Does my service have unspecified behavior?

Kathrin Kaschner and Niels Lohmann

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

Abstract. Services are loosely coupled interacting software components. Since two or more services are usually composed to one software system, the behavior of an implemented service should not differ to its specification. Therefore we propose an approach to test, if the implementation contains unspecified behavior. Due to the interacting nature of services this is a nontrivial task.

1 Introduction

In the paradigm of service-oriented computing (SOC) [1], *services* are encapsulated, self-contained functionalities. Usually, they are not executed in isolation, but interact with each other via a well-defined interface. Thereby, the interaction between services follows complex *protocols* and goes far beyond simple remote-procedure calls (i. e., request/response operations). Arbitrary complex and possibly stateful interactions on top of asynchronous message exchange are common. Finally, a system can be *composed* by a set of services.

Best practices propose that systems should be *specified* prior to their implementation. A specification of a service (e. g., an abstract WS-BPEL process) describes its interface and the possible interactions with other services. Furthermore it already contains all relevant internal decisions of the control flow (e. g., the criteria whether or not a credit should be approved) and data about nonfunctional properties. In contrast, implementation-specific details (e. g., whether amounts are represented as integers or floats) are usually not yet defined.

Ideally, an implementation should *conform* to its specification. This means, all specified properties should be implemented. But it is obviously not always possible to verify the conformance in a formal proof. Furthermore verification is excessively elaborate and therewith often too expensive. Thus, it is customary to *test* the implementation.

As Dijkstra stated, program testing can be a very effective way to show the *presence* of bugs, but is hopelessly inadequate for showing 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.

Testing 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 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. Both might lead to undetected errors.

However, communication is an essential part of the specification of interacting services and consequently it is crucial to check it during conformance testing. That means, any interaction derived from the specification also has to be a correct interaction of the implementation *and* vice versa. In [2], we presented an approach to generate a test suite to test the former. In this paper, we propose how to test if undesired (i.e., unspecified) interactions are implemented in the service. Because the tester does not need knowledge about the code of the implementation, our approach is a *black-box testing* method.

This paper is organized as follows. Section 2 describes how a service implementation can be tested using partners. Section 3 recalls the approach to generate test cases and introduces an example protocol we use to demonstrate the approach. Section 4 provides the main contribution of this paper with the elaboration of test cases that are able to detect unspecified behavior of an implementation. Section 5 concludes the paper and gives directions to future work.

2 Testing interacting services

An adequate software test needs to check all specified properties. Due to the fact that protocols are an essential part of service specifications, they likewise have to be considered during service testing. A protocol describes in which state a service can send or receive which messages (and also which messages are not allowed in that state). In this way, all possible interactions with other services are defined which leads to an implicitly specified set $\mathcal{S}$ of all *partners*. Thereby a partner is again a service. If the specified service is implemented correctly, a partner always interacts deadlock-freely with it. In contrast *non-partners* violate the protocol; that is, during interaction the implemented service might deadlock. For this paper, we assume “correct interaction” between two services means deadlock freedom of their composition. We are well aware that there are other possibilities for defining “correct interaction”. Nevertheless, deadlock freedom will certainly be art of any definition, so this paper can be seen as a step towards a more sophisticated setting.

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 a given service [2]. Thus, testing the protocol means testing whether the implemented service does not exclude specified partners, and also whether it does not include undesired partners. The former is important, because partner exclusion might have a financial impact if the service implements a business process — loosing a partner might result in deadlocks which again yields to down-time and contractual penalties. Testing the latter aims to exclude that additional behavior is implemented. For instance, an online shop should not terminate successfully if a customer did not pay for received goods. Consequently, we distinguish two kinds of test cases: services $P \in \mathcal{S}$ (partners), from which we expect a deadlock-free interaction with the implementation, and services $Q \notin \mathcal{S}$ (non-partners), from which we expect the interaction does not terminate successfully (e.g., a deadlock will occur). Figure 1 illustrates our idea of exploiting the properties of the partners and non-partners for testing services. Ideally, the implementation contains exactly the specified behavior, that means, that $\mathcal{S}$ should be equal to the set $\mathcal{I}$ of correct interacting partners of the implementation ($\mathcal{S} = \mathcal{I}$). Otherwise, the implementation excludes a specified partner ($\mathcal{S} \backslash \mathcal{I} \neq \emptyset$) or the implementation is able to terminate successfully with a non-partner ($\mathcal{I} \backslash \mathcal{S} \neq \emptyset$).

