Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

Using models to test web service-oriented applications: an experience report

Andre Takeshi Endo*, Maicon Bernardino da Silveira†, Elder Macedo Rodrigues†, Adenilso Simao*, Flavio Moreira de Oliveira†, Avelino Francisco Zorzo†

* Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (USP), PO Box 668, 13560-970 São Carlos, SP, Brazil
†Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Avenida Ipiranga, 6681 - Prédio 32, 90619-900 Porto Alegre, RS, Brazil

Technical Report N° 067

Porto Alegre, September, 2012

Abstract

Service-oriented architectures and web services have been widely adopted by companies to pervade integration among software systems. As reliable services are essential to assure that these systems work correctly, formal and systematic testing should be performed. This document reports the application of a model-based approach to test web services in the context of real-world applications of an IT company. The employed approach is called ESG4WSC in which an event-driven model is provided to support the test modeling and generation, as well as an environment to support the test concretization and execution.

1 Introduction

Service-Oriented Architecture (SOA) is an architectural style in which functionalities are decomposed into distinct units named services [11]. These services, distributed over a global or internal network, can be reused and combined to create new and more complex enterprise applications. In this process, a composite service is developed by integrating the functionalities of partner services. There exist various languages and specifications that have been used to describe or execute service compositions, such as WS-CDL [10] and BPEL [4]. Service-oriented applications have been mainly developed using a set of XML standards (WSDL, SOAP, and UDDI), known as Web Services technology, that is implemented by various programming languages.

Service testing has been investigated, since SOA was proposed as an architectural style and web services as its main implementation technology [3, 2]. There is research effort on proposing formal testing approaches that support the verification of single and composite web services [7]. Among them, model-based testing (MBT) has been advocated as a promising approach to provide more formal systematic and automated testing. In MBT, a model of a system under test (SUT) is designed in order to derive test cases automatically. Endo and Simao [6] propose an MBT process for testing service-oriented applications, describing the artifacts, tools and revisiting different steps. A more specific MBT approach to test web service compositions is presented in [1]. The authors propose a modeling technique, named ESG4WSC (event sequence graph for web service composition), that is able to represent communication between the partner services and the composition. Tools are also presented to support modeling and generation, as well as a test execution environment supported by an Enterprise Service Bus (ESB).

In this document, we report an experience of applying a model-based testing approach in the context of service-oriented applications developed within an IT company. The ESG4WSC approach [1] was adopted to model and test the web services, as well as a testing process was followed as in [6]. First, a set of BPEL services were modeled and test suites generated to evaluate the practical application of the test modeling and generation. Second, the complete approach, from modeling to execution, was applied in an ongoing project. We show the achieved results, as well as the lessons learned during this project.

The remainder of this report is organized as follows. Section 2 briefly describes the MBT process adopted and presents the modeling technique applied to design the test models. Section 3 reports the study configuration and adopted tools. Section 4 shows the results, observations and lessons learned. Finally, Section 5 shows the conclusion and discusses future work.

2 Testing Approach

In this study, an MBT process was employed to test service-oriented applications, which can be divided into four main steps: (i) test modeling, (ii) test generation, (iii) test concretization, and (iv) test execution. We adopted the ESG4WSC approach to model and generate the test suites [1].

ESG4WSC model is a directed graph in which nodes are events and edges represent valid sequences of events. Figure 1 shows the ESG4WSC model for the `BCS-05` Service [1]. In web service composition, request messages are represented as light gray circles and response messages as dark gray circles. The messages of the composition itself are represented with a bold line. These events are referred to as public events, while the messages exchanges with the partner services are referred to as private events. There are two special nodes, '[' and ']', that represent the entry and exit nodes, respectively.

A sequence of nodes that is connected by edges is called *event sequence*. Any event sequence that starts with '[' is called partial event sequence (PES). If a PES ends with ']' is called complete event sequence (CES). There are two assumptions in the graph: (i) every event must be reachable, through an event sequence, from the entry node, and (ii) the exit node must be reachable, through an event sequence, from any event.

[1]We replaced the original labels by generic ones along the text.

