Software & Systems Modeling
https://doi.org/10.1007/s10270-017-0647-0

THEME SECTION PAPER

Property-based testing of web services by deriving properties from business-rule models

Bernhard K. Aichernig[1] · **Richard Schumi**[1]

Received: 15 September 2016 / Revised: 9 November 2017 / Accepted: 25 November 2017

Abstract
Property-based testing is well suited for web-service applications, which was already shown in various case studies. For example, it has been demonstrated that JSON schemas can be used to automatically derive test case generators for web forms. In this work, we present a test case generation approach for a rule engine-driven web-service application. Business-rule models serve us as input for property-based testing. We parse these models to automatically derive generators for sequences of web-service requests together with their required form data. Property-based testing is mostly applied in the context of functional programming. Here, we define our properties in an object-oriented style in C# and its tool FsCheck. We apply our method to the business-rule models of an industrial web-service application in the automotive domain.

Keywords Model-based testing · Test case generation · Property-based testing · QuickCheck · FsCheck · Web services · Business-rule models

1 Introduction

Property-based testing (PBT) is a testing technique that tries to falsify a given property by generating random input data and verifying the expected behaviour [8]. Properties can range from simple algebraic equations to complex state machine models. On the basis of the taxonomy of Utting et al. [31], we can characterise the models that are supported by PBT as follows. PBT can handle various forms of input–output models that have a transition-based pre-post modelling paradigm. It supports timed or untimed model characteristics and the models can be non-deterministic. The test case generation technology is based on random generators, but next to simple randomness stochastic distributions may be applied to guide the test-selection process. The test execution is offline, which means that a test case is first generated and then it is executed.

Like in all model-based testing techniques, the properties serve as a source for test case generation as well as test oracles. PBT is a well-known testing practice in functional programming [3,8,9], but nowadays we see a growth of applications outside its traditional domain. Examples include the automated testing of automotive software [4,30], cloud computing [16] and web services [11,18]. In this work, we will focus on the latter.

Many web services store configurations in XML files. Some web services also store workflow details and user-access rules in XML business-rule models [25,27]. These XML definitions can be seen as an abstract specification of the service behaviour which may serve as a basis to verify whether the service complies with this specified behaviour [21]. We present an automated approach that uses these business-rule models to derive FsCheck[1] models and generators that are applied to generate command sequences with random input data. FsCheck is a PBT tool for .NET which supports the definition of properties and generators in both a functional style with the programming language F# and an object-oriented style with C#. We opted for C#, because it is the implementation language of our industrial case study and because the system-under-test (SUT) has an object-oriented

Communicated by Drs. M. Papadakis, S. Ali, and G. Perrouin.

✉ Richard Schumi
rschumi@ist.tugraz.at

Bernhard K. Aichernig
aichernig@ist.tugraz.at

[1] Institute of Software Technology, Graz University of Technology, Graz, Austria

[1] https://fscheck.github.io/FsCheck

Published online: 08 December 2017

architecture. However, the testing approach is general and can be ported to any PBT framework.

The process of our testing approach is illustrated in Fig. 1. The first step is to parse and translate the XML business-rule files to input models for FsCheck. FsCheck supports all kinds of models that have states, transitions, postconditions and optionally preconditions, but in our case the models were extended finite state machines (EFSMs) [6]. These EFSMs are used by the specification builder to create generators and FsCheck interface implementations according to the parsed model. FsCheck transforms these interface implementations into a property to be tested via randomly generated command sequences. This property requires that the state of the model is equal to the state of the SUT after each transition (command). As soon as a command sequence has been generated, it is executed on the SUT as a test case. When the property holds throughout the execution, then the test case was successful resulting in a pass-verdict, otherwise a fail-verdict with a counterexample is produced. The number of test cases can be specified by the user, but if a property fails, then no further test cases are executed.

For our use case, a transition is not a simple action. It represents the opening of a page of a graphical user interface, the entering of data for form fields and saving the page. In the test case generator, the transitions are realised as command classes with attributes representing the associated form data. Our target is to test the underlying requests of the transitions, which are necessary for the interaction with the web-service application.

