2010 Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement Companion

Event- and Coverage-Based Testing of Web Services

André Takeshi Endo*, Michael Linschulte†,
Adenilso da Silva Simão* and Simone do Rocio Senger de Souza*
*Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (USP), Brazil
{aendo, adenilso, srocio}@icmc.usp.br
†Faculty of Computer Science, Electrical Engineering and Mathematics, University of Paderborn, Germany
{linschu}@upb.de

***Abstract*—Service-Oriented Architecture (SOA) fosters the development of loosely coupled applications. Web services have been favored as a promising technology to implement SOAs. Since web services are often involved in complex business processes and safety-critical systems, it is important that they are of high level of reliability. In this paper a strategy for event-based testing of web services is introduced which enables a systematic test case generation for structural testing with the focus on code coverage. The novelty of the approach stems from its integration of an event-based view with a coverage-based view. This forms a combination of black- and white-box testing leading to grey-box testing. A bank service example illustrates the strategy.**

***Keywords*-web service testing; model based testing; structural testing**

I. Introduction

Nowadays, Information Technology landscape of enterprises is mostly heterogeneous, complicating the integration of systems implemented by different technologies. Service-Oriented Architecture (SOA) was introduced to fill this gap by providing a de facto standard enabling communication among those systems. SOA is an emerging approach that aims at fostering loose coupling among applications. It provides a standardized, distributed, and protocol-independent computing paradigm. Software resources are wrapped as "services" that are well-defined and self-contained modules providing business functionalities and being independent from other service states or contexts [1].

Web services are a concrete implementation of the SOA concept. They incorporate applications that provide a set of operations accessed by other applications through the Internet, using XML standards such as SOAP, WSDL, and UDDI. This technology enables to interconnect different enterprises and to build complex business applications which are used in safety critical environments.

SOAs and web services add new factors that need to be considered during software development. Distribution, lack of control and observability, dynamic integration with other applications, and XML standards usage are key features of this new type of software. In this context, software testing is essential to guarantee a high degree of service quality and reliability. Consolidated test approaches have been reused in SOA applications [2]. However, many testing techniques cannot be directly applied due to the dynamic and adaptive nature of SOAs [3]. Moreover, different stakeholders (roles) should be considered along the process of service testing. Canfora and Di Penta [3] identify five roles involved in service testing: developer, provider, integrator, certifier, and user. The developer implements the service and has therefore access to the source code for testing. The provider tests the service against the requirements of a consumer but does not have access to the source code. The integrator tests the confidence of a given service before she/he integrates it into his own applications but has no control over the service in use. The certifier is an optional instance who tests the service to assess the service's fault-proneness, but not within a specific composition as the integrator does. The user has no concern about testing and just wants to be guaranteed that the service works fine. As the majority of roles considers a service as black box where the source code is not available, most of the proposed approaches deal with black box testing.

Currently, Model-based Testing (MBT) has been investigated [4], [5], [6], [7], such as the use of an event-based model to test web services [8]. MBT can be used to reach the desirable level of reliability for every kind of environment. However, those techniques usually neglect the important role of code coverage in testing activity. Actually, this happens because the absence of source code is a common assumption in web service testing. Even applying a formal testing approach, there is no guarantee that all the service code is covered. Structural testing, also called white-box testing, since the source code is assumed to be available, enables the generation of test requirements for covering the internal logic of a program. However, MBT and structural testing tend to be expensive, since they involve complex procedures, notations, and tools. Thus, the application of both approaches seems promising in web services that need high assurance such as banking services and safety critical services.

In this paper, we propose a new testing strategy for web services that combines a formal approach based on events, which provides systematic test case generation, and a structural testing technique, which is concentrated on the code coverage. This combination forms the novelty of the approach as it integrates black- and white-box testing leading to grey-box testing. Our approach is recommended for the service developer because the source code is necessary to measure the code coverage. We illustrate our integrated strategy on the basis of a bank service example, showing evidences that the two approaches are complementary.

The paper is organized as follows. Section II summarizes related work. Section III introduces basic concepts and technical background necessary for our integrated strategy which is described in Section IV. Section V describes an application of our strategy and sums up the results. Finally, in Section VI, we present some conclusions and discuss future work.

