# Testing Service Composition

Antonio Bucchiarone[1]*, Hernán Melgratti[1], and Francesco Severoni[2]

[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 (SOC) is aimed at providing the bases for building software by assembling independent, loosely coupled services. As any software development activity, also building a composite service requires strategies for performing quality assessment of applications and, in particular, 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.
**Keywords**: Testing, Orchestration, Choreography

## 1 Introduction

Many recent efforts, mainly coming from the industry, 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 effort has been spent on services and service composition specification, only recently the focus has been on the validation and verification of service and service composition. Few testing techniques for SOC applications have have been recently proposed. The goal of this research paper is to analyse 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 languages, services can be composed by following the two complementary views of choreography and orchestration. Web services composition 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. In this way, an orchestration exposes the internal logic of

* 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.

a single component by specifying the control-flow structure and the data flow dependencies of that particular component. The specification of the composite service, 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 interaction with other services: (i) *invoking* services, (ii) *receiving* an answer, (iii) *accepting* an invocation, and (iv) *answering* an invocation. The flow of such interactions can be specified either: (i) in a *structured way*, by using the operators of sequence, iteration and branching (as in any procedural language), and the concurrent flows; or (ii) in an *unstructured way*, by specifying the pair-wise dependencies among basic activities (i.e., by writing links). Typical orchestration languages are XLANG, WSFL, and (executable) WS-BPEL.

The *choreography view* exposes the flow of interaction among all involved parties: every participant is aware of the fact that it is taking part of composition and, thus, of the way it should interact. Choreography languages allow for the definition of protocols that parties should follow. There are two main approaches to define choreographies: (i) the *global model*, in which a protocol describes from a global perspective the messages exchanged by all parties, and (ii) 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.

Consequently, we should distinguish between choreography-based and orchestration-based testing. Moreover, since both units (i.e., the services) and assemblies (i.e., the composed services) can be subject of malfunctions, both unit and integration testing must be taken into consideration.

The following sections are organized as follows: Sections 2 and 3 discuss the alternative ways for testing respectively orchestrations and choreographies. Section 4 surveys the approaches proposed in the literature for testing service compositions. Conclusions and future works are drawn in Section 5.

## 2 Testing Orchestrations

This section discusses the possibilities for performing unit and integration testing of orchestrations. We advocate the two well-known levels of testing: unit and integration. We assume that no information about the behaviour of the partners is available (i.e., the choreography is not specified).

### 2.1 Unit testing

Unit testing is intended to find bugs on a particular unit (or basic service). There are two main approaches for testing units: (i) *specification-based* or *black-box*, in which test cases are generated from the specification without any knowledge about the implementation, and (ii) *implementation-based* or *white-box*, in which test cases are selected by taking advantage of the actual code.

An orchestrator codes the behaviour of a particular partner and, hence, it can be seen as a unit of code. When the orchestration is just the specification of a composite service (e.g., when the orchestration is implemented in a language different from the specification), then the test suite generated from the orchestration is part of the black-box testing plan. Differently, orchestrations are executable specifications and can be considered as the actual code of the services. Hence, the generation of test cases from an orchestrator is white-box testing. In this case, we actually test the specification, since we assume the orchestration engine to be correct. Note that the in the first case we actually test the implementation by assuming the specification to be correct. We remark that testing orchestrations could be two-fold:

- Functional black-box testing of implementations: when the orchestrator is 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 orchestrations can be done by applying the principles introduced by white-box testing of code, particularly, test generation from control and data flow abstractions [14]. Either when the orchestrator is specified by structured or unstructured flows, its computation structure can be represented as a flowgraph, as for any ordinary program. (We refer to [1] for the construction of ordinary flowgraphs.) Nevertheless, flowgraphs for orchestration languages should manage the following distinctive features.

- Concurrency: flowgraphs should model parallelism, i.e., nodes that fork the control flow of the program. Consequently, a computation is not represented as a sequence but as a graph (i.e., a partial order).
- Links or synchronization: several parallel paths of executions can be synchronized by links. Those links map directly to edges on the flowgraph.
- Exceptions and transactional scopes: flowgraphs should model flows of executions that are activated when some activity fails.
- Events and timers: flow structures should accommodate executions that are activated at a particular time.