1.1 Related work

PBT was already applied to a number of web-service applications, because it is a good way to verify that a variety of inputs are supported without problems. The most similar approaches are described below.

López et al. presented a domain-specific language (DSL) that allows non-experts to perform an automatic test data generation with QuickCheck [20]. The DSL reuses syntax from the web services description language (WSDL) in order to generate well-formed XML for the input of web services. It supports constraints for different data types and combinators that enable the application of constraints to all kinds of data. The difference in this approach to our work is that it does not consider state machines and that the generator definition must be created manually.

Lampropoulos and Sagonas [18] present a similar approach that automatically reads the WSDL specification of a web service and makes web-service calls with generated data. The approach was implemented with the PBT tool PropEr for Erlang. They support many data types, but only a few constraints for the data. However, they show how additional constraints can be added manually. In contrast to our work, they also do not use state machines to test the service behaviour. They only test if the web-service result is valid and whether no error occurred.

A similar approach was presented by Li et al. [19]. They also show how WSDL can be used to automatically derive generators, but the focus of their work is primarily on evolving web services. Their approach facilitates adapting the

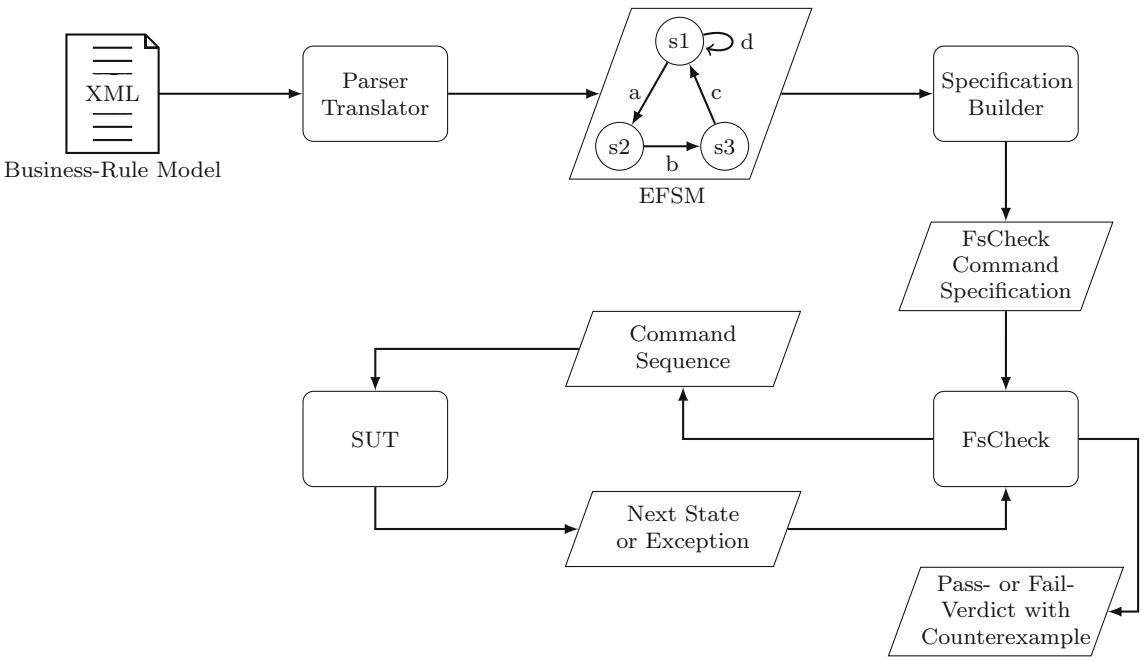

Fig. 1 Overview of the steps for the FsCheck command sequence generation for business-rule models

test environment to a new version of a web service. This is achieved by automatically generating refactoring scripts for the evolving test code. The difference in our work is that their models have to be created manually by the user and that their focus lies on evolving web services.

Frelund et al. [12] present a library called Jsongen, which can generate JSON data to test web services. Many web services communicate via JSON because it is a convenient language to encode data. It is similar to XML, but more compact and more readable. Their library uses JSON schemas with the structure of the data, data types and data constraints to automatically create QuickCheck generators. They use these QuickCheck generators to generate input data that fulfils the requirements of a web-service call. They apply their library to test a small service, where users can post questions and answers.

