Managing test suites for services

Kathrin Kaschner

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

Abstract. When developing an existing *service* further, new functionalities can be added and existing functionalities can be changed or removed. Consequently, also test cases have to be added to or removed from the existing test suite. In this paper, we present an idea how a test suite can be adjusted to these changes. Thereby, we focus on modifications concerning the *communication behavior* of a service.

1 Introduction

Testing is an effective instrument to detect errors in software. However, for thorough testing the number of required test cases increases rapidly with growing software complexity. To reduce the effort for testing, test cases are generated and executed by tool support as much as possible.

In the paradigm of *service-oriented computing* (SOC) [1], modern software systems are composed by a set of loosely-coupled and possibly geographic distributed *services*. Each service implements an encapsulated, self-contained functionality and communicates via message exchange over a well-defined interface with its *partner services*. In earlier work [2] we presented a black box testing approach to test the *communication behavior* for a single service.

The development of a service is rarely finished with its initial release. In most cases, maintenance follows. This includes both, correction (i.e., fixing of discovered bugs) and integration of enhancements and new features. To adequately test the new service version, also the test suite must be revised: New test cases have to be added and existing test cases may be removed. Using the updated test suite, all (new) parts of the new version are taken into account during testing. Further, one can be sure that a failed test case indicates a bug in the implementation and is not caused by the specified changes.

In this paper, we present an idea how a test suite, generated by our approach [2], can be updated. For that purpose, unsuitable test cases are removed directly from the test suite and test cases for new functionalities are created selectively. Consequently, it is not necessary to again generate the complete test suite for a new service version. This can be useful, because there are still some small manual steps involved which do not need to be repeated for those test cases that can stay in the test suite. Especially, if only small changes are specified and most parts of the test cases can remain in the test suite the effort for test case generation can be reduced with this idea.

The rest of this paper is organized as follows: Section 2 introduces basic formalisms. Section 3 recalls the testing approach of [2]. Section 4 describes our idea to update a test suite for a new service version. In Sect. 5, open issues are discussed, before Sect. 6 concludes the paper.

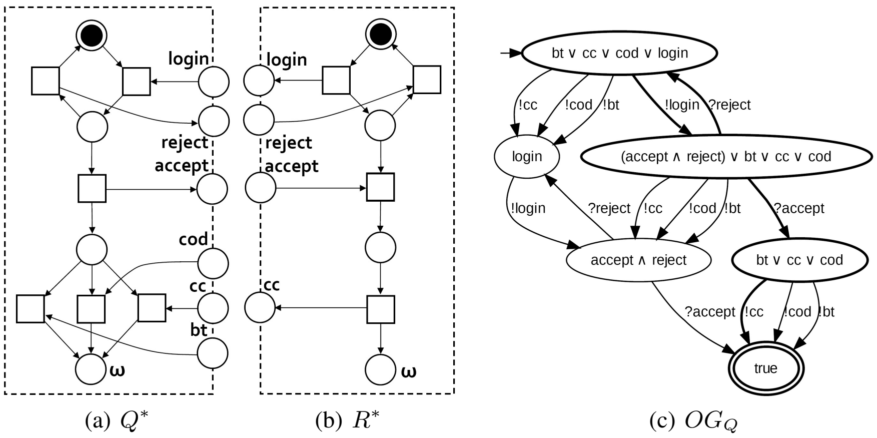
2 Formalization

Black box testing [3,4] means that test cases are created without consideration of the actual code. For an automatic generation of a test case, a formal specification is indispensable. To model the specified communication behavior of a service we use open nets [5], a class of Petri nets. Thereby, the interface is specified by special input and output places which represent the possible message types. In the model, we abstract from the content of messages and focus on sending and receiving events only. They are represented by transitions producing or consuming a token on the interface places of the open net. Furthermore, an open net has an initial and a final marking. A final marking distinguish desired end states from undesired deadlocks. To emphasize the data abstraction we also speak of an *abstract specification* S^* for a given service S.

Figure 1(a) shows the abstract specification Q^* of a simple online shop Q. The shop expects a costumer to log in and to choose one of the payment methods: cash on delivery (cod), credit card (cc), or bank transfer (bt). However, for setting the payment method the login must be accepted first, otherwise the login is rejected and the user has to retry to login. 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.

A *partner* P for a given service S is again a service that interacts deadlock freely with S. That means, no deadlocks (except the final state) can occur in the composition of the both services. To model the communication behavior of a partner P, we use again open nets. Since we also abstract from the data, the net models an *abstract partner* P^* of P.

