Automatic Test Case Generation for Interacting Services

Kathrin Kaschner and Niels Lohmann

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

Abstract. Service-oriented architectures propose loosely coupled interacting services as building blocks for distributed applications. Since distributed services differ from traditional monolithic software systems, novel testing methods are required. Based on the specification of a service, we introduce an approach to automatically generate test cases for black-box testing to check for conformance between the specification and the implementation of a service whose internal behavior might be confidential. Due to the interacting nature of services this is a nontrivial task.
Key words: Testing, Interacting Services, Test Case Generation

1 Introduction

Software systems continuously grow in scale and functionality. Various applications in a variety of different domain 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 building blocks for readily-creating flexible and reusable applications within and across organizational boundaries. A service implements 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 logically or geographically distributed services, and SOC replaces a monolithic software system by a composition of services.

The encapsulation of a functionality further supports the reuse of services. To this end, a *service-oriented architecture* (SOA) proposes a service repository managed by a broker which contains information about services offered by several service providers. Due to trade secrets, a service provider will not publish a verbatim copy of his service containing all business and implementation details, but only an abstract description thereof. This description — called *public view* — only contains the information necessary to interact correctly with the actual implemented service.

While literature agrees that a public view or a similar description is necessary to realize an SOA, only few concrete approaches to generate public views have been proposed, for instance [2]. Even worse, interaction became more complex as services evolved. Instead of simple remote-procedure calls (i. e., request/response operations), arbitrarily complex and possibly stateful interactions on top of asynchronous message exchange are common [3]. In addition control flow, ownerships of choices, semantics of messages, and nonfunctional properties have to be taken into account. Then again, also the service provider is interested in whether the public view of his service implements the same business protocol as the actual implemented service. That means, every interaction that a customer can derive from the public view should also be valid together with the implemented service — even without knowledge about non-disclosed business and implementation details. This is especially important since the public view is used by a service requester to check whether his service fits to the provider's implemented service.

To this end, verification and testing of services received much attention both in academia and industry. However, the communicating nature of services is often neglected. Instead of taking the new characteristics of SOC and SOA into account, existing work on testing classical software systems was slightly adapted to cope with the new languages used to implement services. Consequently, these approaches only take the internal behavior of a service into account, but ignore the interactions with other services. Others in turn are restricted to stateless remote-procedure calls. This in turn might lead to undetected errors.

In this paper, we suggest a novel approach to test whether an implemented service interacts as it is warranted by its public view. Thereby, we focus on the business protocol (i. e., the message exchange and control flow) of the service. Aspects such as semantics of messages or nonfunctional properties are out of scope of this paper, but are orthogonal to our approach. We claim that like a function call is the most natural test case to test a function of a classical program, a *partner service* is likewise the most natural test case for another service. But as mentioned earlier, a public view implicitly describes a large number of correct interactions, and both finding and testing all possibilities is a tedious task. Therefore, we propose an approach to automatically generate test cases (i. e., partner services) that are required to test whether an implementation conforms to a public view.

The rest of the paper is organized as follows. In the next section, we describe how black-box testing can be realized using interacting services. The main contribution of this paper is presented in Sect. 3 where we explain our test case generation approach and present an algorithm to further minimize the generated test set. In Sect. 4, we focus on related work and highlight the differences to our approach. Section 5 concludes the paper.

2 Testing Interacting Services

Best practices propose that complex systems should be *specified* prior to their implementation. A specification of a service describes its business protocol and already contains all relevant internal decisions (e. g., the criteria whether or not a credit should be approved), but lacks implementation-specific details (e. g.,