Earle et al. [10] extend this library so that the JSON schema also includes an abstract specification of the service behaviour. This specification is in the form of a finite state machine (FSM). In the previous work, the FSM definition had to be made separately to the JSON schema for the web-service data. In this work, they show how it can be encoded in the JSON schema. Their FSM is defined with hyperlinks, which represent the events of the FSM and the states can be chosen dynamically. In contrast to our work, the JSON schema for the service has to be produced manually and it cannot be used from the service components directly. Furthermore, their approach was only evaluated with a small test web service; they have not made a meaningful case study.

The most similar work to ours was presented by Francisco et al. [11]. They show a framework that automatically derives QuickCheck models from a WSDL description and OCL semantic constraints. They show how the models can be applied to automatically test both stateless and stateful web services with generated input data. The WSDL description contains information about the required data, the data structures, data types and the possible operations. The OCL constraints define pre- and postconditions for the operations and can be used to describe a state machine for the service behaviour. The used service description is very similar to our business-rule models, but their generators consider only data types, while we also support constraints for the data, like a minimum value for an integer. Another difference is that the OCL semantic constraints are added manually. Our business-rule models were already part of the web-service architecture.

To the best of our knowledge, there is no other work that uses inherent web-service components, respectively, business-rule models to automatically derive PBT models. Although there are some similar publications that show how PBT models can be used for web services, they mostly rely on a manual specification of a model separate to the web-service implementation. In contrast to this, our approach can be directly applied to a service component, which is also used directly on the server-side to verify if a command is permitted in the current state and if the attributes are fitting to the model. Furthermore, the other approaches were all implemented with functional programming languages. Our approach uses C# to define the properties in an object-oriented way.

1.2 Contribution

This article is an extension of our workshop paper on property-based testing with business-rule models [2], where we presented a new test case generation approach that uses XML business-rule models in the form of EFSMs as input for PBT.

Compared to our previous work, the novel contributions of this work are the following. We formalise the underlying concepts and algorithms of PBT with EFSMs and present an extended description of the structure and architecture of our web application under test. A detailed definition of our rule-engine models is given with an abstract grammar. Moreover, we show a formalisation of the translation of our business-rule models to EFSM. Another contribution is the extension of our models to include the functionality to switch between multiple objects and different rule-engine models in order to test whether there are issues, when we consider sets of objects. Furthermore, we present an additional case study, which highlights the usefulness and applicability of our approach by revealing several issues in the SUT.

1.3 Structure

The rest of the paper is structured as follows. First, in Sect. 2, we present our web application under test and define its corresponding rule-engine system and rule-engine models. Then, Sect. 3 will explain the basics of PBT and FsCheck. The contributions start from Sect. 4, where we introduce PBT with EFSMs and we present a small example of model-based testing with FsCheck. Section 5 presents application-specific extensions to our method, like the translation of business-rule models to EFSMs. Then, in Sect. 6 we describe details about the structure and implementation of our approach. Section 7 shows the results of an evaluation, where we performed two case studies for two major modules of an industrial web application. In Sect. 8, we discus the limitations and threads to validity and potential future work. Finally, the work is concluded in Sect. 9.

