Enabling Proactive Adaptation through Just-in-time Testing of Conversational Services

Dimitris Dranidis[1], Andreas Metzger[2] and Dimitrios Kourtesis[1]

[1] South East European Research Centre (SEERC)
Research Centre of the University of Sheffield and CITY College
Thessaloniki, Greece
dranidis@city.academic.gr, dkourtesis@seerc.org

[2] Paluno (The Ruhr Institute for Software Technology)
University of Duisburg-Essen, Essen, Germany
andreas.metzger@sse.uni-due.de

Abstract. Service-based applications (SBAs) will increasingly be composed of third-party services available over the Internet. Reacting to failures of those third-party services by dynamically adapting the SBAs will become a key enabler for ensuring reliability. Determining when to adapt an SBA is especially challenging in the presence of conversational (aka. stateful) services. A conversational service might fail in the middle of an invocation sequence, in which case adapting the SBA might be costly; e.g., due to the necessary state transfer to an alternative service. In this paper we propose just-in-time testing of conversational services as a novel approach to detect potential problems and to proactively trigger adaptations, thereby preventing costly compensation activities. The approach is based on a framework for online testing and a formal test-generation method which guarantees functional correctness for conversational services. The applicability of the approach is discussed with respect to its underlying assumptions and its performance. The benefits of the approach are demonstrated using a realistic example.

Keywords: service testing, online testing, stateful services, test generation, proactive adaptation

1 Introduction

Service-based applications (SBAs) are increasingly composed of third party services available over the Internet [22]. Third party services are owned and controlled by organizations different from the service consumers and thus can change or evolve in ways not anticipated by the service consumer. This means that even if third-party services have shown to work during design-time, they need to be (re-)checked during the operation of the SBA to detect failures. Such failures then should trigger the adaptation of an SBA to ensure that it maintains its expected functionality and quality.

1.1 Problem Statement

Determining when to trigger an adaptation is especially challenging if conversational services are employed in a service composition. A conversational service is one that only accepts specific sequences of operation invocations. This is because some operations may have preconditions which depend on the state of the service. The set of all acceptable invocation sequences is called the *protocol* (or choreography) [10] of the service. An example is the shopping basket of an online store. The shopping basket is initialized first. Then, some items are added to the basket, some might be removed, until a checkout is performed.

The first invocation of a conversational service can work as expected. However, the invocation of that service at a later stage of the interaction sequence could fail due to the service not conforming to the expected protocol. Adapting the SBA to react to such a failure could be very costly: First, the state of the conversational service (e.g., the items in the shopping basket) might need to be transferred to an alternative service. Second, compensation actions might need to be initiated (e.g., if the items have been marked as reserved in a warehouse service, those reservations need to be revoked).

1.2 Contributions of the Paper

This paper introduces an automated technique to determine when to proactively trigger adaptations in the presence of conversational services, thus avoiding costly compensation actions. The proposed technique builds on previous work on testing and monitoring of conversational services [9, 10] and on online testing for proactive adaptation [13, 19].

We advocate performing *just-in-time* online tests of the relevant operation sequences of the constituent services of the SBA. Here, just-in-time means that "shortly" before a conversational service is invoked for the first time within the service composition, the service is tested to detect potential deviations from the specified protocol.

The generation of the test cases is grounded in the formal theory of Stream X-machines (SXMs). SXMs have been utilised [9] for the automated generation of test cases, which, under well defined conditions, guarantee to reveal all inconsistencies among the implementation of a service and its expected behaviour.

To ensure that just-in-time testing can be done with feasible cost and effort, as well as in reasonable time, we propose a new way of reducing the number of test cases, such that we can still guarantee that the conversational service behaves as expected in the context of the concrete SBA. In addition, we propose executing most of the "costly" test preparation activities during deployment time.

The remainder of the paper is structured as follows. In Sect. 2, we discuss related work. Sect. 3 introduces a running example to illustrate our technique for just-in-time online testing. Sect. 4 provides an introduction to the formal underpinnings as well as the available tool support. Based on these foundations, Sect. 5 describes our technique, which is then critically discussed in Sect. 6, focusing on its applicability in realistic settings.

