Structural Test Case Generation Based on System Models

Leandro T. Costa, Avelino F. Zorzo, Elder M. Rodrigues, Maicon Bernardino, Flávio M. Oliveira
School of Computer Science - Pontifical Catholic University of Rio Grande do Sul - PUCRS
Porto Alegre, RS, Brazil
Email: leandro.teodoro@acad.pucrs.br, bernardino@acm.org, {elder.rodrigues, flavio.oliveira, avelino.zorzo}@pucrs.br

***Abstract*—Structural testing, or white-box testing, is a technique for generating test cases based on analysis of an application source code. Currently, there are different tools supporting this type of test. However, despite the benefits of these tools, some tasks still have to be performed manually. This makes the test process time consuming and prone to injection of faults. In order to mitigate these problems, this paper presents a Model-based Testing (MBT) approach for deriving structural test cases for different code coverage tools using UML sequence diagrams. Our approach consists of four steps: *Parser*, *Test Case Generator*, *Script Generator* and *Executor*. These steps are based on the four main features of a Software Product Line for MBT tools, from which we derived two automation tools (PletsCoverageJabuti and PletsCoverageEmma) that generate and execute structural test cases, respectively. We also describe a case study, which defines test cases for an application that manages skills of employees.**

***Keywords*—model-based testing; structural testing.**

I. INTRODUCTION

The evolution and increased complexity of computer systems have made the testing process an activity as complex as the development process itself. In order to overcome this problem, and to increase the effectiveness in the test case generation process, several tools have been developed to automate software testing. Currently, there are several tools supporting different types of testing, for example, structural testing (or white-box testing), in which the source code of the system is inspected; or, functional testing (black-box testing), in which the functionality of the system is verified. In the last decade, many commercial and academic tools have been developed and used to support testing activities, such as, Java Bytecode Understanding and Testing (JaBUTi) [1], Semantic Designs Test Coverage [2], IBM Rational PurifyPlus [3], EMMA [4], Quick Test Professional [5], EvoSuite [17] or Randoop [15].

However, despite the benefits brought about by these testing tools, it is still necessary to perform several manual or semi-automated activities, for example, to provide test cases or to analyze the test results from running test cases. Furthermore, manual or semi-automated test case generation makes the testing process time consuming and prone to introduction of faults, even by experienced professionals. A solution proposal for this issue is to automate the test case generation process through software testing techniques, such as Model-based Testing (MBT) [6]. This technique consists in the generation of test cases and/or test scripts based on system models, which can include the specification of the characteristics that will be tested. MBT adoption presents several advantages, such as reducing the likelihood of misinterpretation of the system requirements by a test engineer or decreasing of testing time.

Currently, MBT can be used to generate test cases through the use of a wide range of modeling notations, such as Specification and Description Language (SDL) [7] or Unified Modeling Language (UML) [8]. UML provides a notation for modeling some important characteristics of applications, allowing the development of automatic tools for model verification, analysis and code generation.

In this context, this paper presents an MBT approach to drive the automatic generation of test cases and test drivers for measuring test coverage. Our approach uses sequence diagrams to identify the classes/methods under test and to generate test sequences based on the order of execution between the classes and methods described in the sequence diagram. Then, generates strucural test cases with a random test case generation tool, and finally generates test drivers to run the test cases and measure their coverage with the code coverage tools EMMA and JaBUTi. Furthermore, our approach is embedded in a Software Product Line (SPL) and new testing products are generated automatically. Our approach consists of four steps: (a) Parser: extracts test information about the classes and methods to be tested from UML sequence diagrams; (b) Test Case Generator: applies a random test data generation technique to generate an abstract structure, *i.e.*, a text file that describes the test case information in a tool-independent format; (c) Script Generator: generates test scripts/test driver for a specific code coverage tool from the information present in the abstract structure; (d) Executor: represents the test execution for a specific code coverage tool using the test driver generated in the previous step. Although we have applied our approach to object-oriented languages, it is straightforward to apply it to other programming paradigms.

