- Interface: presents the coverage for the operations of the component's interface.
- Classes: presents the coverage for each class of the component.
- Component: presents the coverage for the whole component.

The tester component may also expose other operations to provide users with more testing facilities, but it must at least expose operations to define the boundaries of a test session (control) and to perform coverage analysis (observation). It is important to notice that the operations to support the testing activities are not inserted into the testable component, but they are exposed by the tester component. The interface of the regular component remains the same and it still can be used only through the interface.

4) Packing The Testable And The Tester Component: the goal of this activity is to pack all resources related to the testable component. The component provider must pack the testable component and its libraries and resources along with the tester component and the resource (database or file, for example) used to record the test requirements of the testable component. All these assets must be packed together because the testable and the tester component will be executed at the user side. The tester component, for example, needs to access the test requirements to generate the coverage analysis report.

B. Guidelines To Use Testable Components At The User Side

Component users receive the package with the testable and the tester component along with all resources and libraries required. There is no difference between using a regular component or a testable component in a regular mode. Testable components in regular mode have no testing facility activated and the tester component is not required.

The component user cannot use the testing facilities provided by testable components directly. The user can only invoke the regular operations of the testable component. The tester component must be called to put the testable component in testing mode before conducting structural testing activities. Figure 4 shows an illustration of this process.

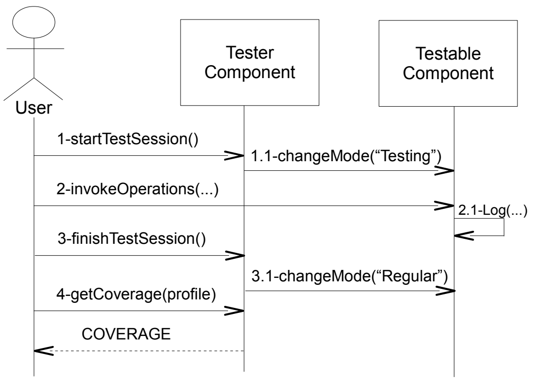


Fig. 4. An illustration of a test session conducted by a testable component user

The user calls an operation of the tester component to start a test session. The tester component access the testable component and change it to the testing mode. From this point on, every execution of the testable component will be logged by its probes. Then, the user executes a test set against the testable component, calling its regular operations. Next, the user invokes the tester component to finish the test session. The tester component returns the testable component to the regular mode. Finally, the user invokes the operation of the tester component to produce a coverage analysis report, which should be generated by the tester component.

IV. A Java Implementation of BISTFaSC

BISTFaSC is a generic approach and may be applied for any component implementation, technology or target platform. We validate the main concepts of the approach by means of an implementation of testable and tester components written in Java. We show, in this section, how the regular Java components are instrumented and how they are controlled and observed by the tester components. The use of testable components generated by this particular implementation is presented in the next section.

For the sake of simplicity, regular components are implemented in Java without considering any specific details of component platforms (such as EJB). Components are packed into JAR files along with resources and libraries required.

The instrumentation process of this particular implementation inserts probes into regular components to collect control (all-nodes and all-edges) and data-flow (all-uses) data. Performing this process manually requires much effort and is error prone. Thus, we decided to develop a tool called BITGen to transform regular components into testable components. Figure 5 presents a simplified architecture of the BITGen tool.

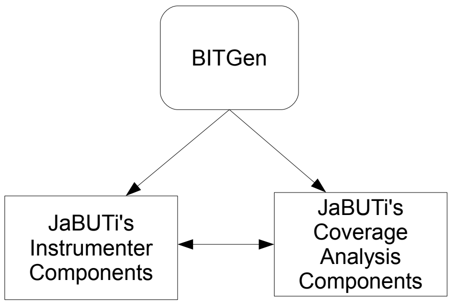

BITGen
JaBUTi's Instrumenter Components
JaBUTi's Coverage Analysis Components

Fig. 5. The simplified architecture of the BITGen tool.