2 Related Work

Various approaches have been introduced in the literature to determine when to trigger the adaptation of SBAs. Typically, in such a setting, monitoring is used to identify failures of the constituent services (see [2] for a comprehensive survey) by observing SBAs during their actual use and operation [5]. However, monitoring only allows for a reactive approach to adaptation (cf. [13]), i.e., the application is modified *after* a failure has been monitored. To support pro-active adaptation, prediction of the future functionality and quality of the SBA is needed. Initial solutions are available for predicting the violation of SLAs (e.g., [18]). However, the focus of that work is on quality attributes and not on protocol conformance.

In our previous work, we have introduced online testing as an approach to predict the future quality of the SBA and thus to trigger pro-active adaptation [13, 19]. Online testing means that the constituent services of an SBA are systematically fed with test inputs in parallel to the normal use and operation of the SBA. Although our approach [19]—different from other uses of online testing (cf. [23, 6, 3, 1])—has shown how to employ online testing for triggering pro-active adaptation, it has only focused on stateless services and violations of quality contracts (such as performance). Thus, those techniques, are not yet capable of addressing the challenges introduced by predicting whether conversational services will violate their functional contracts in the context of an SBA.

Our proposed online testing approach also differs from existing testing and monitoring techniques for services (see [20] for a comprehensive survey). Those techniques only perform tests before or during deployment, whereas after deployment, they resort to monitoring to assess the quality (e.g., see [12] where an approach for protocol testing and monitoring of services is introduced).

3 The e-Shop Example

The example presented in this paper concerns an e-Shop SBA which is composed of three services: CartService, ShipmentService and PaymentService. Clients select the items they wish to purchase, then provide the shipment details, and finally pay. The composition has the following workflow (i.e., main flow of service invocations):

- An order is created by adding items to the order via the CartService.
- Using the ShipmentService, a shipment is created given the origin, destination, date and weight of the order; the shipment rate is calculated based on delivery type.
- The customer is charged the total cost by the PaymentService.

There are many alternative third-party services which could provide the ShipmentService, such as UPS, FedEx, and DHL. There are also alternative third-party services which could provide the PaymentService, such as Paypal, and VISA payment. Therefore, we consider this composition as a realistic example for demonstrating our approach for just-in-time testing for proactive adaptation.

Since we will use the ShipmentService as an example for how to perform just-in-time online testing, we describe that service and its operations in more detail. Table 1 lists the complete interface of the service.

Table 1. The interface of the ShipmentService

Operations	**Inputs**	**Outputs**
create	origin, destination, shipmentDate, weight	-
getShipmentRate	shipmentType	rate
oneStepShipment	shipmentType	-
cancel	-	-
confirm	-	-
getConfirmedRate	-	rate

The ShipmentService is initiated by calling the `create` operation with the origin and the destination of the delivery, the date of the shipment and the total weight of the package as arguments. The shipment cost is calculated for different types of delivery (e.g. normal or express) via the `getShipmentRate` operation. The `confirm` operation finalizes the shipment. Once confirmed, the operation `getConfirmedRate` is used instead of the `getShipmentRate` operation for getting informed about the cost of the shipment. The shipment can be canceled with the `cancel` operation, unless it has been confirmed. The ShipmentService provides also the `oneStepShipment` operation, which does not expect a confirmation from the client.

The ShipmentService is a typical example of a conversational service in which only specific sequences of operation invocations (protocol) are allowed. The following restrictions specify the protocol:

– initially no operation can be called except the `create` operation;
– `confirm` can only be called after at least one `getShipmentRate` call;
– `getShipmentRate` can be called many times to get different rates for different types of delivery but not after `confirm` or `oneStepShipment`;
– `getConfirmedRate` can only be called after `confirm` or `oneStepShipment`;
– `cancel` cannot be called after `confirm`, nor after `oneStepShipment`;
– no operation can be called after `cancel`.