One of the advantages of our approach is related to the reuse of test information, *i.e.*, information described in the abstract structure can be reused to generate test scripts for several code coverage tools, *e.g.*, academic: JaBUTi [1] or EMMA [4]; commercial: Semantic Designs Test Coverage [2] or IBM Rational PurifyPlus [3]. Therefore, a company that is using tool A can, motivated by a technical or managerial decision, easily change to a testing tool B without having to create new test cases. Another advantage is related to the use of UML models to generate test cases. Models provide a representation of the test information at a high level, facilitating the understanding by the test expert responsible for implementing and executing test cases. Moreover, differently from others studies that only describe the process to generate test cases through MBT, our approach is able to instantiate them to generate test drivers that could be executed by different code coverage tools.

Based on our approach, we developed two tools: PletsCoverageJabuti and PletsCoverageEmma. Both tools automatically extract test information from sequence diagrams, generate an

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

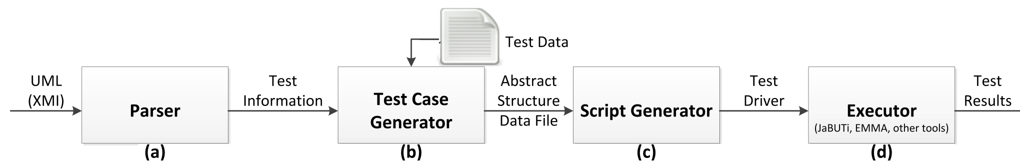

Fig. 1. Approach for generating structural test cases

abstract structure, instantiate the information present in this structure to generate and execute concrete test cases, respectively, for the target tools JaBUTi [1] and EMMA [4]. The tools presented in this paper were derived from a **P**roduct **L**ine for Mod**e**l-based **T**esting tool**s** (PLeTs) [9]. PLeTs supports the generation of products (MBT tools) that automate the generation and execution of test cases. We also applied our approach to a case study, in which we have used two generated tools to test classes and methods of an actual application.

This paper is organized as follows. Section II discusses related background. Section III presents the details of our approach. Section IV describes a case study. Section V discusses related work in structural test case generation using UML. Section VI presents some conclusions and lessons learned.

II. Background

MBT is a technique for automating the generation of test artifacts based on system models [6]. Using MBT it is possible to represent the structure and the system behavior, in order to be shared and reused by the test team members. Therefore, it is possible to extract the test information from models to generate new test artifacts, such as test cases, scripts and scenarios. The MBT adoption requires the creation of models based on system requirements specified by software engineers and test analysts. The purpose is that these models include information that frequently is implicit in traditional specification documents, for example, through comments and/or annotations.

One approach to improve the system specification is the use of UML models [8]. UML models can improve the system specification through stereotypes and tag definitions. Stereotypes is one of the UML extensibility mechanism that may have properties referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties are referred to as tagged values. Hence, all information added to the model, through stereotypes and tagged values, can be used to derive new artifacts, such as test cases.

To the best of our knowledge, early studies focused on MBT were limited to functional testing. Nowadays, models are able to abstract other information, *e.g.*, parameters and input data, thus allowing MBT to be applied to perform other testing techniques, *e.g.*, the structural testing [1].

Structural testing is a technique for generating test cases from the source code analysis. It seeks to evaluate the internal details of implementation, such as test conditions and logical paths. In general, most criteria based on structural analysis use a graph notation named Control Flow Graph (CFG) [1], which represents all the paths that might be traversed during the program execution. These criteria are based on different program elements that can be connected to the control flow and data flow in the program. Control-flow uses the control features of a program to generate test cases, *i.e.*, loops, deviations or conditions, while criteria based on data flow use data flow analysis of the program to generate test cases.

Structural test case generation consists of selecting values from an input domain of a program that satisfies specific criteria. For instance, the All-nodes criterion groups in a domain all the input values that execute a specific node. The selecting input values task could be made using data generation techniques, *e.g.*, random [10], based on symbolic execution [11] or dynamic execution [12]. In this paper, we apply a random technique due to be practical and easier to automate, which provided a useful test case generation for specific code coverage tools.

Currently, there is a diversity of commercial, academic, and open source code coverage tools that assist the testing process. However, most of these tools were individually and independently implemented from scratch based on a single architecture. Thus, they face difficulties of integration, evolution, maintenance and reuse. In order to reduce these difficulties, it would be interesting to have a strategy for automatically generating specific products, *i.e.*, tools that perform tests based on the reuse of assets and a core architecture. This is one of the main ideas behind SPLs [13].