BITGen receives a regular component packed within a JAR file and generates another JAR file containing its testable version. If BITGen receives a JAR file called `Comp.jar`, for example, it generates a package with the name `Comp_BIT.jar`. Figure 6 presents the model of the testable component package generated by BITGen. The package contains the tester (BITTester) and the testable component, an XML file for the test requirements, a class called `CoverageMode` and the `CoverageAnalysis` components of the JaBUTi tool. The `CoverageAnalysis` components of JaBUTi are used by `BITTester` to calculate the coverage and `CoverageMode` is used to format the coverage report according to the profiles suggested by BISTFaSC (see Section III-A3).

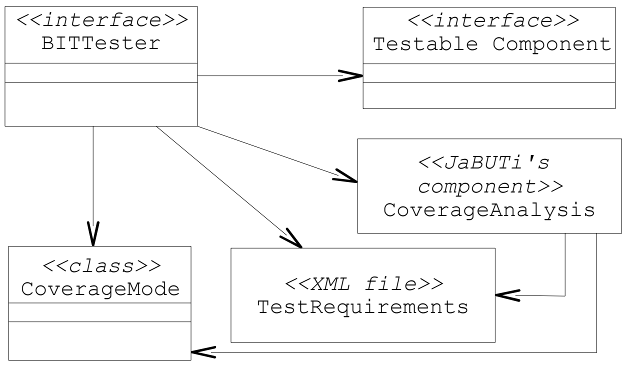

<<interface>> BITTester
<<interface>> Testable Component
<<JaBUTi's component>> CoverageAnalysis
<<class>> CoverageMode
<<XML file>> TestRequirements

Fig. 6. Model of the testable component package generated by BITGen.

BITGen uses the instrumenter components of the JaBUTi (Java Bytecode Understanding and Testing) tool [26] to instrument the regular component. JaBUTi is a structural testing tool that implements intra-method control-flow and data-flow testing criteria for Java programs. The instrumentation is based on the Java bytecode and it is supported by BCEL (Byte Code Engineering Library). During instrumentation, the test requirements regarding the control and data-flow criteria are written to an XML file.

The tester component associated to the testable component is automatically generated by BITGen. Figure 7 shows the interface of the tester component. The tester component is called `BITTester` and exposes operations to control (`startTesting` and `stopTesting`) and to observe (`getCoverage`) the testable component.

```
<<interface>>
BITTester

BITTester()
void startTesting(sessionID)
String startTesting()
void stopTesting()
String getCoverage(sessionID, covMode)
String getCoverage(sessionID, covMode, className)
```

Fig. 7. Interface of the tester component generated by BITGen.

The operation `String startTesting()` initiates a test session and puts the testable component in the testing mode, i.e., it turns on the probes of the testable component. Control and data-flow data are collected by probes and recorded into a trace file when the testable component is executed in this mode. The return of this operation is an identifier automatically generated for the test session initiated. This identifier is also used to name the trace file generated during the testable component execution. If the test session is identified by ”1234”, for example, the trace file generated is the following: `trace_1234.trc`.

The operation `void startTesting(sessionID)` has the same effect as the operation mentioned before, but it receives a session identifier instead of generating it. This operation is useful mainly when the tester wants to perform one coverage analysis for several test sessions recognized by the same identifier. In this case, all information regarding the testable component execution is stored into the same trace file.

The operation `void stopTesting()` finishes a test session by turning off the probes. No information is recorded about the testable component execution from this moment on.

The operation `String getCoverage(sessionID, covMode)` is used to achieve a coverage analysis of a test session identified by the parameter `sessionID`. The report is presented according to the parameter `covMode` (see Section III-A3). In this particular operation, there is no difference between the `Operations` and `Interface` coverage mode. The coverage will be presented for all operations of the component for both profiles.

The operation `String getCoverage(sessionID, covMode, className)` also generates a coverage analysis. The difference from the previous operation is that a class name may be specified as an input parameter. A coverage analysis only for the specified class is generated when the `Classes` profile is used. A coverage analysis is generated only for the operations of the specified class as well when the `Interface` profile is employed.

When one of the `getCoverage` operation is called, the tester component (`BITTester`) finds the trace file associated with the test session identifier and sends it to the `CoverageAnalysis` components of JaBUTi. This component uses the XML file that contains the test requirements of the testable component to calculate the coverage according to the implemented criteria. Finally, `BITTester` uses the class `CoverageMode` to format the coverage report according to the requested profile (Component, Classes, Interface or Operations).

