2009 33rd Annual IEEE Software Engineering Workshop

JaBUTiService: A Web Service for Structural Testing of Java Programs

Marcelo Medeiros Eler, Andre Takeshi Endo, Paulo Cesar Masiero, Marcio Eduardo Delamaro, Jose Carlos Maldonado
Instituto de Ciencias Matematicas e de Computacao – Universidade de Sao Paulo
P.O. 668 – Sao Carlos – Brasil – 13560-970
Email:{mareler, aendo, masiero, delamaro,jcmaldon}@icmc.usp.br

Auri Marcelo Rizzo Vincenzi
Universidade Federal de Goias
Caixa-Postal: 131 - CEP: 74001-970 - Goiania, GO - Brasil
Email: auri@inf.ufg.br

Marcos Lordello Chaim, Delano Medeiros Beder
Escola de Artes, Ciencias e Humanidades
Universidade de Sao Paulo
Avenida Arlindo Bettio, 1000 Ermelino Matarazzo
03828-000 - Sao Paulo, SP - Brasil
Email: {chaim,dbeder}@usp.br

***Abstract*—Web services are an emerging Service-Oriented Architecture technology to integrate applications using open standards based on XML. Software Engineering tools integration is a promising area since companies adopt different software processes and need different tools on each activity. Software engineers could take advantage of software engineering tools available as web services and create their own workflow for integrating the required tools. In this paper, we propose the development of testing tools designed as web services and discuss the pros and cons of this idea. We developed a web service for structural testing of Java programs called JaBUTiService, which is based on the stand-alone tool JaBUTi. We also present an usage example of this service with the support of a desktop front-end and pre prepared scripts. A set of 62 classes of the library Apache-Commons-BeanUtils was used for this test and the results are discussed.**

I. Introduction

Software testing is an important activity in the process of software quality assurance and consists of running a program under test with the aim of revealing faults [1]. Programs are tested against test cases that are created according to specific testing techniques and criteria. Manual generation of test cases is expensive and error prone. Thus, various testing tools have been developed to support this activity. Generally, each testing tool implements mechanisms to test software according to a specific technique. The combination of techniques and criteria usually requires that the tester uses different tools, each one with its own local installation, platform and programming language, and also with specific input and output data definitions. These characteristics hamper integration of tools and the integrated and automatic use of the obtained results. This leads to the need of combining manually the results or using a tool developed specifically for this purpose. According to Wicks and Dewar [2], software engineering tool integration remains an open topic and more work is necessary to improve it.

Web services are an emerging Service Oriented Architecture (SOA) technology to integrate applications using open standards based on XML. SOA is an architectural style that uses services as the basic constructs to support the development of rapid, low-cost, loosely-coupled and easily integrated applications even in heterogeneous environments [3]. We claim that testing tools, as well as other software engineering tools, can be provided as web services suitable for integrating heterogeneous systems. In this context, Ghezzi and Gall [4] proposed a service platform, devising software analysis tools as web services. Following a similar approach, the QualiPSo (Quality Platform for Open Source Software) project has developed a new concept of forge based on SOA [5]. All functionalities and tools in the forge will be provided and integrated through web services.

Our research group has been working with software testing for many years and several tools for software testing have been developed in this period. One of these tools is JaBUTi (Java Bytecode Understanding and Testing) [6]. Initially it was developed as a stand-alone to support structural unit and integration testing of Java and AspectJ programs [7], [8].

In this paper, we propose the idea of testing tools designed as web services, discuss its pros and cons and present as a proof of concept a web service to support structural testing of Java programs called JaBUTiService, which has been developed on top of the stand-alone tool JaBUTi. Our claim is that using widespread availability of testing tools as services with standardized formats of inputs and outputs of their operations, the integration among different tools and the combination of their results would be facilitated.

This paper is organized as follows. Section II presents a discussion about sharing tools as web services. Section III shows a brief overview of software testing, describing the JaBUTi tool. Section IV presents our testing service called JaBUTiService. Section V presents a usage scenario of our service, including a desktop front-end application and pre prepared scripts. Section VI discusses related work and Section

VII presents the conclusion and future work.

II. Testing Tools as Web Services

Providing testing tools as web services could bring many advantages as the ones presented as follows. Possible disadvantages and difficulties are discussed in the conclusion.

Ease of use: testing tools available as services could be integrated in many development environments prepared to access other web services, thus being independent of platforms, languages, operating systems, etc. Services can be accessed through desktop applications, web applications, other services, service composition and orchestration, etc. Section V shows a possible way to use a testing service.

Availability: developers can use the testing web service without spending time with installation and deployment. Moreover, the results of testing and the reached coverage may be made available for analysis tools to keep history, tracking faults, extracting metrics, etc.