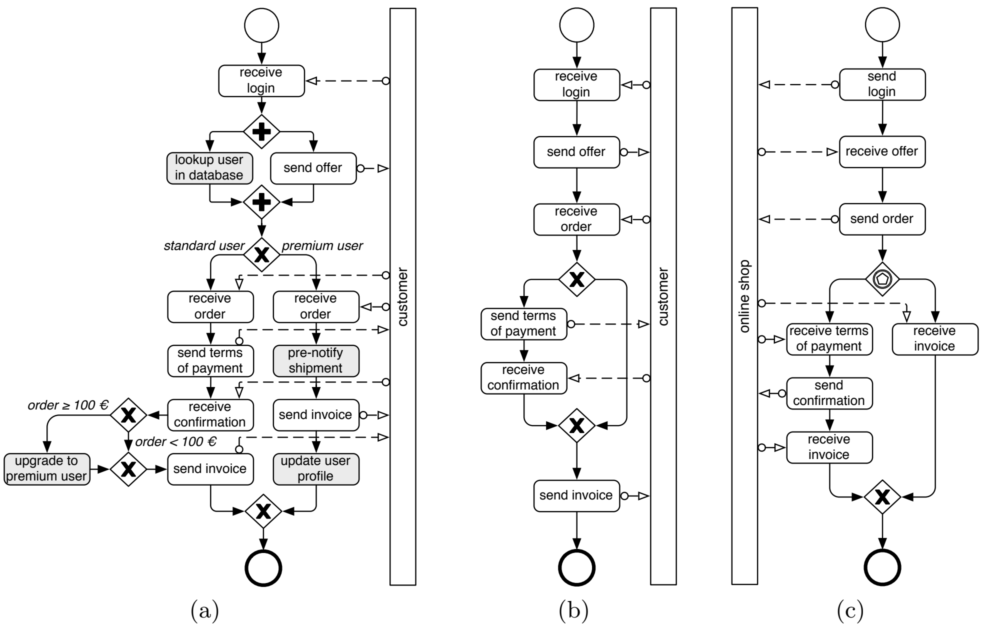


Fig. 1. A specification (a) and a public view (b) of a simple online shop together with a customer service (c). Internal actions are depicted gray.

whether amounts are represented as integers or floats). Finally, the public view lacks both details about internal decisions and implementation, and only describes the business protocol that is realized by the services. Ideally, the business protocol defined by the public view, the specification, and the implementation coincides. In particular, if *any* service behavior that is derived from the public view (resp. the specification) is a valid interaction with the implemented service, we call the implementation *conformant* with the public view (resp. the specification).

As the running example of this paper, consider an online shop which processes an order of a customer. Its specification is depicted as a BPMN diagram [4] in Fig. 1(a). After a customer logs in to the online shop, its user name is looked up in the shop's database. Depending on previous orders, the customer is rated as standard or premium customer. On the one hand, standard customers have to confirm the terms of payment and, based on the amount of the ordered goods, can be upgraded to premium users. On the other hand, the shipment to premium users is handled with priority. An actual implementation of this specification (e. g., in BPEL [5]) is straightforward.

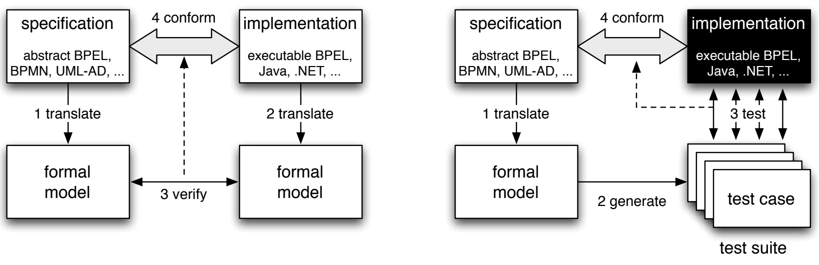
To interact correctly with the online shop, certain information does not need to be disclosed and Fig. 1(b) depicts a public view of the online shop. Non-communicating activities (depicted gray in the specification) as well as branching conditions were removed. This public view now describes the online shop's business protocol, but does not disclose unnecessary internal information. A customer can check whether his service (see Fig. 1(c)) fits to this public view.

The example shows that sheer mirroring of the public view's message flow is insufficient to achieve deadlock-free interaction, because the customer cannot influence the shop's decision whether to be recognized as standard or premium customer. Therefore, the customer must be ready to both receive the terms of payment *and* the invoice after he sent his order to the shop; in Fig. 1(c), this is expressed by an event-based exclusive gateway.