An SPL can be defined as a set of systems that share common and manageable features in order to meet the needs of a specific domain, which may be a market segment or mission [13]. The aim is to explore the similarities among systems in order to manage variability aspects and thus determine a higher reusability level of software artifacts. Through the reuse of artifacts, an SPL allows to create a set of similar systems, thus reducing time to market, cost and, hence, to achieve a higher productivity and quality improvement.

In the testing context, we developed an SPL of MBT tools called PLeTs [9]. This SPL supports the derivation of MBT tools that allow automatic generation and execution of test cases. The purpose of PLeTs is not only to manage the reuse of artifacts and software components, but also to make the development of a new tool easier and faster. Until now, PLeTs was able to generate performance testing products. In this paper, we extend PLeTs to develop structural testing products.

III. Approach to Structural Test Case Generation

As mentioned in the previous sections, MBT techniques have been used to improve software testing through automation of test case generation. Furthermore, using UML models it is possible to automate the test case generation through annotation of test information using stereotypes and tags. Stereotypes and tags can be included in different parts of an UML model to represent test case information [8]. In our previous work [14], we have used UML use cases and activity diagrams as SUT models to automatically generate performance test cases from the information annotated on these diagrams. When conducting performance or even functional testing, UML use cases and activity diagrams were sufficient. However, an understanding about the ordering of execution between program units (*e.g.* methods/functions) is

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

needed to execute structural testing. In this context, we propose an approach to automate the generation and execution of structural test cases based on UML sequence diagrams. Thus, test sequences are generated according to the order of the methods described in sequence diagrams. As mentioned in Section I, we divided our approach in four steps (see Figure 1): `Parser`, `Test Case Generator`, `Script Generator (`*`Test Driver`*`)`, and `Executor`. These steps are based on the four main features of PLeTs.

In order to generate and execute the *Test Driver*, our approach must retrieve information, about classes and methods, annotated in an UML sequence diagram. It is important to highlight that the diagrams must be well-defined, *i.e.*, they have to contain information about classes and methods parameters (name, type), as well as, each method return type. Besides, it is also necessary to annotate the diagrams with additional information, *e.g.*, a variable that will be used to specify the path of the classes that will be tested. This information will be used to generate the *Abstract Structure* (more details about how the diagram is annotated will be presented in Section IV). The UML sequence diagram is annotated with the following tags: ≪**`TDexternalLibrary`**≫: specifies the libraries path of the SUT; ≪**`TDclassPath`**≫: specifies the path of the classes to be tested; ≪**`TDtoolPath`**≫: specifies information about the chosen code coverage tool, *e.g.*, the installation directory and the path of its launcher; ≪**`TDimportList`**≫: specifies a list of imported classes.

The advantage of annotating the sequence diagram with these tags is that they are used to provide information used to automatically generate Test Drivers, such as libraries, dependencies among classes and import list. Each tag can define a fixed value or a variable that can be replaced when generating the actual test case or driver for a specific tool. For example, the previously mentioned four tags must be annotated in the sequence diagram with the following parameters: `@externalLibrary`, `@classPath`, `@toolPath` and `@importList`. However, these parameters are just a reference and have no actual information about the code coverage tool, class path, external library or import list. After this annotation process, all information described in the UML sequence diagram is exported to a *XMI* file, which is the input of the first step in our approach.

The first step (`Parser`) consists of parsing the *XMI* file in order to extract the information necessary to generate a data structure in memory, which we call *Test Information* (see Figure 1a). The *Test Information* describes the test sequences generated from the sequence diagram and it has information about the methods and classes to be tested. The second step (`Test Case Generator`) receives as input the *Test Information* and a XML file called *Test Data* (Figure 1b).