Version control: testing tools handle complex data structures and could be very dependent of technology. These issues could cause the release of new versions frequently. However, since the service interface does not change frequently, the cost of managing version control will be lower. That is, the use of traditional tools requires to reinstall the testing tool any time an update is released. This differs from web services because updates can be released only in the service server and it is likely that client tools will not perceive the change.

Integration: each company or project has its own development process and the software engineering tools are used during different stages. In this context, it is essential to adapt the tools for each scenario. If software engineering tools were available as web services, software engineers could design workflows to integrate different tools from different providers. This scenario could avoid dependency from a dominant general purpose tool and its vendor [16]. If we consider the testing activity, testers can build workflows and scripts to run a test set in a series of tools to obtain the test results and analyze the criteria coverage for different techniques. The combination of different techniques for testing software is highly recommended [1].

Orchestration and choreography of software engineering services: we envisage that in the future, there will be environments with many software engineering services registered at specific brokers, and software engineers will be able to select and compose services to assemble and customize their software development environment. For example, testing services would be integrated with other services that provide features such as extraction of metrics, logging and tracking of faults.

Comparison: conduction of experiments to compare two or more tools with the same functionality would be facilitated. Furthermore, software engineers could easily try several tools and select those that best fit each specific task.

The simple availability of testing tools as services will offer the same benefits brought by the SOA. The benefit of easy integration among services also requires that suppliers agree with the standardization of the service's operation and its parameters (input and output data). Some efforts have been carried out to use taxonomies and ontologies for the integration of tools within a certain field and with tools from other software engineering fields [4].

A platform for sharing testing services is being developed by our research group. To classify the different testing tools to be registered into the service broker, a testing ontology is being used, called OntoTest [10]. We are also studying the feasibility of extending the ontology to standardize the interfaces of the registered testing tools. The results of this extended and standardized ontology could be queried for generating reports using languages like SPARQL [11].

A web service is useful if it can be reused in different contexts. Figure 1 and Figure 2 show two possible ways of using it. In Figure 1, a tester can use JaBUTiService directly for testing a particular project. The tester can access the JaBUTiService using any device that is able to access a web service and create scripts in XML specific for the task at hand.

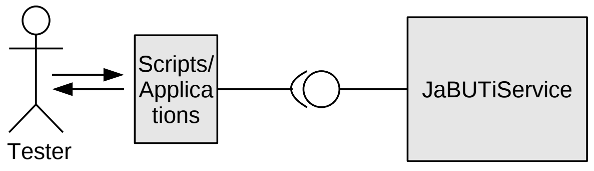


Fig. 1. The tester can use JaBUTiService for a particular project.

Figure 2 shows the usage of JaBUTiService in the context of a testing workflow. Suppose that a tester designed a test set and wants to evaluate it using different techniques. The tester sets up a testing workflow and use the JaBUTiService to perform structural testing. If other testing tools were available as web services, these services could be orchestrated or composed using for example a BPEL process.

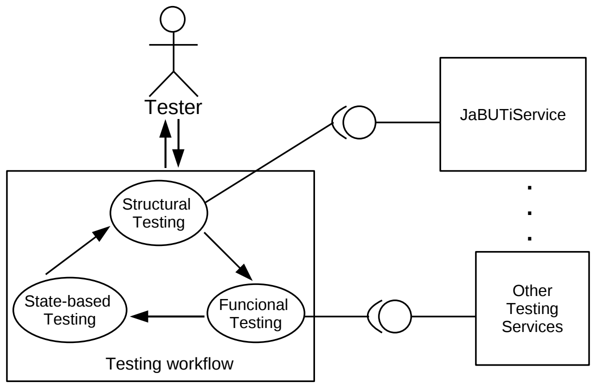


Fig. 2. JaBUTiService is used in a testing workflow.

III. Software Testing and JaBUTi

Software development processes involve many activities, techniques, methods, tools and people. Many types of faults can be introduced in the software by each activity. Quality assurance activities have been introduced along the development process to solve this problem. Among these activities, software testing is one of the most used, providing reliability evidences in complement to other activities like review, validation and verification techniques [12].

Testing is the process of executing a program with the intent of finding faults [1]. It is not possible to test software against all possible inputs because they generally tend to be infinite. Several techniques and criteria have been developed to better create meaningful test cases. A testing criterion defines which properties or requirements need to be tested for evaluating the quality of the tests [13]. Four techniques stand out on testing researches: functional (or black-box) testing, structural (or white-box) testing, state-based testing and fault-based testing.

Structural testing is a technique in which the tester build test cases by means of the program internal logic [1]. Usually, most of the criteria from this technique use a program representation called control flow graph or program graph. The criteria that only consider characteristics of the execution control are named control flow criteria. The most known criteria of this class are all-nodes, all-edges, and all-paths [13]. Moreover, there are data flow criteria that use information about the program data flow to derive test requirements. Rapps and Weyuker [14] proposed an extension of the control flow graph, called Def-Use Graph (DUG), to add information related to variable usage.