In the following, we concentrate on checking conformance between a specification and an implementation of a service; that is, we focus on the service provider's point of view who has access to both the specification and the implementation. Nevertheless, the approach is likewise applicable to check conformance between a public view and an implementation of this public view. This might be relevant for service brokers who want to assert conformance between their stored public views and the respective implemented services.

2.1 Verification vs. Testing

A widespread realization of services are *Web services* [6] which use WSDL to describe their interface and SOAP to exchange messages. While usually the specification language (e. g., BPMN, UML activity diagrams, EPCs) differs from the implementation language (e. g., Java, .NET languages), the Web Service Business Process Execution Language (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.



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

Fig. 2. Approaches to check conformance between specification and implementation.

One possibility to check conformance is *verification*, see Fig 2(a). Thereby, the implementation and its specification have to be translated into a formal model (1, 2). Conformance between the models (3) then implies conformance between specification and implementation (4). Obviously, this approach can only be applied if both models are available.

However, there are several scenarios in which a verification is impossible, because the formal model of the implementation is not available. 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.

To this end, we translate the specification into a formal model (1). This transformation should be possible, because the specification is 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 test fails, we can conclude that the implementation does not comply to the specification (4). Because the tester does not need knowledge about the code of the implementation, our approach is a *black-box testing* method.

Since a large number of test cases may be required for an adequate test, they should be automatically generated. This does not only save time and costs during the development, but furthermore a systematic approach can generate test cases which are hard to find manually. These test cases in turn allow for the detection of errors that were missed by manually designed test cases.

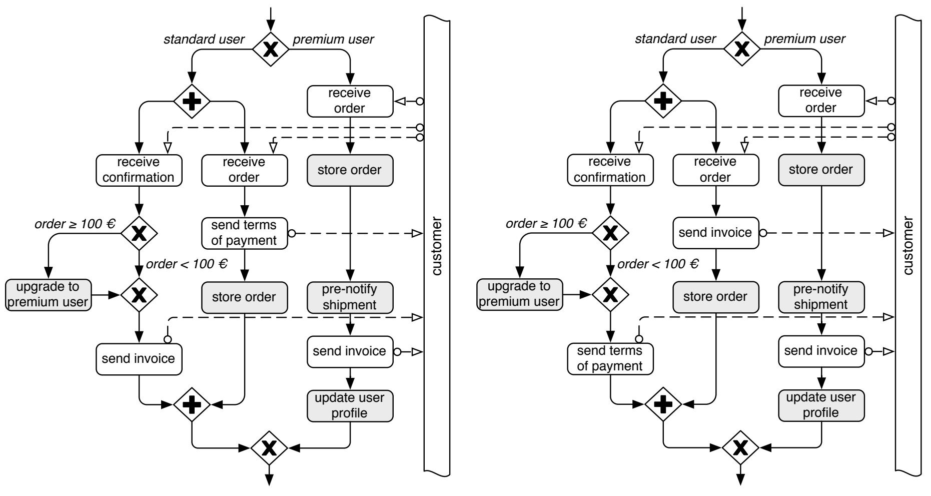
2.2 Testing Services by Partner Services

As motivated in the introduction, we propose to use services to test an implemented service. Considering the example online shop, any partner service of the specification of Fig. 1(a) can serve as one test case to test conformance of an implementation of the online shop. An example for a test case is the customer service depicted in Fig. 1(c). Its composition with the specification is free of deadlocks, and the same should hold for its composition with the implemented online shop.

A general test procedure for interacting services can be sketched as follows. The service under test is deployed in a testing environment together with the test suite. To process a whole test suite, the contained test cases are executed one after the other. Thereby, each test case follows the business protocol derived from the specification and interacts by message exchange with the service under test. The testing environment then is responsible for logging and evaluating exchanged messages. One implementation of such a test environment is the tool BPELUnit [7]. However, in BPELUnit the test suite has to be created manually.

Two implementations for the example online shop are depicted in Fig. 3.[1] Beside insertion of internal activities such as "store order", also reordering of activities does not necessarily jeopardize conformance [8]. For example, implementation in Fig. 3(a) receives the order and the confirmation message concurrently,

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



(a) conformant implementation (b) incorrect implementation

Fig. 3. Parts of two implementations of the online shop.

yet still is conformant to the specification in Fig. 1(a). In contrast, the implementation in Fig. 3(b) further exchanges the invoice and the terms of payment message. This implementation does not comply with the specification.