The *Test Data* file has the actual values about libraries used to the application execution, the path of classes to be tested and the package list to be imported. However, the *Test Data* file has no tool information, since the first two steps of our approach are tool-independent. Moreover, the *Test Data* also describes a set of different parameter values for all classes and methods of the application to be tested. Based on that, the `Test Case Generator` applies a random test data generation technique [10] under the parameter values presented on *Test Data* and only for the classes and methods described on *Test Information*. The random technique generates input values for each method described in a test sequence. The reason for choosing this technique consists of selecting specific parameters for each one of these classes and methods. It was used due to its practicality and to be easier to automate. However, other techniques are presented in the literature, *e.g.*, symbolic execution [11], dynamic execution [12] and feedback-directed random testing [15]. After applying the random test data generation technique, the `Test Case Generator` also produces the *Abstract Structure* and the *Data File*, which are the input of the third step. The *Abstract Structure* is a text file that describes, in a sequential and tool-independent format, the entire data flow of the classes and methods to be tested (see Figure 3 for an example of file that contains the *Abstract Structure*). The *Abstract Structure* is divided in three groups: **1) Tool Configuration:** defines the `@toolPath` parameter, which specifies the information about the code coverage tool that will be used for the test; **2) Test Configuration:** defines the `@classPath`, `@externalLibrary` and `@importList` parameters, which define the information used for a specific test case; **3) Sequential Flow Configuration:** defines the sequential flow of the methods that will be tested.

Each one of these parameters is a reference to the actual data that is stored in the *Data File*, which is a text file that contains the information (values) used to instantiate test cases for a given code coverage tool (see Figure 4 for an example of a file that contains actual values for a specific tool). The test case instantiation is performed by the step `Script Generator` (see Figure 1c), which consists of automatically generating the *Test Driver* for a specific code coverage tool. Therefore, when the *Abstract Structure* and *Data File* are instantiated to generate *Test Driver*, a class file named *TestDriver.java* is generated. This file contains a class that makes calls to the methods that will be tested and also includes a set of information to be used as input of these methods. In our approach, the input information is generated automatically using the random test data generation technique previously mentioned. Furthermore, in the step `Script Generator` the user must provide all information about a specific code coverage tool, *e.g.*, the path of its launcher.

One of the advantages of using a file to store the actual values, which are used in the instantiation of the class file, is that it is not necessary to include, in the UML sequence diagram, the parameter values of the methods that will be tested. Thus, to generate new test cases with different input values, it is only necessary to generate new test data using any kind of data generation technique. Moreover, the advantage of using the *Abstract Structure* is related to the ability to reuse information for different code coverage tools. In this sense, if a company decides to migrate to a different code coverage tool, due to a management strategy, it will be able to use the test cases previously generated. Besides that, the *Abstract Structure* presents the test information in a clear format, making it simple and easy to understand. Therefore, it is easier to automate the *Test Driver* generation for several tools. The last step (`Executor` - see Figure 1d) consists of performing the test with a specific code coverage tool. Therefore, all the class files generated on step three are used for the test execution. The generation of the class files will be further described in Section IV.

IV. CASE STUDY: SKILLS - WORKFORCE PLANNING

This section describes how we have applied our approach to test an application to manage profiles of employees from

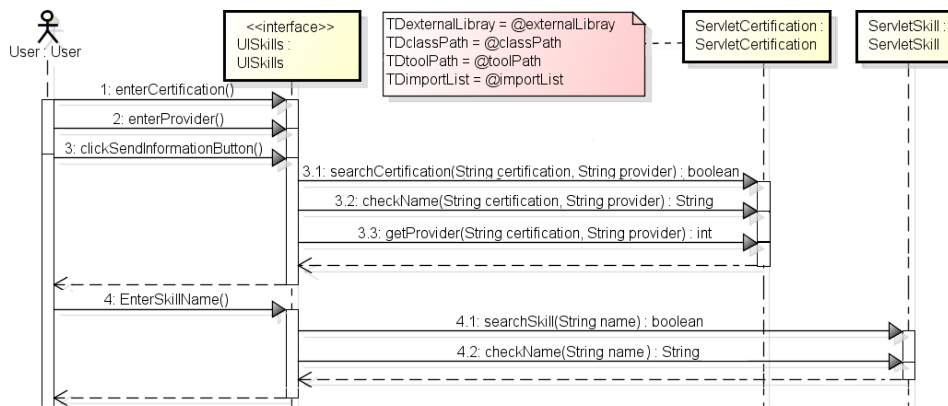

Fig. 2. Sequence diagram

any company. The main goal is to assess the efficacy and the functionality of our approach through presenting how we derived two tools (PletsCoverageJabuti and PletsCoverageEmma) that generate test cases from UML sequence diagrams and execute *TestDrivers* using two coverage tools, *e.g.*, JaBUTi and EMMA. Through the use of our approach we were able to reuse components from steps 1 and 2 (Section III) for both testing tools.