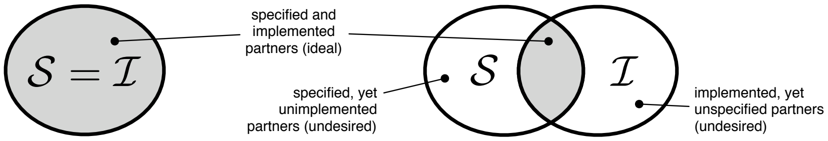

$\mathcal{S} = \mathcal{I}$
specified and implemented partners (ideal)
specified, yet unimplemented partners (undesired)
$\mathcal{S}$
$\mathcal{I}$
implemented, yet unspecified partners (undesired)

Fig. 1. Possible relations between the specified and implemented behavior $\mathcal{S}$ and $\mathcal{I}$.

To calculate $\mathcal{S}$ we can transform the specification into a formal model from which a description (operating guideline) of all partners can be generated [3]. Unfortunately, the set $\mathcal{I}$ cannot be calculated in the setting of black box testing. Thus, we need to test how $\mathcal{I}$ behaves with respect to $\mathcal{S}$.

The general test procedure for interacting services can be sketched as follows. We have a test suite containing the test cases $P \in \mathcal{S}$ and another one containing the test cases $Q \notin \mathcal{S}$. The service to be tested is deployed in a testing environment together with the two test suites. To process both test suites, the contained test cases are executed one after the other. Thereby, each test case interacts by message exchange with the service under test. The testing environment then is responsible for logging and evaluating exchanged messages. We thereby assume the test environment is able to detect whether the implementation terminates properly; for instance, whether a BPEL process instance has completed successfully.

A test run with a service $P \in \mathcal{S}$ fails if the test run terminates, but the implementation did not reach a final state. Then a deadlock has occurred and the implementation is incorrect. P can serve as a witness that the implementation excludes at least this specified partner service ($\mathcal{S} \backslash \mathcal{I} \neq \emptyset$). A test run that does not terminate, or terminates, but leaves the implementation in a final state, is inconclusive; that is, the run neither witnesses an error nor can proof overall correctness. Note that our approach is currently centered around deadlock freedom. In this setting, infinite runs are considered correct, because they do not contain a deadlock. Furthermore, the test environment cannot detect infinite runs and has to abort the test at some point, yielding an inconclusive result. But even if, for each partner service $P \in \mathcal{S}$, the run terminates and the implementation reaches a final state, we cannot conclude that the implementation does not exclude a specified partner service. We only know, that there *exists* a successful run for each specified partner. Since during testing the partner service (the tester) cannot control all internal decisions of the service. Hence, an incorrect part of the implementation may not executed during the test at all. However, a systematic test approach with a significant number of test cases can increase the likelihood of detecting an error.

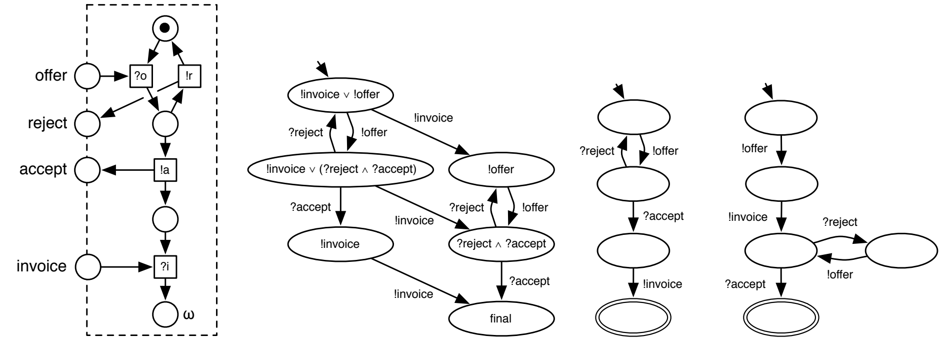
The set $\mathcal{S}$ contains a large number of many services (usually even infinitely many). It is therefore not practical to test a service with *every* $P \in \mathcal{S}$. In [2], we presented an approach how a small subset of $\mathcal{S}$ can be selected. With this reduced test suite, it is possible to detect all errors that could also be detected with the complete set $\mathcal{S}$.

