Proceedings of The 6th IEEE International Symposium on Service Oriented System Engineering (SOSE 2011)

Using structural testing information to support monitoring activities

Marcelo Medeiros Eler, Marcio Eduardo Delamaro, Paulo Cesar Masiero
ICMC/USP
Sao Carlos/SP - Brazil
Email: {mareler,delamaro,masiero}@icmc.usp.br

***Abstract*—Third party services can change without notification as they are usually under the control of external providers. Unexpected changes can clash the integrity of a composition and monitoring approaches have been proposed to detect such changes. Many of these approaches are based on executing all test cases of a regression test set to check whether the behavior of the monitored service remains the same. Such strategy, however, can be very expensive as costs may be associated to the testing activity. In this paper we propose using structural testing information to detect structural changes of monitored services and select only a reduced number of test cases of the regression test set to be executed. We also present three exploratory case studies to evaluate our approach.**

I. Introduction

Third party services used in Service Oriented Architectures (SOA) are usually under the control of different ownership domains and they can change without notification. Monitoring is an important activity of the SOA development process since it is used to detect changes of services to prevent that an unexpected change clashes the integrity of the composition (orchestration or choreography) in which it is integrated. Many approaches have been proposed in the literature to detect changes in third party services concerning both functional and non-functional requirements [1]–[10].

Concerning functional requirements, a common solution is to periodically re-run a set of test cases (regression test) to detect changes of the service [11]. Traditionally, regressing testing is used after a well known change of a service, but in SOA contexts, it has been used to detect changes [11]. This can be expensive as costs may be associated with the execution of test cases. Also, some services only allow a limited number of invocations in a short period of time. Therefore, test cases should be chosen with accuracy and their number should be limited [11], [12].

In this paper we propose using structural testing information to support and improve traditional monitoring activities. We aim at detecting structural changes of monitored services before executing a bunch of test cases. If a modification is identified then a reduced number of test cases are selected to be executed instead of executing the whole regression test set. In such approach, regression test is used after a well known modification instead of being used to detect any change. We also propose using structural testing information to check for untested code into monitored services.

Structural testing is not commonly used in SOA contexts because of the black box nature of services, but previous work have independently developed two approaches to introduce structural testing into a SOA context [13], [14]. The services developed according to their approaches are called testable services and provide their clients with structural testing information. The monitoring strategy we purpose in this paper is suitable to services that provide their clients with structural information through testing interfaces (as the testable services [13], [14]) or through test metadata [15].

This paper is organized as follows. Section II presents the background of our approach. Section III presents the monitoring approach we propose in this paper. Section IV shows the validation of our approach by means of a case study. Section V presents some related work. Section VI presents concluding remarks and future work related to this paper.

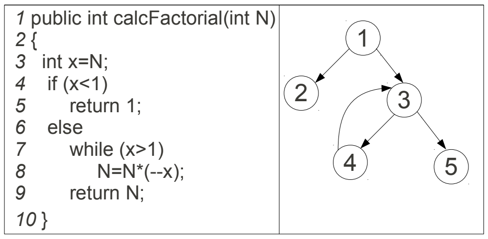
II. Background

A. Structural Testing Criteria

Structural testing focus on testing the whole structure of a program, considering instructions, control and data flow. It derives test data from the implementation according to criteria that are used to determine whether the program under test is completely tested [16], [17]. Structural testing usually adopts a model called control-flow graph (CFG) to represent the structure of a program under test. Each node of a CFG represents a block of instructions and each edge represents a possible transition from a block to another. Each block is an atomic sequence of instructions without flow deviation.

The most known structural testing criteria are the following: all-nodes, all-edges and all-uses. Test cases are created to met test requirements generated for each structural criterion. After executing the test cases, a coverage analysis is performed to measure how many test requirements were covered, which indicates how much of the structure of the program was actually exercised during the test session.