The application used as subject under test is called Skills (Workforce Planning: Skill Management Tool) [14]. This application was developed in a collaboration project between a TDL of a global IT company and our university. The main objective of Skills is to manage and to register skills, certifications and experiences of employees for a given company.

With the purpose of verifying the functional aspects of our approach, we have tested a set of classes and methods of Skills. These classes and methods are represented by four sequence diagrams that describe processes, in which an user performs several operations, *e.g.*: (a) search for a particular certification information; (b) search for a particular skill information; (c) display a list of registered experiences; (d) display information about the user profile; and (e) change the login password. Figure 2 shows part of one of the four sequence diagrams (all sequence diagrams can be found in [16]), in which it is possible to see how tags described in Section III are annotated in the sequence diagram. As can be seen in Table I, these operations are performed through calls of 22 methods of 9 classes (2,561 lines of code). Note that our approach consists of automating the test case generation, in which only the system internal methods are analyzed. Therefore, no method called from the user interaction will be analyzed, since our approach does not implement this feature. In this context, only the information about the methods described in Table I will be used to automatically generate and executing the *Test Driver*.

In order to generate and execute the *Test Driver*, initially, we had to annotate the four sequence diagrams with the tags `TDexternalLibrary`, `TDclassPath`, `TDtoolPath`, `TDimportList` and their respective parameter values: `@externalLibrary`, `@classPath`, `@toolPath` and `@importList`. These tags and values were annotated in the classifier role elements, which represent the nine classes used for this case study. After annotating the sequence diagrams with test information, we exported these test models to a *XMI* file, which is input for PletsCoverageJabuti and PletsCoverageEmma. During their execution, the tools parse the *XMI* file,

```
Abstract Structure: Search for Certification
## Tool Configuration ##
Tool Information : <<TDtoolPath: @toolPath>>
## Test Configuration ##
External Libraries : <<TDexternalLibrary: @externalLibrary>>
Path Classes : <<TDclassPath: @classPath>>
Imported Classes : <<TDimportList: @importList>>
## Sequential Flow Configuration ##
1. ServletCertification
1.1. searchCertification(String certification, String provider):
boolean
1.2. checkName(String certification, String provider): String
1.3. getProvider(String certification, String provider): int ...
```

Fig. 3. Code snippet of the *Abstract Structure*

```
@toolPath = C:\Jabuti\bin; C:\Jabuti\lib\bcel-5.2.jar;
C:\Jabuti\lib\capi.jar; ...
@externalLibrary = C:\Tomcat 6.0\lib\jsp-api.jar; ...
@classPath = C:\CmTool_SkillsTest\web\WEB-INF\classes; ...
@importList = servlets.*; java.io.*; java.util.StringTokenizer
1. ServletCertification
1.1. searchCertification("ActiveX", "BrainBench")
1.2. checkName("ActiveX", "BrainBench")
1.3. getProvider("ActiveX", "BrainBench") ...
```

Fig. 4. Code snippet of the *Data File* for JaBUTi

extracting information from the methods and classes that will be tested in order to generate a data structure in memory (*Test Information*). Based on the *Test Information* and the *Test Data* (a XML file with different parameter values for all classes and methods of the SUT), the tools apply a random test data generation in order to generate the *Abstract Structure* (Figure 3) and *Data File* (Figure 4).

Figure 3 shows a code snippet of the *Abstract Structure* that is divided into three information groups: `Tool Configuration`, `Test Configuration` and `Sequential Flow Configuration`. As mentioned in Section III, all parameters present in each information group are a reference to the actual data that is stored in the *Data File* that contains all values that will be used to instantiate test cases for a given code coverage tool (JaBUTi or EMMA). Figure 4 presents a code snippet with information regarding the parameters values of this file. In this example we defined information on the JaBUTi launcher path (`@toolPath`); for EMMA, we just need to change this value in the *Data File*.

