

Testing Service Composition

Antonio Bucchiarone¹, Hernán Melgratti¹, and Francesco Severoni²

¹ IMT Lucca, Italy

[a.bucchiarone,h.melgratti}@imtlucca.it

² Dipartimento di Informatica, Università di L'Aquila

f.severoni@di.univaq.it

Abstract. Service Oriented Computing is aimed at providing the bases for building software by assembling independent, loosely coupled services. Industry has given birth to several standards for specifying and programming such kind of composite services. As any software development activity, also building a composite service requires strategies for performing quality assessment of applications. One activity that is traditionally used for assuring the quality of software is testing. In this paper, we analyse the main alternatives for testing compositions (either in the form of choreographies or orchestrations), and survey current proposal for doing it.

1 Introduction

Service-Oriented Computing is an emerging paradigm where services are understood as autonomous platform-independent computational entities that can be described, published, categorized, discovered, and dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications. These characteristics pushed service-oriented computing toward nowadays widespread success, demonstrated by the fact that many large companies invested a lot of effort and resources to promote service delivery on a variety of computing platforms (mostly through the Internet in the form of Web Services). The rapid evolution of (Web) service technologies is leading services to play a central role in the software development process. For these reasons Service Oriented Architectures are complementing Software Architectures as a main software model that allows to represent and validate functional and non-functional properties of the system.

The ultimate goal of *Service Oriented Architecture* (SOA) is to enable the old end of developing new components and applications (now services) just by assembling existing ones. Hence, many recent efforts, mainly coming from the industry and mostly oriented to web services, have given birth to several (proposals for) programming/description languages tailored to the specification of web service integration, generally known as *web service composition languages* (WSCL), like BPML [5], XLANG [17], WSFL [13], WS-BPEL [4], WS-CDL [21], and WSCI [22].

While much focus has been recently spent on services and service composition specification, only recently the focus is being oriented to service and service

composition validation and verification. The ability to correctly assemble new systems based on existing services strongly relies on the quality of each service and on the quality of the assembly (i.e., how services are composed). Testing techniques have been recently presented in the domain of SOA. Similarly to component-based testing, new strategies are being devised for testing services in isolation and service composition.

Goal of this research paper is to analyze existing research work on service-oriented testing, so to highlight different alternatives for testing service oriented applications that have been described using standard web service composition languages. In such a languages, services can be composed by following the two complementary views of choreography and orchestration. Thus, service-oriented testing needs to be distinct into choreography-based and orchestration-based testing. Moreover, since both the units (i.e., the services) and the assembly (i.e., the composed services) can be subject to malfunctions, both unit and integration testing must be taken into consideration.

The following of this paper is organized so to present related work on service-oriented testing (Section 5). Based on the state-of-the art in service-oriented testing, Section 4 discusses alternative ways of testing choreographies while Section 3 discusses orchestration-based testing. Conclusions and future work are provided in Section 6.

2 Orchestration and Choreography

Web services composition languages address aggregation by following two complementary views: *Orchestration* and *Choreography*. The *orchestration view* focuses on the description of the computation carried on by a single partner of a composite service. In this way, an orchestration exposes the internal logic of a single component by specifying the control-flow structure and the data flow dependencies of that particular component. From the point of view of an orchestration, the partner that is being specified plays the role of the coordinator of the whole composition, while the remaining services are merely components agnostic to the fact that they are taking part in a larger process. Consequently, the specification of the composite service or, in other words, the coordinator of the composition, which is called the *orchestrator*, states the order (i.e., the flow) in which component services are invoked. Hence, the basic primitives for writing orchestrations concern mainly to the handling of the interaction with other services. In particular, they allow for (i) *invoking* services (like a traditional call or method invocation), (ii) *receiving* an answer (like processing a return), (iii) *accepting* an invocation (like an accept statement), and (iv) *answering* an invocation (like performing a return). The flow of such interactions can be specified either:

- in a *structured way*: by combining flows with the usual operators of sequence, iteration and branching (as in any procedural language), and the possibility of defining concurrent flows (like in concurrent languages).

- in an *unstructured way*: by specifying a partial order as the pair-wise dependencies among basic activities (i.e., by writing links)

Typical orchestration languages are XLANG, WSFL, and (executable processes of) WS-BPEL.