Figure 1 shows a Java method and its CFG. The structural test requirements of WSFactorial are presented in Table I. In the all-uses criterion the requirement (x,4,(3,5)) means that variable `x` is defined in node 4 and used in a decision that takes the control flow from node 3 to node 5. The requirement (x,1,4) means that variable `x` is defined in node 1 and is used in a computation in node 4.



```
public int calcFactorial(int N)
{
  int x=N;
   if (x<1)
      return 1;
   else
      while (x>1)
           N=N*(--x);
      return N;
}
```

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

TABLE I
TEST REQUIREMENTS OF CALCFACTORIAL

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

B. A Testable Service approach

Services are generally provided as black box and Bartolini et. al. [13] and Eler et. al. [14] conceived two similar approaches called BISTWS (Built-in structural testing of web services) and SOCT (Service Oriented Coverage Testing), respectively, to improve the testability of SOA applications by making services more transparent to external testers. Such services have been called *testable services*.

The testable service is a service instrumented to trace its own execution and collect information about the instructions, branches and data exercised during a test session. Testers can set a testable service to a testing mode, carry out a test session and use the testing interface of the testable service to get structural coverage information. Particularly, testable services created with the BISTWS approach [14] can can also provide their clients with testing requirements generated based on the structural criteria all-nodes, all-edges and all-uses.

III. A MONITORING STRATEGY SUPPORTED BY STRUCTURAL TESTING INFORMATION

The most common strategy to monitor services is to execute the whole regression test set to detect changes of the monitored service. Such strategy, however, can be very expensive since costs may be associated with the test of services. There can also be services that allow clients to make only a limited number of invocations during a period of time. Therefore, test cases should be chosen with accuracy and their number should be limited [11], [12].

The monitoring strategy we propose using structural testing information follows these steps: (A) check for structural modification; (B) check for behavioral change; (C) check for untested code; and (D) react. Before executing these monitoring steps, the following activities must be performed: (i) the test requirements of the monitored service must be collected to create a *requirements-baseline*, which will contain the test requirements of the original service; (ii) all test cases of the regression test set must be executed and the test results must be used to create a *results-baseline*; (iii) the structural coverage analysis obtained after the execution of the whole regression test set must used to create a *coverage-baseline*. The *coverage-baseline* shows how much of the service have been exercised by the test cases of the regression test set, concerning the structural testing criteria implemented. In the BISTWS approach, for example, the testable service can provide coverage information regarding the criteria all-nodes, all-edges and all-uses.

A. Check for structural modification

This step is used to identify structural changes of the monitored service. This can be done by comparing the test requirements collected from the monitored service with the *requirements-baseline*. The purpose of the comparison is to identify differences concerning nodes, edges, uses and instructions. The result of this activity is a list of nodes and edges affected by any of the modifications identified (control-flow or data-flow).

Test requirements directly reflects the implementation of the service since a small change of the implementation also changes the test requirements. Regardless of the many possible ways the source code of the service can be modified, the modification of the test requirements are usually the same:

1) **Changing of the control-flow** this means that instructions that deviates the control flow of the program (while, if, switch) were added to or removed from the source code of the operation. Consequently, nodes and edges were added to or removed from the structure of the operation.
2) **Changing of the data-flow:** this means that variables are being used in nodes or edges where they were not being used or variables are no longer being used in nodes or edges where they were being used. The all-uses criterion usually defines the test requirement as a pair definition-use. The definition is the node where the variable is defined and the use is the node or edge where the variable is used. In this approach we are only concerned with changes regarding the uses.
3) **No changes of control or data-flow:** this means that new instructions were added to the source code of the service without affecting nodes, edges and uses; or the mathematical or boolean operators used in computations or decisions have changed; or any other change has been made that does not affect neither the CFG structure nor the uses of the variables, except when the modification is only the definition of a variable. This change can be identified using information other than the test requirements. One possible solution is to use the hash code of the nodes of the CFG [7].

B. Check for behavioral change

This step is used to check whether the behavior of the monitored service has changed. The list of nodes, edges, uses and instructions affected by the modification identified throug the test requirements analysis (previous step) is used to select test cases that cover at least one of the modified test requirements. This strategy allows using regression test in a tradicional way - after a well known modification. The results of the test cases execution are thus compared to the *(*results-baseline) and this will indicate whether the behavior of the monitored service has changed.

