Automatic Test Case Generation for Services

Kathrin Kaschner and Niels Lohmann

Universität Rostock, Institut für Informatik, 18051 Rostock, Germany
{`kathrin.kaschner`, `niels.lohmann`}`@uni-rostock.de`

Abstract. Service-oriented computing (SOC) proposes loosely coupled interacting services as building blocks for distributed applications. This distributed nature makes the guarantee of correctness a challenging task. Based on the specification of a service (the public view), we introduce an approach to automatically generate test cases for black-box testing to check for compliance between the specification and the implementation of a service whose internal behavior might be secret.

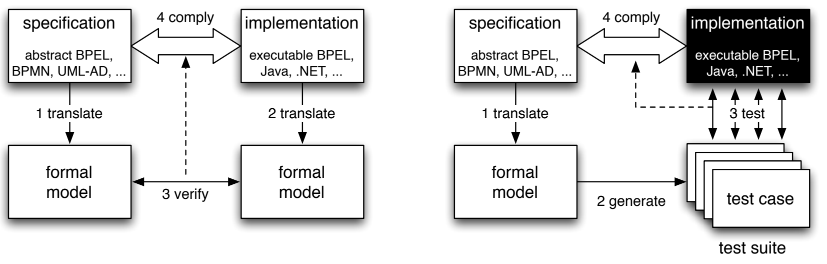
1 Introduction

Software systems continuously grow in scale and functionality. Various applications of a variety of different areas interact and are increasingly dependent on each other. Then again, they have to be able to be adjusted to the rapidly changing market conditions. Accordingly complex is the development and changing of these distributed enterprise applications.

To face these new challenges a new paradigm has emerged: *service-oriented computing* (SOC) [1]. It uses *services* as fundamental elements for readily-creating, flexible and reusable applications within and across organizational boundaries. A service is an encapsulated, self-contained functionality. Usually, it is not executed in isolation, but interacts with other services through message exchange via a well-defined interface. Thus, a system is composed by a set of services.

A well known class of services are *Web services* which use WSDL to describe their interface and SOAP to exchange messages. To deploy a Web service, two steps have to be taken. Firstly, its publicly observable behavior including the partners of the service and the exchanged messages is described. This *specification* — sometimes called business protocol or public view — is then *implemented* in a second step. To this end, all necessary details about the binding, message buffers, instantiation, etc. are added. While usually the specification language (e. g., BPMN, UML activity diagrams, EPCs) differs from the implementation language (e. g., Java, .NET languages), BPEL can be used to both specify and implement Web services. The specification (called an abstract BPEL process) contains placeholders that are replaced by concrete activities in the implementation (the executable BPEL process). The implementation can thereby be seen as a refinement of the specification.

Obviously, the implementation should comply to its specification. In the field of Web services, this is crucial as the public views are used as documentation of how to interact with a service. Thereby, any correct interaction derived from the public view should also be a correct interaction with the implementation. Compliance between an implementation and a specification can be shown by *verification*, cf. Fig 1(a). Thereby, the implementation and its specification have to be translated into a formal model (1, 2). Compliance between the models (3) then implies compliance between specification and implementation (4). Obviously, this approach can only be applied if the two models are available.



(a) verification (b) proposed black-box testing

Fig. 1. Approaches to check compliance between specification and implementation.

However, there are several scenarios in which a the formal model of the implementation is not available which makes verification impossible. For example, parts or the whole implementation might be subject to trade secrets. Furthermore, programming languages such as Java are — compared to high-level languages like BPEL — rather fine-grained and it is therefore excessively elaborate to create a formal model for such an implementation.

As an alternative, the implementation can be tested. The major disadvantage is that by *testing* only the presence of errors can be detected, but not their absence. To nevertheless be able to make a statement about the correctness of an implementation, a test suite with a significant number of test cases is necessary. Because only the interface of the implemented service is known and only the output of the service under test can be observed, this is called *black-box testing* in literature.

In this paper, we will present a novel approach to automatically generate all significant test cases to check compliance of the implementation and its specification, cf. Fig. 1(b). Therefore, we translate the specification into a formal model (1). This transformation is possible, because the specification is publicly available and usually not as complex as for example a Java program. The model of the specification is then the base for generating the test cases (2), with which we test the implementation (3). If a tests fails, we can conclude that the implementation does not comply to the specification (4).

The rest of the paper is organized as follows. In Sect. 2, we present a small example to motivate black-box Web service testing. The necessary formal models for services are recalled in Sect. 3. In Sect. 4, we sketch the test case generation approach. Section 5 is dedicated to related work. Section 6 concludes the paper and gives directions for future research.

2 Example

As an example for a service specification, consider the BPMN diagram in Fig. 2. It models the public view of a simple online shop of [2]. The shop internally decides after the user's login whether the user is a premium customer (upper branch) or a standard customer (lower branch). Standard customers have to accept the terms of payment before receiving an invoice.