4 Fundamentals

4.1 Stream X-Machines

A Stream X-Machine (SXM) [17, 14] is a computational model capable of modeling both the data and the control of a system. An SXM is like a finite state machines but with two important differences: (a) the machine has some internal

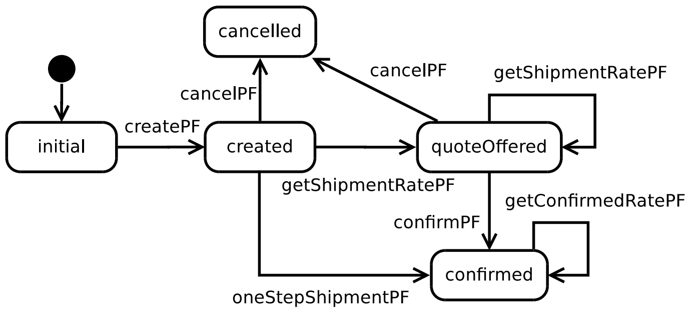

Fig. 1. State transition diagram of the conversational interface of ShipmentService

store, called *memory*; and (b) the transition labels are not simple symbols, but *processing functions* that represent elementary operations that the machine can perform. A processing function reads inputs and produces outputs while it may also change the value of the memory.

An *SXM* [14] is defined as the tuple $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ where:

- Σ is the finite *input alphabet* and Γ is the finite *output alphabet*;
- Q is the finite set of *states*;
- M is a (possibly) infinite set called *memory*;
- Φ is a finite set of partial functions ϕ (called *processing functions*) that map input-memory pairs to output-memory pairs, $\phi : \Sigma \times M \rightarrow \Gamma \times M$;
- F is the next state partial function that given a state and a processing function it provides the next state, $F : Q \times \Phi \rightarrow Q$;
- q_0 and m_0 are the *initial state* and *initial memory* respectively.

Parallels can be drawn between an SXM and a stateful Web service: SXM inputs correspond to SOAP request messages, outputs correspond to SOAP response messages, and processing functions correspond to Web service operation invocations [10, 9].

Example. The conversational behavior of the ShipmentService (see Sect. 3) can be modelled with an SXM. The diagram in Fig. 1 shows the state transition diagram of the SXM (i.e., the next state function F). Each processing function (e.g., cancelPF) is triggered by the corresponding operation (e.g., `cancel`). The memory of the SXM consists of a single numeric variable: the cost of the shipment.

4.2 Test Case Generation

SXMs have the significant advantage of offering a testing method [16, 15, 14] that ensures the conformance of an implementation under test (IUT) to a specification. The production of a finite *test set* is based on a generalization of the W-method [4]. It is proved that only if the specification and the IUT are behaviorally equivalent, the test set produces identical results when applied to both of them. The testing method rests on certain assumptions and conditions which will be explained and discussed in Sect. 6.

The first step of the method consists of applying the W-method on the associated finite automaton $A = (\Phi, Q, F, q_0)$ of the SXM and produces a set $X \subseteq \Phi^*$ of sequences of processing functions:

$$X = S\left(\Phi^{k+1} \cup \Phi^k \cup \ldots \cup \Phi \cup \{\epsilon\}\right) W$$

where W is a *characterization set* and S a *state cover* of A and k is the estimated difference of states between the IUT and the specification. A characterization set is a set of sequences for which any two distinct states of the automaton are distinguishable and a state cover is a set of sequences such that all states are reachable from the initial state.

Finally, the test set $T \subseteq \Sigma^*$ is constructed by converting each sequence of processing functions in X to a sequence of inputs. For this purpose, appropriate input data sequences need to be generated, that trigger the corresponding processing function sequences. The input sequences and the corresponding output sequences produced by the model animation (the model acts as an oracle) provide the test cases that guarantee the equivalence of the IUT to the specification.

Due to space limitations, more information about the test generation method, its application on the specific example, and the generated test cases can be found online at [11].

4.3 Tool Support

A tool suite for the specification of SXMs and automated test case generation [8] is available in Java (JSXM). The JSXM language for modeling SXMs is based on XML and Java inline code. The XML-based specifications in JSXM facilitate easier integration with Web technologies and related XML-based Web service standards. JSXM supports animation of SXM models, model-based test case generation and test transformation. The test case generation implements the testing method introduced in Sect. 4.2 and generates a set of test cases in XML. As such, the test cases are independent of the programming language of the IUT. Test transformation is then used for transforming the XML test cases to concrete test cases in the underlying technology of the IUT. Currently, both a Java test transformer and a Web Service test transformer are available, which automatically generate JUnit and XMLUnit test cases respectively.

Example. Due to space limitations, the corresponding code for representing the SXM of the example in JSXM can be found online at [11].