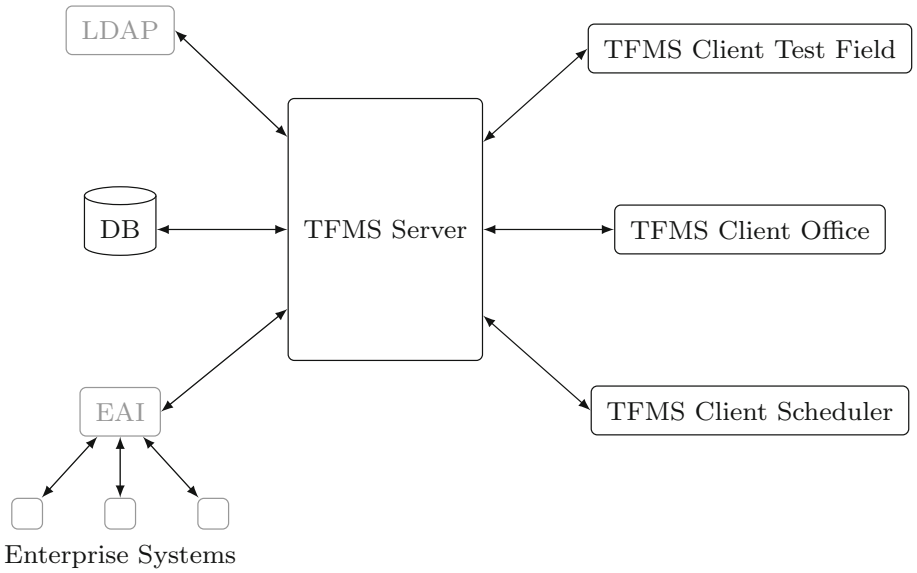

Fig. 2 Client–server architecture of the SUT. *Source*: AVL

2 Web application under test

2.1 System overview

Our approach was developed for a web-service application which was provided by our industrial project partner AVL.[2] This application originates from the automotive domain and is called testfactory management suite (TFMS).

This system supports the process of instrumenting and testing automotive power trains—a core business of AVL. The web-service may capture test field data, activities, resources, information and workflows. A variety of activities can be realised with the system, like test definition, planning, preparation, execution, data management and analysis. The system is a composition of a variety of components and services that serve different purposes. For example, there is a service for the authentication and a database for the data management.

The architecture of the system is illustrated in Fig. 2. It can be seen that it has a client–server architecture. A web server called TFMS server is the central component of the system. The server runs the Internet Information Services (IIS) from Microsoft and provides several Simple Object Access Protocol (SOAP) web services, which are described with the WSDL. It can use an internal service or the *lightweight directory access protocol* (LDAP) for the user authentication and an Oracle database (DB) for the data storage. Furthermore, additional enterprise systems can be connected to the server via *enterprise application integration* (EAI). The system has different types of clients, one for the test field to collect test data, one office client for management activities and a scheduler to control the order of the test activities. The development of the system started about 12 years ago by applying a Capability Maturity Model Integration (CMMI) development process, which is a form of a waterfall process. Now there are about 15 people working on the development and testing of the system applying the principles of the *scaled agile framework*.

The web services on the server are driven by business rules. A rule engine takes this business logic in the form of business-rule models and interprets them on the server. This defines the control-flow of the application. In TFMS, the business-rule models are called Rule-Engine Models (REM). The system consists of multiple modules $Module_1, \ldots, Module_m$. For example, there is a module for test orders called Test Order Manager and one for test equipment called Test Equipment Manager.

Modules can be seen as groups of functionality, and they consist of multiple REMs. These REMs describe what forms can be opened by a user and how they look like, e.g. what form fields they contain. Only one of the REMs can be active, and it determines which forms can be opened by the user in the current state of the system. A module can be defined as a pair $(REMs, activeREM)$ with $REMs$ being a (non-empty) finite set of rule-engine models and *activeREM* being an optionally active REM that represents the activities that can be currently executed by a user. Each REM is represented by an XML file that can be edited via the TFMS server. Hence, the top level structure of the system can be defined as follows:

Definition 1 (System)

$$System =_{df} \{Module_1, \ldots, Module_m\}$$
$$Module =_{df} (REMs, activeREM)$$

[2] https://www.avl.com

$$REMs =_{df} \{REM_1, \ldots, REM_n\}$$

and $activeREM \in REMs \cup \{Nil\}$.

An REM describes the behaviour of a class of objects of the application domain. In order to enable the selection of such objects and also to select the *activeREM*, we introduce a switch mechanism, which will be explained in Sect. 5.2. In the following, we take a closer look at these business-rule models.

2.2 Business-rule models

An application may need various modifications depending on the customer or on the country of deployment. It is infeasible to apply these modifications to the source code, because it would require the development of different versions for each customer. A business-rule engine is a good way to apply the different modifications in the form of rules for different deployments of an application. Business-rule engines are used to integrate these rules in the business logic. They are often combined with business-rule management systems that can be used to store, load and easily modify the rules. There are many frameworks, architectures or systems for web services and applications in general that provide business-rule management functionality [14,23,28].