The activity of testing involves manipulation of a large amount of data. The generation and execution of test cases can be highly expensive and error prone if it will be done manually. It is also necessary to analyze the test results and assess whether the criteria for testing were properly covered. Testing tools are valuable to reduce the time and increase the quality of the testing activity in the design, execution and analysis of test cases. Performing structural testing manually can be infeasible depending on the program's size. Several tools for structural testing have been developed like ASSET [9], ATAC [15], and POKE-TOOL [17].

JaBUTi [6] is a structural testing tool that implements intra-method control-flow and data-flow testing criteria for Java programs. These criteria include all-nodes, all-edges and all-uses. It also considers these criteria with and without taking into account exceptional flows. JaBUTi implements some testing coverage criteria that are used in the context of unit testing, more specifically for testing each method (intra-method). These criteria are classified in exception-independent (ei) and exception-dependent (ed) [6]. Basically, the ei-criteria consider exception-free paths and the ed-criteria require paths with exceptions. Vincenzi et al. [6] proposed the criteria as below:

- All-Nodes-ei: this criterion requires that every node of the DUG reachable through an exception-free path is executed at least once.
- All-Nodes-ed: this criterion requires that every node of the DUG not reachable through an exception-free path is executed at least once.
- All-Edges-ei: this criterion requires that every edge of the DUG reachable through an exception-free path is executed at least once.
- All-Edges-ed: this criterion requires that every edge of the DUG not reachable through an exception-free path is executed at least once.
- All-Uses-ei: this criterion requires that every definition-use association reachable through an exception-free path is executed at least once.
- All-Uses-ed: this criterion requires that every definition-use association not reachable through an exception-free path is executed at least once.

One of the advantages of JaBUTi is that it does not require the Java source code to perform its activities. The instrumentation, test execution and coverage analysis are based on the Java bytecode. This tool was developed as a desktop application that requires human interaction to perform testing activities. The tester must operate the tool in order to create a testing project and to select classes to instrument. The tool instruments the selected classes and generates a set of required elements for each criterion.

The tester may implement test cases according to the JUnit framework [18] for covering the generated required elements. The test cases are submitted to JaBUTi and are executed to generate a trace file that is used to perform coverage analysis. The tester can see a set of reports that includes control flow graphs and coverage analysis by method, criterion or project. The results can be used to evaluate and decide whether the activity shall stop or more test cases should be added to reach a better coverage. All tests carried out by the JaBUTi tool are saved as a JaBUTi project, which can be reloaded at any time for improving its test set, querying required elements and coverage analysis. Extensions to implement control and data flow criteria for testing Java and AspectJ programs have also been developed [6], [8].

IV. JaBUTiService: an Overview

The JaBUTiService is based on the JaBUTi tool implementation. We performed a detailed analysis of the human interactions needed to operate JaBUTi with its GUI and defined the operations of the JaBUTiService interface. We removed the desktop interface of JaBUTi and reused its core components responsible for performing source code instrumentation and coverage analysis.

The architecture of the JaBUTiService is presented in Figure 3. JaBUTiService operations are described by a WSDL file. Any client can get this file and generate the data types and stubs to invoke JaBUTiService operations. The first version of JaBUTiService is publicly available for tests [1].

The JaBUTiService is composed of four components: 1) Axis2 engine [19]; 2) JaBUTiService Controller; 3) a Database (DB); and 4) JaBUTiCore. *Axis2* is a Java-based implementation for both the client and server sides to send, receive and process SOAP messages. The *JaBUTiService Controller* component implements the operations published on the WSDL interface. It is a controller that receives messages, accesses

[1]http://www.labes.icmc.usp.br/˜jabutiservice

the Database and calls JaBUTiCore operations to perform instrumentation and coverage analysis. The *Database* stores testing projects' information, including instrumented classes, test cases and trace files. The *JaBUTiCore* component wraps the core classes of the JaBUTi tool that handle instrumentation and coverage analysis.

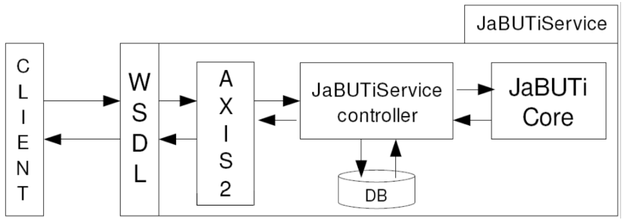


Fig. 3. Architecture of JaBUTiService.

A comprehensive set of operations was defined to provide the structural testing service that would be useful for the service clients. The operations are at a low level of granularity and a tester must use a sequence of operations to perform structural testing using the service (see the JaBUTiService website for more details). These operations can be combined to create workflows for performing unit, pairwise, and pointcut based (integration) testing. Table I shows the list of operations available at the JaBUTiService's interface.

