e-Informatica Software Engineering Journal, Volume 6, Issue 1, 2012, pages: 61–70, DOI 10.5277/e-Inf120105

Towards Automation Design Time Testing of Web Service Compositions

Dessislava Petrova-Antonova*, Sylvia Ilieva*, Ilina Manova**, Denitsa Manova**
*Sofia University, Faculty of Mathematics and informatics
**Rila Solutions
d.petrova@fmi.uni-sofia.bg, sylvia@acad.bg, ilinam@rila.bg, denitsat@rila.bg

Abstract

Service-Oriented Architectures (SOA) allows software applications to interoperate in a new way in distributed environment. Currently, web services are the most widely adopted technology for implementation of SOA. However, they bring a number of challenges to development as well as to testing. Testing web service compositions is one of the major problems in SOA domain that is due to the unknown context, absence of web service source code, multiple provider coordination, lack of tool support, etc. In such context, the paper proposes a framework, named Testing as a Service Software Architecture (TASSA), which aims to provide design time testing of both functional and nonfunctional behavior of web service compositions described with Business Process Execution Language (BPEL). TASSA consists of set of tools that can be used together with existing development environments of service based applications. The paper focuses on an approach for negative testing and unit testing of BPEL processes. The negative testing is supported by TASSA tool, called Fault Injector tool, which implements a fault injection technique providing message delays, wrong message data, etc. The goal of unit testing is to test a BPEL process in isolation from its dependent web services. The isolation technique is implemented in another TASSA tool, named Isolation tool.

1. Introduction

Service-Oriented Architecture (SOA) is a dominant paradigm for design and development of distributed and interoperable software applications. The most widely adopted approach to SOA implementation is web services based on standards such as SOAP, WSDL and UDDI. Testing such implementations is challenging for various reasons. It is difficult to simulate all possible configurations and loads during testing process due to dynamic nature of web services and their consumers, varying load on SOA infrastructure and underlying network [1]. In addition, web services are outside of the control of consumers, leading to potential misunderstandings between parties.

The need of automation of SOA testing process results in targeting of many research efforts to SOA domain. There are separate testing tools and complex proprietary frameworks that can be used for testing of web service compositions, but open, a complete solution that meets SOA testing challenges is still missing. This paper addresses this problem by proposing a framework, named Testing as a Service Software Architecture (TASSA). The main goal of TASSA is to support the testing, validation and verification of both functional and nonfunctional behavior of web service compositions at design time [2]. It consists of set of tools that can be used together with existing development environments of service based applications. This paper focuses on two of TASSA tools, namely Fault Injection tool and Isolation tool that respectively provide functionality for negative testing and unit testing of web service compositions described with

Business Process Execution Language (BPEL). The goal of the negative testing is to test the BPEL process in case of message delays, errors in message data, wrong business logic, etc. The unit testing aims to test the BPEL process in isolation from its partner web services. The essence of fault injection and isolation techniques as well as their automation and application to real scenario are presented in the paper.

The content of the paper from this point forward is organized as follows. Section 2 introduces current approaches for web service composition testing. Section 3 presents TASSA tools. Section 4 shows experimental results from execution negative test cases over a sample BPEL process in TASSA framework. Finally, section 5 concludes the paper.