Starting from a flowgraph, test cases can be derived by taking advantage of the usual adequacy criteria (see [1]), concerning both:

- *control-flow*, i.e., by taking into account the parts of the graph that are exercised, like *statement*, *branch*, *condition*, and *path* coverage.

- *data-flow*, i.e., by considering the data dependencies among activities, such as *definition-use*, *all-definitions*, *all-uses*, *all-definition-use-path* coverage.

Moreover, orchestrators are reactive systems, since basic operations are related to message exchanges. Hence, a test case should specify the order of the interactive events [16], for instance, by considering a test case as a particular subtree $s$ of the flowgraph extended with a causal order among the concurrent events of $s$. Such causal orderings can be selected by following some adequacy criteria, like *some-interleaving* and *all-interleavings*.

Finally, any test case should be equipped with the expected result: a *test oracle*, i.e., a mechanism for determining the behavioural correctness of executions. As tradition, oracles could be generated from software specifications (see [16]) and, in this case, it will depend on the type of testing we are performing:

- For black-box testing of implementations: 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 is given by the observed outputs in the order defining the test case.
- For white-box testing of the specification: 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 the choreographies the orchestrator is involved in.

**Test execution** The execution of orchestration testing resembles ordinary testing since it requires:

- the construction of drivers and stubs, or mock objects, simulating the behaviour of other parts of the system. Since orchestrations provide no information about the behaviour of other partners, they should be defined by other specifications.
- monitoring the execution, i.e., the unit under test should be run with the corresponding inputs, and all observable effects should be collected.
- Once the test case has been run, it should be decided whether the unit behaves as expected for that particular test case or not.

## 2.2 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. 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*.

For testing a single orchestrator we should overcome two main problems:

- the lack of information about the behaviour of involved components: an orchestrator is like the description of a top-level module in a functional decomposition. Hence, the behaviour of particular components is underdefined. From the orchestration point of view, the protocol followed by components is given by the interactions they have with the coordinator.
- impossibility of exercising third party services in testing mode: some components 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.

The second problem may prevent or impose particular constraints to the use of some integration strategies, since the executions of particular components should be avoided or minimized in some way. Clearly, this may prohibit the use of strategies like big-bang. Instead, the first point makes strategies like bottom-up and threads infeasible, since we lack from a description of the complete structure of the systems. 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.

## 3 Testing Choreographies

Choreographies capture the interactions among services. Consequently, testing choreographies resembles testing of systems described by interaction models. This section reviews alternative strategies for doing this task.

### 3.1 Unit Testing

A unit is a particular partner of the choreography, and unit testing is aimed at evaluating whether a particular partner follows an interaction pattern that is conformant to the agreed protocol (i.e., the choreography). This testing is known as *conformance testing*, since it is intended to certify the capability of interaction of a party in the composition. Conformance testing is a kind of *functional black-box testing*, because test cases are generated from the specification of the behaviour of partners, without any knowledge about their implementations.

**Test case generation** Global and local styles of choreography descriptions provide an operational definition of the protocols followed by services. Global protocols describe the interaction among all agents, while local protocols define the order in which messages are received and sent by a particular 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 discuss the main aspects of applying them to choreographies.

Firstly, local choreographies provide a kind of state machine description of the protocol run by a particular service. For instance, abstract processes of BPEL are structured programs containing non determinism, which can be modelled by state machines. Since such programs handle data and their behaviours may depend on the particular values of some variables, a comprehensive testing model should consider advanced notions of state machines or, alternatively, suitable abstractions, similarly to object oriented systems. The second point is whether a service is described by 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 (i.e., the party plays different roles in different choreographies). In the first case, test cases are generated from the unique description of the protocol (i.e., one state machine), while in the second case, tests should consider the information provided by all choreographies. In such situation, there are two alternatives: (i) to consider any model in isolation, or (ii) to consider the complete abstract behaviour of the partner synthesized from all choreographies associated with that party.

Instead, global choreographies are partial orders of message exchanges, like interaction diagrams. Hence, the state machines describing components could be inferred by using standard techniques for deriving state machines from interaction diagrams [7,20]. Also here, we may generate state machines either from a unique description or from multiple models.

Given the state machine model of a unit, a test case is obtained by selecting a particular path of the state machine. Analogously to control-flow coverage, we have adequacy criteria based on the coverage of the state machine, like *all-states*, *all-transitions*, or the *W-method* [6].