ID	Operation name
op1	createProject
op2	updateProject
op3	deleteProject
op4	ignoreClasses
op5	selectClassesToInstrument
op6	getAllRequiredElements
op7	getRequiredElementsByCriterion
op8	getGraph
op9	addTestCases
op10	getInstrumentedProject
op11	sendTraceFile
op12	getCoverageByCriteria
op13	getCoverageByClasses
op14	getCoverageByMethods
op15	getAllCoveredAndUncoveredElements
op16	clearProject

TABLE I
OPERATIONS OF JABUTISERVICE INTERFACE.

JaBUTiService is a stateful web service and needs to follow a sequence of operation execution. The state machine for a project tested with the JaBUTiService is presented in Figure 4. We refer to the ID used at the enumeration of the JaBUTiService's operations (Table I) to simplify the operation calls in Figure 4. Operations on a higher level of granularity can be developed using the current operations.

The state machine starts when a project is created (op1). The machine moves to the state "Created" and to its sub state "Idle". The project may be updated (op2), cleared (op16) or removed (op3) at any time. Any update leads to "Idle" again. The operation deleteProject (op3) ends the project's life cycle.

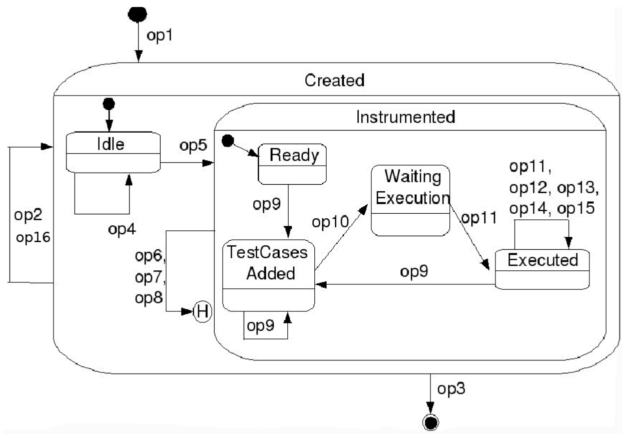


Fig. 4. State Machine for a project in the JaBUTiService.

Starting at the "Idle" state, the user may select classes to be ignored for instrumentation (op4). The user also needs to select classes to be instrumented by JaBUTiService (op5). This operation leads the project to the "Instrumented" state and to its nested state "Ready". The operations to get required elements (op6), get required elements by criterion (op7) and get the def-use graph of each method (op8) can be called at any time in the "Instrumented" state. These operations do not change the project's state.

The operation to add test cases to the project (op9) leads the project to the "TestCases Added" state. At this moment, the JaBUTiService will create a package with the instrumented classes, test cases and instructions to execute the project to generate the execution trace file. We chose to let the tester execute the instrumented code in the client environment, instead of executing it in the server. The reason is that usually tested programs use external libraries, databases and even other systems. It would be very hard to configure the environment to execute the program under test at the server side. Thus, the tester must get the instrumented code (op10 - the machine moves to the "Waiting execution" state) and execute it locally. A trace file will be generated automatically and it must be sent to the JaBUTiService (op11).

At this moment, the machine moves to the state "Executed". In this state, the user can add new test cases (op9 - that leads to "TestCases Added" again) and repeat the process; get the overall testing coverage of the project (op12); get the testing coverage by class (op13); get the testing coverage by method (op14); and get the covered and uncovered required elements (op15).

Other operation not shown in this example is one to set infeasible requirements that are not to be taken into account on coverage analysis. This can be used to reach a 100% coverage, for example.

V. Usage Example

We designed an example to validate the usage of the JaBUTiService that includes the following steps:

1) Choose a Java program to test;

2) Design or reuse test cases;
3) Create a script to invoke JaBUTiService's operations and perform structural testing; and
4) Present the coverage analysis according to the selected criteria.

We chose to use the classes of the Apache-Commons-BeanUtils library [20] from the Apache Software Foundation to show how to use JaBUTiService. This library contains 62 classes. We decided to perform unit testing (considering a method as a unit) using the following criteria: all-nodes-ed, all-nodes-ei, all-edges-ed, all-edges-ei, all-uses-ed and all-uses-ei. The library under test has 478 methods and comes together with a set of 243 test cases distributed in 27 JUnit classes [20]. The script to invoke JaBUTiService's operations was designed with this sequence of operations:

1) createProject (op1);
2) selectClassesToInstrument (op5);
3) getAllRequiredElements (op6);
4) addTestCases (op9);
5) getInstrumentedProject (op10);
6) sendTraceFile (op11); and
7) getCoverageByCriteria (op12).

