Towards the Generation of Test Cases for Executable Business Processes from Classification Trees

Thilo Schnelle[1,2], Daniel Lübke[1,3]

[1] FG Software Engineering, Leibniz Universität Hannover, Germany
[2] innoQ Deutschland GmbH, Germany
[3] innoQ Schweiz GmbH, Switzerland

Abstract. Testing executable business processes is essential when developing mission-critical process solutions. However, developers and testers are confronted with two major challenges: Keeping an overview of functional test coverage during test case creation and adopting existing test cases to process changes during maintainance.
Our aim is to develop a generator-based approach that forces the tester to create a classification tree, which keeps track of functional test coverage. Using this classification tree the generator combines test fragments into test cases. Both the classification tree and the code fragments are easier to change than a complete test suite thereby allowing easier test case development and maintance.

1 Introduction

Testing is essential for producing high quality and reliable software [6]. Testing executable business processes that orchestrate services poses special challenges:

1. Processes are usually triggered by messages from external sources and communicate to Web services. External sources can also include user facing applications, like task managers. For sending messages to a process instance and checking messages that are sent by process instances, all partner services need to be mocked. This issue was already addressed by Mayer & Lübke [9] and resulted in the open source tool BPELUnit.
2. Many different flows through the process model, covering both control-flow and data-flow variants, have to be tested. This causes test cases to contain many messages that are sent to the process. Therefore, BPELUnit test suites consist of many lines of code. This amount of code decreases clarity and makes it difficult to keep track of requirement coverage. Finding out whether all requirements from the specification of the process are covered in a test suite with thousands of lines of code poses a significant challenge.
3. A lot of test cases only diverge in later parts of the process. Therefore many messages – especially in the beginning – are repeated often at the start of different test cases. Applying even simple changes to the process can therefore result in changes to every test case.

O. Kopp, J. Lenhard, C. Pautasso (Eds.): 9th ZEUS Workshop, ZEUS 2017, Lugano, Switzerland, 13-14 February 2017, published at http://ceur-ws.org/

In this paper we first give an overview over related work done in the field of executable business process testing. Afterwards we propose a generator-based approach that helps testers by reducing the aforementioned problems. We show how the generator is used and the test models are created before we describe two exploratory empirical studies that were conducted to validate and enhance the presented concept before we conclude and give an outlook for future work.

2 Related Work

The use of classification trees was proposed by Grochtmann et al [4]. Their classification-tree editor is used to systematically divide black-box tests up into their requirements. Therefore one subtree per requirement is created and further divided into concrete values that can appear for this requirement. Afterwards they are able to automatically connect these pieces into test cases by picking one value per requirement.

Lübke et al [8] proposed a way to measure test coverage for white-box testing on executable business processes. This approach works by measuring which parts of the code are executed. Therefore it differs from the black-box testing proposed in this paper but can be an additional measure for increasing the quality of test cases for processes.

Another model-based approach by Lübke and van Lessen [7] is based on BPMN-modeled scenarios. This approach also addresses communication with stakeholders and is not focused on testers only. However, no test coverage metrics are included in this approach and also the problem of test case maintainance is not addressed.

3 Classification Trees and Test Meta-Model

Our approach is based on a classification tree as the main management artifact. The structure of this classification tree and the attached technical information is presented in this section. The data structured according to this metamodel is used by a generator in order to create an executable BPELUnit test suite.

The meta-model consists of two parts: First the formalization of the classification tree (figure 1) and second the technical bindings attached to it (figure 2).

The classification tree consists of several categories (first level child nodes) that correspond to different requirements. The categories can have subcategories and values. Values are the leafs of the tree. For example, an image processing software might distinguish between different shapes and colors. Both are independent requirements so they represent categories in the classification tree. Their values are concrete shapes (rectangle, circle, ...) and colors (red, yellow, ...). Possible test cases combine these, e.g. a first test case could use red rectangles and a second yellow circles.