The *choreography view* is aimed at exposing the whole flow of interaction among all parties involved in the composition. In this context, all participants are aware of the fact that they are taking part of composition and, thus, of the way they should interact with each other. For this reason, choreography languages allow for the definition of protocols that parties should follow in order to achieve a common goal. There are two main approaches to define choreographies:

- The *global model*, in which a protocol describes from a global perspective the messages exchanged by all parties, and
- The *interaction model* in which each service describes the temporal and logical dependencies among the messages it exchanges, i.e., a kind of interface definition.

WS-CDL adopts the global model style, while WSCI and abstract processes of WS-BPEL are instances of the interaction model.

Example 1. Consider a *senior citizen* that is subscribed to a *health service* (HS), which helps her in monitoring a particular disease, e.g., diabetes. The patient measures at home her blood sugar level, her blood pressure and her weight, and then she sends such data to the HS. Once the HS received the data, it controls with the *subscription service* (SS) whether the sender is a subscribed patient, and then it forwards the received data to a *team of doctors* that are in charge of checking the received values. After analysing the received values, the team sends a confirmation or an adjustment of the medication doses. In addition, in sever situations, the team may contact the *emergency support* to arrange an emergency procedure.

The services involved in this scenario are listed below:

Patient (P): (i) subscribes to the HS and (ii) sends the data about their health condition. In particular, they send data about blood sugar (BS), blood pressure (BP), and weight (W).

Subscription service (SS): responsible for handling the subscriptions of users.

Health Service (HS): It receives data from patients, and processes those corresponding to subscribed users. Any time the HS receives data from a subscribed patient, it forwards them to the team of doctors.

Accounting Service (AS): It is in charge of keeping track of the frequency in which the service HS is used by patients. So, HS forwards the information about who contacts it and when.

Team of Doctors (T): analyses the data forwarded by the HS and takes adequate actions, like confirming or adjusting medication, or activating the emergency support.

Emergency support (ES): This entity is in charge of providing health services (e.g., an hospital). It handles requests for serving medical emergencies (e.g., sending a doctor to the home of the patient).

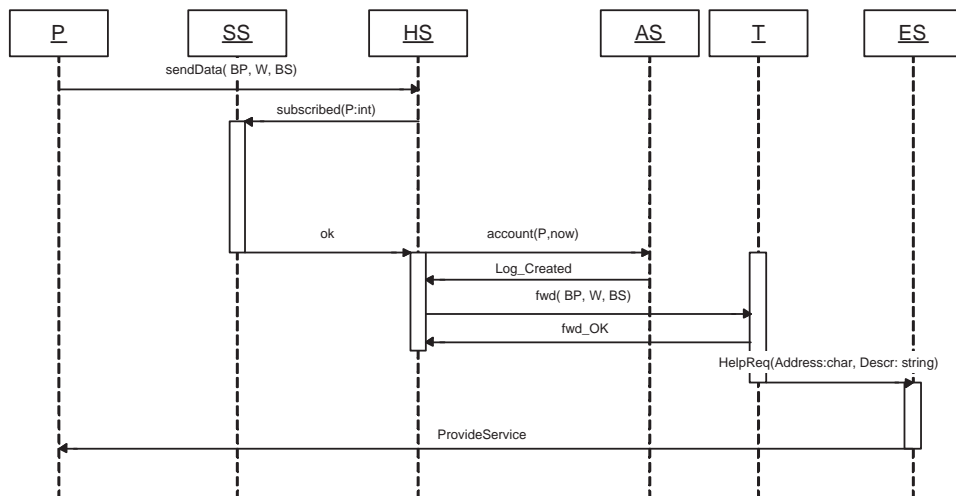


Fig. 1. Emergency Scenario.

Figure 1 and 2 show the flow of interaction among previous services, in two different scenarios. The first one shows some the case in which a registered patient contacts the service HS, which causes the team of doctors to activate the emergency support. The second one, shows the case in which the information is sent by a user that is not a registered patient and, in such case, the service HS sends the notification to the user.

Figure 3 depicts the definition of the orchestrator corresponding to the service HS.