5 Just-in-time Online Testing

As motivated in Sect. 1.2, we want to check that all constituent services will behave according to their contract before they are invoked in the context of the SBA, thus increasing confidence that the execution will not be aborted due to a

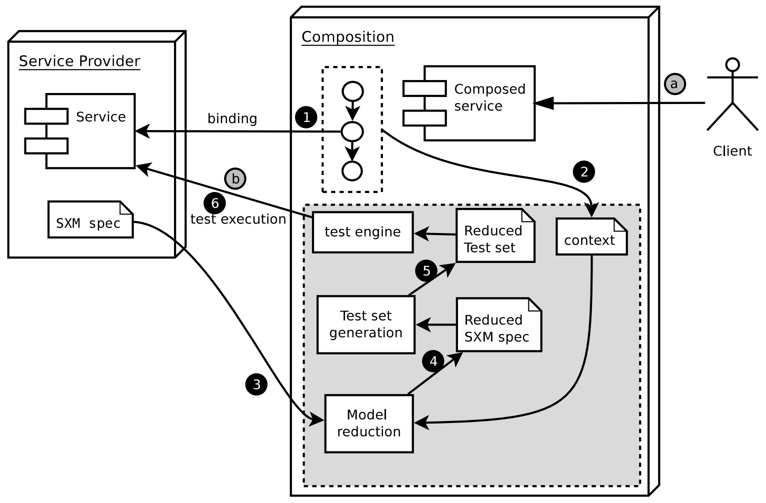

Fig. 2. Just-in-time online testing during composition deployment and execution.

failure at a later point. We achieve this by means of just-in-time online testing, which means that the protocol of the service is tested “shortly” before the service is actually invoked from the service composition (the SBA).

Example. In our running example, the execution of the composition begins with invoking the CartService. This leaves some time[3] for on-line testing of the ShipmentService and the PaymentService. In the case, for example, that the ShipmentSevice is not functionally correct, an adaptation will be triggered. A possible adaptation might suggest to choose a ShipmentService from an alternative provider.

5.1 Test Generation during Composition Deployment

Test generation is a time-consuming process which could slow down the execution of the service composition if tests were generated during the operation of the SBA. Therefore, we propose performing the test case generation during the design or deployment of the service composition.

During the definition of the business process (or workflow) and the deployment of the service composition, specific service providers for each constituent service are chosen and bound to the composition (step 1 in Fig. 2).

The provider provides both the service implementation (possibly as a Web service) and the service specification (as an SXM model) and claims that the implementation conforms to the specification.[4] Applying the test generation

[3] This is especially feasible when a user takes some time to enter the items via a GUI.
[4] A method to derive an SXM specification from a semantically annotated Web service utilizing ontology-based and rule-based descriptions is proposed in [21].

method to the SXM model generates a complete test set for the whole protocol. This test set will thus also include invocations of services not used by the current composition. Testing these operations is unnecessary and increases the testing time during test execution.

Determining the Composition Context. The *context of the service w.r.t the composition* (shortly called the *composition context*) is defined as the subset of all operations of the constituent service which are invoked by the composed service (step 2 in Fig. 2).

Example. If the composition uses only the 2-step shipment facility and moreover without ever using the option to cancel a shipment, then the operations `oneStepShipment` and `cancel` do not belong to the composition context, which in that case consists of the following operations: `create`, `getShipmentRate`, `confirm`, and `getConfirmedRate`.

Reducing the Model. To improve the efficiency of on-line testing our technique reduces the number of test-cases to those which will guarantee the correctness of the behavior in the current composition context. To this end, the original SXM model is retrieved from the provider (step 3 in Fig. 2) and it is reduced to an SXM model for the composition context (step 4 in Fig. 2).[5]

The reduced model $(\Sigma_r, \Gamma_r, Q_r, M, \Phi_r, F_r, q_0, m_0)$ is constructed from the original model $(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ as follows:

1. Σ_r : Inputs from the input alphabet Σ which correspond to service operations not in the composition context are removed;
2. Φ'_r : Processing functions triggered by removed inputs are removed from Φ;
3. F'_r : Transitions labeled with removed functions are removed from F;
4. Q_r : States which are not reachable from the initial state via the transitions in F'_r are removed from Q;
5. F_r : Transitions starting from a removed state or ending to a removed state are removed from F'_r;
6. Φ_r : Processing functions which do not label any remaining transitions in F_r are removed from Φ'_r;
7. Γ_r : Outputs which are not the result of any remaining processing functions in Φ_r are removed from Γ.