After creating the project, selecting the classes to instrument and adding the test cases, the instrumented package is automatically sent to a specific directory (testing directory). The test cases of the instrumented package are also automatically executed and the script remains blocked until the generation of the trace file is done. In the next step the script finds the trace file in the testing directory and sends it to the service to get the coverage by criteria. Since the configuration of the scripts for each program to be tested is not trivial, we created a user interface to help this activity. A screenshot of the application can be seen in Figure 5.

The tab `Create Project` is selected to inform the project name, the user name, the address where the service is installed for testing, and to select classes to be instrumented. Tab `Add TestCases` is where the tester informs test cases to be used. Tab `Configurations` is used to indicate the place where the package with the instrumented code will be sent and run together with the libraries it needs. Tab `Reports` is where the tester must select the desired reports (which are the required elements and coverages for each selected criterion). Reports are generated in HTML and can be exported in the XML format.

Using the desktop application, the user does not need to know details of how to send the trace file to the JaBUTiService. When the button `Run` is pressed, the instrumented classes are sent to the service, along with the test cases. Next, the application requests from the service the instrumented package and downloads it to the selected folder using the tab `Configurations`. The instrumented package is then run against the test cases and using the selected libraries, if necessary. The tracing file is sent automatically back to JaBUTiService and the selected reports are displayed in the HTML format and all data is saved in XML.

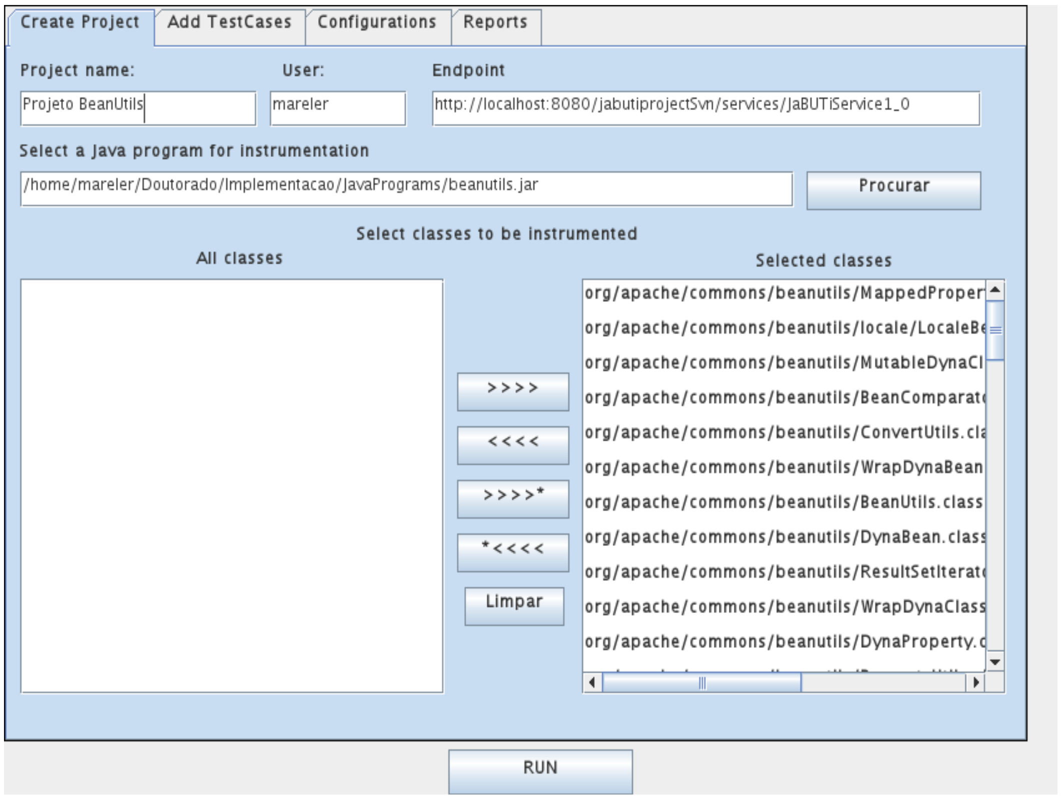
A snippet of our script to invoke JaBUTiService's operations is shown in Listing 1. The JaBUTiService is represented by a stub that encapsulates the service invocation. Line 1 shows the invocation of the getInstrumentedProject operation and the result is returned in `resp`. The response is a package containing the instrumented code and the test cases. The package is saved in the testing directory (`TestDir`) (lines 2-6). A command line to execute the package against the test cases is defined (lines 7-9) and executed (line 10). The script remains blocked (line 11) until the execution is finished. After the execution, a trace file is created and sent to the service (lines 12-18).