3 Testing Orchestrations

This section discusses the possibilities of performing testing over web service compositions specified with orchestrations. We advocate the two well-known levels of testing: unit and integration. The remaining of this section assumes that no information about the behaviour of the partners is available (i.e., there is no definition of the choreography of the composition).

3.1 Unit testing

We recall that unit testing is intended to find bugs on the implementation of a particular unit (or basic component), and it is performed after the unit has been implemented. There are two main approaches for testing units: (i) *specification-based* or *black-box* testing, in which test cases are generated from the specification of the unit without any knowledge about its implementation, and (ii)

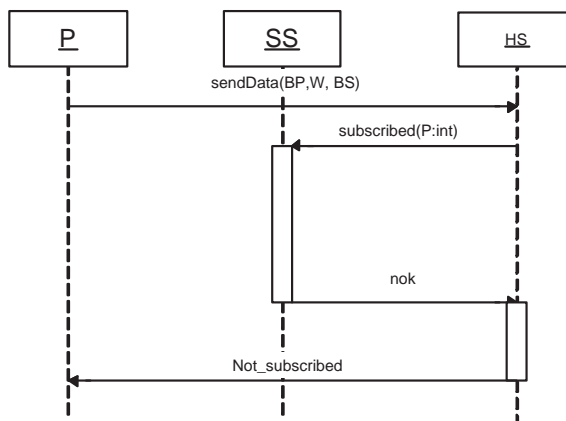


Fig. 2. Subscription Error Scenario.

implementation-based or *white-box* testing, in which test cases are selected by taking advantage of the actual code that implements the specification.

An orchestrator is a program (or code) that defines the behaviour of a particular partner, hence, it can be seen just as unit of code. Therefore, any orchestrator can be taken as a unit. Moreover, if the orchestration is considered just as the specification of the composite service then the test suite generated from such specification is part of the black-box unit testing plan. This is case when the orchestration is actually coded in a language different from the specification. For instance, a BPEL specification coded as a Java program. However, since orchestrations are executable specifications, they can be considered as the actual code of the services, e.g., a BPEL specification is a piece of code that can be executed by any BPEL *engine*. In this case, the activity of generating test cases from the orchestrator is white-box testing, but what we actually test is the specification, since we assume the implementation of the orchestration engine to be correct. In other words, we assume that the software that will execute our specification is correct and, hence, that the execution of a correct specification produces the correct behaviour. This contrasts with the first case, in which we actually test the implementation by assuming the specification to be correct. We remark that testing orchestrations is two-fold:

- Functional black-box testing of implementations: when the orchestrator is being implemented in a language different from the specification language.
- Functional white-box testing of the specification: when the description of the orchestrator coincides with its implementation.

Test case generation Test case generation from orchestration can be done by applying the principles introduced by white-box testing of code, particularly, test generation from control and data flow abstractions of the code, as described

```

Receive : sendData(BP, W, BS)

invoke : ans := subscribed(P)

                if ans = ok
then
                else
flow           reply Not_subscribed

invoke : ans := fwd(BP, W, BS)  invoke : ans := account(P, now)

reply processing

```

Fig. 3. The orchestrator of HS.

in [14]. Depending on the style of the orchestration language we use, an orchestrator may be a program built-up from: (i) structured flows (as in any ordinary programming languages), (ii) links describing unstructured flows (a sophisticated version of goto statements guarded by conditions), or (iii) by a combination of structured and unstructured flows. In any case, the computation structure of an orchestrator can be represented as a kind of flowgraph, as done for programs written in any ordinary programming language. We omit here the definition of the flowgraph construction (we refer the interested reader to [1] for details of the ordinary construction), and we informally discuss the peculiarities of flowgraph generation for orchestration languages. Orchestration languages provide the following distinctive features that should be managed when generating the corresponding flowgraphs.

- Concurrency: Flowgraph for orchestration languages should model parallelism, i.e., there can be fork nodes from which several edges may leave that represent a fork in the control flow of the program. Consequently, a particular computation is not represented anymore with a sequence, but with a graph, i.e., a partial order.
- Links or synchronization: In orchestration languages, several parallel path of executions can be synchronized by stating links. The links of the orchestration map directly to edges on the flowgraph.
- Exceptions and transactional scopes: orchestration languages usually have to deal with exceptions and compensations, that is flows of executions that are

activated either when some activity in the program or within a transaction fails. Hence the flowgraph should also model these situations.