Oracle generation can be performed as for functional black-box testing of orchestrations: the expected outputs can be recovered from the particular path in the state machine. Nevertheless, such information may be incomplete. For instance, a choreography may specify the receiver of a message but not the exact content. When such information is relevant, extra information should be provided by other specifications.

| **Style** | **Testing Level** | **Model** | **Approach** |
|---|---|---|---|
| Orchestration | Unit | BPEL | Mayer and Lübke [15] |
| | | | Yuan *et al.* [23] |
| | | | Zheng *et al.* [24] |
| | | OWL-S | Huang *et al.* [12] |
| Local Choreography | Unit | Finite State Machines | Bertolino and Polini [3] |
| | | Graph Grammars | Heckel *et al.* [9] |
| | Integration | Finite State Machines | Bertolino and Polini [3] |
| | | Graph Grammars | Heckel *et al.* [10] |
| Global Choreography | Unit | Scenarios | Tsai *et al.* [18]<br>Tsai *et al.* [19] |
| | Integration | Scenarios | Tsai *et al.* [18]<br>Tsai *et al.* [19] |
| Global Choreography and Orchestration | Unit | BPEL, State Machines | Li *et al.* [14] |

**Table 1.** Classification of Proposals for Testing WS Composition

### 3.2 Integration Testing

In addition to the strategies for doing integration testing of orchestrations, we can take advantage of the description of the interactions among partners for selecting test cases.

**Test case generation** For generating test cases we may reuse techniques like [2] that take advantage of the interaction model for selecting cases that exercise interesting patterns of interactions. In particular, these approaches start from the behavioural 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. The obtained test cases are paths of the state machines that cover the given directive (or scenario). In other terms, each test case indicates the inputs of the components that allow us to obtain the interaction expressed by the test directive. For choreography testing, the specification of the behaviour of each service can be obtained either from the local choreography or derived from the global choreography as in the case of unit testing. The test directive may be synthesized from the global choreography.

## 4 Classification of proposals

This section discusses the approaches proposed in the literature and classifies them accordingly to the alternatives introduced in the previous sections. 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. Nevertheless, non systematic way for defining test cases is presented. Differently, Yuan *et al.* [23] provide a technique for white-box testing generation based on an adaptation of classical control flow coverage criteria. With the same aim, Yuan *et al.* [23] present a tool that uses model checking for generating test cases satisfying flow and data coverage criteria.

Huang *et al.* [12] propose an integrated process for translating automatically an OWL-S specification (i.e., the composition model) into a C-like specification language, which can be processed by a concurrent version of the model checker BLAST. They can generate positive and negative test cases while performing model checking of a particular formula. Such formula is provided by the user and not by the specification.

A general framework for doing unit black box testing of a local choreography is presented in [3]. This proposal introduces a testing phase before a service is registered into the Universal Description Discovery and Integration (UDDI) registry. The idea is that UDDI registering role is extended to play also the role of an external testing organism that validates the conformance of the service w.r.t. the published interface (interfaces are modelled as local choreographies in the form of finite state machine). In the line of conformance testing, a black box approach for units is presented in [9], where choreographies are contracts describing the mutually agreed behaviour among two partners. Contracts are represented as graph grammars, and test cases are derived automatically from them.

A basic form of integration testing checking the compatibility among two services that are described by local choreographies has been addressed in [3,10]. These approaches use an operational model of the interface of the services (e.g., a finite state machine or a graph grammar) 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 [10] uses graph grammars.

The proposal in [18] tackles the generation of test cases from scenarios (i.e., a kind of global choreography). The generation method is inherited from previous works of the authors for testing object oriented software. No particular attention to aspects related with service composition (mainly correlation and long running transactions) are taken into account. The proposal in [19] is analogous.

Li *et al.* [14] propose a testing technique of BPEL specifications that uses a mix of orchestration and local choreography information for generating test cases at the unit level. 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 to and handling answers from particular services.

## 5 Conclusion and Future works

This work is an initial effort for understanding the current state of testing of 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 (implementations or specifications) when deriving test cases and, consequently, no much concerns about the generation of oracles. 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. How do we test a service involved in different choreographies? We consider such specifications as independent or all 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.

## Acknowledgment

Authors thank Henry Muccini for valuable insights about this topic.

## 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.bpmi.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 statecharts. 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 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. ACM Press, 2000.
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.