Note that using this strategy the execution of the test cases depends on the structural changes identification. If no change is identified then no test case are executed. Even when a structural change is detected and test cases are launched, the number of test cases executed is reduced because only the test cases related to the parts of the code that have change are executed.

C. Check for untested code

This step is used to check whether there are parts of the service that have not been tested yet. The coverage information obtained after the execution of the test cases selected in the previous step is compared to the *coverage-baseline*. A structural coverage modification indicates that something has change within the service, even when no behavioral change is detected in the previous step. Sometimes new pieces of code are added to the service but it does not affect its behavior. The coverage analysis, however, can identify that there are new parts of the code and that they were not executed yet because the path followed by each test case would be different, i.e., the test requirements covered by each test case is different when the implementation changes.

D. React

Changes of a service are not necessarily a bad thing. A bad change is when defects are introduced in the service. We can also consider a bad change when the business rules are modified clashing the integrity of the orchestration or choreography that uses the service. The good change is when the service evolves without changing its behavior and possibly improving its functionalities. Table II presents possible changes detected by the proposed approach. This mapping also indicates whether and how a reaction should be performed according to the change detected.

TABLE II
CHANGES AND RESPECTIVE SUGGESTED REACTIONS

	All TC passed	**At least 1 TC failed**
Coverage has not changed	1	2
Coverage is higher	1	2
Coverage is lower	3	2

Here we present possible actions for each situation:

- Situation 1: This is a good change (considering only functional and structural requirements). All test cases (TC) passed and the coverage is the same or higher. The integrator should do nothing.
- Situation 2: This is a bad change since failures have been found due to the changes detected. The integrator should replace the erroneous service by other service that delivers the same functionalities. In the absence of other service for replacement the provider has to be notified. The integrator may also adapt the composition to comply with new or different business rules if this is the case.
- Situation 3: This is neither a good nor a bad change - it is neutral. The change of the service has not affected the behavior of the service, which is good. The problem is that the non executed code can hide faults as in the case of the component reused in the Ariane 5 rocket. In this case the integrator should create more test cases to test the parts of the code that have not been executed yet.

E. Implementation

Performing monitoring activities manually is difficult and error-prone. We have developed a small framework to automatize setps A to C. The instantiation of this framework consists on extending a class called `BaseTest`, which is an abstract JUnit test class. The instantiation must inform to the framework which service should be tested, which services should be monitored and which test cases are part of the regression test set.

Figure 3 shows an illustration of a particular instantiation of our monitoring framework. The framework was set to test the orchestration `Shopping Service` and to monitor the services `CatalogService`, `ZipCodeService` and `ShippingService`. In this scenario, at each modification of any of the monitored services that is identified the framework selects suitable test cases to verify whether the behavior of `ShoppingService` has changed. It is also possible to set test cases for each monitored service.

IV. VALIDATION

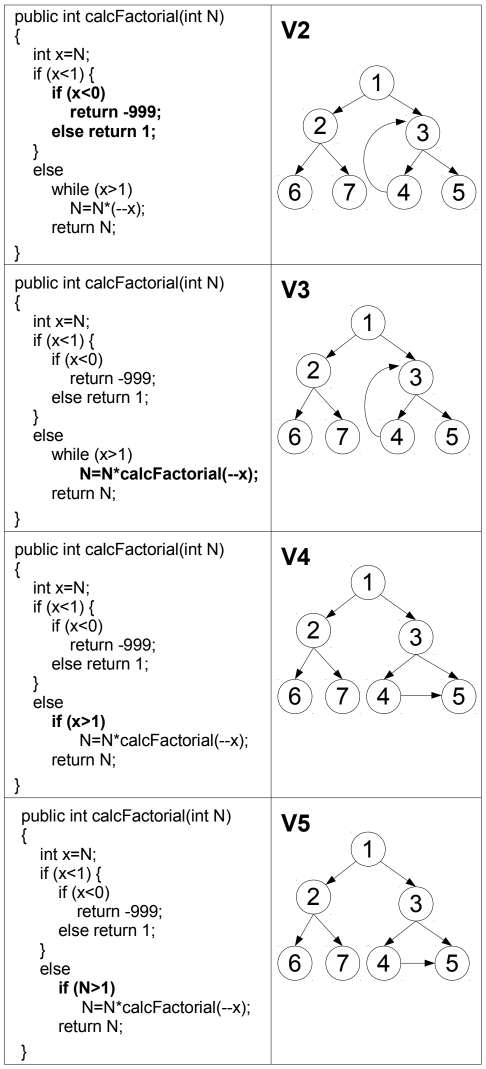
In this section we present three exploratory case studies to validate our approach. In all these three case studies we perform the monitoring of services that have been transformed in testable services using the BISTWS approach [14].

