An Aspect-based Environment for COTS Component Testing

Macario Polo

Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad 4
13071 Ciudad Real, Spain
macario.polo@uclm.es

Alejandra Cechich

Departamento de Ciencias de la Computación
Universidad Nacional del Comahue
Buenos Aires 1400
8300 Neuquén, Argentina
acechich@uncoma.edu.ar

Abstract: Component qualification is one of the major steps in the development of component-based applications. Several techniques are used to enrich documentation by structuring and classifying components' metadata. In that line, aspect information has been used to help implement better component interfaces and to encode knowledge of components' capabilities. In this paper, we describe an environment to generate and execute aspect-dependent test cases, which is based on an aspect-based categorisation of information for testing.

1 Introduction

Several authors add metadata to components to describe their static and dynamic aspects, what is useful for dealing with several software engineering tasks: in [SW99], annotations are used to perform dependence analysis over both the static and dynamic descriptions; in [Ha01] and [Or00], metadata are added to components to provide generic usage information about a component (class and methods names, for example), as well as information for testing. The common idea in these approaches is to add the component a set of information to help the component user during its use, composition, testing, etc.

Our proposal includes the addition of some aspect-based metadata to facilitate the component testing. In our approach [CP02], we have extended the notion of using metadata for testing COTS components by classifying testing information using aspects. A number of meta-procedures for black box testing have been introduced to allow us to reason about completeness of data supplied for testing, and eventually facilitate the use of test cases along with automated tools for black box testing. Following this approach,

we have developed a tool to add testing metadata to Enterprise Java Beans (EJB) components, additionally generating aspect-depending test cases.
Many techniques for generating test cases take as input the source code of the program and/or a set of formal specifications [DF94][CTC01][Ed01][Of03]. For example, the work in [Of03] generate test data from state-based specifications of systems that must be formally described. The use of symbolic execution has been also widespread used to generate test data, mainly to reach different coverage criteria [Me01][La98][BOP00].
To generate test cases automatically, most of the afore-mentioned approaches require the existence of a separated algebraic specification of the class under test, which must include the functional description of the behaviour of the class operations. However, it is also possible to extract the structure of a class under testing using Reflection, a characteristic of many interpreted programming languages that makes possible to obtain the structure of the class under test, what includes the complete description of its public services, parameter types, return types, visibility, superclasses or implemented interfaces, etc.

In this paper, we propose a method that allows us to generate test cases from Java compiled bytecode. In section 2, we briefly introduce our approach to include aspect-based metadata for testing. Section 3 presents a Java environment that supports and automates our method. Finally, conclusions and future works are addressed in section 4.

2 Modelling aspect-based metadata for testing

A key feature of an aspect-based categorisation of component characteristics is the idea that some components provide certain aspect-related services for other components (or end users) to use, while other components require certain aspect-related services from other components. Each component aspect has a number of aspect details that are used to describe component characteristics relating to the aspect. These details aim to increase developers and end users knowledge about components by providing a more effective categorisation and codification mechanism for the component services.
Therefore, when reasoning about operations of components, we analyse them in terms of particular aspects. For example, developers can describe interfaces in terms of a collaborative work aspect, persistency-related aspect, or user interface aspect. Note that some aspect categorisations may overlap (a service might be considered a user interface aspect as well as a collaborative work aspect). Figure 1 illustrates how some aspects map onto some services. The *Accounting View* component has only services described in terms of user interface aspects, meanwhile the *Editing Balance* component has services described in terms of one aspect as well as overlaps between the user interface aspect and the collaborative work aspect, and the collaborative work aspect and the persistency aspect.