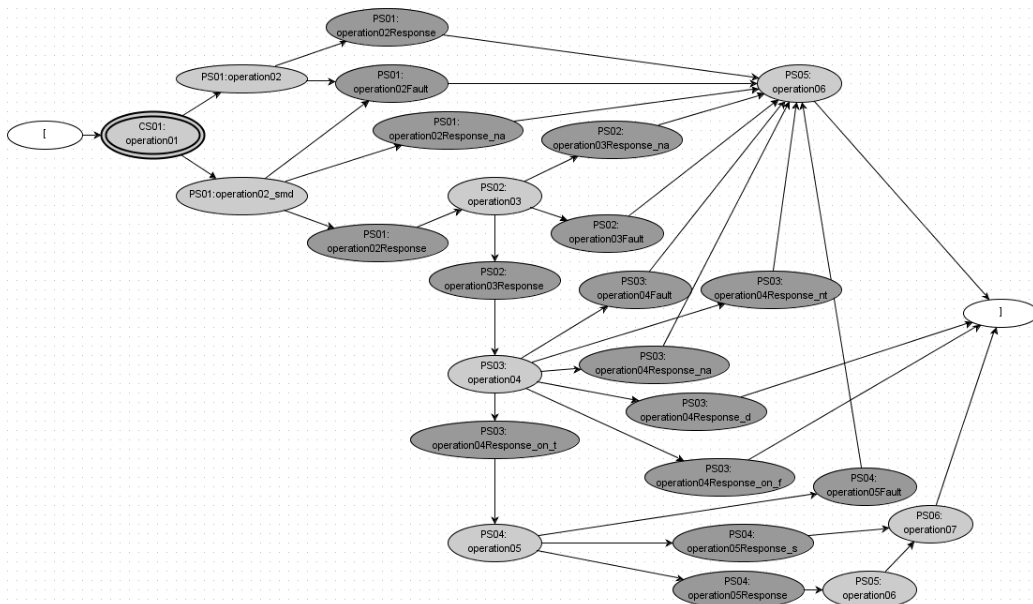

Figure 1: ESG4WSC model.

It is possible to associate decision tables (DTs) with public request events when input parameters may cause different events. In Figure 1, event `CS01:operation01` has two next events that are provoked according to an input parameter. Events with associated DTs are double circled. Table 2 shows a simple DT for event `CS01:operation01`. It contains a constraint $constraint01 =' smd'$, which can be evaluated as true or false, and two rules (R1 and R2). R1 means that if constraint $constraint01 =' smd'$ is evaluated as true, the next event should be `PS01:operation02_smd`. The ESG4WSC model has other features, such as refined events, associated refining ESG4WSCs, and parallel representation. See [1] for more details on the model.

Table 1: Decision table for event `CS01:operation01`.

	rules	
constraints	R1	R2
$constraint01 =' smd'$	True	False
next actions (events)		
`PS01:operation02`		✓
`PS01:operation02_smd`	✓	

2.1 Positive Testing

To generate the positive test cases, a test suite composed of CESs is produced automatically to cover all edges (event pairs) in the test model. The constraints in the DTs are also solved and new edges are added and covered if necessary [1]. Using the model in Figure 1, 14 test cases were generated, as shown in Table 2.1.

Table 2: Positive test cases for the BCS-05 service.

Test case
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Fault, PS05:operation06 ⟩
⟨ CS01:operation01(constraint01!='smd'), PS01:operation02, PS01:operation02Fault, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response_na, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response_na, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Fault, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Response_na, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Fault, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Response_d⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Response_on_t, PS04:operation05, PS04:operation05Response, PS05:operation06, PS06:operation07⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Response_nt, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Response_on_f⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Response_on_t, PS04:operation05, PS04:operation05Fault, PS05:operation06⟩
⟨ CS01:operation01(constraint01='smd'), PS01:operation02_smd, PS01:operation02Response, PS02:operation03, PS02:operation03Response, PS03:operation04, PS03:operation04Response_on_t, PS04:operation05, PS04:operation05Response_s, PS06:operation07⟩
⟨ CS01:operation01(constraint01!='smd'), PS01:operation02, PS01:operation02Response, PS05:operation06⟩

Figure 2 shows a representation for the first test case in Table 2.1 as an XML file.

```
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R1" >
            <Param name="constraint01">smd</Param>
        </Event>
        <Event label="PS01:operation02_smd" type="request" public="false" />
        <Event label="PS01:operation02Fault" type="response" public="false" />
        <Event label="PS05:operation06" type="request" public="false" />
    </CompleteEventSequence>
</TestCase>
```

Figure 2: XML file for a test case #1.