BISTFaSC is intended to be used by component providers to produce testable components for their clients. In this implementation, however, component users can also transform Java COTS into testable components since BITGen instrumentation is based on Java bytecode and the source is not required.

V. Exploratory Study

An exploratory study was conducted to investigate the feasibility of BISTFaSC and to understand the effects of the testable component on a test session executed at the user side. The investigation was performed considering the Java implementation of the approach and a component called `XmlWriter` was used. `XmlWriter` is an open source component that is publicly available in the component provider website[1]. This component is used to output XML code and the user may layer other functionalities on top of the core writing, such as on the fly schema checking, date/number formatting, specific empty-element handling and pretty-printing.

A. *Instrumentation*

This study was performed from the point of view of a component user, but first we had to transform `XmlWriter` into a testable component using the BITGen tool. Figure 8 shows the graphical user interface of BITGen. The tool requires the name and the JAR file with the component to be instrumented, and a local path to write the testable package and to store the data collected by the probes (trace file). After pushing the `Generate` button, BITGen instrumented `XmlWriter` and generated a package called `XMLWriter_BIT.jar`. This package contains all classes, components and resources presented in Figure 6.

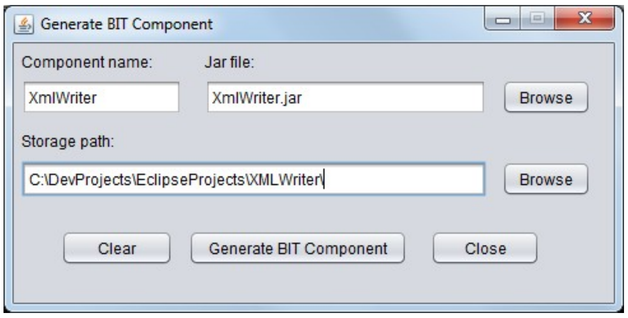

Generate BIT Component
Component name: XmlWriter
Jar file: XmlWriter.jar
Storage path: C:\DevProjects\EclipseProjects\XMLWriter

Fig. 8. User Interface of BITGen

Once the regular component is fully developed and packed along with its libraries and resources, the effort to transform it into a testable component is really low. Considering this implementation, the component provider only has to provide the information required by BITGen and the testable component is generated automatically.

B. *Test Session*

An Eclipse project was created to evaluate the testable version of `XmlWriter`. The package `XmlWriter_BIT.jar` was included into the library references of the project and a test scenario was created. The providers of `XmlWriter`

[1]https://code.google.com/p/osjava/

also published a JUnit test set to test its operations. Instead of creating new test cases to XMLWriter, we used this test set, which is presented in Listing 1. The set up and a tear down method was included in this investigation to control and observe the testable component following the sequence diagram presented in Figure 4. In the set up phase the tester component is invoked to start a test session and in the tear down it is called to stop the test session and to get a coverage analysis.

The `setUp` method runs only once before the test cases execution because of the annotation `@BeforeClass`. The tester component `BITTester` is instantiated in Line 8 and a test session is started in Line 9. The identifier of the test session is recorded by the variable `sessionID`.

The `tearDown` method runs only once after the test cases execution because of the annotation `@AfterClass`. The test session is finished in Line 14 and the coverage report is requested in Line 16. In this case, the coverage was requested to be presented for the whole component profile (`CoverageMode.COMPONENT`). The coverage returned as a String and it was written in the output console of the application (line 17).