As an example we consider a partner R for the service Q (cf. Fig. 1(a)). Its abstract version R^* is depicted in Fig.1(b). To check deadlock freendom between Q and R it is sufficient to compose their abstract versions Q^* and R^*. For this purpose, we only need to merge interface places with the same label. In the composed system, these merged



(a) Q^* (b) R^* (c) OG_Q

Fig. 1: The open net Q^* in (a) specifies the communication behavior of a simple online shop. In (b) an abstract partner of Q^* is shown and in (c) the operating guideline OG_Q derived from Q^* is depicted.

interfaces place become to ordinary places. In our example the composition of Q^* and R^* is free of deadlocks (except the final marking).

The set of *all* partners of a given service S can be characterized by its operating guideline OG_S [6]. Formally, an operating guideline is an *annotated service automaton*. That is a finite state automaton which edges are labeled with sending or receiving events and states are annotated with a Boolean formula. The labels determine which events of the partner may occur in a certain situation and the Boolean formulas defines which combinations thereof are allowed. Each service P belonging to the set characterized by an operating guideline OG_S fulfills the following requirements: First, the reachability graph of the abstract version P^* is a subgraph of OG_S (including an initial state). Further, each node n of the reachability graph satisfies the Boolean formula of its corresponding state in OG_S. The interested reader is referred to [6]. If a services P is characterized by an operating guideline OG_S then we also say: P *matches* with OG_S. Operating guidelines can be generated from abstract specifications using the tool Wendy[1] [7].

Figure 1(c) depicts the operating guideline OG_{Q^*} of the abstract specification Q^* (see Fig. 1(a)). In the graphical representation of operating guidelines, sending events are preceded by "!" and receiving events are preceded by "?". Initial states have an incoming arc from nowhere. Final states are double-lined. Since the reachability graph of R^* is a subgraph of OG_{Q^*} (see the bolded part of OG_{Q^*}) and the Boolean formulas of the "touched" states are fulfilled, R is characterized by OG_{Q^*}; that is, R is a partner of Q.

3 Testing Communication Behavior

To test whether the implemented service interacts correctly with its environment we create partner services as test cases. As they are derived from the specification they interact by design deadlock freely with the service under test (SUT). If, however, during testing a deadlock still occurs, we can conclude that the implementation contains (at least) one error.

The generation of test cases for a service S bases on the operating guideline OG_S. As there is usually a high number of abstract partners characterized by OG_S, it is not recommended to take every partner for testing. But among the test cases there is some redundancy, such that is possible to to select a subset without reducing the test suite's quality [2]. That means, it is still possible to find all errors, that can be found using the whole set of (abstract) partners. After we have selected the required abstract partners from the operating guideline we have to fill the message content with test data, and finally, transform them into "real" services. The latter can be done automatically by tool support, e.g. by oWFN2BPEL[2] [8] for generating a BPEL process. Data are not integrated into the concept of operating guidelines at the current state. However, data for sending messages can be generated randomly and then easily added to an abstract test case. Therefore, only the message types need to be respected. More sophisticated data have to create manually.

[1] Available at `http://service-technology.org/wendy`

[2] Available at `http://service-technology.org/owfn2bpel`

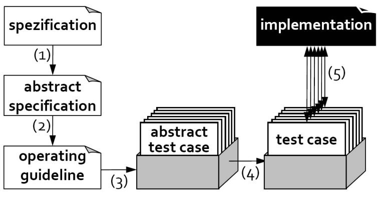


Fig. 2: A graphical overview of the general procedure for generating test cases.

Figure 2 gives again a graphical overview of the general procedure of generating test cases. First, the service specification is transformed (1) into a formal model. Based on this abstract specification the operating guideline is constructed (2) and a set of abstract partners is selected (3). To achieve executable test cases, the abstract partners are enriched with data information and retranslated into real services (4). The actual test procedure (5) can be finally sketched as follows: The service to be tested is deployed in a testing environment together with the test suite. To process the test suite, the contained test cases are executed one after the other. Thereby, each test case interacts as designed 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. Except the generation of test data in (4), all steps can be executed automatically.

4 Adjust existing test suites to new specifications

As already mentioned in the introduction, it can be required to modify an existing service from time to time. By introducing enhancements and new features the specification is changed. This entails existing test cases to become invalid. That makes the test suite inconclusive. Further, after adding new functionalities, there are parts in the new version of the service which are not taken into account by the old test suite.

To ensure a thorough testing, the test suite needs to be revised. Instead of discarding all old test cases and deriving a new test suite from the scratch, we aim to keep as much old test cases as possible. This is desirable because the procedure for generating the test case (see Sect. 3) is not fully automatic. Thus, this strategy helps to reduce the manual effort. Further, it is reasonable to use the same data in the remaining test cases. By generating test cases from scratch, these data information would be lost.

First, we describe in the following how invalid test cases can be removed from the test suite. Afterward, we suggest how the new test cases can be found.