- Events, or timers: Orchestration languages have also the possibility of activating some flows of executions related to some time constraints. Hence, a full representation of the flow structure of an orchestration should also model time constraints.

Once we have a flowgraph representation of the behaviour of an orchestrator we can derive test cases by taking advantage of the usual adequacy criteria concerning both:

- *control-flow*, i.e., by taking into account the parts of the graph that are exercised by a particular test case. Typical control-flow adequacy criteria include *statement*, *branch*, *condition*, and *path* coverage.
- *data-flow*, i.e., by considering the data dependency among the instructions of the program. Typical data-flow adequacy criteria are *definition-use*, *all-definitions*, *all-uses*, *all-definition-use-path* coverage.

Further details on flowgraph coverage criteria can be found in [1].

We also remark that an orchestrator is essentially a reactive system. In fact, basic operations are related to the reception and generation of messages to other services. Hence, a general form for the input of a test case is not just a partition of the domain of the data, but also an specific ordering of the interactive events. In cases in which the flowgraph fixes a total order over such events, then by selecting a particular path in the flowgraph will also fix a precise sequence of events. Differently, when orchestrations use concurrency primitives (like parallel composition, or timed events such as alarms) a particular path does not fix a total order over events, and hence a test case should also fix the actual sequence of interactions [16]. A straightforward approach should be to consider the inputs of a test case as a particular path p of the flowgraph extended with a causal ordering of events that are concurrent in p . Such causal orderings can be selected by following some adequacy criteria: immediate candidates are *some-interleaving* and *all-interleavings* covering.

Finally, any test case should be equipped with a definition of the expected result, or in other words, with a *test oracle*, i.e., a mechanism for determining the behavioral correctness of executions. In particular any test oracle has two parts: the *oracle information* that specifies which is the correct behaviour, and the *oracle procedure* to verify the results of the test execution against the expected ones.

As tradition when testing, oracles could be generated from software specifications (see [16]). In particular, the generation of oracles for testing orchestration will depend on the type of testing we are performing:

- Functional black-box testing of implementations: In this case oracles could be generated directly from the orchestrator definition, i.e., the actual implementation of the orchestrator should mimic its definition. Hence, the expected result could be expressed as the path in the orchestrator that defines the test case.

- Functional white-box testing of the specification: In this case, the oracles should be generated from a different specification of the behaviour of the component. In the service realm, they could be generated from the definitions of all choreographies the orchestrator is involved in. Alternatively, they can be inferred from other any other kind of specification as usually done for software testing [16].

Test execution The execution of tests at the unit level in of SOA applications involves the same activities as ordinary unit testing, which are mainly:

- the construction of drivers and stubs, or mock objects, simulating the behaviour of other parts of the system. In particular, for testing an orchestration we need to simulate the behaviour of any other service in the process. Since the only information we have is the definition of the orchestrator the behaviour of other partners should be defined by the user.
- monitoring the execution of the test, i.e., once all additional elements have been defined, then the unit under test should be run with the corresponding inputs, and all observable effects of such execution should be collected.
- the decision about the result of the testing, once the test case has been run, then it should be decided whether the unit behaves as expected for that particular test case.

As usual, testing environments are fundamental for giving support to the automation of such activities.

3.2 Integration

Integration testing is aimed at exercising the interaction among components and not just single units. Hence, an integration test case involves the execution of several components. Moreover, a plan for integration testing is based on an integration strategy, i.e., the order in which components are put together. The traditional strategies are: *top-down*, *bottom-up*, *threads*, *big-bang*, *critical modules*.

Performing integration testing of a single orchestrator should overcome two main problems:

- lack of information about the behaviour of involved components: an orchestrator provides a complete description of the coordinator of the system (like the description of a top-level module in a functional decomposition). Nevertheless, the behaviour of particular components is underdefined. In fact, from an orchestrator it can be inferred only a partial view of protocol followed by component services: that one that is related to the interactions with the coordinator.
- impossibility of exercising third party services in testing mode: some services that are taking part of the composition are out of the sphere of control of who is developing the orchestration, and hence it is frequently unlikely that

those services can be used for running tests for free. For instance, consider a service for doing online shopping that will accept credit card payments. Hence, when testing the online shopping service, it is unlikely the credit card service can be run in a testing mode.