Listing 1. Test set of XmlWriter.

```
01 public class XmlWriterTest{
02
03 private static BITTester bitTester;
04 private static String testSessionID;
05
06 @BeforeClass
07 public  static void setUp(){
08    bitTester = new BITTester();
09    testSessionID = bitTester.startTesting();
10 }
11
12 @AfterClass
13 public static void tearDown(){
14    bitTester.stopTesting();
15    String coverage;
16    coverage=bitTester.getCoverage(testSessionID ,
      CoverageMode.COMPONENT);
17    System.out.println(coverage);
18 }
19
20 @Test
21 public void testXmlWriter01() throws IOException {
22    StringWriter sw = new StringWriter();
23    XmlWriter xw = new SimpleXmlWriter(sw);
24    xw.writeEntity("unit");
25    xw.endEntity();
26    xw.close();
27    assertEquals(sw.toString(), "<unit/>");
28 }
29
30 @Test
31 public void testXmlWriter02() throws IOException {
32    StringWriter sw = new StringWriter();
33    XmlWriter xw = new SimpleXmlWriter(sw);
34    sw = new StringWriter();
35    xw = new SimpleXmlWriter(sw);
36    xw.writeXmlVersion("1.0", "UTF-8");
37    xw.writeComment("Unit test");
38    xw.writeEntity("unit");
39    xw.writeEntity("test").writeAttribute("order","1").
      writeAttribute   ("language","english").endEntity();
40    xw.writeEntity("again").writeAttribute("order","2").
      writeAttribute("language","english").writeEntity("
      andAgain").endEntity().endEntity();
41    xw.endEntity();
42    xw.close();
43
44    assertEquals(sw.toString(), getTest2Output() );
45 }
46
47 private String getTest2Output() {
48   return "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" +
49          "<!--Unit test-->" +
50          "<unit>" +
51          "<test order=\"1\" language=\"english\"/>" +
52          "<again order=\"2\" language=\"english\"><
   andAgain/</again>" +
53          "</unit>";
54  }
55 }
```

Lines 20 to 55 show common JUnit test cases. These test cases were created by the component provider and reused in this investigation.

Table II presents the coverage analysis obtained from the execution of the test set. The first column shows the testing criteria considered during the coverage analysis. The second column displays the amount of test requirements for a specific criterion considering the whole component, i.e., it is the sum of the number of test requirements generated for all operations of all component classes. The third column presents the amount of test requirements that was covered by the test session execution considering the whole component. The fourth column shows the coverage percentage, which is calculated by `CovTReq` over `TReq`.

TABLE II. COVERAGE ANALYSIS OF XMLWRITER

Criterion	TReq	CovTReq	Coverage
All-nodes	100	33	33%
All-edges	84	23	27%
All-uses	238	56	23%

The coverage achieved is low for all criteria. Listing 2 shows an excerpt of the code used to get the coverage report at different granularity levels (presentation profiles). In these cases, the tester can investigate which classes and/or operations are not being exercised satisfactorily.

Listing 2. Script used to get different presentations of the coverage.

```
01 coverage = bitTester.getCoverage(testSessionID ,
     CoverageMode.ALL_CLASSES);
02 System.out.println(coverage);
03
04 coverage = bitTester.getCoverage(testSessionID ,
     CoverageMode.ALL_OPERATIONS);
05 System.out.println(coverage);
06
07 coverage = bitTester.getCoverage(testSessionID ,
     CoverageMode.INTERFACE_OPERATIONS, "SimpleXmlWriter");
08 System.out.println(coverage);
```

Table III presents the coverage obtained for each class of the component (Lines 1 and 2 of Listing 2). The table shows the coverage percentage and the number of covered test requirements over the number of test requirements (CovTReq/TReq) for each class and criterion. This report shows that the coverage reached for the class `XmlUtils` is practically 0% for all criteria. `XmlUtils` is not exercised probably because `SimpleXmlWriter` uses only one of its operations. That is why the coverage of the whole component is too low, even when the coverage for `SimpleXmlWriter` and `AbstractXmlWriter` are reasonable.

TABLE III. COVERAGE ANALYSIS OF XMLWRITER CLASSES

Classes	All-nodes	All-uses	All-edges
SimpleXmlWriter	72% (29/40)	70% (22/31)	66% (56/84)
AbstractXmlWriter	50% (3/6)	100% (1/1)	-
XmlUtils	1% (1/54)	0% (0/52)	0% (0/154)

Table IV presents the coverage obtained for each operation of `SimpleXmlWriter` (Lines 7 and 8 of Listing 2). This table shows only the coverage percentage for each operation and criterion. The sign "-" indicates the absence of test requirements for that criterion. This usually happens when the method has only one node. Then, there are no test requirements for edges and uses. The tester can use this report to see which operations were not exercised and which operations need to be extensively tested.