M, q_0, and m_0 remain the same.

Example. Fig. 3 illustrates the state transition diagram of the reduced SXM. Removed states, transitions, and processing functions are shown in gray.

[5] Although filtering the set of test cases (generated from the original SXM model) would be possible in theory, this leads to practical problems if the state cover and characterization sets contain operations which are not part of the operational context (as in such a situation, relevant test cases might be filtered out).

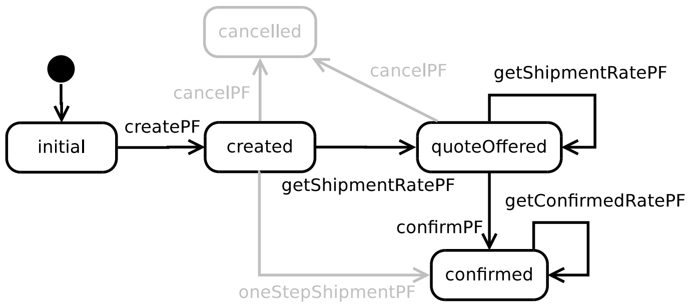

Fig. 3. State transition diagram of the reduced SXM for the composition context

Generating the Test Cases. The test generation method (Sect. 4.2) is applied to the reduced SXM resulting to a reduced test set (step 5 in Fig. 2). For the test generation, both the state cover and the characterization sets need to be calculated for the reduced SXM, as, in the general case, they will be different from the respective sets of the original model.

Example. Test sets for the complete and the reduced SXMs for the ShipmentService example can be found online at [11].

Testing during Deployment. The final step during deployment involves testing the service (step 6 in Fig. 2). This ensures that the service behaves according to its specification, at least concerning the part of its interface which belongs to the composition context and thus ensures that a “correct” SBA is being deployed. However, as motivated in Sect. 1, even if a service was shown to work during design-time, it needs to be (re-)checked during the operation of the SBA to detect deviations (especially if such a service is provided by a third party).

5.2 Just-in-time Testing after Deployment

After deployment, when the client starts executing the composition (step a in Fig. 2), the testing engine is invoked to perform online tests.

To this end, the testing engine will read the stored, reduced test set and execute the test cases (step b in Fig. 2) until all the tests have passed or one of them has failed. In the latter case an adaptation is triggered.

6 Critical Discussion

This section reflects on the applicability of our proposed technique in realistic settings by discussing its underlying assumptions, as well as scalability and efficiency.

6.1 Assumptions of the Testing Approach

As mentioned in Sect. 4.2, the SXM test case generation relies on certain assumptions to guarantee its results. Specifically, the testing method assumes that the implementation of the individual processing functions is correct, and, based on this assumption, ensures that the system correctly implements their integration. As suggested in [16] this assumption could be checked in practice through a separate quality assurance process (e.g., using traditional black-box testing techniques or formal verification).

Additionally, the testing method relies on the following "design for test" conditions [16]: *controllability* and *observability.* Controllability (also called input-completeness) requires that any processing function can be exercised from any memory value using some input. Observability requires that any two different processing functions will produce different outputs if applied on the same memory/input pair.

In our case observability can be achieved by having each operation providing a distinct response (which is usually the case with SOAP Web services).

Controllability depends on the model. The ShipmentService used as an example in this paper is an input-complete (thus controllable) model. Furthermore, the proposed model reduction preserves completeness, since processing functions are not modified but merely removed from the model. If the initial processing functions were input-complete, then the remaining functions are also input-complete.

Moreover, a fundamental premise for the just-in-time online testing approach presented here—as well as for service testing in general— is that service providers offer some practical way for their services to be tested without triggering real world side effects, such as shipping items or charging credit cards.

There are several ways that this can be realised from a technical point of view. In general, it is accomplished by allowing services to be executed in a special testing environment or configuration mode, often called a "sandbox", which allows the functionality of services to be fully exercised in isolation from the real production environment and databases. The sandbox environments of Paypal and Amazon Web Services are typical examples.