2.1 Meta-Procedures for Aspect-based Testing

Metadata and aspects can extend the documentation of a component to define and validate its test cases. For example, in our approach documentation to define test cases is grouped into a meta-object used as a descriptor of a base-object.
To introduce the approach, a number of concepts must be defined. We firstly define the notion of *aspect scope* of a method as the set of all aspects that influence a particular component's method. For the example introduced previously, the aspect scope of all the methods in *Editing Balance* is the following:

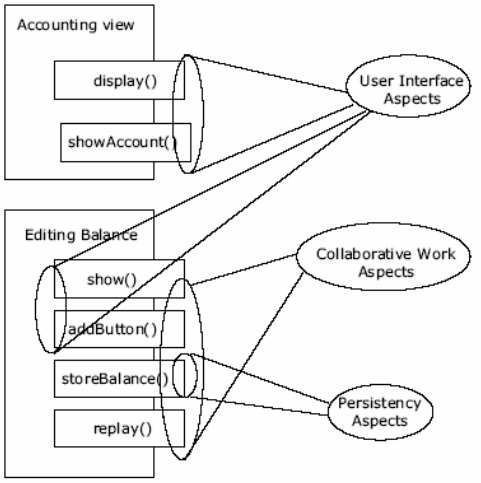

Figure 1. Illustration of component aspects

ASc = {(show(), {User Interface, Collaborative Work}), (addButton(), {User Interface, Collaborative Work}), (storeBalance(), {Collaborative Work, Persistency}), (replay(), {Collaborative Work, Persistency})}

Identifying and creating test cases relevant for a component involve analysing test cases for a single method as well as test cases for multiple-methods. Defining test cases depends on several factors such as the component's state, the invariants on the functionality associated to the component, and preconditions and post conditions of each method. These factors are extended when we consider aspects by including the aspect scope as we defined previously.
Then, a single-method treatment for aspect-oriented generation of test cases can be enunciated[1]. For example, the method show() in Figure 1 is affected by two aspects – "User Interface" and "Collaborative Work". We firstly select the pre/post conditions relevant to the aspect "User Interface" and find test cases by using traditional techniques for black box testing. After that, we proceed doing the same for the aspect "Collaborative Work". Once all aspects have been considered, we analyse all test cases

[1] We refer the reader to [CP02] for a detailed specification of treatments.

to identify incompatibility or inconsistency among them generating a new set of test cases, where tests are compatible. Finally, several analyses of properties of test cases can be done. In particular, a completeness analysis that checks that all aspects have been considered and hence all domains have been consolidated could be useful for identifying a wider spectrum of test cases.

A more complete selection of test cases also involves analysing valid ranges from a combined view of methods, so aspects affecting methods can be used as a way of partitioning the testing space. Then, we have defined the notion of *relevant methods* as the set of all methods affected by a particular aspect. For the example, relevant methods of all the aspects are defined as follows:

RM = {(User Interface, {show(), addButton()}), (Collaborative Work, {show(), addButton(), storeBalance(), replay()}), (Persistency, {storeBalance()})}

Finally, methods are relating each other by sharing a set of aspects that influence their behaviour. This situation leads to an "aspect overlap'" meaning that methods tested according to previous procedures are also influenced by compatibility of common ranges. To clarify this point, just consider our example of Figure 1. Firstly, we produced test cases for show() from the point of view of "User Interface", addButton() from the point of view of "User Interface", show() from the point of view of "Collaborative Work", and so on. Then, we continued evaluating methods from the same point of view, such as show() and addButton() from the point of view of "User Interface". But show() and addButton() should also be analysed considering that both methods share different points of view or aspects ("User Interface" and "Collaborative Work"), which could constrain our test case domain.
Then, we have defined the notion of *common aspects* as the set of relations between all methods affected by a particular set of aspects. For the example, common aspects are defined as follows:

CA = {({User Interface, Collaborative Work}, {show(), addButton()}), ({Persistency, Collaborative Work}, {storeBalance()})}

2.3 Main data structures for testing

Following the ideas of [DF94] and [CTC01], a test case will be composed by a sequence of calls to methods in the component under test. Each method in the sequence will take the adequate values to be compiled and executed. After the test case execution, it is important to compare the obtained component's state with the expected one; so, the methods in the component will be annotated with a set of postconditions that could be used to check both states. Obviously, depending on the granularity of these annotations, the result and goodness of the comparison can be more or less effective. However, it is always important to add the adequate constraints in order to automate the comparison stage.