Canonically, a test run with a non-partner service $Q \notin \mathcal{S}$ fails, if the test run terminates and the implementation is in a final state. Thereby, Q is a witness that the implementation supports undesired partner services ($\mathcal{I} \backslash \mathcal{S} \neq \emptyset$). Test runs of the composition of the implementation and the test case that neither terminate nor leave the implementation in a final state are inconclusive.

In Sect. 4 we will explain how the non-partner can be derived from the specification. This is a nontrivial task, because not all partner services $Q \notin \mathcal{S}$ are suitable for testing. As demonstrated by the following example, there exist some $Q' \notin \mathcal{S}$ which can fail even trough the implementation is correct.

3 Example

To formally reason about the specification of a service, an exact mathematical model is needed. We use *open nets* [4], a special class of Petri nets [5] as a formalism for protocols. Using existing translations, they can be easily derived from industrial languages such as WS-BPEL or BPMN. As running example for this paper, consider the open net from Fig. 2(a). It models a protocol of a buying service which receives and evaluate offers. It either accepts an offer and waits for an invoice, or rejects the offer and returns to its initial state. The final marking of this net is the marking $[\omega]$ which only marks the place ω: the control flow reached its end and all message channels are empty. Final markings distinguish desired end states from undesired deadlocks.



(a) specified protocol (b) operating guideline (c) test case (d) test case

Fig. 2. A buyer protocol modeled as an open Petri net (a). Its operating guideline (b) characterizes $\mathcal{I} \backslash \mathcal{S}$-test cases, for instance the seller services (d) and (e).

From this model, an *operating guideline* (OG) [3] can be calculated (see Fig. 2(b)). An operating guideline is a finite automaton whose edges are labeled with message events (sending events are preceded by ! and receiving events are preceded by ?) and whose states are annotated with Boolean formulae. These formulae express which edges of a partner must be present to guarantee deadlock freedom. For example, "!invoice $\vee$!offer" expresses the requirement that every partner has to initially send an invoice or an offer, or make a decision between both during runtime. Likewise, "?reject $\wedge$?accept" states that the receipt of *both* messages must be possible in that state. The operating guideline finitely characterizes the infinite set of all deadlock-free interacting partners $\mathcal{S}$ of the buying service and hence all test cases for the specified protocol.

Two test cases are depicted in Fig. 2(c) and 2(d). While the first test case sends the invoice only after receipt of an acceptance message, the second exploits asynchronicity and sends the invoice right after sending the offer. In both cases, the OG's annotations are fulfilled.

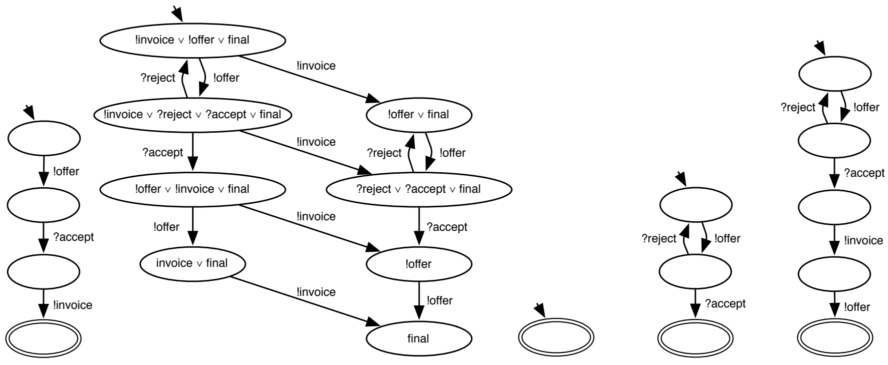
4 Testing for unspecified behavior

As mentioned earlier, the operating guideline can used to generate the test suite containing all necessary partners the test if $\mathcal{S} \backslash \mathcal{I} = \emptyset$ holds. To be able to make a statement about these $\mathcal{I} \backslash \mathcal{S}$ deviations (i.e., unspecified partners), a different approach is needed. This is due to the fact that a service that is not characterized by the does not *necessarily* yield a deadlock. For example, a service without the ability to receive a rejection message (see Fig. 3(a)) may deadlock with the buyer service if the latter rejects the offer. If, however, the offer is accepted, no deadlock occurs. Hence, any result of this test case would be inconclusive.