The research questions of our case studies are the following:

- **a** Did the monitoring approach identify all changes in the monitored service?
- **b** Is the number of test cases selected to be executed again satisfactory, i.e., is it less than the total number of test cases of the regression test?
- **c** Are the test cases selected to be executed again suitable to reveal changes in the monitored service? To answer this question we have to measure:
  - **c.1** The number of test cases that indicated the behavioral change.
  - **c.2** The number of test cases that were not selected and would reveal any change if they were executed.

A. Exploratory study 1: monitoring a testable service as a single service

The first version of WSFactorial (source code and CFG) is presented in Figure 1. The test requirements of the operation `calcFactorial` are presented in Table I. Figure 2 shows four versions of WSFactorial (V2 to V5). For each version we



```
public int calcFactorial(int N)
{
   int x=N;
   if (x<1) {
      if (x<0)
         return -999;
      else return 1;
   }
   else
      while (x>1)
         N=N*(--x);
      return N;
}
```

```
public int calcFactorial(int N)
{
   int x=N;
   if (x<1) {
      if (x<0)
         return -999;
      else return 1;
   }
   else
      while (x>1)
         N=N*calcFactorial(--x);
      return N;
}
```

```
public int calcFactorial(int N)
{
   int x=N;
   if (x<1) {
      if (x<0)
         return -999;
      else return 1;
   }
   else
      if (x>1)
         N=N*calcFactorial(--x);
      return N;
}
```

```
public int calcFactorial(int N)
{
   int x=N;
   if (x<1) {
      if (x<0)
         return -999;
      else return 1;
   }
   else
      if (N>1)
         N=N*calcFactorial(--x);
      return N;
}
```

Fig. 2. Changes of WSFactorial

discuss how the monitoring framework detected the modification and how the test cases of the regression test were selected to be executed again.

The first time the monitoring framework was executed if creates the whole *baseline* required by our strategy: requirements (see Table I) , results (see Table III) and coverage (100% for all criteria - see Table V).

TABLE III
TEST SET OF WSFACTORIAL

TC-ID	Input	Expected result	Path exercised	Status
TC-01	0	1	1-2	Passed
TC-02	1	1	1-3-5	Passed
TC-03	4	24	1-3-4-3-5	Passed

After creating the *baseline* information, the monitoring framework started to execute the monitoring steps periodically and detected all modifications presented in Figure 2. Table IV summarizes the changes detected by the monitoring framework.

TABLE IV
CHANGES DETECTED BY THE MONITORING FRAMEWORK IN WSFACTORIAL

	Main changes			
Transition	**Nodes**	**Edges**	**Uses**	**T.C. selected**
V1 to V2	+6, +7	+(2,6), +(2,7)	+(x,1,(2,6)), +(x,1,(2,7))	TC-01
V2 to V3			+(return,1,4)	TC-03
V3 to V4		-(4,3), +(4,5)		TC-02, TC-03
V4 to V5			-(x,1,(3,4)), -(x,1,(3,5)) +(N,1,(3,4)), +(N,1,(3,5))	TC-02, TC-03

When the service changed from *V1 to V2*, the framework detected that two new nodes and two new edges were created. TC-01 is the only test case that covers node 2 (see Table III), which was selected to be executed to check whether the behavior of WSFactorial is the same after the modification. TC-01 was executed and it passed. This indicates that the behavior of WSFactorial had not changed, at least considering the result of TC-01.