The second problem may prevent or imposed particular constraints to the use of integration strategies, since the executions of particular components should be avoided or restricted/minimized in some way. Clearly, this may prohibit the use of some strategies like big-bang integration. On the other hand, the first point makes strategies like bottom-up and threads unfeasible, since we lack from a description of the complete structure of the systems. Therefore, threads of interactions and bottom-up strategies cannot be discovered from the specification. Consequently, the most viable integration strategy is the top-down, in which components are incrementally introduced. Information about critical components (for instance, the number of interactions with the coordinator) can be used for deciding the order in which such components are introduced.

4 Testing Choreographies

A choreography models the interactions among a set of services either from the viewpoint of an ideal observer who oversees all interactions or from the perspective of a particular service, capturing only those interactions that directly involve it. Hence, testing choreography basically resembles the testing of systems described in terms of interaction models. In the following we review the most widely known strategies for testing from these kinds of specifications.

4.1 Unit Testing

As for orchestrations, a unit is a particular partner or service of the choreography. Specifically for choreographies, the testing at the unit level is aimed at evaluating whether a particular partner follows an interaction pattern that is conformant to the agreed protocol (i.e., the choreography). This kind of testing for web services is also referred to as *conformance testing*, since it is intended to certify the capability of interaction of any party in the composition. It is assumed that two peers that are proved conformant to the same protocol will actually interoperate by producing a legal conversation. Note that conformance testing is a kind of *functional black-box testing*, because test cases are generated from the specification of the behaviour of the partners, without any knowledge about their implementations.

Test case generation Note that both global and local style of choreography description provide an operational definition of the protocols followed by services. As aforementioned, in the first case such protocol is given by describing the interaction with other agents, while in the second case, the protocol is just a definition of the order in which messages are received and sent by the component.

Consequently, we can reuse techniques developed for testing object or component based systems at the unit level. Most proposals are based on test case generation from finite state machines describing the behaviour of units. We omit here the details about these techniques and refer the interested reader to [11,8], and we only discussed the main aspects of applying them to choreography descriptions.

Firstly, local view choreographies actually provide the state machine description of the protocol of a particular service. For instance, in the case of BPEL abstract processes, what we actually have is the definition of a structured program containing non determinism, which can be modelled by a state machine. Clearly, such programs could be hardly considered finite state machines, since they handle data and its behaviour may depend on the particular values of some variables. Hence, a comprehensive testing model should also consider advanced notions of state machines or suitable abstractions, as done for instance when considering object oriented systems. The second point is whether a service is described using (i) a *single point of view*, i.e., like participating in only one choreography or (ii) a *multiple point of view*, i.e., when the service is described by several choreographies. In the first case, there is a unique description of the of the protocol followed by the service (i.e., one state machine), while in the second one, such descriptions may be not unique (for instance, the partner plays different roles in different choreographies) and, therefore, test cases should be derived by using the information provided by all choreographies. Here, the approaches may derive tests by consider any model in isolation, or they may synthesis the complete abstract behaviour of the partner, for instance, by considering the parallel composition (if defined) of the machines corresponding to all choreographies associated with a party.

When dealing with global choreographies, what we actually have is a one or a set of interaction descriptions, like interaction diagrams. Consequently, the state machine associated to a component can be inferred by using standard techniques for deriving state machines from interaction diagrams, such as [7,20]. In this case, we also have the problem of considering state machines generated from a unique interaction or from several.

Once a state machine model of the behaviour of the unit is obtained, test generation can be approached by using standard techniques that provide criteria for selecting input sequences by following the paths of the corresponding state machine. By analogy with control-flow coverage, we have adequacy criteria based on the coverage of paths in the state machine. Typical criteria are, e.g., *all-states*, *all-transitions*, or the *W-method* [6].

Example 2. Figure 4 shows the state machine modelling the behaviour of the Healt Service in Example 1. In this case, we can select the following suite containing two test cases $\{[s1, s2, s3, s4, s5, s7, s8, s1], [s1, s2, s3, s6, s1]\}$ that satisfies the *all-states* and *all-transitions* criteria.