Based on the information described in the *Abstract Structure* and *Data File*, the *TestDriver.java* class is generated. This class is the same for both JaBUTi and EMMA. Since JaBUTi and EMMA perform structural analysis on the bytecode, PletsCoverageJabuti and PletsCoverageEmma create a Java

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

TABLE I. Coverage Information for JaBUTi

Classes	Methods	Lines of Code	Coverage Percentage (%)			
			One run	*Two runs*	*Three runs*	*Four runs*
ServletCer tification	searchCertification	128	100	100	100	100
	checkName	96	100	100	100	100
	getProvider	134	69	75	89	100
ServletSkill	searchSkill	119	56	100	100	100
	checkName	90	100	100	100	100
ServletEx peri- ence	getUserExperiences	125	100	100	100	100
ServletProfile	getUsers	122	80	85	93	100
	printResult	95	100	100	100	100
Servlet Password	checkPassword	129	100	100	100	100
	changePassword	121	90	95	100	100
ServletTree	searchSkillNode	120	100	100	100	100
	searchCertificationNode	126	59	72	95	100
ServletIndus try-Domain	getRoleChildren	115	48	57	81	100
ServletForgot Password	sendEmail	137	100	100	100	100
	checkEmail	121	66	84	94	100
	checkUser	119	48	63	86	100
ServletGen eralSearch	getSelectedUsersCertifications	122	25	50	75	100
	getSelectedUsersExperiences	129	71	82	100	100
	getSelectedUsersSkills	113	74	89	100	100
	printCertifications	100	75	100	100	100
	printExperiences	102	62	100	100	100
	printSkills	98	90	100	100	100

process to compile the driver class. In order to perform test cases with EMMA, automating the generation of the *TestDriver.java* class is enough. However, in order to perform test cases with JaBUTi it is necessary to generate a project file. PletsCoverageJabuti generates this project file by creating a Java process. This process runs a JaBUTi's internal class called `br.jabuti.cmdtool.CreateProject`, in which some information such as paths of the JaBUTi's internal libraries is used as input parameter.

Once these two files are generated, the test execution consists in the internal call of the `probe.DefaultProber.probe` and `probe.DefaultProber.dump` methods for JaBUTi. At the end, the PletsCoverageJabuti creates a Java process for running JaBUTi, which is responsible to calculate and to show the updated coverage information for the defined test case. Based on that converage information, the tester could continue running the PletsCoverageJabuti in order to generate more test cases and increase code coverage. In this context, the tool executes several tests until the code coverage is reached. The tester has also the possibility of terminating the PletsCoverageJabuti execution in any moment and then, finalize the test. An advantage of using PletsCoverageJabuti is that it could generate several tests avoiding redundant test cases, since each test case generated by the random technique is saved by the tool. This ensures that a test case will not be repeated. Table I shows the coverage results after four test runs. It is important to mention that all classes and methods were analyzed based on All-nodes criterion. As can be seen in the table, some methods were covered after one run, while others needed for runs to be covered.

In order to generate and execute *Test Drivers* using the PletsCoverageEmma, we have used the same sequence diagram. However, we have not annotated it with test information, because this task had been done previously for PletsCoverageJabuti. Furthermore, all test cases generated for PletsCoverageJabuti were also used for our second tool. In the same way as PletsCoverageJabuti, the user/tester has the possibility of continuing to run the tool in order to generate and execute more *Test Drivers*. The results for EMMA are similar to the ones for JaBUTi presented in Table I.

These results show that our approach allowed the same diagrams, and test cases to be used in different tools producing similar results. Furthermore, our approach was able to generate a second tool (PletsCoverageEmma) with less effort. The reason is that our approach is based on an SPL, which allowed the reuse of components (*Parser* and *Test Case Generator*) already developed. Although we have developed different components (*Script Generator* and *Executor*) for our both tools, this task required less effort compared to development of the two first components. In this case, we had to automate the calls of internal routines and subcommands of JaBUTi and EMMA. Furthermore, once familiar with the functional features of the PletsCoverageJabuti tool, it was possible to perform tests with little learning effort using PletsCoverageEmma, since both tools share several features, *e.g.*, GUI, test data generation technique and the *Abstract Structure* format.

The results also show the importance of performing structural testing, since it covers faults that are difficult to meet with other testing techniques, *e.g.*, the functional testing. For instance, if a test team does not ensure that all methods were fully covered during the structural testing activity, it is possible that when applying the functional testing, a specific functionality cannot be assessed (unreachable statement) due to a code inconsistency, *e.g.*, infinite loops or conditions that never occur. Therefore, structural testing is useful in combination with functional testing, since it helps to reveal faults that may not be evident with black-box testing alone.