2. Related work

The BPEL inherently brings a challenge for testing due to its specific syntax, dynamic binding during execution and the fact that it integrates web services implemented by various providers. This section presents a review of various techniques, methods and tools that meet this challenge. The generation of the test suite for basis path testing of WS-BPEL and an accompanying tool that can be used by service testers are presented in [3]. The proposed testing tool does not support all XML schema data types in the generation of test data (only integer, float, boolean, and string are supported). Also, only sequence, condition, and repetition patterns of control are allowed. The tool does not consider infeasible paths that cannot be accessed. In [4] the authors propose a gray-box testing approach that has three key enablers: test-path exploration, trace analysis, and regression test selection. In order to improve the preciseness of the generated test paths IBM BPEL extensions, like Java snippets, need to be handled. The experimental results show that the test-generation time is linear to the number of test paths searched. Thus a more efficient generation algorithm is needed to avoid the performance problem for complex processes. In [5] a formal model for an abstract-based workflow framework that can be used to capture a composed web service under test is introduced. It is focusing on verifying, based on structural-based testing strategies that a composed web service can function correctly according to its semantic, activities and data dependencies. In [1] the authors use High-level Petri nets (HPNs) to model BPEL web service composition. The relationship between BPEL conceptions and HPNs is specified in four levels according to inter-service, intra-service, inter-activity, and intra-activity. In [6] a model-driven approach toward generating executable test cases for the given business process is presented. Its drawback is that the generated test cases still needs some effort to develop the adapter and codec to run. In [7] WSA is proposed to model concurrency, fault propagation, and interruption features of BPEL process. A model checking based test case generation framework for BPEL is implemented. An open issue is to prove the correctness of the model transformation. The approach in [10] is more applicable to programs without complex variable sharing or process interaction patterns. The messages' maximum enablement is limited to one time during the transformation of BPEL process into Extended Control Flow Graph XCFG. Also, the exception handling logic does not affect the other running threads, which run to undisturbed completion. An advantage of the approach is that it is modularized so that it can be used together with other testing technologies. It avoids the state space explosion problem and is applicable for programs in which concurrent computation units have only very few or no shared variables or other types of synchronization. In [8] an approach to unit testing of WS-BPEL and a tool prototype extending JUnit are presented. The proposed BPEL-Unit provides the following advantages: allow developers simulate partner processes easily, simplify test case writing, speed test case execution, and enable automatic regression testing. In [9] the authors propose a layer-based approach to creating frameworks for repeatable, white-box BPEL unit testing, which is applied to new testing framework. The framework does not provide much support in test case creation and the monitoring of the PUT. Devel-

Table 1. Comparison of BPEL testing approaches

Approach	EH	FH	A	ER	TCG	NT
Lertphumpanya [3]	no	no	yes	yes	yes	no
Li [4]	yes	yes	yes	yes	yes	no
Karam [5]	no	no	no	no	no	no
Yuan [6]	no	no	yes	yes	yes	no
Zheng [7]	yes	yes	yes	no	yes	yes
Li [8]	no	no	yes	yes	yes	no
Mayer [9]	no	no	yes	no	yes	yes
Dong [1]	no	no	yes	yes	yes	no
Yan [10]	yes	yes	no	yes	yes	no
Karam [5]	yes	yes	yes	yes	yes	yes

opers have to manually prepare large amount of coherent XML data and XPath expression to compose a test case. This is a painstaking task considering the complex structure of involved XML data. The results from comparison analysis of the current BPEL testing approaches are shown in Table 1. More detailed results can be found in [11].

The most of the authors propose to transform the BPEL process into intermediary model using CFG, HPN, etc. in order to find the executable paths of the process and generate test cases. Some of the approaches do not cover all BPEL activities during transformation. That is why the table has columns that show which approaches consider event handling (EH) and fault handling (FH) BPEL activities. The forth column of the table, called Automated (A), shows which approaches are implemented as tools or frameworks that are ready to use by testers. The next table column, named Experimental results (ER) indicates which of the approaches are proved via case studies, experimental results, etc. Only one of the approaches does not provide test case generation (TCG). This can be seen from column before the last one. And finally, the last column shows which of the approaches supports negative testing (NT).

3. TASSA tools

The TASSA framework consists of several tools that can be used jointly to achieve end-to-end testing of BPEL processes. The architecture of TASSA framework is presented on Figure 1.