From the set of methods in both interfaces of the component under test, a simple algorithm can be used to generate all the possible combinations of call to methods. Afterwards, each combination will be applied to a new algorithm that generate the test cases depending on the parameters' types. Following the testing terminology [FG99], we call *Test script* to each of these combinations.

Since Java is the platform where EJBs run, we have used some of the classes included in the *java.lang.reflect* package to have a suitable representation of the data structures involved in the testing process, that makes possible the easy manipulation of interfaces, methods, types, etc. So, we have reused the *TClass* (*T*esting *Class*) defined in a previous work [PPR01][JPP03] to manage the component under test. *TClass* knows a *java.lang.Class* object, from which the tool can recover the set of methods in both interfaces. *TClass* also includes the *getTestScripts(int maxLength, String regularExpression)* method, that generates all the possible combinations of calls to methods in the interface (i.e., test scripts): the first parameter represents the maximum number of calls in the test script, and the second one is a regular expression that avoids the generation of undesired combinations (for example: if we are testing a banking account, maybe we do not want to test the sequences of calls starting with a withdrawal).

Therefore, there is also a *TestScript* class in our system, that includes the *getTestCases(String valuesFileName)* to generate test cases. Its only parameter represents the path to a file containing the values to be used to generate the test cases, although it can be null if the user desires to provide the testing values by hand. The method generates test cases making all the possible combinations of testing values with methods in the corresponding test script. Therefore, the number of test cases can be easily huge and, if we generate a testing method for each test case, the testing file could not be manageable, readable and, even, compileable by a powerful computer. Fortunately, as all the test cases proceeding from the same test script will only differ in the parameter values, it is possible to group all of them in a single method that has so many arrays of values as parameters in all the methods. The assignment of values to parameters is controlled by a set of loops.

The *TestCase* class is in charge of constructing the testing methods, that are saved to a standard Java file. Besides the calls to the methods in the interfaces of the component under test, each test method contains also the code corresponding to the pre and postconditions of the method.

In [DF94] and [CTC01], "test cases consist of a pair of sequences along with a tag", that can be "equivalent" or "non-equivalent". Both sequences of the test case are applied to the same object; so the tag is "equivalent" or "non-equivalent" depending on the state of the objects. For us, a test case will be composed by a message sequence that will be checked against the possible set of pre and post conditions annotating each method in the sequence according to its influencing aspects. In this way, there is no a single *expected value* for the test case, but a set of possible right values, whose cardinality depends on the amplitude of the constraints being tested.

3 An aspect-based environment for testing

Following our approach, we have developed a tool to add metadata for testing EJB components. Using two predefined interfaces (*EntityBean* and *SessionBean*), developers can build three types of EJBs:

- Stateless beans receive requests from client applications, process them and send them a response. They do not maintain information about the connection with the client.
- Stateful beans maintain information about the connection and can then receive multiple related requests at different moments and provide a set of also-related responses.
- Entity beans maintain information about the connection but, moreover, their instances represent persistent objects that are saved in some kind of secondary storage.

Stateless and stateful EJBs must implement the *SessionBean* interface, whereas entity EJBs implement the *EntityBean* one. EJBs are managed by a component container, which receives client requests through the *Remote* interface of the component. Previously, the client must have located the desired component via its *Home* interface. So, the *Remote* interface includes the declaration of the business methods, which are those used by the clients to make use of the component functionalities. The *Home* interface also includes the header of some methods to find instances and to provide persistence to instances (in Entity EJBs). In practice, the component must implement the methods defined in both the *Remote* and the *Home* interface, although this point is later checked at runtime by the component container since there is no direct implementation relationship from the component to its interfaces, that is, the compiler does not detect whether a method in either the *Remote* or the *Home* interface is not implemented by the component (even, the names of the methods in the *Home* interface are not equal in the component).

This fact is illustrated in Figure 2, which shows in the right side the indirect nature of the implementation relationship among the *AccountEJB* component and its two interfaces. Both interfaces are actually exposed by the container as a mean to communicate the component with clients: in fact, the client application uses the component functionality by using the *Account* interface (the *Remote* one), being the component container in charge of linking the interface with the component.