To this end, the operating guideline of the specified protocol cannot be used to derive $\mathcal{I} \backslash \mathcal{S}$-test cases. To characterize all services that are expected to deadlock when composed to the implementation, we need to reinterpret the final states of the protocol: If we complement the set of final markings, then the operating guideline of this *anti-protocol* (called the *anti operating guideline*) characterizes partners that interact without deadlock with the anti-protocol. That is; the interaction terminates in a final state of the anti-protocol. This in turn is a deadlock of the original protocol. Hence, the anti operating guideline characterizes partners that are expected to deadlock with the original protocol.

For the example of Fig. 2(a), any marking but $[\omega]$ would be a final marking of the anti-protocol. Note that the structure of the anti-operating guideline of Fig. 2(b) is very similar to the original operating guideline (cf. Fig. 3(b)) and we see that the interactions that would lead to the a final state in Fig. 2(b) (e.g. !offer ?accept !invoice) lead to the only state without final in the annotation in Fig. 3(b).

Three test cases characterized by the anti operating guideline are depicted in Fig. 3(c)–3(e): the first two exploit the fact that premature termination will lead to a deadlock with a conformant implementation, whereas the last test case avoids a valid final marking by sending another offer.



(a) (b) anti opereating guideline (c) test (d) test (e) test

Fig. 3. The service (a) is not characterized by the operating guideline (cf. Fig. 2(b)), but does not necessarily deadlock with the protocol from Fig. 2(a). The anti operating guideline (b) for the protocol characterizes $\mathcal{S}\backslash\mathcal{I}$-test cases. For example, communication should deadlock if a seller sends no message (c), aborts the interaction prematurely (d) or sends too many messages (e).

5 Related Work

Several works exist to systematize testing of Web services (see [6] for an overview). Test case generation can be tackled using a variety of approaches such as control flow graphs [7], model checking [8], or XML schemas [9]. 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 [9]. Bertolino et al. [10] 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 [11] or Parasoft SOAtest [12] support testing of services, but do not specifically focus on the business protocol of a service. In contrast, Mayer et al. [13] underline the importance of respecting the business protocol during testing, but do not support the generation of test cases in their tool BPELUnit.

6 Conclusion

We presented an approach to generate test cases from a specification protocol and show how they can be used to check whether an implementation introduces unspecified behavior. This test suite is not complete in the sense that we cannot detect all deviations from the specification. The reason for this lies in the nature of services in which decisions may not always be enforceable by the environment. To this end, we introduced anti operating guidelines to characterize $\mathcal{S}\backslash\mathcal{I}$ test cases that must deadlock when composed to a compliant implementation. The results of this paper are defined in terms of open nets and are independent of a concrete protocol specification language. With existing translation, e.g. from BPEL to open nets and vice versa [14, 15], the approach is easily applicable to industrial languages.

In future work, we plan to propose a test case coverage criterion to select a small number of test cases which still can detect the same errors that are detectable with the complete set of test cases. Indications are that this criterion should be similar to the criterion we proposed for $\mathcal{I}\backslash\mathcal{S}$ test cases [2]. Finally, the test case selection procedures need to be implemented to validate the test suite generation with industrial case studies.

References

1. Papazoglou, M.P.: Agent-oriented technology in support of e-business. Communications of the ACM **44**(4) (2001) 71–77
2. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: ICSOC Workshops 2008. LNCS, Springer (2009)
3. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. Volume 4546 of LNCS., Springer (2007) 321–341
4. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
5. Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985)
6. Baresi, L., Nitto, E.D., eds.: Test and Analysis of Web Services. Springer (2007)
7. 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
8. 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
9. Hanna, S., Munro, M.: An approach for specification-based test case generation for Web services. In: AICCSA, IEEE (2007) 16–23
10. Bertolino, A., Angelis, G.D., Frantzen, L., Polini, A.: Model-based generation of testbeds for web services. In: TESTCOM/FATES 2008. LNCS 5047, Springer (2008) 266–282
11. Oracle: BPEL Process Manager. (2008) `http://www.oracle.com/technology/products/ias/bpel`.
12. Parasoft: SOAtest. (2008) `http://www.parasoft.com`.
13. Mayer, P., Lübke, D.: Towards a BPEL unit testing framework. In: TAV-WEB '06, ACM (2006) 33–42
14. 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
15. 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)