In this section the cooperation of TASSA tools with the focus of *Fault Injection* and *Isolation tools* is presented. The main task of *Fault Injection tool*, called *faultInjector*, is to simulate faults during message exchange in order to generate negative test cases. The possible situations that are simulated are (1) overload of the communication channel that leads to delay of sending or receiving a message, (2) failure of the communication channel that leads to impossibility of sending or receiving a message, (3) noise in communication channel that leads to receiving a message with syntax and structure errors, and (4) wrong business logic of particular web service that leads to sending or receiving a message with syntax errors in its data. *faultInjector* takes as input a BPEL process under test, a list with failure parameters that describes the above situations and a string with values, which correspond to the arguments of the activity causing the failure. It returns a transformed BPEL process with simulated failure. The fault injection process consists of the following steps:

- identification of message exchanged when the failure is simulated,
- modification of communication channel, so that the failures expected by the tester occur,
- modification of an activity that corresponds to the message in order to send message to the proxy created between the message sender and receiver,
- serialization of input arguments of the real receiver (marshalling),
- invocation of the proxy,
- deserialization of output arguments and sending to the real receiver (unmarshalling).

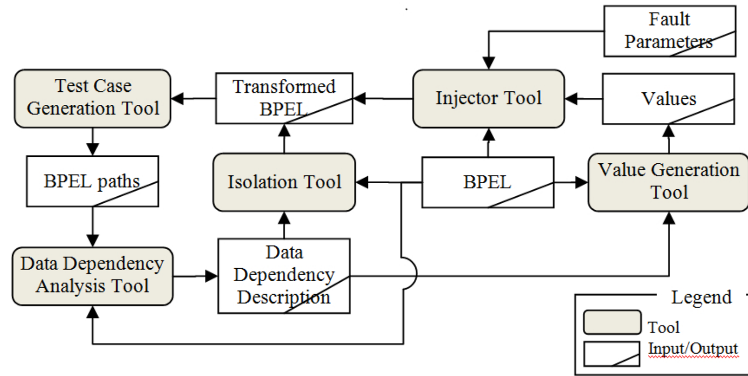

Figure 1. Architecture of TASSA framework

Similar steps are performed for the response of the invocation.

The formal representation of the process of marshalling and unmarshalling is as follows:

$o = invoke(i_1, i_2, ..., i_n)$

$o = Unmarshal(ProxyInvoke(Marshal(i_1, i_2, ..., i_n), R))$

where $i_1, i_2, ..., i_n$ are the real arguments of the modified Invoke activity, o is the original output data, *Marshal* and *Unmarshal* are the embedded BPEL functions for marshalling and unmarshalling, and *ProxyInvoke* is the call of the proxy with failure parameters specified by *R*.

The proposed approach is applicable only to invoke activities because their corresponding exchange of the messages is initiated by the BPEL process. It is necessary condition for the realization of the approach because activities for marshalling and unmarshalling need to be placed round the initiator of the message exchange.

The values passed to faultInjector are generated from a tool, called *Value Generation Tool* (VGT). Its goal is to generate valid values for all field of a given variable defined with XML Schema Definition (XSD). The main functionality of *VGT* is provided by a tool, called WS-TAXI, which is developed by a research team of Software Engineering Research Laboratory at the ISTI - Istituto di Scienza e Tecnologie dell'Informazione A.Faedo in Pisa. WS-TAXI generates compliant XML instances from a given XML Schema by using well-known Category Partition technique [12]. *VGT* takes as input a BPEL process under test and an array with identifiers of variables, whose values need to be generated.

The array of variables is produced by a tool, called *Data Dependency Analysis Tool* (DDAT) [2]. For a given path of the BPEL process *DDAT* finds all conditional activities along the path and specifies which variables affect those conditional activities. It receives as input a BPEL process and an array of unique identifiers of activities, describing the path that the BPEL process needs to follow. The path is generated by a tool, called Test Case Generation Tool (TCGT). TCGT solves two tasks. Its first task is to identify all paths of a given BPEL process in order to assist the tester in the process of test case generation. The second task of TCGT is to ensure management capabilities and storage for test cases.