For every test case only one value per category may be selected. The goal is to have every value used in at least one test case but to create as few test cases

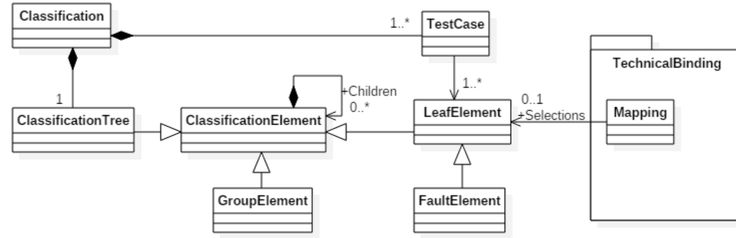

Fig. 1. Meta-Model for Test Classification Trees

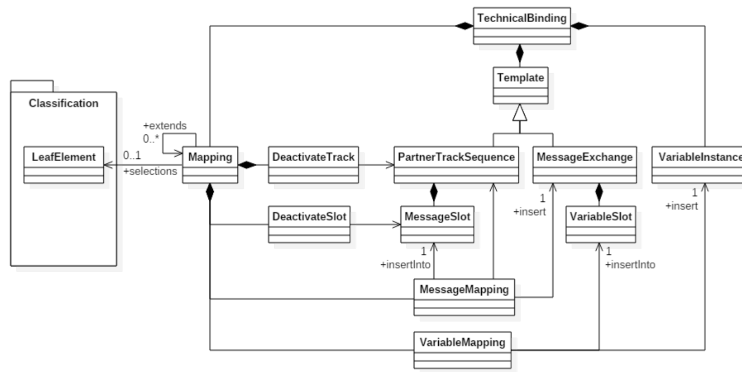

Fig. 2. Meta-Model for Technical Bindings

as necessary. Values that represent errors should always get their own test case as errors may influence the execution of the process drastically.

In the example three test cases are needed to cover all values because the largest category (Shape) contains three values. The maximum number of test cases is 6 (3 shapes times 2 colors).

Although categories should be independent from each other, the pracical use has shown that this strict requirement is very problematic in practice: Often values are independent business requirements but are technically not exclusive. Following the strict theoretical model leads to few categories with many values. In order to allow more readable classification trees, we allow semi-depedent categories and let the testers define conditions that prohibit certain combinations of values. Such extensions have also been proposed by Lehmann & Wegener [5].

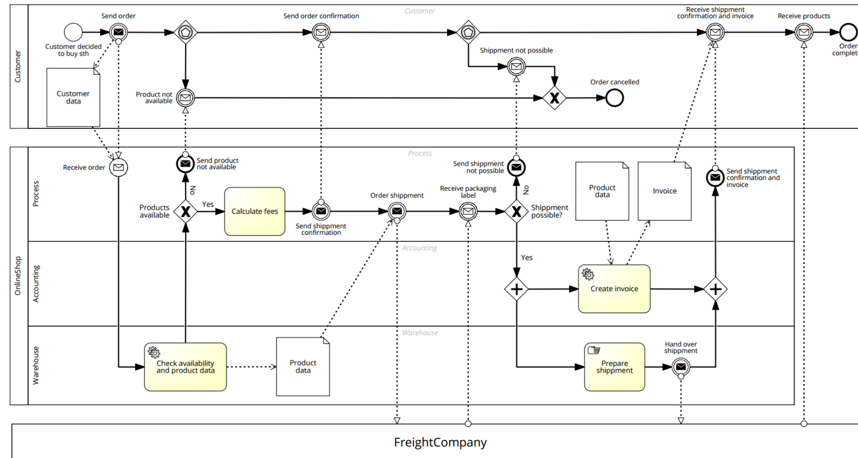

Fig. 3. A simple Example Process for an Online Shop

For every value there is a mapping that defines which test fragments are added to a test case that contains this value. This part is called "technical binding".

Mappings define which messages should be sent at specific points in the execution of the test cases. They also define instances for variable slots. For example in a message there might be a variable slot "purchase date". Possible instances for this slot are concrete dates from which one is chosen. There is also the possibility to deactivate slots for messages and even complete partners of the process, so that any communication to and from a specific service is disabled.

4 Generator Architecture & Example

We implemented a first version of a generator that follows the described meta-model: The classification tree is edited and saved in a spreadsheet and the technical binding information is stored in XML files. The generator reads all files and produces an executable BPELUnit test suite.

In the following, we discuss a simple model of an online shop as a business process related example. Figure 3 shows the corresponding BPMN diagram.

The classification for this online shop is shown in figure 4. There are three independent categories with in total nine values. In contrast to classification trees in "classical" software, the categories can represent process instance data (e.g. message contents) or decisions made in the process (e.g. messages in deferred choices.)