The generation of test oracles in this case is analogous to the case of functional black-box testing of orchestrations, in which the expected outputs of the party are determined when the test case is selected. That is, we can recover from the

state machine model the outputs that should be generated by the party. Clearly, such information may be incomplete for orchestrations. For instance, what we can obtain from a choreography is the description of the receiver, but the exact content of the message may be not defined. If such content should be checked, we would need some extra specification of the behaviour of the system.

The problems of test execution are the same as those of executing testing generated from orchestrations.

4.2 Integration Testing

All the discussion done for integration testing for orchestrations apply also for choreographies. Nevertheless, in this case we also have a description of the interactions among partners and, hence, we can apply techniques that take advantages of such models for selecting test cases.

Test case generation The generation of test cases that take advantage of the interaction model for exercising interesting patterns of interactions may reuse techniques for performing *model-based testing* [2] proposed for object or component-based systems. Such kind of techniques start from the behavioral specification of the services (e.g., state machines) and a test directive (e.g., a sequence diagram), and produces as output a set of test cases representing the shortest paths in the state machine that covers the given directive. In other terms, each test case contains the sequence of messages expressed by the test directive. The main idea behind model based testing is that of providing a set of particular messages that should be exercised. For instance for the composition in Example 1, we would like to exercise the messages in Figure 5. Hence, model based testing will allow to automatically build (by using the information of the local behaviours) the shortest interaction sequence (shown in Figure 6) that exercise such messages.

5 Classification of proposals

This section discusses the approaches proposed in the literature and classifies them accordingly to the alternatives introduced in the previous sections (Figure 7 graphically depicts such hierarchies). Table 1 summarizes the results of our classification.

Unit testing for orchestration have been addressed by several works. Mayer and Lübke [15] have proposed a framework for performing white-box unit testing. Although they describe a particular framework for giving support to this activity, non systematic way for defining test cases is presented. Differently, Yuan *et al.* [23] define a technique for white-box testing generation base on an adaptation of classical control flow structure coverage criteria. With the same aim, Yuan *et al.* [23] present a model checking based tool for generating test cases based on flow and data coverage criteria.

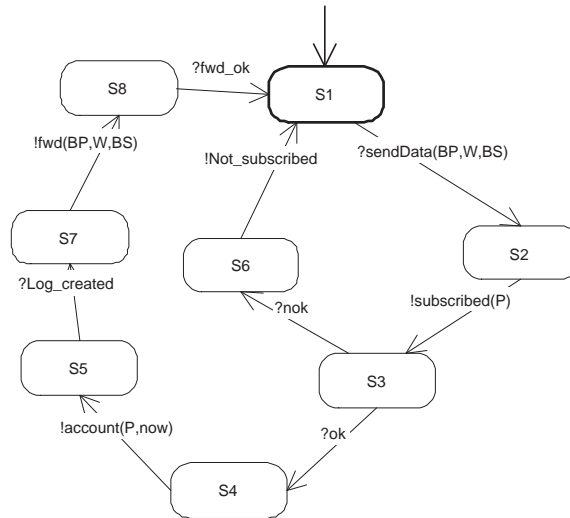


Fig. 4. Behavior of the Health Service.

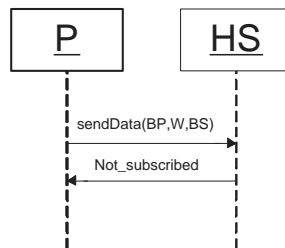


Fig. 5. Test Directive.

Luang et al. in [12] proposed an integrated process to automatically translate OWL-S specification (i.e., the composition model) of composite WS into a C-like specification language that can be processed by concurrent version of BLAST model checker, which can generate positive and negative test cases during model checking of a particular formula. Nevertheless, this method requires the specification of properties to prove that are outside of the standard specification of the orchestration.