The output of DDAT is also needed for a tool, named *Isolation Tool* (IsT). IsT provides temporary removal of BPEL process dependencies from one or more external web services. This allows the tester to control the web service returned results and pre-determine the possible routines in the BPEL process, as well as to continue testing even if a particular web service is missing. The BPEL process's dependency upon external services can be described as follows:

- synchronous execution of operation provided by an external service (Invoke activity in the BPEL process description),
- asynchronous execution of operation provided by an external Service (combination of Invoke and Receive activity in the BPEL process description),
- unforced message receipt from external service (Pick activity),

Table 2. Replacement of the BPEL process activities

Original Activity	Replacement Activity
Synchronous Invoke	Assign
Asynchronous Invoke	Empty
Receive	Assign
Reply	Empty
Pick/OnAlarm	Wait and OnAlarm branch
Pick/OnMessage	Assign and OnMessage branch
HumanTask	Invoke

- sending message to external service (resulting from an ingoing message),
- HumanTask activity, which requires human intervention and which affects the application through its output data (operator-entered values).

Invoke activity is modeled with following expression:

$o = f(i_1, i_2, ..., i_n, R)$

Herein the letter f denotes the functionality of the operation provided by the external service, $i_1, i_2, ..., i_n$ are the input parameters of the operation, o is the returned result, and R is additional parameters of the activity not directly related to the operation execution.

To eliminate the dependency upon f the following modifications are necessary to isolate the BPEL process from operation 1:

- Modification of the process, where the relevant Invoke activity is replaced with Assign activity to assign the output variable o specific values set by the user;
- When isolating the process from one activity there is created a test artifact (a variant of the BPEL process, in which the Invoke activity is replaced by an Assign activity).

The other dependencies are handled in a similar way, e.g. in the asynchronous mode for operation call (Invoke and Receive), the Invoke activity is replaced by the Empty activity (as it does not influence it) and Receive activity is replaced by Assign activity. Table 2 illustrates the mechanisms for isolation of the process from the different dependencies.

Through the cooperation of the above described tools the automation of the functional testing is largely achieved. Furthermore, lack of functionality for automation of testing in conditions of poor or unavailable communication channels with remote services is to a great extent overcome. Automation of negative testing is also supported.

4. Application of fault injection and isolation techniques in TASSA framework

This section presents the interoperability between faultInjector and IsT of TASSA framework. The tools are verified through testing of sample BPEL process, called Order Data Verifier Business Process (ODVBP). The process consists of four web services that are described in Table 3.

ODVBP uses the web services presented in Table 3 to validate the clients order data, namely email, credit card number and Zip code. It also retrieves the state abbreviation form Zip code and converts the total amount of the order into appropriate currency according to current rate.

Listing 1 shows an invoke activity, called CardValidatorInvoke, that is responsible for invocation of web service for validation of client credit card number.

```
<invoke name="CardValidatorInvoke"
    partnerLink="CardValidatorPartner"
    operation="Validate_CreditCard"
    xmlns:tns="http://www.Softwaremaker.Net/
    WebServices/" portType=
    "tns:ValidatorSoap"
    inputVariable="Validate_CreditCardIn"
    outputVariable="Validate_CreditCardOut">
</invoke>
```

Listing 2 shows transformation of the above activity after execution of faultInjector and IsT of TASSA framework.

```
<assign name="Assign1">
  <copy><from>
```

Table 3. Web services called from Order Data Verifier Business Process

Web service	Description
Email Validator	Validates email addresses for client applications
Credit Card Validator	Validated credit card number and type
Currency Convertor	Get conversion rate from one currency to another currency
Zip Code Validator	Validate Zip code and returns USA state abbreviation, latitude (decimal degrees) and longitude (decimal degrees)

```
      sxxf:doMarshal($Validate_CreditCardIn.parameters)
    </from><to>
      $ProxyInvokeOperationIn.operationIn/tassaP:part1
    </to></copy>
  <copy><from>
      'http://www.softwaremaker.net/webservices/
      swm/validator/validator.asmx?WSDL'
    </from><to>
      $ProxyInvokeOperationIn.operationIn/
      tassaP:endpoint
    </to></copy>
  <copy><from>20</from><to>
      $ProxyInvokeOperationIn.operationIn/tassaP:wait
    </to></copy>
  <copy><from>0</from><to>
      $ProxyInvokeOperationIn.operationIn/
      tassaP:errorsFactor
    </to></copy>
</assign>
<invoke xmlns:tns="http://www.rila.com/tassa/ProxyInvoke"
      inputVariable="ProxyInvokeOperationIn"
      name="ZipCodeInvoke"
      operation="ProxyInvokeOperation"
      outputVariable="ProxyInvokeOperationOut"
      partnerLink="PartnerLink1"
      portType="tns:ProxyInvokePortType"/>
<assign name="Assign2">
  <copy>
    <from>
      sxxf:doUnMarshal($ProxyInvokeOperationOut.part2)
    </from>
    <to part="parameters"
      variable="Validate_CreditCardOut"/>
  </copy>
</assign>
```