TABLE IV. COVERAGE ANALYSIS OF SIMPLEXMLWRITER OPERATIONS

Operations	All-nodes	All-uses	All-edges
close	66%	50%	30%
closeOpeningTag	100%	100%	100%
endEntity	87%	77%	76%
getDefaultNamespace	0%	0%	0%
getWriter	0%	-	-
openEntity	100%	-	-
setDefaultNamespace	0%	-	-
writeAttribute	100%	100%	100%
writeAttributes	100%	100%	100%
writeCData	0%	-	-
writeChunk	100%	-	-
writeComment	100%	-	-
writeEntity	66%	50%	44%
writeText	0%	-	-
writeXmlVersion	80%	50%	55%

The component user has not spent much effort to use the testing facilities of the testable component to conduct structural testing activities. Considering this particular implementation and the JUnit framework, the user had to add only 12 extra lines (Lines 06 to 10 and Lines 12 to 18) into the test set code.

C. Performance Overhead Analysis

We performed an analysis to measure the performance overhead brought by the testable component probes. We measured the time it took to execute the test set presented in Listing 1 100 times. First, we tested the regular version of `XmlWriter`. Next, we executed the test set against `XmlWriter_BIT` in regular mode, i.e., the `startTesting` operation was not invoked then the probes were deactivated. Finally, we tested `XmlWriter_BIT` in testing mode, i.e., the `startTesting` operation was called before the probes were activated. Table V shows the results of this analysis.

TABLE V. ANALYSIS OF THE PERFORMANCE OVERHEAD

Component	Time	Overhead
XmlWriter	44 ms	0%
XmlWriter_BIT (probes off)	47 ms	7%
XmlWriter_BIT (probes on)	460 ms	1045%

The overhead brought by the testable component is minimal when the probes are off. When the probes are on, however, the component execution is 10 times slower. Even with this great overhead brought by the probes when they are on, we believe that it is not significant in general because the probes will be on only when the component is under test. The probes are turned off when the component is integrated in an operational environment and in this case the overhead is lower and have less significant impact on the overall performance. However, it may be critical to many real time systems.

D. Size Overhead Analysis

The testable components generated by `BITGen` have their size enlarged by extra code and libraries. The component classes are extended by the probes inserted during instrumentation. Table VI shows a comparison between the size (in bytes) of `XmlWriter` classes before and after the instrumentation.

TABLE VI. ANALYSIS OF THE SIZE OVERHEAD

Classes	Size before	Size after	Overhead
SimpleXmlWriter	4531	5515	22%
AbstractXmlWriter	1417	1795	26%
XmlUtils	3739	5027	34%
All classes	9687	12337	27%

`SimpleXmlWriter` presents the smallest overhead of the Component because it has 15 short operations which does not require too many probes to be instrumented. `AbstractXmlWriter` has only 4 small operations which are usually calls to abstract or interface operations. The size overhead presented by this class is in the average considering all classes (27%).

`XmlUtils` presents the greatest overhead size of the component. `XmlUtils` is smaller and has less operations than `SimpleXmlWriter`, but its operations are bigger and have more control flow deviations and variable usage. This makes a difference regarding the test requirements generated for the structural testing criteria (see Table III). `XmlUtils` generates more test requirements than `SimpleXmlWriter` considering all criteria. The more test requirements are generated for a class the more probes are required to trace its execution.

The size overhead is not so important when components are used in enterprise environments where space and memory are usually widely available. However, it may be crucial when it is embedded into devices with space and memory restrictions.

Regarding size overhead, the major problem of the testable components generated by `BITGen` is the size of the libraries and components required to perform the coverage analysis. Coverage analysis requires components of the JaBUTi tool and libraries to manipulate XML data and Java bytecode. These libraries and components add at least 1MB to the testable component size. Again, it is not a significant overhead in enterprise environments, but it is relevant for restricted devices.

The overhead problem brought by extra components and libraries, however, is not related to the generic approach `BISTFaSC`. The restriction is imposed by its Java implementation which could be improved by reducing the number of library dependencies. The alternative to overcome this situation is to use two versions of the component: a testable version to conduct testing activities and a regular version to embed into devices with restrictions.

VI. RELATED WORK

Several approaches have been proposed in the literature as an attempt to improve component testability and support testing activities on the user side. The already mentioned BIT approaches (see Section II-B) introduce testing facilities into software components to control state machines and execute/generate test cases, validate contracts and invocation sequences, for example [9], [10], [11], [22], [23], [12], [13], [20].