A general framework for doing black box testing at the unit level from a local choreography is introduced in [3]. This proposal is aimed at introducing a testing phase before the registration of services into UDDI registry. The idea is that UDDI registering role is extended to play also the role of an external testing organism which validates the WS conformity to the published interface, such interface is a local choreography modelled as a finite state machine. A different approach for generated black box unit tests is presented in [9], where choreographies are presented as contracts, i.e., as mutually agreed behaviour

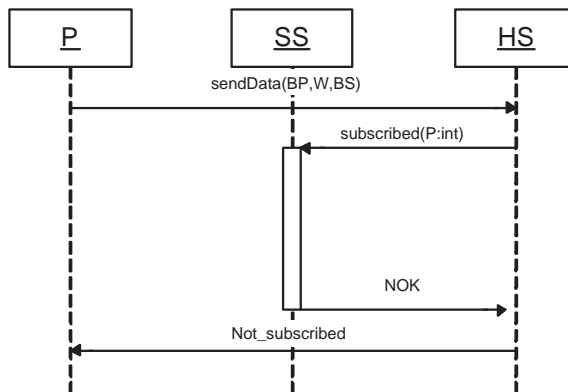


Fig. 6. Test Case.

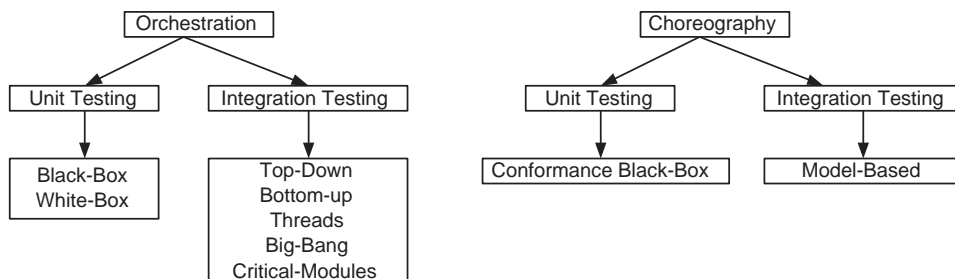


Fig. 7. Alternatives for Testing Service Composition.

among two partners. Such contracts are represented then as graph grammars, and then test cases are derived from them.

A basic form of integration testing out of local choreographies that checks the compatibility among two services have been addressed in [3,10]. Those approaches use an operational model of the interface of the services (e.g., a finite state machine or a graph grammar model) for generating test cases that check the compatibility of a published service against the requirements of a particular client. As in the previous case, such role is played by the UDDI registering role. In the case of [3] the behaviour of a service is modelled as a finite state machines, while in [10] component interfaces are described as graph grammars.

The proposal in [18] analyses the problem of generated unit testing from scenarios (i.e., a kind of global choreography), by providing a framework for testing. The test generation is not presented in detailed, but uses techniques previously developed by authors for testing object oriented software. In this case, no particular attention to aspects related with service composition (mainly correlation and long running transactions) are taken into account. In the same line is the proposal in [19].

Style	Testing Level	Model	Approach
Orchestration	Unit	BPEL	Mayer and Lübke [15] Yuan <i>et al.</i> [23] Zheng <i>et al.</i> [24]
		OWL-S	Yuan <i>et al.</i> [23]
		Finite State Machines	Bertolino and Polini [3]
Local Choreography	Unit	Graph Grammars	Heckel <i>et al.</i> [9]
		Finite State Machines	Bertolino and Polini [3]
	Integration	Graph Grammars	Heckel <i>et al.</i> [10]
		Scenarios	Tsai <i>et al.</i> [18] Tsai <i>et al.</i> [19]
Global Choreography	Unit	Scenarios	Tsai <i>et al.</i> [18] Tsai <i>et al.</i> [19]
	Integration	Scenarios	Tsai <i>et al.</i> [18] Tsai <i>et al.</i> [19]
Global Choreography and Orchestration	Unit	BPEL, State Machines	Li <i>et al.</i> [14]

Table 1. Classification of Proposals for Testing WS Composition

Li *et al.* [14] propose a testing technique from BPEL specifications. This proposal uses a mix of orchestration and local choreography information for generating test cases. This proposal is focused on the creation of a test framework for giving support to the execution of test cases, providing an infrastructure for invoking and handling answers from particular services.

6 Conclusion and Future works

This work is an initial effort for understanding the current situation of testing techniques for standard approaches to web service composition. The main conclusion is that, although several techniques and methodologies can be reused in this particular context, the discipline is still quite immature. Many aspects of testing from orchestrations and choreographies are still in dark corners. As far as orchestration is concerned, we highlight the fact that typical aspects of orchestration languages, such as compensations, timed events, correlation sets are overlooked when mapping orchestrations to traditional control and data flow models. At the same level, there is no much insight about the conceptual kind of testing done (testing implementations or specifications) when deriving test cases from BPEL definitions and, consequently, no much concerns about oracles generation are given in such cases. Similarly, to the best of our knowledge no work discusses strategies for doing integration testing from orchestrations.