II. Related Work

Web service testing has been intensively investigated due to the SOA importance to many types of application [9]. Service testing aims at finding faults and assuring conformance between a service and its specification [10]. More recently, MBT has been applied in the context of web services. The main focus of application is on web services that operate with data persistence. In this scenario, the service response does not depend only on the input, but also on the state of the service [5]. However, the WSDL interface does not provide the behavioral specification of the service. The behavioral specification is a key requirement for successful interactions with this type of service. It enables the discovery of invoked actions in relation to the service, the process or temporal aspects of service interaction [11].

In [12], the authors propose a high quality service registry that incorporates automated web service testing before the registration. Graph transformation rules are used to specify the behavior of the web service. The test case generation uses a domain based strategy, named partition testing [13].

In [5], an Extended Finite State Machine (EFSM) is generated by means of a WSDL-S specification [14]. WSDL-S adds pre and post-conditions for each operation, enabling a behavioral specification. An algorithm that derives an EFSM using the service specification is also presented.

Frantzen et al. [10] use *Symbolic Transition Systems* (STS) to model and test the coordination of web services. Initially, the test is carried out for the interaction between a consumer and a web service. Then, adaptations and extensions are presented for a context with multiple ports and multiple web services.

Bertolino et al. [7] propose a model based approach to generate stubs for web services. The stubs are generated following Service Level Agreement contracts and functional contracts (State Machines). STS notation is used to specify the right values for invocation parameters, the right order of messages and corresponding features of a message provided by the service.

Zheng et al. [6] and Keum et al. [4] tackle the control and data flow coverage for their respective models, but the code coverage is not considered. Except for [5], the previous cited work assumes the manual creation of a formal model. In this context, Bertolino et al. [15] propose a method named StrawBerry to automatically derive an automaton model from the service implementation using synthesis and testing techniques.

In summary, to our knowledge there is no work addressing the problem whether or not the MBT approach covers the code of the web service. Our present work uses a graph model-based approach based on events for test case generation and structural testing for measuring code coverage. Our goal is to take advantage of proposed black-box approaches complemented by code coverage information, proposing a new testing strategy for the service developer.

III. Background

In this section, we introduce basic concepts and technical background which forms the basis of our strategy, presented in Section IV.

A. ESG4WS

Belli et al. [16] introduced Event Sequence Graphs (ESGs) for testing graphical user interfaces. ESGs enable to describe user interactions in a simplified way ignoring the internal state of a system under test (SUT) which is often hard to identify. Nevertheless, ESGs allow generating test cases systematically and not only for the desirable behavior but also for the undesirable one providing a holistic testing. Moreover, an extension of ESGs, named ESG4WS, was proposed for testing web services [8].

ESGs are directed graphs; their nodes represent events whereas edges represent valid sequences of events. An example can be seen in Figure 1. In ESG4WS, an event is specified as a request (call of an operation) to the SUT or a response (server response message to a "request"). Two pseudo nodes, '[' and ']', symbolize entry and exit where any node can be reached by entry, and any node can reach the exit. Any sequence of nodes starting at the pseudo vertex '[' and ending with the pseudo vertex ']' is called Complete Event Sequence (CES). CESs are successful runs through the ESG, i.e., they are expected to arrive at the exit of the ESG that models the SUT. For *positive testing* of SUT, CESs are used as test inputs.

Figure 2 depicts an ESG for the banking service example. Consider its solid lines at first glance. The dashed lines will be introduced after the explanation of the SUT. This

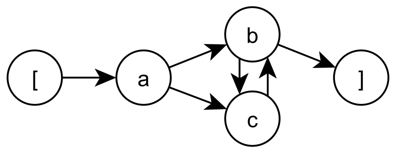

Figure 1. An ESG with pseudo nodes [,].

ESG exemplifies a web service implementing a banking service. The service allows to open an account, withdraw and deposit money, and to close the account. In Figure 2, requests and responses that belong to the same operation are circled. Note that the operation "withdraw" is successful only if enough money is saved. Furthermore, a successful account creation is required before money can be transferred.