A requirement that logically follows, if we are to increase our confidence for some service which is tested under such a "non-production" mode, is that the special test-mode instance of the service which is actually executed during testing is identical to the actual production instance that will be invoked shortly after.

6.2 Scalability and Efficiency

Table 2 shows the sizes and the lengths of the test set for different values of k, both for the complete and the reduced model. The actual numbers are much smaller than the (theoretically estimated) maximums.[6]

[6] Based on [16], the maximum number of test sequences, i.e., $card(X)$, is less than $n^2 \cdot r^{k+2}/(r-1)$, where $n = card(Q)$, $r = card(\Phi)$, and (continued on next page)

The complete SXM models a service with an interface of 6 operations whereas the reduced SXM models an interface of 4 operations. The time savings from the reduction range between 55–60%.

As it has been discussed above, even if only the reduced set of test cases is executed, testing still takes some time to complete and thus might take longer than the time that is available before the conversational service is invoked for the first time.[7]

A possible solution to this problem is to perform some kind of gradual testing: Online testing can begin with the execution of test cases for $k = 0$. If there is still time available execution of test cases for $k = 1$ can begin, and so on. This gradually increases the confidence that the implementation is correct.

In the worst case scenario there might be not enough time even for the execution of the $k = 0$ test set. In that case there are two possibilities: (1) delay the service execution in order to complete testing, or (2) stop the testing and start service execution. Those choices could be offered in a framework configuration and left to the designer of the composition to take the decision.

To take a more informed decision, the framework could provide some measurements from previous test executions. During test at deployment, gradual testing may be performed and the required times for the test execution for different values of k could be measured and stored. These values could be used for deciding how the framework should behave in cases there is not sufficient time to test. For example, the designer of the composition could decide that even when there is not enough time, tests for $k = 0$ should always be executed first before service execution.

In addition to time, testing costs can become a critical factor for the applicability of our technique, as the invocation of third-party services can be costly. Furthermore, intense online testing can have an adverse effect on the provisioning of the services, as testing might reduce the resources available on the provider's

(continued from previous page) k the estimated number of extra states in the implementation. The total length l of the test set is less than $card(X) \cdot n'$, where $n' = k+n$. Actual numbers are much lower than these maximums, mainly due to the fact that JSXM removes all those sequences which are prefixes of other sequences in the test set.

[7] Indicatively, it took about 0.5 seconds to execute the test set of length 98 to a Web service implementation of the ShipmentShervice in a LAN.

Table 2. The sizes and the lengths of the test set for the complete and reduced model for different k values

card(X), l	k									
	0		**1**		**2**		**3**		**4**	
complete SXM	38	122	61	234	89	398	122	624	160	922
reduced SXM	17	54	25	98	36	168	50	270	67	410
savings %	55	56	59	58	60	58	59	57	58	56

side and thus could, for instance, impact on performance (cf. [7]). This means that the number of online test cases that can be executed is limited by economic and technical considerations. Especially if more than one SBA instance is running, applying our technique to each of those SBA instances individually can lead to redundant online tests, which should be avoided. One solution would be to use a "central" component in our framework that governs the online testing activities and, for instance, does not execute a test if an identical test has already been performed (provided that it can be assumed that its outcomes are still representative of the current situation).

7 Conclusions

This paper introduced a novel technique for just-in-time online testing of conversational services, which enables proactively triggering adaptations of service-based applications. Such a technique is especially relevant in the setting of the "Internet of Services", where applications will increasingly be composed from third party services, which are not under the control of the service consumer and thus require that they are (re-)checked during the operation of the SBA to detect failures.

We are currently integrating the presented technique into the PROSA framework (PRO-active Self-Adaptation [13]) in the context of the S-Cube project. Although, possible adaptation strategies have not been in the focus of this paper, they can be considered to further extend our proposed techniques. For example, if an alternative service is chosen to replace a faulty one, the alternative service may be (pre-)tested to ensure it provides the same functionality before the actual adaptation will be performed.

Acknowledgments. We cordially thank the anonymous reviewers for their constructive and detailed comments on improving the paper.

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube). For further information please visit http://www.s-cube-network.eu/.