The coverage analysis step shows that the coverage reached is different, as we show in Table V. The coverage is lower because there are new nodes, edges and data which were not exercised. Indeed, we can read the code and realize that the service has changed because in this version the developer is handling invalid entries (negative numbers), but the test cases of the regression test cannot detect the modification because none of them tests for invalid entries.

TABLE V
COVERAGE ANALYSIS OBTAINED FOR EACH VERSION OF WSFACTORIAL

Version	all-nodes	all-edges	all-uses
V1	100%	100%	100%
V2	85%	85%	92%
V3	85%	85%	93%
V4	85%	85%	92%
V5	85%	85%	92%

Since the monitoring framework detected a modification in WSFactorial, the test requirements, the test results and the coverage information regarding the second version of WSFactorial (V2) has become the new *baseline*. The transition from *V2 to V3* has added a new use to node 4: the return of the recursive call of (`N=N*calcFactorial(--x);`). TC-03 is the only test case affected by this modification because it is the only test case whose path includes node 4. TC-03 failed when it was executed pointing that the behavior of the third version (V3) of WSFactorial has changed. For the purpose of this study, we kept the monitoring framework running to detect other changes, even with the failure identified.

When WSFactorial changed from *V3 to V4*, the monitoring framework detected that an edge was missing and another was created. Test cases TC-02 and TC-03 were executed and passed. This means that the defect inserted in the previous version was removed.

In the transition from *V4 to V5*, two uses were created and two uses were removed. In V4, variable `x` is used in the statement `if (x>1)`, but in V5, variable `N` is used instead: `if (N>1)`. TC-02 and TC-03 were executed again and passed. The coverage analysis for the three criteria is also the same (Table V).

The monitoring framework implementing our monitoring steps detected all changes made in WSFactorial. This answers our research question A. The monitoring framework executed 1 out of 3 test cases in the first change, 1 out of 3 in the second, 2 out of 3 in the third and 2 out of 3 in the fourth. Thus it have been executed 6 test cases instead of 12 (50%), which answers the question B. Regarding question C, 3 out of the 6 selected test cases indicated the behavioral change (50%). We also executed the test cases which were not selected and they have not revealed any change in WSFactorial, indicating that they should not be selected to be executed.

B. Exploratory study 2: monitoring a testable service exhaustively changed by mutation in the context of a composition

Figure 3 presents the context in which this study was performed. ShoppingService is an orchestration that uses three testable services: CatalogService, ZipCodeService and ShippingService. ShoppingService provides its clients with the operation `buyItems`, which takes a list of products (IDs) and a zip code as input parameters. The result of the operation is the total price of the shop including the shipment tax to the zip code informed. ShoppingService uses the three testable services to: (i) get the price and the weight of each product of the list (CatalogService); (ii) get the shipment address using the zip code (ZipCodeService); and (iii) get the shipment price (ShippingService).

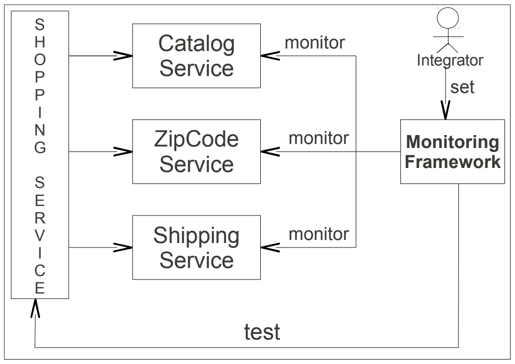


Fig. 3. Illustration of ShoppingService

We used the tool MuJava [1] to generate mutants of ShippingService. We randomly selected two mutants of each operator and each time a mutant was generated we published a new version of ShippingService. The results of this study is presented in Table VI. The first column is the name of the mutant according to the operator used to generate it. The second column indicates whether the framework detected the modification (MOD). The third column is the number of test cases selected to be executed again (#S-TC) and the fourth column is the number of test cases that failed (#S-TC-F). The fifth column indicates whether the other test cases (not selected) have failed (#NS-TC-F). To get this last information we also run those test cases that were not selected to be executed.