A node of the ESG representing an operation call with input parameter is refined by a Decision Table (DT), i.e., input parameters and their structure are modeled by DT. Thus, refined nodes are augmented by DT and in the corresponding ESG they are double-circled. A DT allows the modeling of valid system behavior depending on input parameters. Table I illustrates an example of a DT that represents the operation *withdraw* of Figure 2. The top left cell identifies the operation represented by the DT. The constraint part (rows 2-5) specifies (a) valid domains to input parameters and (b) additional constraints. These constraints are the pre-conditions of a contract and have to be resolvable to true (T) or false (F). The action part (rows 6-7) specifies valid system responses with respect to constraint sets, called rules (see column "R1", "R2", ...). The action part represents the post-conditions of a contract. Thus, every rule of the DT defines a contract.

Table I
DECISION TABLE FOR THE WITHDRAW OPERATION.

withdraw(id, v)	R1	R2	R3	R4	R5	R6
id $\in N$	T	F	T	F	T	T
v $\in R^+$	T	T	F	F	T	T
id $\in$ existingAccountID	T	-	-	-	F	T
v $\leq$ AccountBalance	T	-	-	-	-	F
withdrawSuccessful	$\surd$					
withdrawNotSuccessful		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

As mentioned before, the WSDL does not describe valid or invalid sequences in which service operations are supposed to be called. Therefore, testing of invalid sequences of events is also of substantial interest. To test the undesirable behavior, the ESG is to be completed by additional edges (dashed lines in Figure 2), namely: (1) "Response"→"Request", (2) "["→"Request", (3) "Request"→"Request". The additional edges symbolize Faulty Event Pairs (FEPs) as they should result in a SOAP fault. Wherever another system response arises (perhaps one of the regular responses), the corresponding tests are failed ones (*negative testing*). For enabling testing the system reaction upon an unexpected, undesirable event, FEPs are to be extended to Faulty Complete Event Sequences (FCESs). A FCES starts with the pseudo vertex '[' and ends with the FEP under consideration. Hence, for *negative testing* of SUT, FCESs are used as test inputs.

B. Structural Testing

During the testing activity, it is necessary to determine the point in time where, if ever, the executed tests have in fact enabled reaching the desired level of quality of the software under test; this is necessary to stop testing. A testing criterion defines which properties or requirements need to be tested to evaluate the quality of the tests [17]. These properties or requirements are called test requirements. A limitation of the testing activity is that test requirements may be infeasible, i.e., no input data can cause their execution [17].

The criteria can be classified into different techniques, depending on the information used to derive the test requirements. Structural testing, also called white-box testing, is a technique in which test cases are derived from the program internal structure [18]. Most of the structural testing criteria use a program representation called Control Flow Graph (CFG) 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 and all-edges [17]. In addition, there are data flow criteria that use information about the program data flow to derive the test requirements. Rapps and Weyuker [19] propose an extension of the CFG, called Def-Use Graph (DUG), to add information related to the program data flow. A data-flow association is established between a variable assignment (definition) and a subsequent reference (use) through a path without redefinition.

In this work, we consider the testing criteria for Java programs proposed by Vincenzi et al. [20]. In [20], the authors present JaBUTi, a structural testing tool that implements control-flow and data-flow testing criteria for Java programs. JaBUTi implements testing coverage criteria that are used in the context of unit testing, more specifically for testing each method (intra-method). The coverage criteria used in this paper are as follows:

- **Control-flow criteria:**
  - **All-Nodes:** this criterion requires that every node of the DUG reachable is executed at least once.
  - **All-Edges:** this criterion requires that every edge of the DUG reachable is executed at least once.
- **Data-flow criteria:**
  - **All-Uses:** this criterion requires that every