This inconformance can be detected by composing the test case of Fig. 1(c) to the implementation in Fig. 3(b). 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 test case after receiving the terms of payment. The services wait for each other; their 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.

The next section explains how test cases such as the customer service of Fig. 1(c) can be automatically generated.

3 Generating Test Cases

In this section, we present how test cases can be automatically generated given a service specification. Firstly, we describe how all test cases can be compactly characterized. Among the test cases, some are redundant; that is, their test result are already implied by other test cases. We conclude this section by defining some criteria when a test case can be considered redundant and show how this redundancy criterion can be used to minimize the generated test suite.

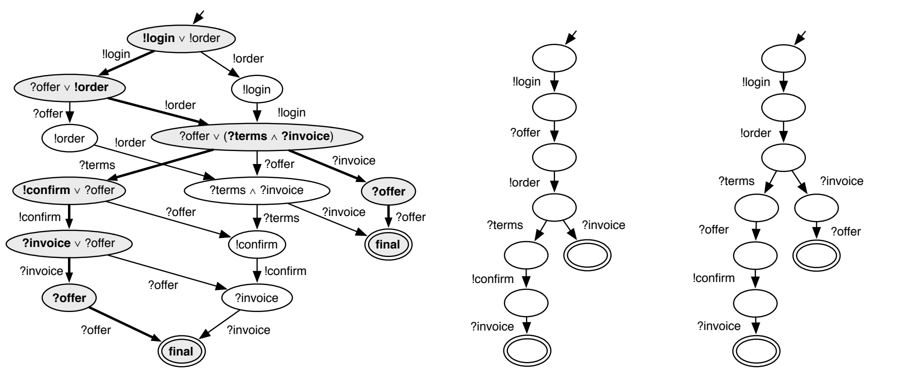
3.1 Characterizing Conformant Services

To formally reason about the behavior of a service, an exact mathematical model is needed. In this paper, we use *open nets* [9], a special class of Petri nets [10].

For BPEL, a formal semantics based on open nets was specified which allows to translate BPEL processes into open nets [11]. Furthermore, open nets can be translated back to BPEL processes [12]. As both translations are implemented[2], results in the area of open nets can be easily applied to real-life Web services.

A fundamental correctness criterion for services is *controllability* [13]. 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 [14]. 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* [15].



(a) OG of specification of Fig. 1(a) (b) a test case (c) another test case

Fig. 4. The operating guideline (OG) (a) describes all 1151 test cases, e. g., (b) and (c). The former can be used to show that the implementation in Fig. 3(b) does not comply to the specification of Fig. 1(a).

The operating guideline of the specification of Fig. 1(a) is depicted in Fig. 4(a). It is a finite state machine whose edges are labeled with messages sent to (preceded with "!") or received from (preceded with "?") the customer. The annotations describe for each state which outgoing edges have to be present (see [15] for further details). As mentioned earlier, subgraphs of the operating guideline are only partners if the states also fulfil the respective annotations. One example for a characterized partner is the subgraph consisting of the gray nodes and the bold edges, also depicted in Fig. 4(b). This partner describes the behavior of the customer service of Fig. 1(c). Another test case is depicted in Fig. 4(c). In total, the operating guideline characterizes 1151 partners, i.e. 1151 rooted subgraphs fulfilling the annotations (see [15] for details).

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

The operating guideline of a specified service exactly characterizes the set of conformant services of this specification. Any service that is characterized by that operating guideline must also interact deadlock-free with the implementation and can therefore be seen as a test case. Using the operating guideline of the specification of a service as a characterization of the test cases postulates the requirement that the specification is controllable. If it is uncontrollable, then *any* interaction will eventually deadlock and no (positive) test cases exists. In this case, a diagnosis algorithm [16] can help to overwork the specification.