Listing 1. Script to invoke JaBUTiService's operations.

```
1  GetInstrumentedProjectResponse resp = stub.
       getInstrumentedProject(input);
2  datahandler = resp.get_return().getFile();
3  FileOutputStream fos = new FileOutputStream(new File
       (TestDir + "package.jar"));
4  datahandler.writeTo(fos);
5  fos.flush();
6  fos.close();
7  String execString = "java -cp "+TestDir+"package.jar
       ";
8  execString+=" br.jabuti.junitexec.JUnitJabutiCore -
       trace "+TestDir+"test.trc -cp ";
9  execString=execString+TestDir+"package.jar -tcClass
       "+ TestSuiteClass;
10 Process p = Runtime.getRuntime().exec(execString);
11 p.waitFor();
12 SendTraceFile inputTrace = new SendTraceFile();
13 inputTrace.setProjectId(projectid);
14 inputTrace.setIdUserName(user);
15 fds = new FileDataSource(new File(TestDir + "test.
       trc"));
16 datahandler = new DataHandler(fds);
17 inputTrace.setTracefile(datahandler);
18 SendTraceFileResponse traceResp = stub.sendTraceFile
       (inputTrace);
```

The test of the library Apache-Commons-BeanUtils with JaBUTiService through the desktop application and scripts resulted in the coverages shown in Figure 6. The first column shows the criteria used in the test. The coverage generated in this example takes into account the whole project and the results are expressed as the sum of the number of elements required and covered by all methods of the 62 classes. The second column lists the sum of the required elements to cover the corresponding criteria of the first column. The third column lists the sum of the covered elements and the fourth column shows the information as a percentage of the coverage achieved by the set of test cases. For the all-uses-ed criterion, the number of required elements was 0 because no variable was used in the exception handlers. The obtained coverages could also be shown for each method and for each class. Coverage of each method, for example, could be more useful to show for which ones the test cases are not being effective to cover the required criteria and thus to create more test cases to achieve a satisfactory coverage.

The construction of scripts and the desktop application were decisions that we made for the first evaluation of the service. However, there are several ways to access the service. In the context of the QualiPSo project, the JaBUTiService



Project name: Projeto BeanUtils

User: mareler

Endpoint: http://localhost:8080/jabutiprojectSvn/services/JaBUTiService1_0

Select a Java program for instrumentation: /home/mareler/Doutorado/Implementacao/JavaPrograms/beanutils.jar

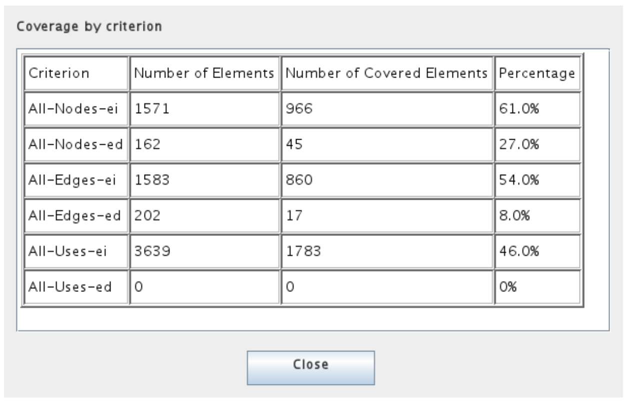
Select classes to be instrumented

All classes

Selected classes

```
org/apache/commons/beanutils/MappedProper
org/apache/commons/beanutils/locale/LocaleBe
org/apache/commons/beanutils/MutableDynaCl
org/apache/commons/beanutils/BeanComparat
org/apache/commons/beanutils/ConvertUtils.cl
org/apache/commons/beanutils/WrapDynaBean
org/apache/commons/beanutils/BeanUtils.class
org/apache/commons/beanutils/DynaBean.class
org/apache/commons/beanutils/ResultSetIterat
org/apache/commons/beanutils/WrapDynaClass
org/apache/commons/beanutils/DynaProperty.c
```

Fig. 5. User interface to configure the scripts.



Coverage by criterion

Criterion	Number of Elements	Number of Covered Elements	Percentage
All-Nodes-ei	1571	966	61.0%
All-Nodes-ed	162	45	27.0%
All-Edges-ei	1583	860	54.0%
All-Edges-ed	202	17	8.0%
All-Uses-ei	3639	1783	46.0%
All-Uses-ed	0	0	0%

Fig. 6. Coverage data for the Apache-Commons-BeanUtils.

has been accessed through a Mashup interface. Mashup is a technology for combining the content from more than one source into an integrated application [21]. Each data source can be represented by a small application called gadget. Thus, the mashup application is a set of gadgets. Some gadgets have been developed for accessing JaBUTiService using EzWeb [22] to manage them in the QualiPSo project.

Currently, JaBUTiService is being used by an educational tool called Progtest [23]. Progtest is going to support testing and programming courses to evaluate students' exercises. The students can submit their programs and test cases and then Progtest generates reports based on control and data flow coverage analysis and the evaluation of each test case. The instrumentation and coverage analysis are performed by JaBUTiService.