2.2 Negative Testing

Using the same model presented in Figure 1, test cases can also be generated for undesirable situations. These cases are referred to as *negative test cases* which can be applied to public or private events. In public negative testing, unexpected cases are tested between public events that were not predicted in the model. The test cases generated for this case is called *public faulty event sequence* (PubFES). For the private negative testing, different fault classes are defined and simulated during the tests. Table 2.2 shows the fault classes we used, in which type of event is applied, and the number of negative test cases generated for the `BCS-05` service. The test cases generated for this case is called *private faulty event sequence* (PrivFES). It is important to emphasize that new fault classes can be defined and integrated in the approach.

Table 3: Number of negative test cases.

Fault Class	Event	#Test cases
no response (NR)	request	8
missing service (MS)		
unexpected fault (UF)		
Longtime response (LR)	response	16
Wrong XML schema (WSc)		
Wrong XML syntax (WSy)		
Right Schema, wrong data (WD)		

Figures 3 and 4 show examples of XML files for negative test cases applied on request events and response events, respectively.

```
no response
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R1" >
            <Param name="constraint01">smd</Param>
        </Event>
        <Event label="PS01:operation02_smd" type="request" public="false" />
        <Event label="PS01:operation02Response" type="response" public="false" />
        <Event label="PS02:operation03" type="request" public="false" noresponse="true" />
    </CompleteEventSequence>
</TestCase>
missing service
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R2" >
            <Param name="constraint01">anyother</Param>
        </Event>
        <Event label="PS01:operation02" type="request" public="false" missingservice="true" />
    </CompleteEventSequence>
</TestCase>
unexpected fault
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R1" >
            <Param name="constraint01">smd</Param>
        </Event>
        <Event label="PS01:operation02_smd" type="request" public="false" unexpectedFault="true" />
    </CompleteEventSequence>
</TestCase>
```

Figure 3: XML files for faults in request events.

```
longtime response
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R1" />
        <Event label="PS01:operation02_smd" type="request" public="false" />
        <Event label="PS01:operation02Response_na" type="response" public="false" timeout="100000" />
    </CompleteEventSequence>
</TestCase>
wrong XML schema
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R1" />
        <Event label="PS01:operation02_smd" type="request" public="false" />
        <Event label="PS01:operation02Response" type="response" public="false" />
        <Event label="PS02:operation03" type="request" public="false" />
        <Event label="PS02:operation03Response" type="response" public="false" />
        <Event label="PS03:operation04" type="request" public="false" />
        <Event label="PS03:operation04Response_on_f" type="response" public="false" wrongSchema="true" />
    </CompleteEventSequence>
</TestCase>
wrong XML syntax
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R1" />
        <Event label="PS01:operation02_smd" type="request" public="false" />
        <Event label="PS01:operation02Response" type="response" public="false" />
        <Event label="PS02:operation03" type="request" public="false" />
        <Event label="PS02:operation03Response_na" type="response" public="false" wrongSyntax="true" />
    </CompleteEventSequence>
</TestCase>
right schema, wrong data
<TestCase>
    <CompleteEventSequence>
        <Event label="CS01:operation01" type="request" public="true" rule="R1" />
        <Event label="PS01:operation02_smd" type="request" public="false" />
        <Event label="PS01:operation02Response" type="response" public="false" wrongData="true" />
    </CompleteEventSequence>
</TestCase>
```

Figure 4: XML files for faults in response events.

3 Study Configuration

This study was conducted in cooperation with an IT company that provided access to its applications, as well as technological support and documentation. The communication with the IT company happened through on-line means (e-mail, instant messaging) and meetings with the development team. Overall, all data presented in this document was produced and collected in one month.

We adopted a set of tools in order to support the conduction of the study and automation. The Test Suite Designer (TSD) tool was used to design the ESG4WSC model and generated the test cases [1]. The Event Runner for Test Execution (ERunTE) tool was adopted to support the test concretization and execution [1]. ERunTE is integrated with mule-ESB [12] to record and control all messages produced during the tests. SoapUI [8] was employed to mock some services that have security issues and create initial stub messages. Finally, Eclipse [5] and JUnit [9] were used to support the development of adaptors and the test execution.

4 Analysis of Results

This study was divided into two parts. First, we focused on the modeling and test generation over BPEL composite services. Second, we analyzed the approach in the context of the `ABC` project, emphasizing the test concretization and execution.

4.1 Test Modeling and Generation

In this part, we used 23 composite services specified in BPEL to evaluate the ESG4WSC modeling capabilities. Then, we evaluated the test suites generated from these models. The models were designed in an exploratory way that is divided into two steps:

1. Exploratory modeling: an initial ESG4WSC model is designed, identifying request and response events. The order among them is also modeled. During this step, the global communication is prioritized and branches and DTs are put aside. Generic events and comments were used to recall that these issues need to be handled in future.