TABLE VI
RESULTS OF THE STUDY WITH MUTATION

Mutant	MOD	#S-TC	#S-TC-F	#NS-TC-F
M-AOIS05	yes	4	2	0
M-AOIS26	yes	8	8	0
M-AOIU01	yes	8	8	0
M-AOIU02	yes	8	8	0
M-ASRS01	yes	2	2	0
M-ASRS06	yes	2	1	0
M-COI01	yes	7	6	0
M-COI08	yes	4	4	0
M-COR02	yes	7	0	0
M-COR08	yes	3	2	0
M-LOI02	yes	4	1	0
M-LOI06	yes	4	2	0
M-ROR14	yes	3	0	0
M-ROR23	yes	8	8	0
Total	14	72	52	0

The results answer all of our research questions. Our monitoring approach detected all changes (question A). The number of test cases was minimized (question B). An approach that executes the whole test set would execute at least 168 test cases (considering only one execution) to detect changes in the monitored service, while our approach have used 72 (43%). The test cases selected to reveal changes are also suitable (question C): 52 out of 72 test cases have failed indicating a change in the behavior of the monitored service.

Another important result is that we executed the test cases that were not selected and none of them revealed any change. This also indicates the accuracy of our algorithm on selecting test cases to be executed again. There were two situations in which no test case failed. This happened because the mutants M-COR02 and M-ROR14 are equivalent mutants.

C. Exploratory Study 3: monitoring a testable service using a more realistic setting

The advantage of using mutation is that mutation allows changing the monitored service exhaustively. The disadvantage, however, is that each mutant operator makes only one change at each time. This is not very realistic considering a real evolution of a service. In this study we performed a more realistic evaluation. We asked three graduate students to make changes in ShippingService. Each student created only one new version of ShippingService based on the original version. The first student created new business rules; the second made a refactoring; and the third introduced a defect in ShippingService.

The results of this case study also answered the three research questions satisfactorily. All changes were identified (question A). The number of test cases was minimized because 39% of the test cases were executed to detect the three changes (question B). The test cases selected were suitable because 78% of the test cases selected to be executed again have failed indicating changes in the behavior of the monitored service

[1]http://www.cs.gmu.edu/ offutt/mujava/

(question C). We executed the test cases that were not selected to be executed and none of them failed.

V. Related Work

Many papers related to monitoring approaches were found in the literature and most of the approaches focuses on monitoring SLAs (Service Level Agreements) rather than functional requirements. Many authors have proposed the idea of continuous monitoring of web services at runtime to check for violations of contracts, SLAs agreements and requirements specifications [1]–[6], [8]–[10].

Liu et. al. [7] presented a regression testing approach for BPEL business processes. They specified an impact analysis rule to identify the test paths affected by changes of the BPEL structures and to generated more test cases to detect inconsistencies of the global behavior of the composition. In their approach, however, it is also requested to identify changes into the third party services used by the BPEL process to identify chanbes in the whole structure.

Harrold et. al. [15] presented an approach to select test cases to perform regression test using structural information provided by component metadata. Lin et. al. [18] extended the approach of Harrold et. al. [15] for services, but they state that their approach cannot be directly applied to web services since it is based in white-box testing. The main difference among our approaches is that we use data-flow criteria (all-uses) and coverage information to detect changes. Additionally, we present a feasible solution proposed in previous work to provide the structural information through testable services.

Bartolini et. al. [13] also presented an approach to monitor testable services. Their approach, however, is based on the execution of all test cases of the regression test set to detect changes by observing only the structural coverage analysis.

VI. Conclusions and Future Work

In this paper we presented an approach to help monitoring approaches to select suitable test cases to detect changes into monitored services. The main purpose of the approach is to detect structural changes in services using testing requirements and to detect behavioral changes using test cases execution. We use the structural change information to select only a minimized number of test cases to be executed. We used the concepts of control and data flow testing to detect changes into the monitored services. Structural testing is usually performed when the tester has access to the source code but in a SOA environment it is possible due to the structural testing information provided by testable services through testing interfaces or through testing metadata of regular services.