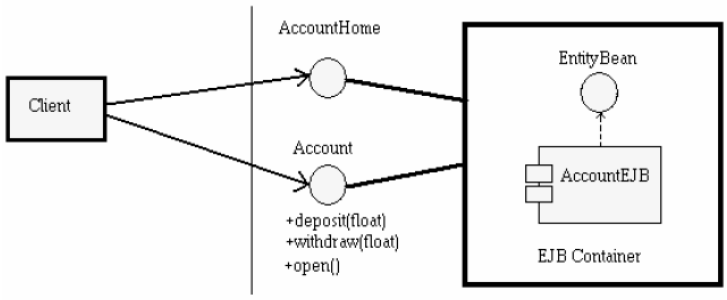

Figure 2. Client and server sides in a generic EJB application

3.1 Description of the tool

Figure 3 shows the main screen of the tool: from here, the user can select the EJB to be tested and sets the Remote and the Home interfaces. Then, the tool extracts the set of locally defined methods of each interface and shows their signatures under each interface name. Through the Aspects group of widgets, the user assigns method from both the remote and the home interface: in the figure, the *getBalance, withdraw, deposit* and *findAll* methods are being assigned to the *Business* aspect. The different groupings in aspects can be seen in a single Aspect editor (accessible via the "Aspect explorer" button), that allows to see what methods in each aspect are, as well as the addition or removing of methods in each aspect.

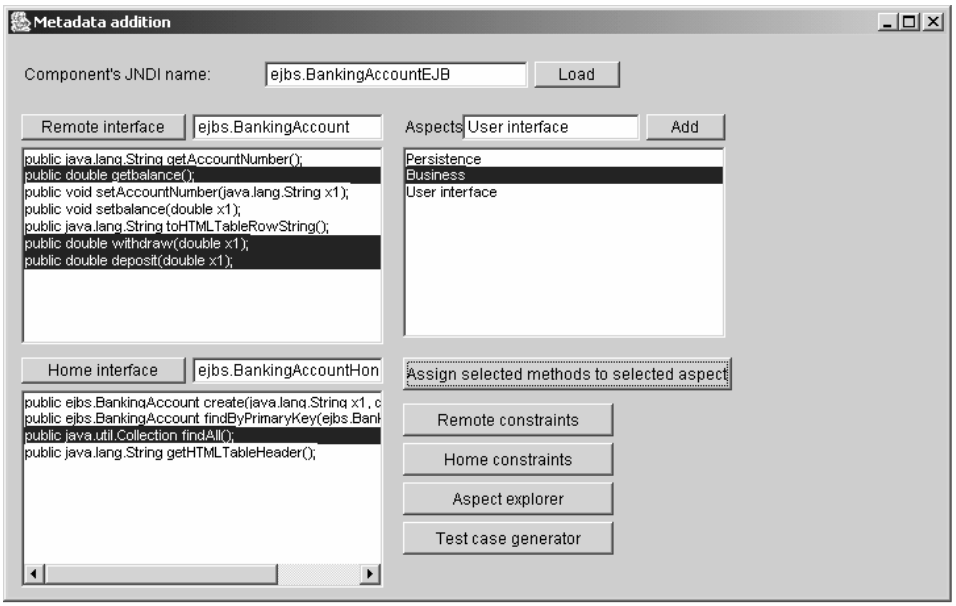

Figure 3. Main screen of the tool

The addition of pre and postconditions to methods is made on screen shown in Figure 4. As it is seen, these constraints are directly written in Java, since they will be later included in the file containing the testing program. In the example, the user has written two preconditions for the *withdraw* operation (actually, two pieces of code to be computed before the call to the method): the first one is a typical precondition to check the validity of an argument; the second one saves in a variable the value of the account balance before executing the method. This variable (*balancePre*) is used in the postcondition code to test whether the account balance has been adequately updated. Pre and postconditions can be saved into a file to be reused in later test cases generations.