In order to generate transformed BPEL process *faultInjector* and *IsT* need a configuration information that describes the simulated failures as follows:

- Wait interval: an integer value that defines the delay of message seconds in seconds,
- Error factor: an integer value that that defines the kind of error will be injected (1–100: insert random errors in the data, which would possible break the XML structure; 0: usually used with Wait interval to delay the message; −1: replace the original values in the message; −2: interrupt the message),
- End point address: an end point address of the partner web service,
- Activity variables: input and output variables of the activity that will be injected.

As can be seen from Listing 1 and Listing 2, the Invoke activity, named CardValidatorInvoke, is enclosed with two additional Assign activities. The first Assign activity initializes the input parameters of ProxyInvoke operation of faultInjector. The parameters are as follows:

- Serialized input arguments of card validator operation of Credit Card Validator web service;
- End point address of the Credit Card Validator web service;
- Wait interval initialized with 20;
- Error factor initialized with 0.

The second Assign activity copies deserialized result from invocation of ProxyInvoke operation of faultInjector to the output variable of the Credit Card Validator web service. In addition, CardValidatorInvoke activity invokes ProxyInvoke operation instead actual Credit Card Validator web service.

faultInjector is validated through four test cases that correspond to its possibility of faults generation:

- Test Case 1: Message delay
- Test Case 2: Interruption
- Test Case 3: Noise in the message structure
- Test Case 4: Noise in the message data

To prove the fault injection against normal behavior of the process first the standard use case should be observed:

Table 4. Configuration data

Data	Description
wait=20	Error factor
error_ratio=0	Wait interval
http://www.softwaremaker.net/webservices/swm/validator/validator.asmx?WSDL	End point address
$Validate_CreditCardIn.parameters=$Validate_CreditCardOut.parameters	Activity variables

Table 5. Expected outputs from test cases

Test case	Description
Test case 0	Meaningful, well formed message that is executed in time interval t_i.
Test case 1	Meaningful, well formed message that is executed in time interval $t_i + T$, where T is the delay given as a failure parameter.
Test case 2	Error message, because of interruption
Test case 3	Error message, because of wrong structure
Test case 4	Well formed message with a random or invalid data

Table 6. Input data of Order Data Verifier Business Process

Test data	Test data values	Remark
Input1	<ord1:firstname>John</ord1:firstname> ... <ord1:currencycode>EUR</ord1:currencycode>	Valid data
Input2	<ord1:creditcardnumber>5374439468966228000 </ord1:creditcardnumber>	Invalid credit card number
Input3	<ord1:email>dessislava.g.petrovagmail.com</ord1:email>	Invalid email
Input4	<ord1:postalcode>070930</ord1:postalcode>	Invalid zip code

– Test Case 0: No fault injection (normal behavior)

The expected outputs from each test case are described in Table 5.

Test data are generated according to the XSD schema of ODVBP. They are presented in Table 6.

Table 7 presents the possible output results from execution of ODVBP.

Test case 0 includes tests representing normal behavior of the BPEL process. The tests correspond to the input data presented in Table 6, namely Input 1, Input 2, Input 3 and Input 4. The results form test executions are respectively Output 1, Output 2, Output 3 and Output 4. All tests are performed approximately for about 5.714 seconds.