3.2 Selecting Test Cases

The operating guideline of the specification characterizes all necessary test cases to test conformance of an implementation with its specification. Unfortunately, even for small services such as the example online shop, there already exist thousands of test cases, each describing a possible customer service derived from the specification/public view. However, there are some *redundant* test cases which can be left out without reducing the quality of the test suite. Thus, even with a limited number of test cases, it is possible to detect all errors that could also be detected with the complete set of test cases.

We define redundancy as follows: A partner service T characterized by the operating guideline is a redundant test case if (1) there exists a set of partner services $T_1, \dots, T_n$ such that each T_i is a proper subgraph of T, and (2) the union of these graphs equals T; that is, $T_1 \cup \dots \cup T_n = T$. We can generate the test cases for the reduced test suite directly by decomposing the most permissive partner (i.e., the operating guideline without the Boolean formulae) into several smaller partner services $T_1, \dots, T_n$, so that each partner service characterized by the operating guideline is redundant to a subset of test cases containing in the test suite. This can be realized by one depth-first search on the most permissive partner and by exploiting the operating guideline's annotations.

We illustrate the procedure on the operating guideline in Fig. 4(a). The annotation "$\mathsf{!login} \vee \mathsf{!order}$" of the initial node demands that any test case has to start with a !login action (as the service in Fig. 4(b)), an !order action (cf. Fig. 4(c)), or both. Thereby, the service containing both actions can choose which message to send on (test) runtime. This is not desirable, because such a test case has to be executed twice: once for each action. In such a case, we can move this runtime decision to the design time of the test case and only consider those services consisting of one of the respective actions. Hence, the service with the choice between the two actions can be considered as redundant, because its behavior is covered by the other two services. Consequently, we have two kinds of test cases: ones starting with a !login action and another one starting with an !order action. By following the !login edge in the operating guideline we reach a node annotated with "$\mathsf{?offer} \vee \mathsf{!order}$". Thus we can refine the first kind of test case to those which either start with a !login followed by an ?offer or start with a !login followed by an !order. With the remaining nodes we proceed in the same manner. Finally, we will detect only nine test cases which are necessary to test the conformance of the implementation to its specification. The test cases are depicted in Fig. 5. All other test cases characterized by the operating guideline are redundant to these nine test cases. Thus the minimized test suite is complete in the sense that it is still able to detect all errors that could have been detected by all 1151 test cases.

Note our notion of redundancy will not split conjunctions. This is important as they model choices that cannot be influenced by the test case. For instance, a customer of the online shop of Fig. 1(a) cannot influence whether or not he is recognized as a premium customer and therefore must be able to receive both the terms of payment or immediately the invoice (cf. Fig. 4(b) and 4(c)).