V. RELATED WORK

There has been some work in the past years related to MBT, UML and structural testing, but to the best of our knowledge none of them has integrated all of them. Furthermore, our work also uses code coverage tools and it is integrated into an SPL.

Regarding test case generation using UML sequence diagrams, Khandai *et al.* [18] propose an approach for generating test cases for concurrent systems using sequence diagrams. Our approach, on the other hand, aims to generate tools that automatically generate and execute test cases based on source code of applications. A strategy similar to Khandai *et al.* can be applied to extend our approach for concurrent systems.

Similarly, Sharma *et al.* [19] convert the UML sequence diagram into Sequence Diagram Graph (SDG), and then traverse the SDG to generate the test cases. Other UML diagrams are also used to collect information that is stored in the SDG. Their approach was extended to combine sequence and use case diagrams to generate system test cases. This extended approach consists of converting UML use cases into a Use Case Diagram Graph and UML sequence diagrams into SDG. Our work, on the other hand, focus on structural testing with coverage criteria based on commands, decisions, classes, and methods, which is not addressed by Sharma *et al.*. Thus, both approaches are complementary since our approach can be used to generate test cases to cover interactions and scenarios faults.

A work from Swain and Mohapatra [20] uses UML sequence and activity diagrams to generate test cases by converting the UML diagrams into an intermediate representation called Model Flow Graph (MFG). This MFG is traversed to generate test sequences that are instrumented in the test case to satisfy a message-activity path test adequacy criteria. Our approach differs from [20] since it is embedded in an SPL and go further than only generating the test cases, *i.e.*, our approach actually executes the test cases in two code coverage tools.

Different from the existing works in test case generation presented in this section, our approach consists not only in the test case generation through MBT, but also on the generation and execution of *Test Drivers* for several tools.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

Furthermore, we applied our approach to a detailed case study in an actual company environment. Moreover, our approach is based on an SPL, which make it easier to reuse code that was not developed for a specific tool. This has happened in the two tools we presented and also in previous tools for performance and functional testing [14]. Furthermore, our approach distinguishes from [14] in two aspects: first, our approach generates test cases from UML sequence diagrams, while their work uses UML use cases and activity diagrams; second, our approach uses a random test data generation technique to select input values from a specific domain, while their work uses sequence test case generation methods, *e.g.*, Harmonized State Identification (HSI) [21].

VI. Conclusion and Lessons Learned

This paper presented an approach for automating test case generation for several coverage tools from UML sequence diagrams. Based on this approach, we incremented an SPL called PLeTs. PLeTs is able to generate testing tools that use academic or commercial tools to execute performance, functional or structural test. One of the advantages of our approach is related to reuse of test information described in UML sequence diagrams. Hence, it is possible to easily migrate to a different testing tool and reuse the test cases previously defined. The tools used to exemplify our approach were: JaBUTi and EMMA. However, commercial tools such as Semantic Designs Test Coverage and Rational PurifyPlus or other academic tools such as Poke-Tool (Poke-Tool) could be used for this purpose. Furthermore, our approach is useful for industry when developers already have designed models. In this context, the models could be reused to automatically generate test cases. Basically, the main lessons we have learned were: **1. Coverage analysis based on bytecode and source code.** Although we have presented a case study, in which we used two code coverage tools (JaBUTi and EMMA) to perform coverage analysis based on bytecode, our approach is able to deal with tools performing tests based on the analysis of source code, *e.g.*, Semantic Designs Test Coverage [2]. Some minor tools related changes should be performed, however. For example, the `@classPath` parameter must indicate the path of source code files instead of the path of class files (bytecode). We decided to use bytecodes, since in some situations the source code could not be available to test an application. **2. The choice of test data generation technique.** When performing structural testing, it is very important to choose an efficient technique for generating testing data. An efficient technique increases the likelihood of meeting the requirements of structural testing. In our approach we have applied a random testing data generation technique to select the parameter values used to instantiate the *TestDriver.java* file. However, our approach could implement other data generation techniques, *e.g.* symbolic execution [11] and dynamic execution [12]. These two techniques are more effective than random generation and guarantee data selection with a higher probability to reveal faults. Nevertheless, a random technique is practical and easier to automate. **3. The needed knowledge on the code coverage tools.** Although there are several ways to automate the generation and execution of tests for different tools, a detailed study of used code coverage tools is still necessary. Sometimes this study may reveal that is not possible to automate the generation and execution of test cases for a particular code coverage tool. For example, open source tools such as JaBUTi and EMMA are easier to automate than commercial ones, because it is possible to get access to their internal functioning. Another point is related to the way tools are executed, *i.e.*, throughout command line or GUI. Command line tools are easier to automate because they, usually, provide a set of subroutines/programs that can be easily parameterized. **4. The advantage to generate testing tools from an SPL.** The SPL concepts were useful to develop testing tools with less effort. A reason for that is related to the possibility to reuse components already developed to generate other testing tools. Furthermore, an SPL can provide other advantages, such as: quality improvement, since it is possible to reuse components already developed and tested; higher productivity, since it is not necessary to develop tools from scratch; and cost reduction, since it is possible to develop tools in large scale.