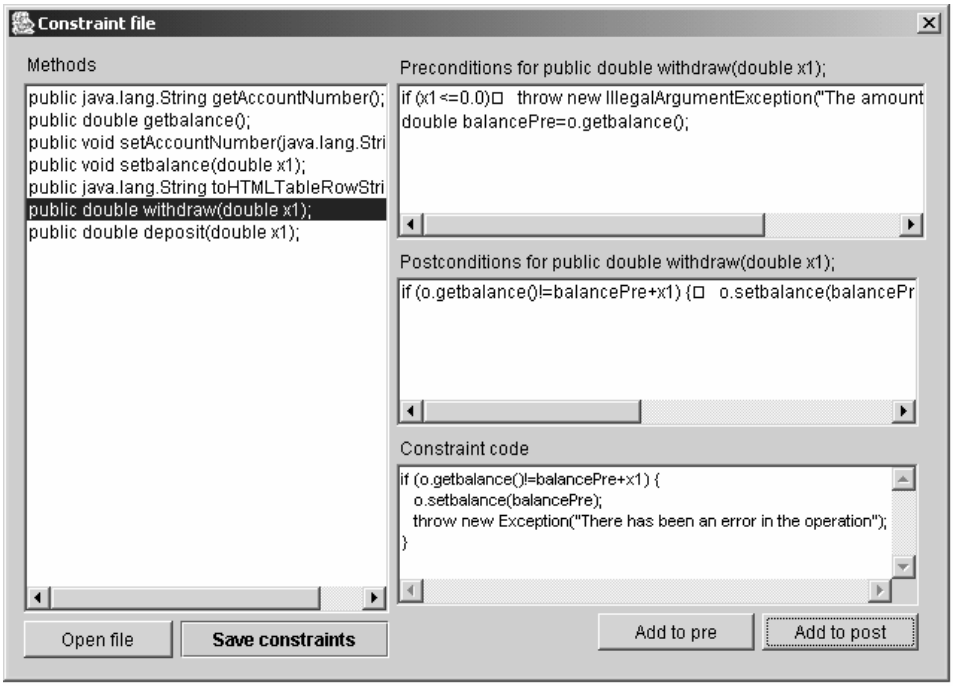

Figure 4. Screen to add pre and postconditions to methods

When pre and postconditions have been added to the operations, the user can go to the test case generation stage. Figure 5 shows the sreen in charge of this task: it shows the signatures of the set of operations in the Remote and Home interfaces of the component under test. The user can fix the maximum number of methods in each test script and build a regular expression to limit the set of test scripts to be generated using the operators | (alternative), * (zero or more elements), + (one or more elements), . (concatenation) and parenthesis. In the figure, the user desires to generate test cases with a maximum length of five methods starting by calls to *create* and *deposit,* followed by any combination of calls to *withdraw* and *deposit.* Pressing the "Get sequences" button, the tool obtains all the test scripts fulfilling these conditions. Each test script receives a unique name. The methods composing each test script can be seen clicking on the corresponding node in the test script tree.

Test cases are generated either for the all the test scripts or only for the selected test script by pressing the "Generate test cases" button. As we said, a test case consists of calls to the methods in its corresponding test script with actual values passed as parameters. Before the generation, the tool prompts the user to introduce test values by hand to each parameter, or to use those saved in a file.