Most of them only differ by the information that can be encoded in the rules. For example, business-rule engines can store constraints, conditions, actions and other business process semantics. Even workflow details can be included, although there is a separate technology, called workflow engine or also business process management system [1,5,26]. The major difference is that workflows/processes define the order or sequence of tasks (actions/operations) and business rules describe conditions and resulting actions.

The web-service application of our case study has a custom implementation of a rule management system. This custom implementation was made, because there were not many existing approaches at the time, when the application was developed. Our business-rule models are similar to other rule definitions. For example, the rule markup language (RuleML) [33] could be applied to encode our models. Note that we talk about rules and not processes as our models have the main purpose of storing costumer-specific business logic, and they specify conditions for enabling certain tasks, which can be seen as condition-action pairs. They do not focus on the sequence of tasks and do not support the composition of services. Moreover, this term is used within the given commercial system.

As mentioned, in this work business-rule models are also called REMs and they are the primary basis for our approach. An REM is a state machine defining the behaviour of a TFMS Object Class. A TFMS Object Class describes objects of our application domain, like incidents or test orders. Each of these objects has a state, an identifier, attribute values/data, and they are stored in the database of our SUT.

The abstract syntax of a rule-engine model can be defined as follows. Its definition corresponds to the concrete XML syntax, but is more concise.

Definition 2 (Rule-engine models)

$$\begin{aligned}
REM &=_{df} rem(AllAttributes, AllTasks, AllStates)\\
AllAttributes &=_{df} Attr*\\
Attr &=_{df} attr(Name, DataType, Parameter*)\\
&\quad \mid attr(Name, DataType)\\
DataType &=_{df} Integer \mid Bool \mid String \mid Enum \mid Object\\
&\quad \mid Date \mid DateTime \mid Float \mid File \mid Reference\\
Parameter &=_{df} MinValue \mid MaxValue \mid EnumItem*\\
&\quad \mid Query \mid Regex \mid \ldots\\
AllTasks &=_{df} Task*\\
Task &=_{df} task(id : Name, attributes : Name*,\\
&\qquad possibleNextStates : Name*)\\
AllStates &=_{df} State*\\
State &=_{df} state(id : Name, possibleTasks : Name*)
\end{aligned}$$

For easier readability, we use *record types* to define composite data: an REM is defined as a record *rem* with three fields: the set of *AllAttributes*, the set of *AllTasks* and the set of *AllStates*. In fact, these sets are represented as sequences, e.g. the sequence of all attributes $Attr*$. An attribute comprises a *Name*, a *DataType* and optionally a sequence of *Parameter*s. Parameters may further restrict a data type, like a maximum value for an *Integer*. A more complex form of restriction of an attribute may be realised via a *Query* to a database, which will be further explained in Sect. 6.1 (reference attributes). This allows to implement a selection of existing values, e.g. a dynamic drop-down menu in a web-form. Another restriction can be applied with a regular expression (*Regex*), which can limit a string attribute to only allow certain patterns.

Tasks represent the behaviour, i.e. the actions or events a user may trigger. For readability, we define tasks with *field names*. A *task* has an identifier, i.e. a *Name*, a number of attributes to be entered into a form and the possible next states a task may reach. If there is more than one possible state, then it can be selected via an external choice by the user; hence, this does not represent non-determinism. Finally, all *States* define the complete state-space with each state being associated with a sequence of tasks that can be triggered in this state.