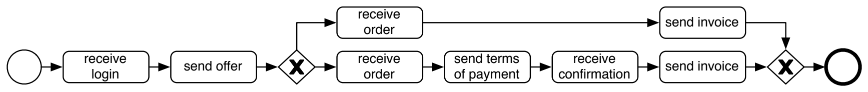
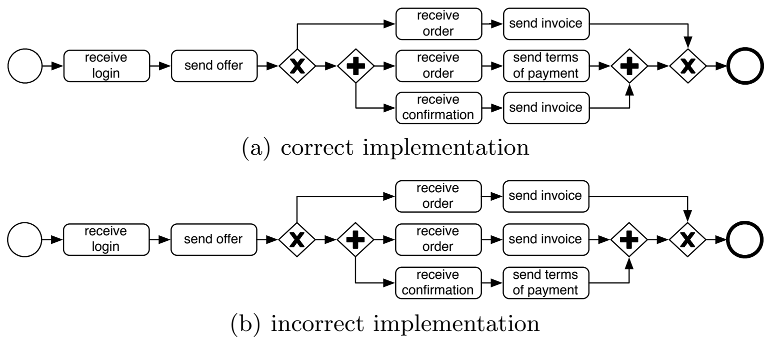


Fig. 2. A specification for a simple online shop.

To implement the online shop, a lot of details that have been left out in the specification have to be filled in. For example, the specification gives no information on how instantiation, message correlation, message queuing, fault handling, or logging is organized. Beside adding internal actions, also the order of communicating activities can be changed without jeopardizing compliance to the public view. After these modifications, it is not any more obvious that the resulting service complies to the original specification. Furthermore, the translation from the specification formalism (e. g., BPMN) into an implementation language (e. g., Java) is not always straightforward.



(a) correct implementation

(b) incorrect implementation

Fig. 3. Two implementations of the specification in Fig. 2.

Figure 3 shows two implementations of the online shop.[1] Implementation (a) executes the tasks for standard customers concurrently, yet still complies to the specification. The second implementation (b) further exchanges the invoice and the terms of payment message. This implementation does not comply to the specification as we will see later.

[1] To ease the presentation, we again use BPMN to diagram the implementations and focus on the message exchange between online shop and customer.

3 Modeling Services

As modeling formalism, we use *open workflow nets* (oWFNs) [3], a special class of Petri nets. They generalize classical workflow nets [4] by introducing an interface for asynchronous message passing. An important correctness criterion is *controllability* [5]. A service is controllable if there exists a partner service such that their composition (i.e., the choreography of the service and the partner) is free of deadlocks.

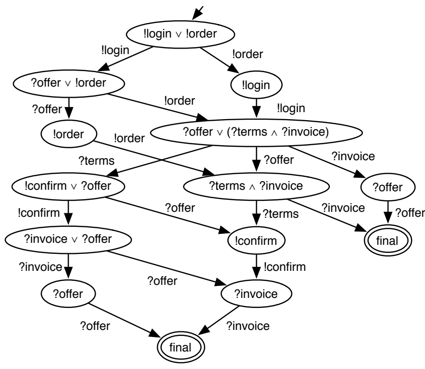
Controllability can be decided constructively: if a partner service exists, it can be automatically synthesized [2]. Furthermore, there exists one canonic *most permissive* partner which simulates any other partner service. The converse does, however, not hold; not every simulated service is a correct partner service itself. To this end, the most permissive partner service can be annotated with Boolean formulae expressing which states and transitions can be omitted and which parts are mandatory. This annotated most permissive partner service is called an *operating guideline* (OG) [6]. With translations back and forth between BPEL and oWFNs [7, 8], results of oWFNs can be directly applied to real-life Web services.

4 Generating Test Cases

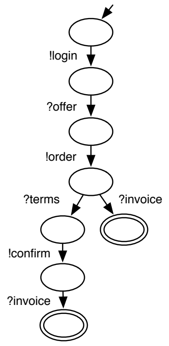
To check whether an implementation of a Web service complies with its specification, we focus on the partner services of the service. That are services which interact without deadlock with the given Web service. The implementation is compliant to the specification if it does not exclude partner services: any service that communicates correctly with the specification (i. e., the public view) should also communicate correctly with the implementation. To this end, any partner service of the specification can be seen as test case. The complete test suite is then characterized by the OG of the model of the specification.

The OG of the specification of Fig. 2 is depicted in Fig. 4(a). It is an automaton whose edges are labeled with messages sent to (preceded with "!") or received from (preceded with "?") the environment. The annotations describe for each state which have to be present (see [6] for further details). In total, the OG characterizes 139 partners. However, it is possible to select a small subset of partners as test cases. On the one hand, the most permissive partner is a canonic test case which can be derived by omitting the formulae of the OG. This partner covers the maximal behavior of the specification. On the other hand, the formulae can be used to "split" the most permissive partner into several smaller test cases. These test cases then can be checked independently of each other, for example using several dedicated test servers. In addition, the set of test cases can be adjusted using behavioral constraints [9]. Constraints allow to only generate certain test cases, for instance only those that use a certain functionality or send messages in a given order.