Making the test suite conclusive. Integrating new functionalities can change the control flow and with it the communication behavior such that existing test cases become invalid. This is demonstrated by the following example. Figure 3 shows a modification Q^*_{new} for the specification in Fig. 1(a): If the credit card (cc) is chosen as payment method, a voucher is sent to the customer. Now, the composition of Q^*_{new} and R^* (see Fig. 1(b)) contains a deadlock because the voucher cannot be received from R^*. When testing the new version of the simple online shop with R the test fails even through the

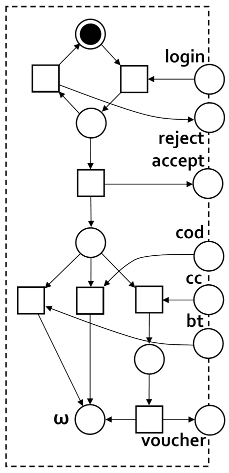


Fig. 3: An modified specification Q^*_{new} for the simple online shop, cf. Fig.1(a).

implementation is correct. In converse, if the implementation is incorrect the test can succeed.

To achieve a conclusive test suite for the new specification we discard the invalid test cases. They can be detected using the matching algorithm for operating guidelines (cf. Sect. 2). For each test case we check whether it matches with the operating guideline of the *new* specification. This can be done using the tool Cosme[3]. Non-matching test cases are invalid and have to be removed. After this procedure we have a conclusive test suite.

Adding missing test cases. To cover new parts during testing, test cases have to be added to the test suite. For a systematic approach we make the following consideration: Let OG_S be the operating guideline of a specification and $OG_{S_{new}}$ the operating guideline of the new specification. Let T_S be the set of test cases characterized of OG_S and $T_{S_{new}}$ be the set of the test cases characterized by $OG_{S_{new}}$. When calculating the difference $T_{S_{new}} \setminus T_S$ we get exactly the set of test cases for testing the new parts.

Unfortunately, the result of the difference operation cannot be represented as an operating guideline. Instead, it can be represented as an *extended annotated service automaton* (EAA for short) [9]. This kind of finite automata extends annotated service automata (cf. Sect. 2) by a global Boolean formula. It constrains the combinations of states in the EAA that have to be "touched" during matching. Further, the structure of the Boolean formulas in the states is not restricted as much as in an operating guideline.

Formally, an operating guideline can easily be transformed into an EAA by setting the global Boolean formula to $true$; that is, there are no additional requirements for the states. For EAAs all basic set operations (including the difference) are defined [9] and supported by the tool Safira[4] [10].

Thus, we are able to calculate the set $T_{S_{new}} \setminus T_S$ based from the corresponding operating guidelines. The result is an EAA that characterizes the test cases to be added.

[3] Available at `http://service-technology.org/cosme`

[4] Available at `http://service-technology.org/safira`

In the next step, the (abstract) test cases needs to extract from the EAA and transformed into executable test cases.

5 Future Work

The set of test cases to be added, can be only represented by EAAs, an extension of operating guidelines. In contrast to operating guidelines, the extraction of test cases from an EAA is a time-consuming task. In [9] we proved that this problem is NP-complete in general. This is caused by the more complex Boolean formula (in comparison to the operating guidelines). In future work, we intend to find a good heuristic such that the selection is practicable for EAAs. Thereby, it can be useful that the EAA is resulted by applying the difference operation to two operating guidelines. Thus, the boolean formulas in the result are less complex, than it could be in an arbitrary EAA.

6 Conclusion

In this paper, we presented a concept to manage test suites for services in an iterative development process. With the contained test cases the communication behavior can be tested thoroughly. We demonstrated how invalid test cases can be detected and removed from the test suite and we principle show how new test cases can be added. Thereby, most steps are supported by existing tools.

References

1. Papazoglou, M.: Agent-oriented technology in support of e-business. Commun. ACM **44**(4) (2001) 71–77
2. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: Service-Oriented Computing 2008 Workshops. LNCS 5472, Springer (2009) 66–78
3. Beizer, B.: Black-box testing: techniques for functional testing of software and systems. John Wiley & Sons, Inc., New York, NY, USA (1995)
4. Black, R.: Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing. Wiley Publishing (2009)
5. Wolf, K.: Does my service have partners? LNCS T. Petri Nets and Other Models of Concurrency **5460**(2) (2009) 152–171
6. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007. LNCS 4546, Springer (2007) 321–341
7. Lohmann, N., Weinberg, D.: Wendy: A tool to synthesize partners for services. In: PETRI NETS 2010. LNCS 6128, Springer-Verlag (2010) 297–307
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 Lecture Notes in Informatics (LNI)., GI (2008) 57–72
9. Kaschner, K., Wolf, K.: Set algebra for service behavior: Applications and constructions. In: BPM 2009. LNCS 5701, Springer-Verlag (2009) 193–210
10. Kaschner, K.: Safira: implementing set algebra for service behaviour. In: Proceedings of the 2nd Central-European Workshop on Services and their Composition. CEUR Workshop Proceedings, CEUR-WS.org (2010) 49–56