Acknowledgment

Research projects: PDTI 001/2014 financed by Dell Computers with resources of Law 8.248/91, and AutoScene supported by Facin/PUCRS. Thanks also to Soraia R. Musse.

References

[1] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong, "Establishing structural testing criteria for Java bytecode," Software: Practice and Experience, vol. 36, no. 14, pp. 1513–1541, 2006.
[2] Semantic Designs, "Semantic Designs Test Coverage," URL: http://www.semdesigns.com, [retrieved: July, 2014].
[3] IBM, "IBM Rational PurifyPlus," URL: http://www.ibm.com/software/awdtools/purifyplus/, [retrieved: July, 2014].
[4] V. Roubtsov, "EMMA: a Free Java Code Coverage Tool," URL: http://emma.sourceforge.net, [retrieved: July, 2014].
[5] S. R. Mallepally, QuickTest Professional (QTP) Interview Questions and Guidelines: A Quick Reference Guide to QuickTest Professional. Parishta, 2009.
[6] P. Krishnan, "Uniform Descriptions for Model Based Testing," in *Proc. ASWEC*, 2004, pp. 96–105.
[7] A. Kerbrat, T. Jéron, and R. Groz, "Automated Test Generation from SDL Specifications," in Proc. SDL Forum, 1999, pp. 135–152.
[8] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide. Addison-Wesley Professional, 2005.
[9] CePES/PUCRS, "PLeTs SPL," URL: http://www.cepes.pucrs.br/plets, [retrieved: July, 2014].
[10] D. Hamlet and R. Taylor, "Partition Testing does not Inspire Confidence," IEEE Transactions on Software Engineering, vol. 16, no. 12, pp. 1402–1411, 1990.
[11] M. Lin, Y. Chen, K. Yu, and G. Wu, "Lazy Symbolic Execution for Test Data Generation," IET Software, vol. 5, no. 2, pp. 132–141, 2011.
[12] R. Dara, *et al.*, "Using Dynamic Execution Data to Generate Test Cases," in Proc. ICSM, 2009, pp. 433–436.
[13] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing, 2001.
[14] M. B. Silveira, *et al.*, "Generation of Scripts for Performance Testing Based on UML Models," in Proc. SEKE, 2011, pp. 258–563.
[15] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in Proc. ICSE, 2007, pp. 75–84.
[16] CePES/PUCRS, "PLeTs Guide," URL: http://www.cepes.pucrs.br/plets/?a=guide, [retrieved: July, 2014].
[17] EvoSuite, "EvoSuite," URL: http://www.evosuite.org, [retrieved: July, 2014].
[18] M. Khandai, A. Acharya, and D. Mohapatra, "A Novel Approach of Test Case Generation for Concurrent Systems Using UML Sequence Diagram," in Proc. ICECT, 2011, pp. 157–161.
[19] M. Sarma, D. Kundu, and R. Mall, "Automatic Test Case Generation from UML Sequence Diagram," in Proc. ADCOM, 2007, pp. 60–67.
[20] S. K. Swain and D. P. Mohapatra, "Test Case Generation from Behavioral UML Models," International Journal of Computer Applications, vol. 6, no. 8, pp. 5–11, 2010.
[21] A. Petrenko, *et al.*, "Nondeterministic State Machines in Protocol Conformance Testing," in Proc. PTS, 1993, pp. 363–378.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4