VI. Related Work

The idea of software tools available as Web services has emerged recently and a few proposals are found in the literature. Some of them envisage the integration of services offered and others have just the provision of a service in particular. The work related to the availability and integration of services is proposed by Ghezzi and Gall [4], Baldamusa et al. [24], and Yap et al. [25]. Bartolini et al. [26] refer to the availability of a stand-alone testing service, without allowing for integration with other similar services.

Ghezzi and Gall [4] proposed a platform that offers services for software analysis. The idea is that stand-alone tools of analysis are encapsulated by a service and registered in the proposed platform. The registered tools are classified by a taxonomy and data input and output are standardized by an ontology. The platform provides mechanisms for the selection of tools and a BPEL workflow is instantiated for their invocation. This is an idea similar to ours, except for being a different type of software engineering tool, which could be integrated and composed with JaBUTiService.

Bartolini et al. [26] developed an approach called SOCT (Service-Oriented Coverage Testing) in which is proposed a web service called TCov to support structural testing services. The developer must manually instrument the service or composition of services to be tested and include calls to TCov to record the paths of execution. Every time the instrumented service runs, the paths of execution will be recorded in TCov. Thus, the client or integrator using the service can run test cases and query TCov to learn the achieved path coverage. For any coverage analysis, the client or integrator should use the data collected from TCov and do it manually or either develop a tool to do this. This work is very similar to ours in the sense that it proposes a service to support structural testing. However, their aim is to test (BPEL) composition of services while our aim is to test any type of Java and AspectJ based software, including web services.

SOCT supports the developer partially because the instrumentation of the service must be done manually. TCov is used to register the trace of executions and the developer must choose which information to log in accordance with the criteria that will have its coverage analyzed. In our approach, the support to the developer is fully automated, because JaBUTiService performs code instrumentation automatically. The support of SOCT to the client or integrator is also partial. The user of TCov must retrieve the data collected and stored by TCov and perform the coverage analysis manually or with another tool not provided by SOCT. In our approach, the JaBUTiService performs the analysis of structural coverage criteria from the traces of the instrumented code.

Baldamusa et al. [24] used web services for creating an environment to integrate a set of formal verification toolkits. There is a verification scripting facility so that users are able to specify how the sub-tasks within any verification run should be carried out. They claimed that web services are suitable for promoting distributed and open integration which fits the decentralization and dynamism required by the formal verification community. Another proposal is to establish directories for publishing web services specifically related to formal verification. Service composition becomes a new service or a workflow and the formal verification services are invoked from within it.

Yap et al. [25] described an approach to support the dynamic discovery, integration and invocation of remote software engineering tools using web services. These tools are facilities for JEdit, an open source Integrated Development Environment. Each facility is encapsulated as a remote web service that is dynamically registered, discovered, integrated, and invoked from within JEdit IDE as required. In this work it does not result in a workflow, but in functionalities that are added to the desktop tool. Our work allows that tools available as services can either be integrated in an workflow (using BPEL, for example) or by a desktop tool similar to the presented in Section V.

VII. Conclusion

In this paper we discussed the idea of providing and using testing tools as web services. We also presented the use of a web service to perform structural testing and support the developer of Java programs in the activity of software testing. A proof of concept was designed and presented.

Despite the advantages listed in Section II, we identified some difficulties in the use of a testing tool as a web service. Currently, there are not many software engineering services available. For this reason, all the potential integration among these types of services cannot be carried out yet. For instance, it is not possible to integrate an issue tracker or version control service because they are not available as web services.

Another limitation is related to interactivity. Software tools with graphical interfaces facilitate user interaction with the features offered by the tool. The JaBUTi tool, for example, allows testers to interact with the control flow graph that represents the test code and see the corresponding source code, nodes and paths not covered, which helps generating new test cases to increase testing coverage. This type of interaction is not possible with the JaBUTiService in its current version.

The present version of the JaBUTiService has some of the problems inherent from the web services technology. For instance, the processing time is worse than the stand-alone version, since there was an increase in the XML parsing overhead and the network communication add some delay to response time. There are other issues to be considered like authentication, security and testability.

As future work, we intend to perform the following improvements in our approach. JaBUTiService will be published in a specialized broker for sharing testing services. An ontology of software testing [10] is being used to classify the published testing services and to standardize input and output data in order to facilitate tool integration. JaBUTiService will have new features to increase the testability of web services coded in Java and facilitate the task of integrators when testing a service to be used in a composition. JaBUTiService will also be able to store results of testing and make them available for certifiers and will be integrated with other QualiPSo factory services like version control and issue tracker.

VIII. Acknowledgements

The authors would like to thank the Brazilian funding agencies: FAPESP (process 2008/03252-2), CAPES, CNPq and to the QualiPSo Project (IST-FP6-IP-034763) for their support.

References

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

[2] M. N. Wicks and R. G. Dewar, A new research agenda for tool integration, Journal of Systems and Software, vol. 80, no. 9, pp. 1569 1585, 2007.