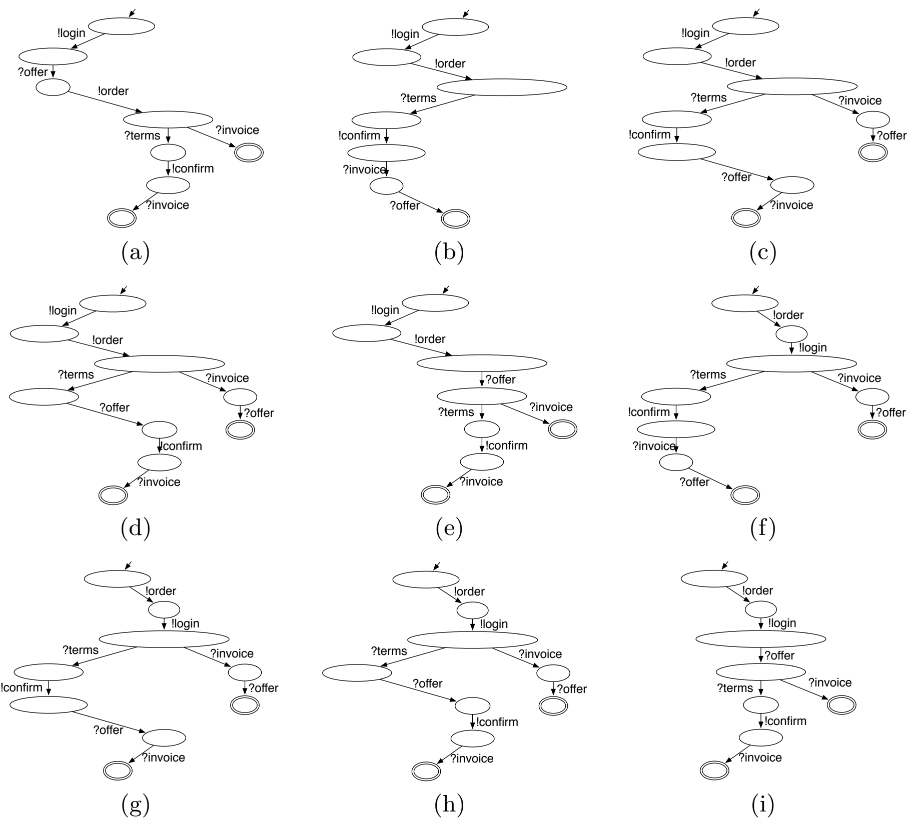


Fig. 5. Minimal test suite consisting of nine non-redundant test cases. Case (a) coincides with Fig. 4(b) and case (d) coincides with Fig. 4(c).

If the specification is given as an abstract BPEL service, an existing tool chain[3] is directly applicable to translate the specification into an open net and to calculate the operating guideline thereof (cf. [14]). Furthermore, the test cases can be automatically translated into BPEL processes [12]. The resulting test Web services can then be executed on a BPEL engine to test the implementation under consideration.

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

4 Related Work

Several works exist to systematize testing of Web services (see [17] for an overview). In [18, 7], 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.

Test case generation can be tackled using a variety of approaches such as model checking [19], XML schemas [20], control or data flow graphs [21, 22]. These approaches mainly focus on code coverage, but do not take the interacting 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 be used for black-box testing in which only the interface of the service under test is available.

To the best of our knowledge, none of the existing testing approaches take stateful business protocols implemented by Web services into account, but mainly assume stateless remote procedure calls (i. e., request-response operations), see for instance [20].

Bertolino et al. [23, 24] present an approach to generate test suites for Web services and also consider nonfunctional properties (QoS). Their approach, however, bases on synchronous communication and is thus not applicable in the setting of SOAs.

Finally, several tools such as the Oracle BPEL Process Manager [25] or Parasoft SOAtest [26] support testing of services, but do not specifically focus on the business protocol of a service. In contrast, Mayer et al. [7] underline the importance of respecting the business protocol during testing, but do not support the generation of test cases in their tool BPELUnit.

5 Conclusion

We presented an approach to automatically generate test cases for services. These test cases are derived from an operating guideline of a specification and can be used to test conformance of an implementation without explicit knowledge of the latter (black-box testing). We also introduced a notion of redundant test cases to drastically reduce the size of the test suite while still ensuring completeness with respect to the specified business protocol. The approach bases on open nets as formal model, and can be applied to any specification language to which a translation into open nets exists. Existing translations from BPMN [27] or UML-AD [28] into Petri nets are likely to be adjustable to open nets easily.

In this paper, we focused on business protocols in which data plays a secondary role. However, existing works such as [29, 30] show that it is principally possible to add those data aspects into a formal model. As a result, we can refine not only the open net, but also the resulting operating guideline. This in turn might further reduce the number of necessary test cases as nondeterministic choices are replaced by data-driven decisions.

In future work, we also want to consider *negative test cases* to test the absence of unintended partners. Such negative test cases can be similarly derived from the operating guideline. For example, annotations can be wilfully left unsatisfied by a test case to "provoke" deadlocks during a test. Furthermore, we plan to validate our test case generation in a BPEL test architecture such as [18, 7] using real-life case studies.

Beside black-box testing, the service test approach is also applicable to several other settings. Firstly, the test case generation can support *isolated testing* of services. In this case, we can use the specification to not only derive test cases, but also to synthesize stub implementation of third-party services. These stub implementations follow the business protocol as specified by the specification, but avoid possible resulting costs of third-party calls. Isolated testing might also help to locate occurring errors more easily, because we can rely on the synthesized mock implementations to conform to the business protocol.