For illustration, Listing 1 shows a simplified version of the XML file of an REM that was used as basis for the example in Sect. 4.2. It can be seen that these models are structured very similarly to our abstract syntax. The main components are:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RuleEngineModel TfmsType="Incident">
3      <AllAttributes>
4          <StaticAttributeInfo Name="ParentFolder"
5          DataType="Reference">
6              <Query Criteria="Class=IncidentFolder">
7                  <RequestedAttributes>
8                      <string>*</string>
9                  </RequestedAttributes>
10             </Query>
11         </StaticAttributeInfo>
12         <StaticAttributeInfo Name="Description"
13         DataType="String" MaxValue="128" />
14         ...
15     </AllAttributes>
16     <AllTasks>
17         <Task Name="IncidentCreateTask">
18             <DynamicAttributesInfo>
19                 <Attribute Name="ParentFolder"
20                 Enabled="true" Required="true" />
21                 <Attribute Name="Description"
22                 Enabled="true" Required="true" />
23                 ...
24             </DynamicAttributesInfo>
25             <PossibleNextStates>
26                 <State Name="Submitted"
27                     NoteRequired="false" />
28             </PossibleNextStates>
29         </Task>
30         ...
31     </AllTasks>
32     <AllStates>
33         <State Name="Submitted">
34             <PossibleTasks>
35                 <Task>IncidentEditTask</Task>
36                 <Task>IncidentCloseTask</Task>
37             </PossibleTasks>
38         </State>
39         ...
40     </AllStates>
41 </RuleEngineModel>
```

Listing 1 Simplified XML representation of a rule-engine model

- attribute definitions with data types and constraints (Lines 3 to 15)
- tasks with enabled and required attributes and possible next states (Lines 16 to 31)
- states with possible tasks (Lines 32 to 40)

Optionally, the models may also include:

- scripts, which can be executed on certain events
- queries for the selections of specific objects
- reports for a good overview of the entered objects

Note that our REMs do not always represent the actual behaviour of the web application under test. REMs determine what tasks are currently active and what attributes are required. However, developers can overrule the constraints that are included in REMs, when they implement a task. For example, a task can lead to different target states that are not specified in an REM. Moreover, a form of the SUT might require additional attributes for special cases that are only implemented in the SUT, but not contained in REMs. It makes sense to search for such cases where REMs are overruled by the implementation in order to find out, if this behaviour is intentional or was introduced by mistake. Moreover, manual adjustments to the test models had to be made so that this deviations are not found repeatedly.

3 Property-based testing

3.1 Overview

Property-based testing (PBT) is a random testing technique that evaluates a system by verifying a given property. A property is a high-level specification of behaviour that should hold for a range of data points. For example, a property might state that a function should have a certain expected behaviour. A test for this property is successful, when the function runs through as expected, otherwise a counterexample is returned. Simple properties can be expressed as functions with Boolean return values that should be true when the property is fulfilled. These functions should work for any input values; hence, a high number of random inputs are generated for the parameters. Another important aspect of PBT is shrinking, which is used to find a similar simpler counterexample, when a property fails. In order to shrink a counterexample, a PBT tool searches for smaller failing counterexamples. The search method can be specified individually for different data types [15,24,29].

A simple example of an algebraic property is that the reverse of the reverse of a list must be equal to the original list:

$$\forall xs \in List[T] : reverse(reverse(xs)) = xs$$

A PBT tool will invoke its built-in generator for *List*s and generate a series of random lists xs, execute the reverse function and evaluate the property. A tester may extend or replace the basic generators with special-purpose generators, e.g. generating extremely long lists. Generators are type based and provide a sample function for the given type. In the simple case, they can be described as follows: $AGen =_{df} gen(sample : A)$, where A is the type of the generator, which provides a sample function that returns an instance of this type. For nested data types, generators take other generators as arguments. For example, a generator for $List[T]$ needs a generator for element values of type T.