The BISTFaSC approach is similar to most of these approaches since it also introduces testable and tester components associated. Testable components have testing facilities that can be turned on and off. The difference is that BISTFaSC introduces testing facilities to support structural testing while the other approaches usually support specification-based testing.

Testable components are intended to generated by the component providers in classical BIT approaches and also in BISTFaSC. When COTS are not equipped with testing facilities by their providers, however, component users can use the Java implementation of the approach to instrument Java COTS. This process is similar to the approach supported by the BIT/J library [22].

Teixeira et. al [27] proposed a tool based on JaBUTi [26] to support the structural testing of Java components. The component provider can instrument the component and create test cases for it. The component user receives the instrumented component that is packed with test cases, test case descriptions and coverage information. The user thus can use the tool to create test cases using the JUnit framework and get a coverage analysis on the test set execution. This coverage can be compared to the coverage reached by the provider during development time and test cases packed with the component can be reused.

There are many differences between BISTFaSC and the tool proposed by Teixeira et. al [27], although they are meant to support structural testing of software components. BISTFaSC follows the concepts of BIT and defines a tester component to the testable component. Moreover, the testing facilities that support structural testing activities are embedded into the component code and libraries therefore no supporting tool is required.

Eler et. al and Bartolini et. al proposed an approach to produce testable services with structural testing capabilities [28], [29]. Their approach is similar to BISTFaSC because they propose to transform regular services into testable services by instrumentation. In this case, however, the testable service has to implement the operations to define the boundaries of a test session and to get the coverage information. The coverage information is generated by a third party service instead of being calculated by the testable service itself. The testable components of BISTFaSC do not have their operations augmented and the testing facilities are implemented by the tester component associated. Moreover, the coverage information is internally calculated then no third party component is required.

VII. Conclusion

This paper presented a BIT solution to introduce facilities into software components at the provider side to support structural testing activities at the user side. The approach is generic and it is called BISTFaSC. A Java implementation of the approach was presented and used to perform an exploratory study by generating Java testable components of third party components. The exploratory study showed that the component provider does not have much effort to generate testable components and the component user has only to add a few extra code into their test class to take advantage of the testing facilities provided by testable components.

The performance overhead brought by the probes added during instrumentation is not significant when the probes are off (regular mode), but it may be significant for real time applications. The size overhead is also not relevant considering enterprise environments, but it may be significant if the target environment is a device with space and memory restrictions.

We believe this approach can bring benefits to CBSE, since it allows users applying both specification and implementation based testing techniques. Combining these two testing techniques may provide higher confidence to component providers and users.

Moreover, there are several software engineering approaches in which components are used as the building blocks of applications, such as object oriented frameworks, service-oriented computing and software product lines. Testable components can contribute with the testing activities conducted for all of these type of applications.

The coverage information alone cannot help testers to improve their test set to increase the coverage when it is low. It only gives a clue of how much of the component was exercised during a test session, which is valuable information itself. As future work, we intend to propose metadata to help testers to understand which test cases should be created to improve the coverage achieved, but without revealing the source code. The idea is to use test metadata as suggested by Harrold et. al [7] and used by Eler et. al [28] for testable services.

We also intend to explore how structural testing facilities could be used to perform component monitoring and regression test case selection and reduction. Moreover, we would like to perform more rigorous evaluation of the approach with bigger components. We also want to evaluate the usability of the approach at the user and at the provider side.

Acknowledgment

The authors would like to thank the Brazilian funding agency CNPq for its support.

References

[1] C. Szyperski, *Component Software, Beyond Object-Oriented Programming*, ser. Component Software. Addison-Wesley, 2002.

[2] J. Cheesman and J. Daniels, *UML Components: A simple process for specifying component-based software*. Addison-Wesley, 2000.

[3] S. Beydeda, "Research in testing cots components - built-in testing approaches," in *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, ser. AICCSA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 101–vii. [Online]. Available: http://dl.acm.org/citation.cfm?id=1249246.1249567

[4] E. J. Weyuker, "Testing component-based software: A cautionary tale," *IEEE Software*, vol. 15, no. 5, pp. 54–59, 1998.