Secondly, the completeness of the operating guideline can be used to support *substitutability* of services. When implementation I of a service is substituted by a new implementation I', we can use the same test cases to check conformance of I with the specification to check conformance of I' with that specification. In case a test fails, we can conclude that I' should not be replaced by I'.

A third scenario are inter-organizational business processes. In this setting, several parties agree on a *contract* which can be partitioned into several services. Each service then is implemented by a party. Then, conformance between the contract (a special kind of a public view) and the implementation can be tested for each party. Additionally, a party can use the global contract to derive test cases to test whether another party's implementation conforms to the contract.

Acknowledgements Niels Lohmann is funded by the DFG project "Operating Guidelines for Services" (WO 1466/8-1).

References

1. Papazoglou, M.P.: Agent-oriented technology in support of e-business. Communications of the ACM **44**(4) (2001) 71–77
2. Martens, A.: Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services. PhD thesis, Humboldt-Universität zu Berlin, Berlin, Germany (2003) in German.
3. Papazoglou, M.P.: Web services: Principles and technology, Prentice Hall (2007)
4. OMG: Business Process Modeling Notation (BPMN) Specification. Final Adopted Specification, Object Management Group (2006) `http://www.bpmn.org`.
5. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. Committee Specification, OASIS (2007)
6. Curbera, F., Leymann, F., Storey, T., Ferguson, D., Weerawarana, S.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR (2005)
7. Mayer, P., Lübke, D.: Towards a BPEL unit testing framework. In: TAV-WEB '06, ACM (2006) 33–42
8. König, D., Lohmann, N., Moser, S., Stahl, C., Wolf, K.: Extending the compatibility notion for abstract WS-BPEL processes. In: WWW 2008, ACM (2008)

9. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
10. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985)
11. 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
12. 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)
13. Wolf, K.: Does my service have partners? Transactions on Petri Nets and Other Models of Concurrency (2008) (Accepted for publication, preprint available at `http://www.informatik.uni-rostock.de/~nl/topnoc.pdf`).
14. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: BPM 2006. Volume 4102 of LNCS., Springer (2006) 17–32
15. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. Volume 4546 of LNCS., Springer (2007) 321–341
16. Lohmann, N.: Why does my service have no partners? In: WS-FM 2008. LNCS, Springer (2008)
17. Baresi, L., Nitto, E.D., eds.: Test and Analysis of Web Services. Springer (2007)
18. Li, Z., Sun, W., Jiang, Z.B., Zhang, X.: BPEL4WS unit testing: Framework and implementation. In: ICWS 2005, IEEE (2005) 103–110
19. 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
20. Hanna, S., Munro, M.: An approach for specification-based test case generation for Web services. In: AICCSA, IEEE (2007) 16–23
21. 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
22. Bartolini, C., Bertolino, A., Marchetti, E., Parissis, I.: Data flow-based validation of web services compositions: Perspectives and examples. In: WADS 2007. (2007) 298–325
23. Bertolino, A., Angelis, G.D., Frantzen, L., Polini, A.: Model-based generation of testbeds for web services. In: TESTCOM/FATES 2008. Volume 5047 of LNCS., Springer (2008) 266–282
24. Bertolino, A., Angelis, G.D., Lonetti, F., Sabetta, A.: Let the puppets move! automated testbed generation for service-oriented mobile applications. In: SEAA 2008. IEEE (2008) 11–19
25. Oracle: BPEL Process Manager. (2008) `http://www.oracle.com/technology/products/ias/bpel`.
26. Parasoft: SOAtest. (2008) `http://www.parasoft.com`.
27. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Information & Software Technology (2008) accepted for publication.
28. Störrle, H.: Semantics and verification of data flow in UML 2.0 activities. Electr. Notes Theor. Comput. Sci. **127**(4) (2005) 35–52
29. Sharygina, N., Kröning, D.: Model checking with abstraction for Web services. In: Test and Analysis of Web Services. Springer (2007) 121–145
30. Moser, S., Martens, A., Görlach, K., Amme, W., Godlinski, A.: Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. In: SCC 2007, IEEE (2007) 98–105