The results from execution of the rest test case are presented in Table 8, Table 9, Table 10 and Table 11. As can be seen from the tables, all tests are passed, which means that the faultInjector successfully detects the faults injected in the business process.

In Test Case 1 (Message delay) the output data is the same as in Test Case 0 (Normal behavior), but the execution time is longer due to simulated delay. It is obvious that the execution time of Test Case1 differs from the execution time of Test Case 0 in the delay given as a failure parameter. For example, the Wait interval of the test for Validate_CreditCard operation is 10s. Therefore, the execution time of this test is equal to the execution time of corresponding test in Test Case 0 plus 10s.

During execution of Test Case 2 and Test Case 3, the BPEL process fails, due to impossibility of sending or receiving a message as well as corruption of message data. These faults are not handled by the sample BPEL process, so here the negative test cases catch a bug in the business logic, which is the main idea of testing BPEL processes against fault injections.

In Test Case 4 (Noise in the message data) faultInjector simulates wrong business logic of a web service by corruption of the received message with invalid data. The first test of this test case

Table 7. Output data of Order Data Verifier Business Process

Result data	Test data values
Output1	<ns0:firstname>John</ns0:firstname> ... <ns0:total>141.74</ns0:total> <ns0: currencycode>EUR</ns0:currencycode> <ns0:ordererrors>Validation is successful.</ns0:ordererrors>
Output2	<ns0: ordererrors >ERROR: Invalid Credit Card</ns0:ordererrors>
Output3	<ns0: ordererrors >ERROR: Invalid email</ns0:ordererrors>
Output4	<ns0: ordererrors >ERROR: Invalid Zip Code</ns0:ordererrors>
Output5	exMessage: disconnected
Output6	BPCOR-6130: Activity Name is CardValidatorInvoke Caused by: javax.xml.soap.SOAPException: javax.xml.stream ... Message: An invalid XML character was found in the element content of the document
Output7	BPCOR-6130: Activity Name is EmailInvoke Caused by: javax.xml.soap.SOAPException: javax.xml.stream ... Message: An invalid XML character was found in the element content of the document
Output8	BPCOR-6130: Activity Name is ZipCodeInvoke Caused by: javax.xml.soap.SOAPException: javax.xml.stream ... Message: An invalid XML character was found in the element content of the document

Table 8. Test Case 1 execution results

Wait interval	Error factor	Operation	Input	Output	Test result	Execution time (s)
wait=10	error_ratio=0	Validate_CreditCard	Input1	Output1	Passed	15.714s
wait=20	error_ratio=0	ValidateEmail	Input2	Output2	Passed	25.714s
wait=15	error_ratio=0	ConversionRate	Input3	Output3	Passed	20.714s
wait=20	error_ratio=0	ValidateZip	Input4	Output4	Passed	25.714s
wait=10, wait=10	error_ratio=0, error_ratio=0	ConversionRate, ValidateZip	Input1	Output1	Passed	25.714s
wait=10, wait=15	error_ratio=0, error_ratio=0	Validate_CreditCard, ValidateEmail	Input1	Output1	Passed	30.714s

Table 9. Test Case 2 execution results

Wait interval	Error factor	Operation	Input	Output	Test result	Execution time (s)
wait=0	error_ratio=-2	Validate_CreditCard	Input1	Output5	Passed	8.425s
wait=0	error_ratio=-2	ValidateEmail	Input2	Output5	Passed	8.145s
wait=0	error_ratio=-2	ConversionRate	Input3	Output5	Passed	8.412s
wait=0	error_ratio=-2	ValidateZip	Input4	Output5	Passed	8.345s

Table 10. Test Case 3 execution results

Wait interval	Error factor	Operation	Input	Output	Test result	Execution time (s)
wait=0	error_ratio=40	Validate_CreditCard	Input2	Output6	Passed	5.194s
wait=0	error_ratio=40	ValidateEmail	Input3	Output7	Passed	6.594s
wait=0	error_ratio=40	ValidateZip	Input4	Output8	Passed	7.194s

Table 11. Test Case 4 execution results

Wait interval	Error factor	Operation	Input	Output	Test result	Execution time (s)
wait=0	error_ratio=-1	ValidateEmail	Input2	Output1	Passed	4.086s
wait=0	error_ratio=-1	ConversionRate	Input2	Output2	Passed	6.411s
wait=0	error_ratio=-1	ValidateZip	Input2	Output2	Passed	4.014s

shows that this leads to wrong data values comparing with the expected ones in Test Case 0, or even to incorrect workflow.