We performed three exploratory studies to evaluate our approach and the results were satisfactory. The monitoring approach is suitable to detect minimal changes in services and it is useful to reduce the number of test cases executed during the regression test. The results of the studies cannot be generalized because we used small examples and the modifications of the monitored services may not be realistic since they were not made in a real environment. As future work we intend to improve the change detection mechanism of our approach using other structural testing criteria and to perform further evaluation using real world services.

VII. Acknowledgments

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

References

[1] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, “Run-time monitoring of instances and classes of web service compositions,” in *ICWS ’06: Proc. of the IEEE Int. Conf. on Web Services*. Washington, DC, USA: IEEE C.S., 2006, pp. 63–71.

[2] L. Baresi, C. Ghezzi, and S. Guinea, “Smart monitors for composed services,” in *ICSOC ’04: Proc. of the 2nd Int. Conf. on Service oriented computing*. New York, NY, USA: ACM, 2004, pp. 193–202.

[3] A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta, “GLIMPSE: a generic and flexible monitoring infrastructure,” in *Proc. of the 13th European Workshop on Dependable Computing*, ser. EWDC ’11. New York, NY, USA: ACM, 2011, pp. 73–78. [Online]. Available: http://doi.acm.org/10.1145/1978582.1978598

[4] M. Bruno, G. Canfora, M. D. Penta, G. Esposito, and V. Mazza, “Using test cases as contract to ensure service compliance across releases,” in *Service-Oriented Computing - ICSOC 2005, Third Int. Conf.*, 2005, pp. 87–100.

[5] Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O’Farrell, and J. Waterhouse, “Runtime monitoring of web service conversations,” in *Proc. of the 2007 Conf. of the Center for Advanced Studies on Collaborative Research*. New York, NY, USA: ACM, 2007, pp. 42–57.

[6] S. Lamparter, S. Luckner, and S. Mutschler, “Formal specification of web service contracts for automated contracting and monitoring,” in *HICSS ’07: Proc. of the 40th Annual Hawaii Int. Conf. on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2007, p. 63.

[7] H. Liu, Z. Li, J. Zhu, and H. Tan, “Business process regression testing,” in *ICSOC ’07: Proc. of the 5th Int. Conf. on Service-Oriented Computing*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 157–168.

[8] S. Ho, W. M. Loucks, and A. Singh, “Monitoring the performance of a web service,” in *Proc. of the EEE Canadian Conf. on Electrical and Computer Engineering*, 1998, pp. 109–112.

[9] S. Rosario, A. Benveniste, and C. Jard, “Monitoring probabilistic slas in web service orchestrations,” in *IM’09: Proc. of the 11th IFIP/IEEE Int. Conf. on Symposium on Integrated Network Management*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 474–481.

[10] Q. Wang, J. Shao, F. Deng, Y. Liu, M. Li, J. Han, and H. Mei, “An online monitoring approach for web service requirements,” *IEEE Transactions on Services Computing*, vol. 2, pp. 338–351, 2009.

[11] G. Canfora and M. Di Penta, *Service Oriented Architecture Testing : A Survey*, ser. LNCS. Springer, 2009, no. 5413, pp. 78–105.

[12] S. shan Hou, L. Zhang, T. Xie, and J. su Sun, “Quota-constrained test-case prioritization for regression testing of service-centric systems,” in *Proc. IEEE Int. Conf. on Software Maintenance*, 2008, pp. 257–266.

[13] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, “Bringing white-box testing to service oriented architectures through a service oriented approach,” *J. Syst. Softw.*, vol. 84, pp. 655–668, April 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2010.10.024

[14] M. M. Eler, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, “Built-in structural testing of web services,” in *Proceedings of the 2010 Brazilian Symp. on Soft. Engineering*, ser. SBES ’10. Washington, DC, USA: IEEE C.S., 2010, pp. 70–79. [Online]. Available: http://dx.doi.org/10.1109/SBES.2010.15

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

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

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

[18] F. Lin, M. Ruth, and S. Tu, “Applying safe regression test selection techniques to java web services,” in *Proc. of the Int. Conf. on Next Generation Web Services Practices*. Washington, DC, USA: IEEE C.S., 2006, pp. 133–142. [Online]. Available: http://portal.acm.org/citation.cfm?id=1262693.1263106