One test case is depicted in Fig. 4(b). This test case interacts without deadlock with both the specification (Fig. 2) and the first implementation (Fig. 3(a)). The communication with the second implementation (Fig. 3(b)), however, deadlocks.



(a) OG of specification of Fig. 2



(b) a test case

Fig. 4. The OG (a) describes all 139 test cases. One of them (b) can be used to show that the implementation in Fig. 3(b) does not comply to the specification of Fig. 2.

If the customer is treated as standard customer, the online shop — after receiving the login and an order and sending the offer and the invoice — is left in a state where it awaits the confirmation message. This will, however, only be sent by the customer after receiving the terms of payment. The services wait for each other; the composition deadlocks. While this test case can be easily derived directly from the specification, manual test case generation becomes an increasingly tedious task for real-life specifications with thousands of partner services.

If the specification is given as abstract BPEL service, an existing tool chain[2] is directly applicable to translate the specification into an oWFN and to calculate the OG thereof (cf. [2]). Furthermore, the test cases can be automatically translated into BPEL processes [8]. The resulting test Web services can then be executed on a BPEL engine to test the implementation under consideration.

5 Related Work

Several works exist to systematize testing of Web services (see [10] for an overview). In [11, 12], white-box test architectures for BPEL are presented. The approaches are very similar to unit testing in classical programming languages, but not applicable in our black-box setting. Furthermore, the authors give no information about how to generate test cases.

There exist several software tools that support testing of Web services, mostly focusing on white-box testing. Some for example Oracle BPEL Process Manager [13] or Parasoft SOAtest [14] allow to automatically create test cases. However, neither of products support a systematic test case generation.

Test generation can be tackled using a variety of approaches such as control flow graphs [15], model checking [16], or XML schemas [17]. These approaches mainly focus on code coverage, but do not take the communicating nature of Web services into account. In particular, internal activity sequences are not necessarily enforceable by a test case. Therefore, it is not clear how derived test cases can

[2] See `http://www.service-technology.org/tools`.

be used for black box testing in which only the interface of the service under test is observable. Furthermore, none of the testing approaches takes the complex communication nature of Web services into account and mainly assume simple remote procedure calls (i. e., request-response operations).

6 Conclusion

We presented an approach to automatically generate test cases for services. These test cases are derived from the OG of a specification and can be used to test compliance of an implementation without explicit knowledge of the latter (black-box testing). The approach bases on oWFNs as formal model, and can be applied to any specification language to which a translation into oWFNs exists. To support other specification formalisms, a translation into oWFNs is necessary. Existing translations from BPMN or UML-AD into Petri nets can are likely to be adjustable to oWFNs easily.

In future work, we also want to consider negative test cases to test the absence of unintended partners. Furthermore, we plan to validate our test case generation in a BPEL test architecture (cf. [11]) using real-life case studies. Another important aspect is to add data (as far as specified) to the test case generation to refine the results.

References

[1] Papazoglou, M.P.: Agent-oriented technology in support of e-business. Communications of the ACM **44**(4) (2001) 71–77

[2] Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: BPM 2006. Volume 4102 of LNCS., Springer (2006) 17–32

[3] Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43

[4] Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66

[5] Schmidt, K.: Controllability of open workflow nets. In: EMISA 2005. Volume P-75 of LNI., GI (2005) 236–249

[6] Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. Volume 4546 of LNCS., Springer (2007) 321–341

[7] Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: WS-FM 2007. Volume 4937 of LNCS., Springer (2008) 77–91

[8] Lohmann, N., Kleine, J.: Fully-automatic translation of open workflow net models into simple abstract BPEL processes. In: Modellierung 2008. Volume P-127 of LNI., GI (2008)

[9] Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: BPM 2007. Volume 4714 of LNCS., Springer (2007) 271–287

[10] Baresi, L., Nitto, E.D., eds.: Test and Analysis of Web Services. Springer (2007)

[11] Li, Z., Sun, W., Jiang, Z.B., Zhang, X.: BPEL4WS unit testing: Framework and implementation. In: ICWS 2005, IEEE (2005) 103–110

[12] Mayer, P., Lübke, D.: Towards a BPEL unit testing framework. In: TAV-WEB '06, ACM (2006) 33–42

[13] Oracle: BPEL Process Manager. (2008) `http://www.oracle.com/technology/products/ias/bpel`.
[14] Parasoft: SOAtest. (2008) `http://www.parasoft.com`.
[15] Yan, J., Li, Z., Yuan, Y., Sun, W., Zhang, J.: BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In: ISSRE 2006, IEEE (2006) 75–84
[16] García-Fanjul, J., Tuya, J., de la Riva, C.: Generating test cases specifications for BPEL compositions of Web services using SPIN. In: WS-MaTe 2006. (2006) 83–94
[17] Hanna, S., Munro, M.: An approach for specification-based test case generation for Web services. In: AICCSA, IEEE (2007) 16–23