2. Test-driven modeling: using the model designed in previous steps, a more detailed analysis was conducted to identify and model DTs (constraints, rules, and actions). Generic events and comments were removed and branches along the model were solved. The goal of this step is to set up a model that is adequate to generate test cases.

This configuration of steps was intuitively conducted during the modeling of the first four services. After a phase of identification, all services were modeled in two steps and therefore with two model versions. The sources of information about the 23 services were the BPEL specification itself, WSDL files, logs of the BPEL engine, and talks to the developers.

Table 4.1 shows the test model's characteristics for each of the modeled services. Observe that all services are asynchronous (request-only) since the number of public response events is zero in all lines. The number of public request events is low, most of them with one or two requests. Only services `BCS-03` and `BCS-20` have more public requests. The number of private request and response events reflects the complexity of communication with the partner services. The overall complexity of the test models can be summarized by the number of events and edges. Clearly, `BCS-11` has the large test model with 447 events and 501 edges, followed by `BCS-22`, `BCS-09`, and `BCS-20`. In services `BCS-11`, `BCS-20`, and `BCS-22`, refined events and refining ESG4WSCs were adopted to deal with the complexity of many events and edges. ESGs in parallel were not used in these models. Most of the branches in the models are caused by different types of responses, instead of input parameters. This is observed by the number of DTs (and their elements, constraints, actions, and rules) that is low. `BCS-20` has the highest number of DTs which is consistent with its public request events; `BCS-10` has the highest number of constraints for one DT. Ten out of 23 services do not have associated DTs.

For the test models, positive and negative test suites were generated. Table 4.1 shows the test suite's information divided by type of testing. For each type of test suite, the number of executed events is also shown as a measure of cost. All test suites were automatically generated using the TSD tool. The cost of positive test cases is highly dependent on number of events and edges in the model. `BCS-11` has the highest number of test cases (40) and `BCS-01`, `BCS-15`, `BCS-17`, and `BCS-23` has the lowest number of test cases (2).

For PubFESs, the cost is related to the number of public request and response events. `BCS-03` and `BCS-20` have the highest number of test cases and cost, 49 test cases executing 98 events and 25 test cases executing 222 events, respectively.

Table 4: Test model information.

Service Name	#Public Request Events	#Private Request Events	#Request Events	#Public Response Events	#Private Response Events	#Response Events	#Refined Events	#Generic Events	#Events	#Edges	#Refining ESG4WSCs	#ESGs in Parallel	#Decision Tables	#Constraints	#Actions (next events)	#Rules
BCS-01	1.0	3.0	4.0	0.0	2.0	2.0	0.0	2.0	8.0	10.0	0.0	0.0	1.0	1.0	2.0	2.0
BCS-02	1.0	8.0	9.0	0.0	7.0	7.0	0.0	0.0	16.0	25.0	0.0	0.0	1.0	2.0	3.0	3.0
BCS-03	7.0	7.0	14.0	0.0	0.0	0.0	0.0	0.0	14.0	21.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-04	2.0	15.0	17.0	0.0	13.0	13.0	0.0	0.0	30.0	42.0	0.0	0.0	1.0	1.0	2.0	2.0
BCS-05	1.0	8.0	9.0	0.0	16.0	16.0	0.0	0.0	25.0	39.0	0.0	0.0	1.0	1.0	2.0	2.0
BCS-06	2.0	15.0	17.0	0.0	14.0	14.0	0.0	1.0	32.0	45.0	0.0	0.0	1.0	1.0	2.0	2.0
BCS-07	1.0	9.0	10.0	0.0	7.0	7.0	0.0	0.0	17.0	26.0	0.0	0.0	1.0	3.0	4.0	4.0
BCS-08	1.0	2.0	3.0	0.0	0.0	0.0	0.0	0.0	3.0	6.0	0.0	0.0	1.0	2.0	3.0	3.0
BCS-09	1.0	38.0	39.0	0.0	33.0	33.0	0.0	0.0	72.0	87.0	0.0	0.0	1.0	1.0	2.0	2.0
BCS-10	1.0	26.0	27.0	0.0	13.0	13.0	0.0	2.0	42.0	56.0	0.0	0.0	1.0	14.0	14.0	14.0
BCS-11	1.0	235.0	236.0	0.0	181.0	181.0	11.0	19.0	447.0	501.0	11.0	0.0	1.0	4.0	4.0	4.0
BCS-12	1.0	24.0	25.0	0.0	22.0	22.0	0.0	5.0	52.0	63.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-13	2.0	5.0	7.0	0.0	4.0	4.0	0.0	0.0	11.0	15.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-14	1.0	5.0	6.0	0.0	5.0	5.0	0.0	0.0	11.0	16.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-15	1.0	2.0	3.0	0.0	1.0	1.0	0.0	2.0	6.0	8.0	0.0	0.0	1.0	1.0	2.0	2.0
BCS-16	2.0	6.0	8.0	0.0	11.0	11.0	0.0	0.0	19.0	30.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-17	1.0	4.0	5.0	0.0	3.0	3.0	0.0	0.0	8.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-18	1.0	7.0	8.0	0.0	9.0	9.0	0.0	0.0	17.0	24.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-19	1.0	4.0	5.0	0.0	4.0	4.0	0.0	0.0	9.0	12.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-20	6.0	26.0	32.0	0.0	13.0	13.0	2.0	6.0	53.0	65.0	2.0	0.0	6.0	6.0	12.0	12.0
BCS-21	1.0	13.0	14.0	0.0	14.0	14.0	0.0	4.0	32.0	44.0	0.0	0.0	0.0	0.0	0.0	0.0
BCS-22	1.0	33.0	34.0	0.0	29.0	29.0	2.0	8.0	73.0	86.0	2.0	0.0	0.0	0.0	0.0	0.0
BCS-23	2.0	3.0	5.0	0.0	0.0	0.0	0.0	0.0	5.0	9.0	0.0	0.0	1.0	2.0	3.0	3.0