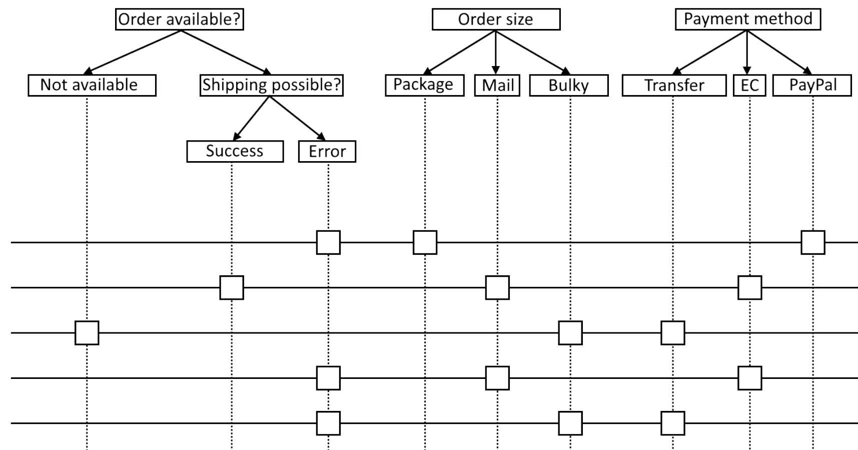

Fig. 4. An Example Classification of an Online Shop

```
<csg:variableDefinitions xmlns:csg="..." xmlns:os="http://www.example.org/OnlineShop/"
xmlns:tes="http://www.bpelunit.org/schema/testSuite">
    <csg:partnerTrackSequence name="ClientPort">
        <csg:messageSlot>OrderAvailable</csg:messageSlot>
        <csg:messageSlot>OrderShipping</csg:messageSlot>
    </csg:partnerTrackSequence>
</csg:variableDefinitions>
```

Fig. 5. Example for a Partner Track Definition

Figure 5 shows the definition of a partner track, which corresponds to a mocked service. For tracks there are slots defined into which message exchanges can be inserted.

These message exchanges are defined like the example in figure 6: The SOAP service, binding and operation are defined like in a normal BPELUnit message exchange but it may additionally contain variable slots. These are named places for the insertion of test case specified data.

This data is defined in variable instances like shown in figure 7. The variable has the same name as the variable slot it can be inserted in.

The connection between the two is made by mappings (see figure 8) and a variable instance is chosen by its name to be inserted into the corresponding variable slots. There are also message exchanges chosen to be inserted into message slots of partner tracks. In addition the figure 8 shows how to deactivate single slots or complete partner tracks if required.

The generator is available as open source at GitHub[4].

[4] https://github.com/ThiloSchnelle/bpelunit-facet-classification-generator

```
<csg:variableDefinitions xmlns:csg="..." xmlns:os="http://www.example.org/OnlineShop/" xmlns:oss=
"www.example.org/OnlineShopSchema/">
    <csg:messageExchange xmlns:tes="http://www.bpelunit.org/schema/testSuite" name="AccountingRequest">
        <tes:receiveSend service="os:Accounting" port="AccountingPort" operation="accountingInformation">
            <tes:receive fault="false" />
            <tes:send fault="false">
                <tes:data>
                    <oss:accountingResponse>
                        <oss:pdfLocation><csg:variableSlot name="InvoiceVar" /></oss:pdfLocation>
                    </oss:accountingResponse>
                </tes:data>
            </tes:send>
        </tes:receiveSend>
    </csg:messageExchange>
</csg:variableDefinitions>
```

Fig. 6. Example of a Message Exchange Definition with a Variable Slot

```
<csg:variableDefinitions xmlns:tes="http://www.bpelunit.org/schema/testSuite"
    xmlns:csg="..." xmlns:oss="www.example.org/OnlineShopSchema/">
    <csg:variableDefinition name="OverallPrice">
        <csg:instance name="Mail">8.99</csg:instance>
        <csg:instance name="Package">93.94</csg:instance>
        <csg:instance name="Bulky">399.99</csg:instance>
    </csg:variableDefinition>
</csg:variableDefinitions>
```

Fig. 7. Example for a Variable Definition

5 Exploratory Empirical Studies

In order to better understand the field of process testing and the implications our approach might have, we conducted two empirical, exploratory studies: A re-implementation of existing test cases of an executable business process taken from an industry project and a small experiment with students.

5.1 Re-Implementation of Industrial Test Cases

The approach was applied to a process of the project Terravis [1], which had a set of unit tests. These tests were re-implemented with the goal of exactly reproducing the existing test suite in order to demonstrate that our approach has the needed expressiveness for defining test cases.

```
<csg:mappings xmlns:csg="...">
    <csg:mapping name="Order available?:Shipping possible?">
        <csg:extends>BaseSelection</csg:extends>
        <csg:variable variableName="ProductsAvailable" variableInstance="Available" />
        <csg:useMessageExchange messageName="NotAvailable" messageSlot="AvailabilityMessageSlot" />
        <csg:deactivateTrack name="AnotherTrack" />
        <csg:deactivateSlot name="SlotX" />
    </csg:mapping>
</csg:mappings>
```

Fig. 8. Example for a Mapping Definition

The size of the original test suite is much larger compared with the size of source files for the generation approach: The original test suite contained 437 declarations of message exchanges that took 3663 lines of code (LOC) while the source files for the generation only declare 53 (12%) message exchanges in 1428 (39%) LOC. The metrics show that the generator approach eliminated much code duplication in the test suite.