References

1. Bai, X., Chen, Y., Shao, Z.: Adaptive web services testing. In: 31st Annual Int'l Comp. Software and Applications Conf. (COMPSAC). pp. 233–236 (2007)
2. Benbernou, S.: State of the art report, gap analysis of knowledge on principles, techniques and methodologies for monitoring and adaptation of SBAs. Deliverable PO-JRA-1.2.1, S-Cube Consortium (July 2008), http://www.s-cube-network.eu/results/
3. Chan, W., Cheung, S., Leung, K.: A metamorphic testing approach for online testing of service-oriented software applications. International Journal of Web Services Research 4(2), 61–81 (2007)
4. Chow, T.S.: Testing software design modelled by finite state machines. IEEE Transactions on Software Engineering 4, 178–187 (1978)

5. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. 30(12), 859–872 (2004)
6. Deussen, P., Din, G., Schieferdecker, I.: A TTCN-3 based online test and validation platform for Internet services. In: Proceedings of the 6th International Symposium on Autonomous Decentralized Systems (ISADS). pp. 177–184 (2003)
7. Di Penta, M., Bruno, M., Esposito, G., et al.: Web Services Regression Testing. In: Baresi, L., Di Nitto, E. (eds.) Test and Analysis of Web Services, pp. 205 – 234. Springer (2007)
8. Dranidis, D.: JSXM: A suite of tools for model-based automated test generation: User manual. Tech. Rep. WPCS01-09, CITY College (2009)
9. Dranidis, D., Kourtesis, D., Ramollari, E.: Formal verification of web service behavioural conformance through testing. Annals of Mathematics, Computing & Teleinformatics 1(5), 36–43 (2007)
10. Dranidis, D., Ramollari, E., Kourtesis, D.: Run-time verification of behavioural conformance for conversational web services. In: Seventh IEEE European Conference on Web Services. pp. 139–147. IEEE (2009)
11. Dranidis, D., Metzger, A., Kourtesis, D.: Enabling proactive adaptation through just-in-time testing of conversational services (supplementary material). Tech. rep., S-Cube (2010), http://www.s-cube-network.eu/results/techreport/sw2010
12. Hallé, S., Bultan, T., Hughes, G., Alkhalaf, M., Villemaire, R.: Runtime verification of web service interface contracts. IEEE Computer 43(3), 59–66 (2010)
13. Hielscher, J., Kazhamiakin, R., Metzger, A., Pistore, M.: A framework for proactive self-adaptation of service-based applications based on online testing. In: Service-Wave 2008. No. 5377 in LNCS, Springer (10-13 December 2008)
14. Holcombe, M., Ipate, F.: Correct Systems: Building Business Process Solutions. Springer Verlag, Berlin (1998)
15. Ipate, F.: Theory of X-machines with Applications in Specification and Testing. Ph.D. thesis, University of Sheffield (1995)
16. Ipate, F., Holcombe, M.: An integration testing method that is proven to find all faults. International Journal of Computer Mathematics 63, 159–178 (1997)
17. Laycock, G.: The Theory and Practice of Specification Based Testing. Ph.D. thesis, University of Sheffield (1992)
18. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: Monitoring, prediction and prevention of SLA violations in composite services. In: IEEE International Conference on Web Services (ICWS) Industry and Applications Track (2010)
19. Metzger, A., Sammodi, O., Pohl, K., Rzepka, M.: Towards pro-active adaptation with confidence augmenting service monitoring with online testing. In: Proceedings of the ICSE 2010 Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS '10). Cape Town, South Africa (2-8 May 2010)
20. Pernici, B., Metzger, A.: Survey of quality related aspects relevant for service-based applications. Deliverable PO-JRA-1.3.1, S-Cube Consortium (July 2008), http://www.s-cube-network.eu/results/
21. Ramollari, E., Kourtesis, D., Dranidis, D., Simons, A.: Leveraging semantic web service descriptions for validation by automated functional testing. The Semantic Web: Research and Applications pp. 593–607 (2009)
22. Tselentis, G., Domingue, J., Galis, A., Gavras, A., Hausheer, D.: Towards the Future Internet: A European Research Perspective. IOS Press, Amsterdam, The Netherlands (2009)
23. Wang, Q., Quan, L., Ying, F.: Online testing of Web-based applications. In: Proceedings of the 28th Annual Int'l Comp. Software and Applications Conference (COMPSAC). pp. 166–169 (2004)