The obtained results show that faultInjector is suitable for generation of different types of faults, which leads to the possibility of using TASSA framework for negative testing of BPEL processes.

5. Conclusion

This paper presents TASSA Framework, which offers approach for web service compositions testing by automating the testing process. Currently, it supports only design time testing, which is provided by five tools, composed as services itself, which integrated together offer to developers and service integrators the complete environment for functional testing of BPEL processes. Through integration of faultInjector tool in TASSA framework the negative testing is also achieved. This allows testing of BPEL processes in conditions of poor or unavailable communication channels with remote services. Furthermore, the testing process can be performed in isolation of external partner web services of the BPEL process under test due to possibility of Isolation Tool to remove dependency from them. The experimental results from validation of the testing approach through simple business process are presented.

Our future work will concentrate on developing complete testing methodology and validation of all TASSA framework tools over more complex BPEL processes.

Acknowledgment

The authors acknowledge the financial support by the National Scientific Fund, BMEY, grant agreement no. DO02–182 and SISTER project, funded by the European Commission in FP7–SP4 Capacities via agreement no. 205030.

References

[1] L. Dong, H. Yu, and Y. Zhang, “Testing BPEL-based web service composition using high-level Petri Nets,” in *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference*, 2006, pp. 441–444.

[2] I. Spassov, D. Petrova-Antonova, V. Pavlov, and S. Ilieva, “DDAT: Data dependency analysis tool for web service business processes,” in *Second International Workshop on Software Quality SQ*, June 2011, pp. 232–243.

[3] T. Lertphumpanya and T. Senivongse, “Basis path test suite and testing process for WS-BPEL,” *WSEAS Transactions on Computers*, Vol. Volume 7, No. 5, pp. 483–496, 2008.

[4] J. Li, H. Tan, H. Liu, J. Zhu, and N. Mitsumori, “Business-process-driven gray-box SOA testing,” *IBM Systems Journal*, Vol. Volume 47, pp. 457–472, 2008.

[5] M. Karam, H. Safa, and H. Artail, “An abstract workflow-based framework for testing composed web services,” in *International Conference on Computer Systems and Applications (AICCSA)*, 2007, pp. 901–908.

[6] Q. Yuan, J. Wu, C. Liu, and L. Zhang, “A model driven approach toward business process test case generation,” in *10th International Symposium on Web Site Evolution (WSE)*, 2008, pp. 41–44.

[7] Y. Zheng, J. Zhou, and P. Krause, “An automatic test case generation framework for web services. journal of software,” *Journal of Software*, Vol. Volume 2, No. 3, pp. 64–77, 2007.

[8] J. Li and W. Sun, “BPEL-Unit: JUnit for BPEL processes,” in *Service-Oriented Computing, ICSOC*, 2006, pp. 415–426.

[9] P. Mayer and D. Lubke, “Towards a BPEL unit testing framework,” in *Proceedings of the workshop on Testing, analysis, and verification of web services and applications*, 2006, pp. 33–42.

[10] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang, “Bpel4ws unit testing: Test case generation using a concurrent path analysis approach,” in *Proc.*

of ISSRE. IEEE Computer Society, 2006, pp. 75–84.

[11] D. Petrova-Antonova, I. Krasteva, and S. Ilieva, "Approaches facilitating WS-BPEL testing," in *17th Conference on European Systems and Software Process Improvement and Innovation*, September 2010, pp. 5.1–5.17.

[12] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, "WS-TAXI: A WSDL-based testing tool for web services," in *International Conference on Software Testing Verification and Validation*, 2009, pp. 326–335.