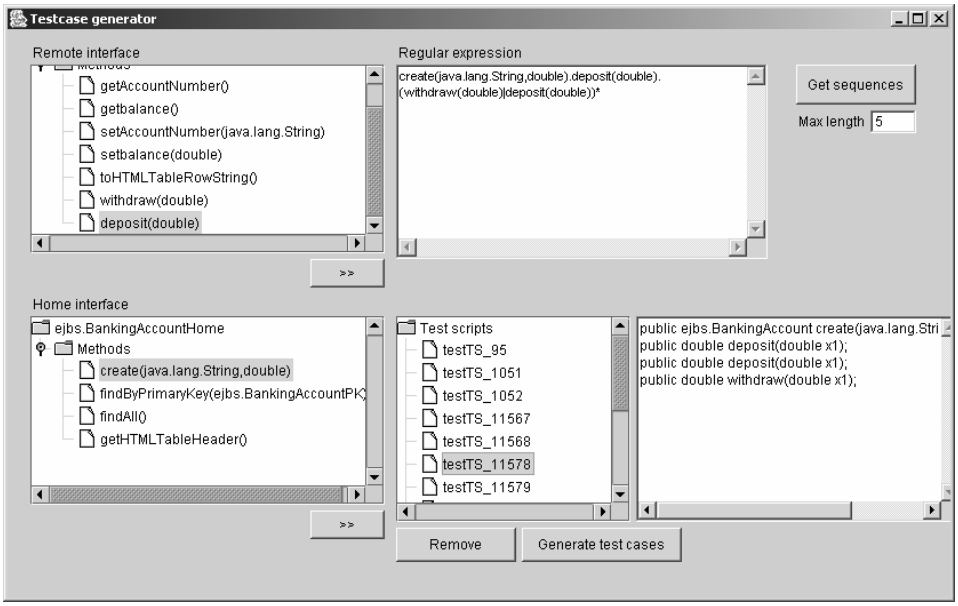

Figure 5. Screen for test case generation

In our example, test cases are written into a *BankingAccountTestCases* Java class. Each test case is written into a method whose name contains the number of test script it proceeds from, and a test case number. Figure 6 shows the test case number 1 for the test script 11568. It contains a call to the *create* method with "1234567890" and 50.0 as parameter values; a call to *deposit* passing 10.0, a call to *withdraw* and again a call to *deposit.* As it is seen, the call to *withdraw* is placed between the precondition and the postcondition described in Figure 4. It is also possible to group all the test cases corresponding to the same test script into just a method. In this case, the Java file produced is much smaller, but also more difficult to be read by a person.

The tool generates also a web page to execute the methods included in the *BankingAccountTestCases* class (in our case). Figure 7 shows this web page after having executed the test case shown in Figure 6. We suppose here there is an error in the implementation of the *withdraw* operation that puts the component in a state that does not fulfil the postcondition attached to this method, so the corresponding exception is thrown and shown in the page. Test cases can also be executed all together, showing on the web page the execution results for all of them. The web page includes also information about the aspects tested by the test cases contained in the test script.

```
public void testTS_11568_1() throws Exception {
        InitialContext ic = new InitialContext();
        Object objRef = ic.lookup("ejbs.BankingAccountEJB");
        ejbs.BankingAccountHome home=(ejbs.BankingAccountHome)
                PortableRemoteObject.narrow(objRef, ejbs.BankingAccountHome.class);

        java.lang.String x1=(java.lang.String) "1234567890"; double x2=(double) 50.0;
        ejbs.BankingAccount o=home.create(x1, x2);

        double x3=(double) 10.0;
        o.deposit(x3);

        double x4=(double) 100.0;
        if (x4<=0.0)
                throw new IllegalArgumentException("The amount must be greater than 0");
        double balancePre=o.getbalance();
        o.withdraw(x4);
        if (o.getbalance()!=balancePre+x4) {
                o.setbalance(balancePre);
                throw new Exception("There has been an error in the operation");
        }

        double x5=(double) 100.0;
        o.deposit(x5);
}
}
```

Figure 6. One of the test cases generated

Methods exposed by the interfaces can take non-primitive parameters. In our example (see Figure 5), the *findByPrimaryKey* method requires an object of *BankingAccountPK* class representing the primary key value of the tuple associated with the instance to be created.

In order to pass values of complex data types to test methods, the tool includes a repository of serialized objects. When the tool needs this type of values to generate test cases, it looks for files whose name is the same than the data type required. In the example, the tool could keep instances of *BankingAccountPK* in several files called *BankingAccountPK_1.ser*, *BankingAccountPK_2.ser,* etc. Obviously, it is indispensable that classes corresponding to these objects implement the *java.io.Serializable* interface.

4 Conclusions and future work

Splitting metadata according to aspects brings the possibility of applying separation of concerns to improve test selection and generation. Particularly, complexity of analysis might be reduced through the use of categorised metadata. In this paper, we have presented an environment to automate the generation of aspect-based test cases from Java code.