During the re-implementation two findings were made:

1. One test case of the existing unit tests used incorrect business data which worked in the process because the data from this service call were not used in this scenario. As a result, the process and the test suite was fixed. Because the generator requires standardized message contents, this defect was found, which is an advantage of our approach.
2. One test case that used anonymized data from production incidents was found. The data was not harmonized with the other test cases. These regression tests should probably not be re-implemented in real-life projects or should be transformed to (re-)use already existing message contents and test data. This also shows that the generator approach probably works best when unified test data is used so that the different binding pieces can be easily joined together and be reused.

5.2 Student Experiment

The experiment took part in a university course. The participants started with no webservice knowledge. They learned to model, implement and test executable business processes. Their task was to test two example processes: One with BPELUnit and one with the presented generator. Afterwards, the students voluntarily answered a questionaire.

Students on average created 2.89 test cases with pure BPELUnit compared to 1.67 test cases with the generator. Some found working with the generator to be faster in the long run and the additional effort to be small. Others found the additional effort big and felt slowed down. The majority judged that the generator is complicated but also increases the overview of the test coverage.

We think that the time and the experience the participants had were not sufficient for the additional complexity introduced by the generator to pay off. This is why developing test cases was slower with the generator. On the other hand an important task of the generator - increasing the overview over test coverage - has been valued by the participants.

From the feedback it also became clear that better tool support is needed. As of now there is no GUI and XML files need to be written manually. This is why we expect better productivity when tooling is in place.

6 Conclusions and Outlook

In this paper we proposed a generator-based approach to testing executable business processes. An initial version of the meta-model and a generator have

been developed and has been open sourced. The toolchain has been tried initially in an industrial project.

Moreover, an exploratory student experiment has provided further necessary research and development directions: Especially the missing editor support seems to influence the development speed of the students negatively. Additional tool support needs to be provided and an empirical validation has to be made for it.

Conceptually, the generator-based approach offers the possibility to generate missing test cases by combining different, yet unused combinations of values of the classification tree. One possible algorithm to explore has been presented by Cohen et al [2]. Research with "traditional" software systems has shown that combining all possible values of two attributes with each other yields the highest cost-usage benefits [3]. Further studies need to show whether these findings are also valid for executable business processes.

We will follow up on the open research questions and will especially look at the efficiency (how many defects have been found and comparison between classification tree coverage and code coverage) and would like to join with other academic and industrial partners to further enhance and validate our approach.

References

1. Berli, W., Lübke, D., Möckli, W.: Terravis – large scale business process integration between public and private partners. In: Plödereder, E., Grunske, L., Schneider, E., Ull, D. (eds.) Lecture Notes in Informatics (LNI), Proceedings INFORMATIK 2014. vol. P-232, pp. 1075–1090. Gesellschaft für Informatik e.V., Gesellschaft für Informatik e.V. (2014)
2. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An approach to testing based on combinatorial design. IEEE Transactions on Software Engineering 23(7), 437–444 (1997)
3. D. Richard Kuhn, Dolores R. Wallace, A.M.G.J.: Software fault interactions and implications for software testing. IEEE Transactions of Software Engineering 30(6) (2004)
4. Grochtmann, M.: Test case design using classification trees. In: Proceedings of STAR 94. pp. 93–117 (1994)
5. Lehmann, E., Wegener, J.: Test Case Design by Means of the CTE XL. In: Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark (2000)
6. Lübke, D.: Test and Analysis of Service-Oriented Systems, chap. Unit Testing BPEL Compositions. Springer (2007)
7. Lübke, D., van Lessen, T.: Modeling Test Cases in BPMN for Behavior-Driven Development. IEEE Software Sep/Oct 2016, 17–23 (Sep/Oct 2016)
8. Lübke, D., Singer, L., Salnikow, A.: Calculating BPEL Test Coverage through Instrumentation. In: Workshop on Automated Software Testing (AST 2009), ICSE 2009 (2009)
9. Mayer, P., Lübke, D.: Towards a BPEL Unit Testing Framework. In: TAV-WEB '06: Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications, Portland, USA. pp. 33–42. ACM Press, New York, NY, USA (2006)