PBT constitutes a flexible and scalable model-based testing technique because it is random testing, and it has been shown that it generates a large number of tests in reasonable time [32]. The first PBT tool was QuickCheck [8] for Haskell. There are many other tools that are based on the concepts of QuickCheck, e.g. ScalaCheck [22] or Hypothesis[3] for Python. For our approach, we work with FsCheck.

[3] https://pypi.python.org/pypi/hypothesis

3.2 FsCheck

FsCheck is a PBT tool for .NET based on QuickCheck and influenced by ScalaCheck. Like ScalaCheck, it extends the basic QuickCheck functionality with support for state-based models. A limitation of the current version is that it does not consider preconditions when shrinking command sequences. Due to this limitation, we initially had to disable shrinking for our case study, because it was not possible to receive proper counterexamples. However, this feature is included in an experimental release and we were able to apply shrinking with this experimental version. With FsCheck, properties can be defined both in a functional programming style with F# and in object-oriented style with C#. Similar to QuickCheck, it has default generators for basic data types and more complex ones can be defined via composition. It has an arbitrary instance that groups together a shrinker and a generator for a custom data type. This makes it possible to use variables of this data type as input for properties. New arbitrary instances can be dynamically registered at run time, and then the new data type can be directly used for the input data generation. Furthermore, FsCheck has extensions for unit testing, which support a convenient definition and execution of properties like normal unit tests.

4 Property-based testing with extended finite state machines

4.1 State machine properties

PBT can also be applied to models in the form of extended finite state machines (EFSMs) [17].

Definition 3 (EFSM) An EFSM can formally be defined as a 6-tuple $(S, s_0, V, I, O, T) \in State\_set \times State \times Variable\_set \times Input\_set \times Output\_set \times Transition\_set$.

S	is a finite set of *State*s,
$s_0 \in S$	is an initial *State*,
V	is a finite set of *Variable*s,
I	is a finite set of *Input*s,
O	is a finite set of *Output*s,
T	is a finite set of transitions, $t \in T$ can be defined as a 5-tuple (s, i, g, op, s'),
s	is the source *State*,
i	is an *Input*,
g	is a guard of the form $e \circ e'$, where e and e' are algebraic expressions and $\circ \in \{<, >, \neq, =, \leq, \geq\}$,
op	is a sequence of output and assignment operations of the form $v = e$ with
$v \in V$	and e is an expression,
s'	is the target *State* [17].

An example EFSM is presented in Sect. 4.2 in Fig. 4. Semantically, a guard g is a Boolean function that takes the variable valuations v as input and returns a Boolean value. An operation op is a function mapping the current variable valuations to a pair of new valuations and an optional output $o \in O$.

In order to perform PBT for an EFSM, a state machine specification *spec* has to be provided. This *spec*ification includes functions to set the initial state of the model and the SUT, a set of commands *cmds* and a *next* function that builds a command generator *CmdGen* for a given model state:

Definition 4 (State machine specification)

$$\begin{aligned} Spec \;=_{df}\; & spec(initalModel : () \to Model, \\ & initialActual : () \to Sut, \\ & cmds : Cmd\_set, next : Model \to CmdGen) \end{aligned}$$

Algorithm 3 in Sect. 4.2 outlines an example specification for the incident manager as it is required for FsCheck.

A *Model* object consists of fields representing the current EFSM state s, the valuations for the variables v, the transition set T, the last output o and a *doStep* function that performs the execution of a transition.

$$\begin{aligned} & Model \;=_{df}\; model(s : State, v : Variable \to Val, T : \\ & Transition\_set, o : Output, doStep : Input \to Model) \\ & doStep(in) \;=_{df}\; model(s', v', o', doStep) \; such\; that \\ & (s, in, g, op, s') \in T \wedge g(v) = True \wedge (v', o') = op(v) \end{aligned}$$

Note that the SUT is defined in the same way as the model and is, therefore, omitted.