[5] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability*, pp. n/a–n/a, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.1470

[6] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," Tech. Rep., 1990. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.1990.101064

[7] M. J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, and M. L. Soffa, “Using component metadata to support the regression testing of component-based software,” Tech. Rep. GIT-CC-00-38, 2000.

[8] Y. Wang, G. King, and H. Wickburg, “A method for built-in tests in component-based software maintenance,” in *Proceedings of the Third European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 1999, p. 186.

[9] C. Atkinson and H. gerhard Gros, “Built-in contract testing in model-driven, component-based development,” in *In ICSR-7 Workshop on ComponentBased Development Processes*, 2002.

[10] Y. Wang and G. King, “A european COTS architecture with built-in tests,” in *Proceedings of the 8th International Conference Object-Oriented Information Systems*. London, UK: Springer-Verlag, 2002, pp. 336–347.

[11] J. Hornstein and H. Edler, “Test reuse in cbse using built-in tests,” 2002.

[12] H.-G. Gross, *Component-Based Software Testing with UML*. Springer, 2005.

[13] L. C. Briand, Y. Labiche, and M. M. Sówka, “Automated, contract-based user testing of commercial-off-the-shelf components,” in *Proceeding of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 92–101.

[14] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2004.

[15] B. Beizer, *Software testing techniques (2nd ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[16] H. Zhu, P. A. V. Hall, and J. H. R. May, “Software unit test coverage and adequacy,” *ACM Computing Surveys (CSUR)*, vol. 29, no. 4, pp. 366–427, 1997.

[17] S. Rapps and E. J. Weyuker, “Selecting software test data using data flow information,” *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, Apr. 1985.

[18] S. Beydeda and V. Gruhn, “State of the art in testing components,” in *International Conference on Quality Software. IEEE Computer*. Society Press, 2003, pp. 146–153.

[19] C. Atkinson, J. Bayer, and D. Muthig, “Component-based product line development: the kobra approach,” in *Proceedings of the first conference on Software product lines : experience and research directions: experience and research directions*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 289–309. [Online]. Available: http://dl.acm.org/citation.cfm?id=355461.357556

[20] H. S. Lima, F. Ramalho, P. D. L. Machado, and E. L. Galdino, “Automatic generation of platform independent built-in contract testers. Simposio Brasileiro de Componentes, Arquiteturas e Reutilizacao de Software,” 2007.

[21] D. Brenner, C. Atkinson, B. Paech, R. Malaka, M. Merdes, and D. Suliman, “Reducing verification effort in component-based software engineering through built-in testing,” in *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 175–184.

[22] F. Barbier, N. Belloir, and J. M. Bruel, *Chapter: Incorporation of Test Functionality into Software Componentes. In book: COTS-Based Software Systems*. Springer, 2003.

[23] J.-M. Bruel, J. Araújo, A. Moreira, and A. Royer, “Using aspects to develop built-in tests for components,” in *The 4th AOSD Modeling With UML Workshop*, 2003.

[24] W. T. Tsai, J. Gao, X. Wei, and Y. Chen, “Testability of software in service-oriented architecture,” in *Proceedings of the 30th Annual International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 163–170.

[25] L. O’Brien, P. Merson, and L. Bass, “Quality attributes for service-oriented architectures,” in *Proceedings of the International Workshop on Systems Development in SOA Environments*. Washington, DC, USA: IEEE Computer Society, 2007, p. 3.

[26] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong, “Establishing structural testing criteria for java bytecode,” *Software Practice & Experience*, vol. 36, no. 14, pp. 1513–1541, 2006.

[27] V. S. Teixeira, M. E. Delamaro, and A. M. R. Vincenzi, “Fatesc - uma ferramenta de apoio ao teste estrutural de componentes,” in *Sessão de ferramentas - XXI SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE*. New York, NY, USA: ACM Press, 2007b, pp. 7–12.

[28] M. Eler, A. Bertolino, and P. Masiero, “More testable service compositions by test metadata,” in *6th IEEE International Symposium on Service Oriented System Engineering*, vol. 1, no. 1. Washington, DC, USA: IEEE Computer Society, 2011, pp. 204 –213.

[29] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, “Bringing white-box testing to service oriented architectures through a service oriented approach,” *Journal of Systems and Software*, vol. 84, pp. 655–668, April 2011.