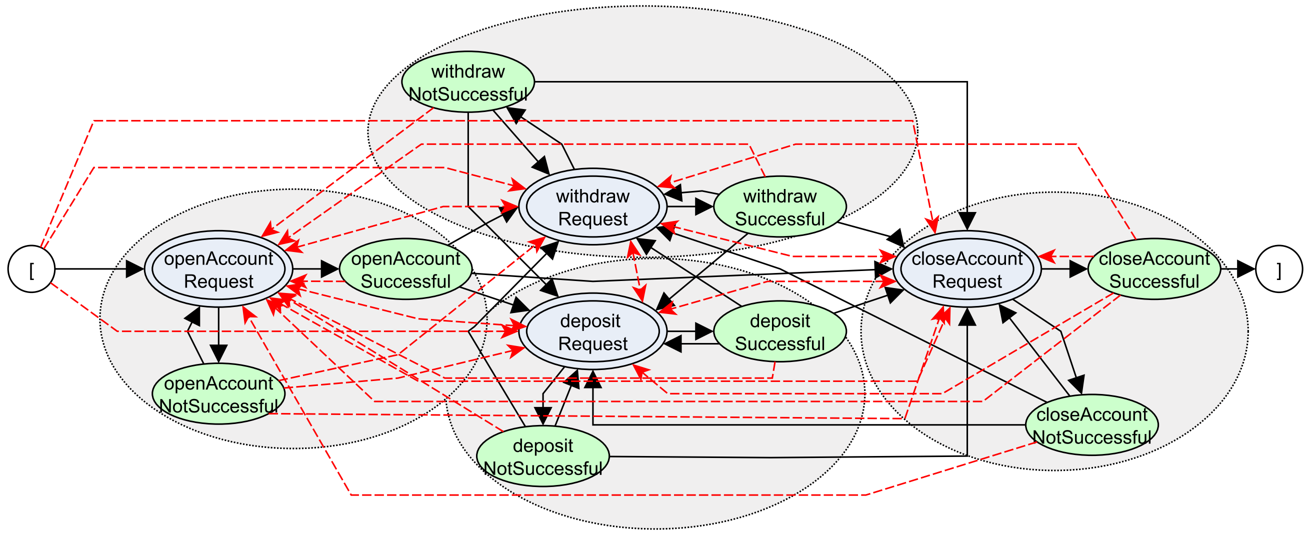

Figure 2. ESG of the banking service example.

definition-use association reachable is executed at least once.

– **All-Pot-Uses:** this criterion requires that every definition-potential-use association reachable is executed at least once. The definition-potential-use extends the definition-use association considering implicit references.

Using coverage criteria brings also another benefit: They can be used to prioritize test cases during regression testing [21], [22].

Regression testing is a costly process that validates a modified software and detects whether new faults were introduced into previously tested code [23]. In the SOA context, changes and evolution are common issues and new versions are deployed constantly. A complete re-execution of the tests may be infeasible due to schedule and budget constraints. Hence, mechanisms for ranking and selecting a subset of the tests are useful. In our strategy, we propose some ranking strategies that could be useful to select an appropriate subset of generated test cases for the regression testing.

IV. Integrating Both Approaches: Event-based Structural Testing

ESG4WS uses the specification of an SUT to generate test cases which will be executed on the SUT. In contrast, structural testing concentrates on the implemented system and measures the achieved coverage of specific system executions. Therefore, the combination of both approaches (Event-Based Structural Testing) enables to identify differences between the specification of a system and its real implementation. In a first step, the model helps to identify significant faults in the system behavior. If all tests are successfully run, in the second step the structural testing provides additional information about the test coverage of the system.

Observing the coverage information like non-covered test requirements and low coverage of some test cases, some problems can be identified like: (1) additional functionality in the system which is not specified; (2) missing, incomplete or faulty specifications; and (3) special fault handling methods, which are hard to trigger, e.g. for erroneous database interactions.

The testing strategy introduced in this paper can be used to identify the faults mentioned before. The presented strategy can be seen in Figure 3 and is divided into the following steps:

1) `Event-based modeling`: An ESG model for the service is to be designed.
2) `Test case generation`: The test cases are generated based on the ESG model using a support tool. The generated test set is divided into positive testing and negative testing.
3) `source code instrumentation`: The service code is instrumented to include instructions for recording the execution trace. In our work, we use JaBUTi tool. This step can be executed in parallel to steps 1 and 2.
4) `test cases execution with coverage analysis`: The abstract test cases generated by ESG should be realized in an executable format. We use the framework JUnit. The testing coverage is evaluated considering control and data flow criteria as described in Section III-B. In this point, the coverage is measured for each test case and for all

test cases. The pre-definition of a desired coverage is essential for determining whether other activities will be carried out.