Sixteen out of 23 services have only one test case.

The PrivFESs are divided in accordance with the fault classes described in Section 2.2: no response (NR), missing service (MS), unexpected fault (UF), long-time response (LR), wrong XML schema (WSc), wrong XML syntax (WSy), and right schema wrong data (WD). The test suites for the fault classes NR, MS, and UF are dependent on the number of private request events. They have different costs because each fault class has its own characteristics during the execution. For instance, in class NR, the affected request event happens and no response is produced, while in class UF a response event is provoked. `BCS-11`, `BCS-09`, and `BCS-22` have the highest number of test cases.

The test suites for the fault classes LR, WSc, WSy, and WD are dependent on the number of private response events. `BCS-11`, `BCS-09`, and `BCS-22` also have the highest number of test cases. `BCS-03`, `BCS-08`, and `BCS-23` have no

test case for these fault classes since their models do not contain private response events (as observed in Table 4.1).

The last two columns show the total number of PrivFESs, including all seven fault classes. As the negative testing for private events produces test suites that cover all request and response events in combination with the fault classes, a high number of negative test cases is generated. For all models, its cost exceeds the cost of positive testing.

Table 5: Test suite information.

Service Name	#Positive Test Cases	#Executed Events	#Public Faulty Event Sequences	#Executed Events	#Private Faulty Event Sequences (NR)	#Executed Events	#Private Faulty Event Sequences (MS)	#Executed Events	#Private Faulty Event Sequences (UF)	#Executed Events	#Private Faulty Event Sequences (LR)	#Executed Events	#Private Faulty Event Sequences (WSc)	#Executed Events	#Private Faulty Event Sequences (WSy)	#Executed Events	#Private Faulty Event Sequences (WD)	#Executed Events	Total # #Private Faulty Event Sequences	#Executed Events
BCS-01	2	7	1	2	3	12	3	9	3	15	2	6	2	8	2	8	2	8	18	68
BCS-02	8	47	1	2	8	36	8	28	8	44	7	33	7	33	7	33	7	33	53	242
BCS-03	7	14	49	98	7	14	7	7	7	21	0	0	0	0	0	0	0	0	70	140
BCS-04	10	80	6	28	15	103	15	88	15	118	13	54	13	67	13	67	13	67	103	592
BCS-05	14	97	1	2	8	46	8	38	8	54	16	80	16	96	16	96	16	96	89	508
BCS-06	11	85	6	28	15	103	15	88	15	118	14	56	14	70	14	70	14	70	107	603
BCS-07	9	39	1	2	9	30	9	21	9	39	7	18	7	25	7	25	7	25	56	185
BCS-08	3	5	1	2	2	4	2	2	2	6	0	0	0	0	0	0	0	0	7	14
BCS-09	15	159	1	2	38	327	38	289	38	365	33	210	33	243	33	243	33	243	247	1922
BCS-10	14	53	1	2	26	78	26	52	26	104	13	26	13	39	13	39	13	39	131	379
BCS-11	40	822	1	2	235	3923	235	3688	235	4158	181	2623	181	2804	181	2804	181	2804	1430	22806
BCS-12	10	173	1	2	24	285	24	261	24	309	22	243	22	265	22	265	22	265	161	1895
BCS-13	4	24	5	17	5	23	5	18	5	28	4	13	4	17	4	17	4	17	36	150
BCS-14	5	27	1	2	5	22	5	17	5	27	5	16	5	21	5	21	5	21	36	147
BCS-15	2	5	1	2	2	6	2	4	2	8	1	2	1	3	1	3	1	3	11	31
BCS-16	10	55	4	8	6	28	6	22	6	34	11	40	11	51	11	51	11	51	66	285
BCS-17	2	13	1	2	4	18	4	14	4	22	3	10	3	13	3	13	3	13	25	105
BCS-18	7	47	1	2	7	36	7	29	7	43	9	38	9	47	9	47	9	47	58	289
BCS-19	3	15	1	2	4	14	4	10	4	18	4	10	4	14	4	14	4	14	29	96
BCS-20	9	112	25	222	26	250	26	224	26	276	13	88	13	101	13	101	13	101	155	1363
BCS-21	11	140	1	2	13	103	13	90	13	116	14	103	14	117	14	117	14	117	96	765
BCS-22	8	132	1	2	33	402	33	369	33	435	29	310	29	339	29	339	29	339	216	2535
BCS-23	4	7	4	8	3	6	3	3	3	9	0	0	0	0	0	0	0	0	13	26