Figure 7. Web page for executing test cases

The tool reuses several data structures of a previous tool for testing standard Java classes. This one had a wider set of functionalities that we are now adding to the current tool, such as:

- Generation of mutants for the test cases, such as [GM01] claim. This includes the generation of text files that could be used as reports, to compare the results of the original and the mutated test cases.

- Implementation of a new user interface for the test case executor. The current one (web page) requires a web server and does not provides any specific contribution.

Other considered functionalities include:

- Addition to the tool of the *EntityTesting* class, an abstract class we defined in a previous work [CP04] and that can be used to create components specially designed and ready for testing .

- Addition of a specific module for managing the test cases according to the aspects they are related to.

Acknowledgments

This work is partially supported by the CyTED project VII-JRITOS2; by the UNComa project 04/E048; and by the MÁS project (TIC 2003-02737-C02-02).

Bibliography

[BOP00] Buy, U.; Orzo, A., and Pezzè M.: Automated Testing of Classes. In Proc. International Symposium on Software Testing and Analysis, Portland, Oregon, 2000.

[CP02] Cechich, A. and Polo M.: Black-box Evaluation of COTS Components using Aspects and Metadata. In Proc. 4th Int. Conf. on Product Focused Software Process Improvement, Springer-Verlag LNCS 2559, pages 494-508, 2002.

[CP04] Cechich A. and Polo M.: COTS Component Testing through Aspect-based Metadata. In Building Quality into COTS Components – Testing and Debugging. Beydeda S. and Gruhn V. (Eds), Springer-Verlag, 2004 (to appear).

[CTC01] Chen H.Y.; Tse T.H.; and Chen T.Y.: TACCLE: A methodology for object-oriented software testing at the class and cluster levels. ACM Transactions on Software Engineering and Methodology, 10(4):56-109, 2001.

[DF94] Doong, R.K. and Frankl P.G.: The ASTOOT approach to testing object-oriented programms. ACM Transactions on Software Engineering and Methodology, 3(2): 101-130, 1994.

[Ed01] Edwards S.H.: A framework for practical automated black-box testing of component-based software. Software Testing, Verification and Reliability, (11):97-111, 2001.

[FG99] Fewster M. and Graham D.: Software Test Automation, Addison-Wesley, 1999.

[GM01] Ghosh S. and Mathur A.: Interface mutation. Software Testing, Verification and Reliability (11): 227-247, 2001.

[Ha01] Harrold, M. et.al.: Using Component Metadata to Support the Regression Testing of Component-Based Software. Technical Report GIT-CC-01-38, College of Computing, Georgia Institute of Technology, 2001.

[JPP03] Jiménez M., Polo M., and Piattini M.: Una técnica de descripción formal para la generación y ejecución automática de casos de prueba. Proceedings of the 6th Workshop Iberoamericano de ingeniería de Requisitos y Ambientes de Software, pages 97-108, 2003.

[La98] Lapierre, S. et.al.: Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees. In Proc. Int. Conf. on Software Maintenance, 1998.

[Me01] Meudec C.: ATGen: automatic test data generation using constraint logic programming and symbolic execution. Software Testing Verification and Reliability, 11(2): 81-96, 2001.

[Of03] Offut J. et.al.: Generating test data from state-based specifications. Software Testing, Verification and Reliability, (13):25-53, 2003.

[Or00] Orso A.: Component Metadata for Software Engineering Tasks. In Proc. 2nd Int. Workshop on Engineering Distributed Objects, Springer-Verlag LNCS 1999, pages 126-140, 2000.

[PPR01] Polo M., Piattini M., and Ruiz F. : Automating Testing of Java Programs using Reflection. In Proc. of ICSE 2001 Workshop WAPATV, IEEE Press, 2001.

[SW99] Stafford J. and Wolf L.: Annotating Components to Support Component-Based Static Analyses of Software Systems. Technical Report CU-CS-896-99, University of Colorado at Boulder, 1999.