5) `identifying improvement points`: Points of the code (methods/functions) with no coverage are identified. An analysis should be done to identify whether these points are not executable or the model misses some scenario. Test cases with a low coverage also deserve attention since some requirements might not be completely implemented. For instance, it is expected that a certain test case has a high coverage but it does not happen because some requirement is not implemented.
6) `refining the model`: If the coverage goal is not reached, additional test cases are implemented. The model could be extended to include these new tests.
7) `ranking test cases`: This activity uses the information generated by step 4 to rank test cases. We adopt some strategies to minimize the generated test set w.r.t. the code coverage:

   - *Random (R)*: The random strategy starts with an arbitrary test case and adds new test cases until a selected coverage criterion reaches a pre-defined coverage level. It is possible that some test cases do not increase the coverage, e.g., test cases testing different scenarios of already covered code. In this case, the test case under consideration is excluded and a new one is selected.
   - *Random with proportion (RP)*: The R strategy does not consider the proportion of positive, negative, and structural tests. A second strategy is to consider that each type of test (positive, negative, and structural) is important and test cases should be selected respecting the test proportion. It is similar to R strategy, but it selects a subset for each type of test.
   - *Cost with proportion (CP)*: An issue that should be considered is that some operation calls of the web service can be more expensive than others. Thus, we propose to associate a cost with each operation. The main motivation for it is that in a real-world service, financial costs and processing time are important facts in the testing activity. A simple cost function for each test case would be: $cost_{tc} = \sum_{op=1}^{n} x_{op} * c_{op}$, where x_{op} is the number of calls of the operation op and c_{op} the corresponding cost. Using the coverage average (avg_{tc}), we propose that each test case has $(1/cost_{tc}) * avg_{tc}$ points. The selection is based on the test cases with more points.

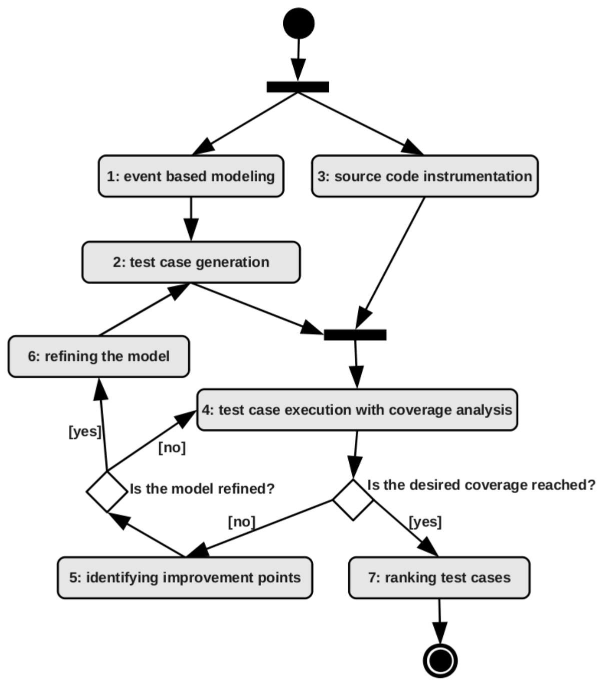
1: event based modeling
3: source code instrumentation
2: test case generation
6: refining the model
4: test case execution with coverage analysis
[yes]
[no]
Is the model refined?
Is the desired coverage reached?
[no]
[yes]
5: identifying improvement points
7: ranking test cases

Figure 3. Test process overview.

The information collected in Step 7 is useful for regression testing. A common scenario is that the TCs can be expensive to run and therefore only a subset of test cases is desired for regression testing. We propose three different strategies (R, RP and CP) from which the most adequate strategy for the web service under test can be selected. Other types of strategies for regression testing can be adapted, e.g., [24], [23].

V. Example

We exemplify our approach using the banking service example (extracted from [4]) presented in Section III. Banking service under consideration is a web service with four operations: openAccount, withdraw, deposit, and closeAccount. The ESG model for the service can be seen in Figure 2 and a DT example in Table I. The ESG model was constructed exclusively to support the testing activity.

In the following, we present some results reached by the application of our strategy in the banking service example. The testing criteria coverage showed in this section does not consider the infeasible test requirements. We defined as coverage goal to reach 100% coverage for the presented criteria. The coverage was measured for each method (unit test) and summarized for the complete web service (all the classes). Each test case (TC) can have different numbers of operation calls, since they represent different scenarios modeled in the ESG.