Test model information gives an idea on the human effort that would be spent since the tester is supposed to design it manually. Although the modeling effort is directly related to the application's size and complexity, we observe that other factors may also influence on it. The restricted access to sources of information and unclear test purposes may increase the cost during the modeling. We do not investigate this hypothesis, but to design two or three models for different test purposes instead of a large one (as in `BCS-11`) may be more cost-effective.

The cost of generating the test suites is low since it is automatically performed by the tool. For the largest model, the tool took less than 11 seconds to produce

the positive and negative tests. We have provided the test suite information as an additional measure to predict the cost of execution. It is important to emphasize that a development effort is also needed to implement adaptors during the test concretization. This topic was not investigated in this part of the study. We provide further discussion on it in Section 4.2.

During the modeling of the 23 services, limitations were identified on the ESG4WSC model and tool. Although these limitations can be handled from a practical point of view, we describe them as a contribution for future improvements in the modeling technique and tool:

- BPEL-ForEach(parallel=yes) and OracleBPEL-flowN: the activity ForEach (with parallel=yes) from BPEL 2.0 and flowN extension of Oracle BPEL engine introduce, e.g., the possibility of executing n request events in parallel. As the parallelism is implicitly introduced and only decided in runtime, ESGs in parallel are not able to directly represent this case in the ESG4WSC model. The tester needs to define the number of instances before the test generation and replicate the ESGs in parallel.

- Private request/response events within loops: this case is similar to the previous one, as the number of iterations is decided in runtime. The tester should identify the number of iterations and repeat the instances in the model before generating the tests.

- Global/internal variables: BPEL engines have the concept of global variables that are independent and assigned outside the composition However, they can be referred within the BPEL and define different branches. Internal variables are more common in composite services implemented in traditional programming languages, instead of BPEL. These composite services tend to interact with databases and modify the workflow depending on internal variables. These variables cannot be represented in the ESG4WSC model and, in some way, introduce preconditions in the test cases.

- Event branch solved by distant predecessor event or parameter: when some event branch is solved by some event or parameter (DT) that happens previously in the workflow. In the current solution, the intermediate path between the solving event and the branch needs to be replicated.

4.2 Test Concretization and Execution

In this part, we applied the ESG4WSC approach in the ABC application, specifically in its composite service ABCService (ABCS). This service interacts with three other services: PartnerService01 (PS01), PartnerService02 (PS02), and PartnerService03 (PS03). Table 4.2 shows the tested services, if they are composite or not, their total number of operations, and number of operations involved in the tests.

Table 6: Information about services.