As far as choreographies is concerned, the situation is quite similar. In particular, the mappings from real orchestration languages to particular formalisms (e.g., graph grammars and finite state machines) are still unclear. Crucial points are the mapping of infinite models to finite ones, for instance, by disregarding data. In such cases, how we generate oracles from such models to compare actual

executions. Have those data a low relevance when testing? How different choreographies should be tested? Independently, or by considering them together? In the later case, how we combine the information given by different models.

Moreover, to the best of our knowledge there has been no effort in providing guidelines, criteria, approaches (better if supported by evidence) for building testing plans for service compositions. Our work is a contribution in this line, that attempts to classified the main point to be covered by a test plan and the current approaches proposed in the literature. As a by product, we identified some obscure points for guiding our future works.

Acknowledgment

Authors thank Henry Muccini for his valuable insights about this problem. Antonio Bucchiarone is also supported by the Marie Curie Host Fellowships for Transfer of Knowledge (FP6-2002-Mobility 3 Proposal n FP6-14525) and he collaborates with Nokia Siemens Networks, Lisboa, Portugal.

References

1. B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
2. A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electr. Notes Theor. Comput. Sci.*, 116:85–97, 2005.
3. A. Bertolino and A. Polini. The audition framework for testing web services interoperability. In *EUROMICRO-SEAA*, pages 134–142, 2005.
4. BPEL Specification. version 1.1. Available at <http://www.ibm.com/developerworks/library/ws-bpel>, May 2003.
5. Business process modelling language (BPML). Available at <http://www.bpml.org>.
6. T. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.
7. S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991.
8. S. Gnesi, D. Latella, and M. Massink. Formal test-case generation for uml state-charts. pages 75–84. IEEE Computer Society, 2004.
9. R. Heckel and M. Lohmann. Towards contract-based testing of web services. *Electr. Notes Theor. Comput. Sci.*, 116:145–156, 2005.
10. R. Heckel and L. Mariani. Automatic conformance testing of web services. In *FASE*, pages 34–48, 2005.
11. H. Hong, Y. Kwon, and S. Cha. Testing of object-oriented programs based on finite state machines. *apsec*, 00:234, 1995.
12. H. Huang, W. Tsai, R. Paul, and Y. Chen. Automated model checking and testing for composite web services. In *ISORC*, pages 300–307, 2005.
13. F. Leymann. WSFL Specification. version 1.0. Available at <http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>, May 2001.

14. Z. Li, W. Sun, Z. Jiang, and X. Zhang. Bpel4ws unit testing: Framework and implementation. In *2005 IEEE International Conference on Web Services (ICWS 2005)*, pages 103–110, 2005.
15. P. Mayer and D. Lübke. Towards a bpel unit testing framework. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 33–42. ACM Press, 2006.
16. D. Richardson, S. Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 105–118, New York, NY, USA, 1992. ACM Press.
17. S. Thatte. XLANG: Web Services for Business Process Design. Available at http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.
18. T. Tsai, R. Paul, L. Yu, A. Saimi, and Z. Cao. Scenario-based web service testing with distributed agents. E86-D(10):2130–2144, 2003.
19. W. Tsai, R. Paul, W. Song, and Z. Cao. Coyote: An xml-based framework for web services testing. In *HASE*, pages 173–176, 2002.
20. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 314–323, New York, NY, USA, 2000. ACM Press.
21. Web Services Choreography Description Language. Version 1.0. Available at <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>, April 2004.
22. WSCI Specification. version 1.0. Available at <http://www.w3.org/TR/wsci/>, August 2002.
23. Y. Yuan, Z. Li, and W. Sun. A graph-search based approach to bpel4ws test generation. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, page 14. IEEE Computer Society, 2006.
24. Y. Zheng, J. Zhou, and P. Krause. A model checking based test case generation framework for web services. In *Proceedings of the International Conference on Information Technology (ITNG'07)*, pages 715–722. IEEE Computer Society, 2007.