- **Positive Tests**

Using the ESG approach, 3 TCs were generated for the positive testing. In Table II, we show the coverage reached by positive and negative test sets. We note that the positive tests covered almost all the test requirements. In order to reach a full 100% coverage of all testing criteria, we added 3 new TCs to cover test requirements of the data flow criteria (all-uses and all-pot-uses). The added TCs basically execute test requirements related to scenarios with no created account and two or more accounts. This is necessary because our ESG model considers interactions within a single account. We note that structural testing can improve the tests since additional cases were not considered by the ESG model.

Table II
COVERAGE FOR POSITIVE AND NEGATIVE TESTS.

Type	All-Nodes	All-Edges	All-Uses	All-Pot-Uses
positive	100%	100%	98%	95%
negative	97%	95%	96%	90%

- **Negative Tests**

For the negative testing, two test sets were generated. The first set with 16 TCs covers the faulty pairs of the ESG. The second set, also with 16 TCs, covers the cases with parallel access combining the possible operations. The two sets had a testing coverage lower than the positive set (Table II), not covering test requirements non-covered by the positive set. We added 3 new structural TCs for the negative test set to reach the 100% coverage. Two of them are equal to the TCs used to complete the positive test set.

- **Ranking and Minimizing Test Cases**

Using ESG4WS for the example, 35 TCs were generated. Then, 4 additional TCs were manually created to cover the structural criteria. We obtain a test set with 39 TCs. For our example, we defined that our test goal is to cover 100% of all-nodes, all-edges, all-uses, and all-pot-uses criteria. Table III presents the results achieved for the example. All structural TCs were included in all strategies because these tests were created for the test sets to reach 100% of coverage. We obtained test sets with 26-36% from the initial test set.

Table III
RESULTS FOR MINIMIZING STRATEGIES.

	R	RP	CP
# positive TCs	1	2	3
# negative TCs	5	9	7
# structural TCs	4	4	4
total	10 (25.6%)	15 (38.4%)	14 (35.8%)

- **Discussion of the Results**

Structural coverage criteria have been studied as a technique to measure and select test cases [19], [17]. As the test requirements are based on the source code, the sequence of operations is not considered. Generating test cases based on the DUG can be a complicated and error-prone activity. The main application of structural testing is to evaluate the test quality of existing test cases.

The black-box approaches for testing web services usually neglect the code coverage since they assume that the source code is not available. Using the ESG4WS approach has all the benefits of the MBT, such as automatic and systematic test case generation. However, the test coverage of the source code is not considered. Moreover, possible faults or missing requirements in the model are difficult to identify.

According to Myers [18], the black box and white box (structural) techniques are complementary. As it can be seen in the study, our approach brings for the service developer the benefits of both techniques. The coverage information and ESG model can be provided for the other roles (provider, certifier, and integrator) as additional artifacts to support their test activities. For instance, the coverage information can help the integrator to select a subset of the test cases when a new version of the service is available. Step 7 of our strategy discusses this issue and provides some forms to handle it.

We note that generating positive and negative tests did not result in a complete 100% of coverage. The structural testing helped to extend the TCs reasonably. In contrast, even with a high code coverage rate, a different order of operation calls could reveal a new fault. Thus, both approaches complement each other in a systematic and reasonable way.

VI. LIMITATIONS, CONCLUSIONS AND FUTURE WORK

In this paper, we presented a strategy to combine MBT and structural testing applied for web services. An event-based technique called ESG4WS and structural testing criteria for Java programs were used in this context. The proposed strategy is relevant for service testing since most of existing approaches handle a web service as black box. The approach introduced in this paper is novel in the sense that it integrates event-based, black-box testing with coverage-based testing, thus white-box testing. We consider this as a step towards formalizing a grey-box testing strategy.

We identify some limitations of our approach and ways to overcome them. As we use testing criteria and supporting tool for Java programs, our strategy is currently limited to Java-based web services. Nevertheless, it is not difficult to find structural testing tools for other languages and adapt them to be used in our strategy. We particularly adopt criteria for unit testing, but integration testing appropriate criteria are to be used. Another limitation is the necessity of learning a new modeling technique (ESG and DTs) and some effort to model the service. A trained engineer took around about one hour to set up the model and corresponding DTs for the banking service example given in this paper. We believe that it is an acceptable trade-off, since the approach has all the benefits of MBT like automatic test case generation and early detection of faulty requirements.