Service name	composite service?	#operations	#involved operations
ABCService	yes	8	2
PartnerService01	no	26	2
PartnerService02	no	12	1
PartnerService03	no	10	1

Based on performance test scripts and on meetings with development team, the test model shown in Figure 5 was designed. It focuses on the flow of messages triggered by the operations operation01 and operation02 of ABCService. This model was augmented with SOAP messages for the public request events and private response events. These messages are necessary to provoke some sequences.

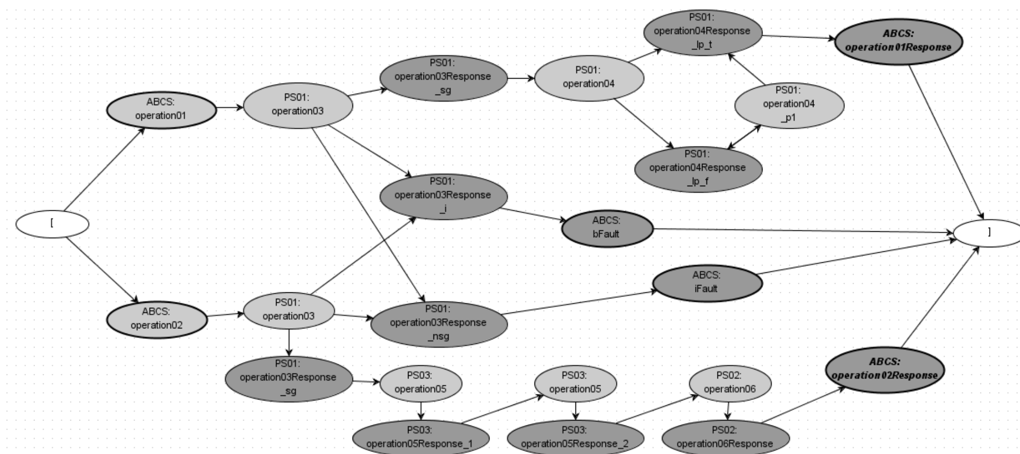

Figure 5: ESG4WSC model for ABC application.

The four services involved in the tests were deployed in the mule-ESB. Thus,

all messages produced during the tests will be passed through the bus and controlled by the module ERunTE-esbcomp. As services `PartnerService01`, `PartnerService02`, and `PartnerService03` have some security issues, we used SoapUI to mock them. Simple modifications were performed in `ABCService`, modifying the original service endpoints to the ones provided by the ESB. Thus, all messages produced during the tests are controlled by the ESB.

Using the model in Figure 5, test cases were generated using the TSD tool. Table 4.2 summarizes the test suites generated for positive and negative testing. In total, 68 test cases were generated.

Table 7: Number of positive and negative test cases.

Positive testing		
Test Suite	**#Test cases**	**Execution time**
#CESs	7	$\approx$ 13s
Negative testing		
Test Suite	**#Test cases**	**Execution time**
#PubFESs	4	$\approx$ 5s
#PrivFESs (NR)	7	$\approx$ 75s
#PrivFESs (MS)	7	$\approx$ 515s
#PrivFESs (UF)	7	$\approx$ 13s
#PrivFESs (LR)	9	$\approx$ 96s
#PrivFESs (WSc)	9	$\approx$ 12s
#PrivFESs (WSy)	9	$\approx$ 5s
#PrivFESs (WD)	9	$\approx$ 12s

The next step was the concretization of the tests. By doing so, two adaptors were implemented, PublicEventAdaptor and MessageCheckingAdaptor. PublicEventAdaptor implements the calls for public events, i.e., invoking the composite service, and checks its responses. Each event has an associated method that is annotated with the event name. MessageCheckingAdaptor implements individual verifications for all events. Its purpose is, after executing a test case, to verify if all messages were produced and in the defined order. Moreover, a couple of additional classes were implemented to configure and run the tests. The modules ERunTE-runner and ERunTE-service were used to support the test execution with the developed adaptors. Table 4.2 shows the number of lines of code and cyclomatic complexity for the two adaptors and the entire project. The metrics

were collected using Eclipse Metrics plugin [2].

Table 8: Code metrics for the test project (adaptors, setup code).

	Lines of Code	**average cyclomatic complexity**
PublicEventAdaptor	94	1.5
MessageCheckingAdaptor	231	1.8
Entire test project	900	1.6

All test cases were successfully executed; the approximate execution time is also presented in Table 4.2. The test suite for fault class MS took more time than others (around 515 seconds. This happened due to a limitation in the ERunTE tool that requires more time to simulate a missing service. The test suites for fault classes NR and LR also took more time since the ERunTE tool simulates timeouts for these tests.