[3] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Kramer, Service-oriented computing: A research roadmap, in Service Oriented Computing, ser. Dagstuhl Seminar Proceedings, F. Curbera, B. J. Kramer, and M. P. Papazoglou, Eds., vol. 05462. Internationales Begegnungs- und Forschungszentrum fur Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[4] G. Ghezzi and H. Gall, Towards software analysis as a service. in ASE Workshops. IEEE, 2008, pp. 110. [Online]. Available: http://dblp.uni-trier.de/db/conf/kbse/asew2008.html

[5] QUALIPSO, 2009, qualipso - Quality Plataform for Open Source Software. Available on-line at: http://www.qualipso.org/.

[6] 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. 15131541, 2006.

[7] I. G. Franchin, O. A. L. Lemos, and P. C. Masiero, Pairwise structural testing of object and aspect-oriented java programs, in The 21th Software Engineering Brazilian Symposium, Joao Pessoa, PB, Brazil, 2007.

[8] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero, Control and data flow structural testing criteria for aspect-oriented programs, Journal of Systems and Software, vol. 80, no. 6, pp. 862882, 2007.

[9] P. Frankl and E. Weyuker, A data flow testing tool, in Proceedings of IEEE Softfair II Conference on Software Development Tools, Techniques, and Alternatives, Dec. 1985, pp. 4653.

[10] E. F. Barbosa, E. Y. Nakagawa, and J. C. Maldonado, Towards the establishment of an ontology of software testing, in SEKE, K. Zhang, G. Spanoudakis, and G. Visaggio, Eds., 2006, pp. 522525.

[11] E. Prudhommeaux and A. Seaborne, 2008, sPARQL Query Language for RDF. W3C Recommendation.

[12] J. C. Maldonado, Potential uses criteria: A contribution to software structural testing, PHD thesis, DCA/FEE/UNICAMP, Campinas, SP, 1991, (in portuguese).

[13] 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. 366427, 1997.

[14] S. Rapps and E. J. Weyuker, Selecting software test data flow information, IEEE Transactions on Software Engineering, vol. 11, using data no. 4, pp. 367375, Apr. 1985.

[15] J. R. Horgan and S. London, Data flow coverage and the C language, in TAV4: Proceedings of the symposium on Testing, analysis, and verification. New York, NY, USA: ACM, 1991, pp. 8797.

[16] E. Kapsammer, T. Reiter, and W. Schwinger, Model-based tool integration - state of the art and future perspectives, In proc. of the 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications (CITSA 2006), 20-23, Orlando, USA, 2006. [Online]. Available: http://www.bioinf.jku.at/publications/2006/0706.pdf

[17] M. L. Chaim, Poke-tool a tool for supporting structural testing based on data flow analysis of programs, Masters thesis, DCA/FEEC/UNICAMP, Campinas, SP, Apr. 1991, (in portuguese).

[18] K. Beck and E. Gamma, 1998, jUnit Test Infected: Programmers Love Writing Tests - Java Report, July 1998, Volume 3, Number 7. Available on-line at: http://JUnit.sourceforge.net/doc/testinfected/testing.htm.

[19] T. A. S. Foundation, 2009, apache Axis2 Users Guide. Available on-line at: http://ws.apache.org/axis2.

[20] BeanUtils, 2009, apache Common BeanUtils. Available on-line at: http://commons.apache.org/beanutils/.

[21] S. Cetin, I. N. Altintas, H. Oguztuzun, A. H. Dogru, O. Tufekci, and S. Suloglu, A mashup-based strategy for migration to service-oriented computing, in IEEE International Conference on Pervasive Services, 2007, pp. 169172.

[22] EZWEB, 2009, ezWeb - Developing Approach. Available on-line at: http://ezweb.morfeo-project.org/EzWeb-Info/Tutorial/.

[23] E. Barbosa, M. Silva, C. Corte, and J. Maldonado, Integrated teaching of programming foundations and software testing, in 38th Annual Frontiers in Education Conference, Oct. 2008, pp. S1H5S1H10.

[24] M. Baldamusa, J. Bengtsona, G. Ferrari, and R. Raggi, Web services as a new approach to distributing and coordinating semantics-based verification toolkits, in First International Workshop on Web Services and Formal Methods (WSFM), 2004.

[25] N. Yap, H. C. Chiong, J. Grundy, and R. Berrigan, Supporting dynamic software tool integration via web service-based components, in ASWEC 05: Proceedings of the 2005 Australian conference on Software Engineering. Washington, DC, USA: IEEE Computer Society, 2005, pp. 160169.

[26] C. Bartolini, A. Bertolino, and E. Marchetti, Introducing serviceoriented coverage testing. in ASE Workshops. IEEE, 2008, pp. 5764. [Online]. Available: http://dblp.uni-trier.de/db/conf/kbse/asew2008.html