Our strategy is in the first place recommended to service developers since the structural test technique used requires access to the code. We note that, although ESG4WS generates a test set with a high coverage, structural testing could provide an extra measurement to assure a web service with high quality. Moreover, our strategy could be adapted and applied to other type of software, since ESG can model other systems and there are structural testing tools for many programming languages.

As future work, we consider the evolution of the proposed strategy to deal with service compositions and the conduction of more sophisticated case studies to investigate the fault detection capabilities. An additional improvement is to extend the ESG4WS approach for increasing the testing coverage, identifying unhandled scenarios like the multiplicity of test objects (in our example, bank account). A further goal is to investigate other mechanisms for ranking test cases.

Acknowledgments

The authors are thankful to the anonymous reviewers for providing useful comments and suggestions on this paper. This work was financially supported by FAPESP/Brazil (process 2009/01486-9 and 2008/04614-5) and CNPq/Brazil (process 305236/2009-1).

References

[1] M. P. Papazoglou and W.-J. Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The International Journal on Very Large Databases (VLDB)*, vol. 16, no. 3, pp. 389–415, July 2007.

[2] A. Bucchiarone, H. Melgratti, and F. Severoni, "Testing service composition," in *8th Argentine Symposium on Software Engineering (ASSE)*, Mar del Plata, Argentina, 2007.

[3] G. Canfora and M. Di Penta, "Testing services and service-centric systems: Challenges and opportunities," *IT Professional*, vol. 8, no. 2, pp. 10–17, 2006.

[4] C. Keum, S. Kang, I.-Y. Ko, J. Baik, and Y.-I. Choi, "Generating test cases for web services using extended finite state machine," in *18th IFIP International Conference on Testing of Communicating Systems (TESTCOM)*. Springer, 2006, pp. 103–117.

[5] A. Sinha and A. Paradkar, "Model-based functional conformance testing of web services operating on persistent data," in *Workshop on Testing, analysis, and verification of web services and applications (TAV-WEB)*. New York, NY, USA: ACM, 2006, pp. 17–22.

[6] Y. Zheng, J. Zhou, and P. Krause, "An automatic test case generation framework for web services," *Journal of Software*, vol. 2, no. 3, pp. 64–77, set 2007.

[7] A. Bertolino, G. D. Angelis, L. Frantzen, and A. Polini, "Model-based generation of testbeds for web services," in *20th IFIP International Conference on Testing of Communicating Systems (TESTCOM)*, ser. Lecture Notes in Computer Science, vol. 5047, Tokyo, Japan, 2008, pp. 266–282.

[8] F. Belli and M. Linschulte, "Event-driven modeling and testing of web services," in *IEEE International Computer Software and Applications (COMPSAC)*, Aug 2008, pp. 1168–1173.

[9] L. Baresi and E. Di Nitto, *Test and Analysis of Web Services*. Springer, 2007.

[10] L. Frantzen, J. Tretmans, and R. d. Vries, "Towards model-based testing of web services," in *International Workshop on Web Services - Modeling and Testing (WS-MaTe)*, A. Polini, Ed., Palermo, Italy, 2006, pp. 67–82.

[11] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton, "Reference model for service oriented architecture 1.0," 2006. [Online]. Available: http://www.oasis-open.org/committees/soa-rm/

[12] R. Heckel and L. Mariani, "Automatic conformance testing of web services," in *Fundamental Approaches to Software Engineering (FASE)*, ser. Lecture Notes in Computer Science, 2005, pp. 34–48.

[13] L. White and E. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, pp. 247–257, May 1980.

[14] W3C, "Web service semantics – WSDL-S," 2005. [Online]. Available: http://www.w3.org/Submission/WSDL-S/

[15] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic synthesis of behavior protocols for composable web-services," in *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. New York, NY, USA: ACM, 2009, pp. 141–150.

[16] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study: Research articles," *Software Testing, Verification & Reliability*, vol. 16, no. 1, pp. 3–32, 2006.

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

[18] G. J. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

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

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

[21] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, Oct 2001.

[22] F. Belli, M. Eminov, and N. Gokce, "Prioritizing coverage-oriented testing process - an adaptive-learning-based approach and case study," in *Annual International Computer Software and Applications Conference (COMPSAC)*, vol. 2, July 2007, pp. 197–203.

[23] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, April 2007.

[24] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, Feb 2002.