During the tests, no fault was found in the SUT. This can be explained by the application' stability. The application has been released for more than two years and many cycles of testing/maintenance were performed. Although most of the generated test cases (scenarios) were likely covered by previous tests (performed by the development team), there is a lack of automated solutions for testing web services. Our impressions are that the MBT approach can be useful in scenarios, similar to the presented one, that have complex workflows and mocking different services and sequences of messages are too complex and error-prone. However, more robust and automated tools would be essential to a large scale adoption.

In this context, we observed that there is still room for improvements in the approach automation. In the adaptors development, MessageChekingAdaptor may use the XML schema in the WSDL files to support automatic verification of messages. Most of the written code can be generated automatically. Moreover, XPath queries that currently are evaluated in the adaptor code can be included directly in the model (and in the XML test cases as well). Thus, ERunTE-runner would be in charge of reading XPath queries and evaluated them, working as a test oracle. ESB configurations may also be automatically performed and integrated with the development environment.

We also noticed opportunities for supporting performance testing. For instance, the module *esbcomp* (integrated with the ESB) of ERunTE could be extended to capture metrics used to evaluate and monitor performance, such as response time and throughput. Extensions could also be carried out in the model

[2]http://metrics.sourceforge.net/

in a way that performance testing information is introduced and used to guide the generation of model-based performance tests.

5 Conclusion

In this technical report, we have presented an experience report on applying a model-based approach to test web services. A set of real-world applications of an IT company was used in the study. In the first part, the results on modeling and test generation of 23 composite services have been described. In the second part, we have provided more details on the test concretization and execution for the `ABC` application.

The experience reported in this document has given evidences that model-based testing, more specifically the ESG4WSC (event sequence graph for web service composition) approach, is applicable to test service-oriented applications in real and less controlled scenarios within an IT company. A set of lessons learned and limitations has been identified in order to improve the approach and tools. More investigation on how to deploy the approach in an ongoing project and its cost-effectiveness is still necessary. Another topic is to research the automatic generation of partial event-driven models out of BPEL specifications. This will reduce the initial effort to produce test models to verify this kind of applications and/or to maintain legacy systems.

Acknowledgments

Andre Takeshi Endo is financially supported by FAPESP/Brazil (grant 2009/01486-9) and CAPES/ Brazil (grant 0332-11-9). In this study, he was also partially supported by the project PROCAD/ CAPES 191/2007 – *"Integrando e aprimorando atividades de pesquisa, ensino/treinamento e transferência tecnológica em teste e validação de software"*. Study partially developed by the Research Group of the PDTI 001/2012, financed by Dell Computers of Brazil Ltd. with resources of Law 8.248/91. We thank CNPq/Brazil, CAPES/Brazil, INCT-SEC, and Dell for the support in the development of this work.

6 Agradecimentos

Avelino F. Zorzo possui bolsa de produtividade CNPQ/Brasil e CAPES/PROCAD. Elder M. Rodrigues possui bolsa CAPES e é pesquisador integrante do INCT-SEC.

References

[1] F. Belli, A. T. Endo, M. Linschulte, and A. Simao. Model-based testing of web service compositions. In *IEEE 6th International Symposium on Service Oriented System Engineering (SOSE 2011)*, pages 181–192, Irvine, CA, USA, 2011. IEEE Computer Society.

[2] M. Bozkurt, M. Harman, and Y. Hassoun. Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2012.

[3] G. Canfora and M. Di Penta. Service-oriented architectures testing: A survey. In *Software Engineering: International Summer Schools (ISSSE)*, pages 78–105, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] D. Jordan et al. OASIS web services business process execution language (WSBPEL) v2.0, 2007.

[5] Eclipse.org. Eclipse, 2012. [9 August 2012].

[6] A. T. Endo and A. Simao. Model-based testing of service-oriented applications via state models. In *IEEE International Conference on Services Computing (SCC 2011)*, pages 432–439, Washington, DC, USA, 2011. IEEE Computer Society.

[7] A. T. Endo and A. S. Simao. A systematic review on formal testing approaches for web services. In *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, pages 89–98, Natal, Brazil, 2010.

[8] Eviware. soapUI, 2012. [9 August 2012].

[9] junit.org. JUnit, 2012. [9 August 2012].

[10] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language version 1.0, 2005.

[11] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton. OASIS reference model for service oriented architecture 1.0, 2006.

[12] MuleSoft. Mule ESB: Open source ESB and integration platform. [9 August 2012].