A command $Cmd_{in} \in spec.cmds$ encodes a set of transitions T_{in} with the same input in. They encapsulate preconditions, postconditions and the execution semantics of these transitions. Preconditions *pre* define the permitted transition sequences by enabling the command only in states where the input in is allowed. Postcondition *post* can verify the effects of the command, e.g. by comparing the state of the model and the SUT. The execution semantics are encoded via the functions *runModel* and *runActual* for executing the *Model* and the SUT. The definition of a command is shown in Fig. 3. Note that in this definition we show various possible checks in the postcondition, i.e. we analyse the current state of the SUT, variable valuations and the output. In reality this may not be feasible, because the SUT might not provide all this information. Hence, in many cases it may only be possible to check the output in the postcondition. An example implementation of a command is presented in Sect. 4.2 in

$$
\begin{aligned}
Cmd_{in} &=_{df} cmd_{in}(T_{in} : Transition\_set, pre_{in} : Model \to Bool, post_{in} : (Model, Sut) \to Boolean,\\
&\qquad runModel_{in} : Model \to Model, runActual_{in} : Sut \to Sut)\\
T_{in} &=_{df} \{(s, i, g, op, s') \mid (s, i, g, op, s') \in T \wedge i = in\}\\
pre_{in}(model) &=_{df} \begin{cases} True & \text{if } \exists (s, i, g, op, s') \in T_i.\ s = model.s \wedge g(model.v) = True \\ False & \text{otherwise} \end{cases}\\
runModel_{in}(model) &=_{df} model.doStep(in)\\
runActual_{in}(sut) &=_{df} sut.doStep(in)\\
post_{in}(model, sut) &=_{df} \begin{cases} True & \text{if } model.s = sut.s \wedge model.v = sut.v \wedge model.o = sut.o \\ False & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 3 Command definition

Algorithm 4. This example demonstrates the function definitions for an *IncidentCreateTask*.

A property of an EFSM is that for each permitted path, the postcondition of each transition, respectively, command of the path must hold. In order to verify this property, a PBT tool produces random command sequences and checks the postconditions after each command execution.

The state machine property must hold in all settings of the model *Model_set* and the SUT *SUT_set* that are reachable via valid command sequences. A command sequence is valid if all its preconditions are satisfied. Hence, given a specification *spec*, a state machine property for EFSMs can be defined as follows:

Definition 5 (State machine property)

$$
\begin{aligned}
&\forall cmd_{in} \in spec.cmds, model \in Model\_set :\\
&\exists sut \in SUT\_set : cmd_{in}.pre(model) \implies\\
&\qquad cmd_{in}.post(cmd_{in}.runModel(model),\\
&\qquad\qquad cmd_{in}.post(cmd_{in}.runActual(sut))
\end{aligned}
$$

Algorithm 1 shows the pseudocode of the test case generation for such a property. The algorithm takes a *spec* and a *size* parameter for the length of the test case as input and returns a *testSequence*, which is a sequence of (*Cmd*, *Model*) pairs. In the first step, the initial model is created with the *initialModel* function of the *spec*. Next, there is a loop over the size parameter. In each iteration, a command generator *gen* is built with the *next* function of the *spec*. This function takes the model (Line 9) and creates a subset of all commands by checking their precondition. The function returns an *Elements* generator, which selects one element of this set with a uniform distribution (Line 11). The sample function of this generator is called to produce a command (Line 4). This command is executed with *runModel*, which returns a new model that incorporates the state change. Note that we need a new model and not only change the current one, because future changes should not affect the old stored model instances. This new model and the command are stored in the *testSequence*. Finally, after the loop is finished we return the *testSequence*, which represents a test case.

Algorithm 1 Pseudocode of the test case generation.

Input:
spec: state machine specification
$size \in \mathbb{N}_{>0}$:parameter for test-case length
Output:
$testSequence : (cmd_1, model_1), \ldots, (cmd_n, model_n)$
1: $model \leftarrow spec.initialModel()$
2: **for** $i \in \{1, ..., size\}$ **do**
3: $gen \leftarrow spec.next(model)$ ▷ next returns a *cmd* generator
4: $cmd \leftarrow gen.sample()$ ▷ command is generated
5: $model \leftarrow cmd.runModel(model)$ ▷ command is executed
6: $testSequence[i] \leftarrow (cmd, model)$ ▷ build test sequence
7: **end for**
8: **return** *testSequence*
9: **function** spec.next(*model*)
10: $cmdSet \leftarrow \{cmd_{in} \in spec.cmds \mid cmd_{in}.pre(model) = True\}$
11: **return** *Gen.Elements*(*cmdSet*)
12: **end function**

Algorithm 2 Pseudocode of the test case execution.

Input:
$testSequence : (cmd_1, model_1), \ldots, (cmd_n, model_n)$
spec : state machine specification
Output:
Pass, if the test case is successful, *Fail* otherwise
1: $sut \leftarrow spec.initialActual()$
2: **for** $(cmd_i, model_i) \in testSequence$ **do**
3: $sut \leftarrow cmd_i.runActual(sut)$ ▷ command is executed
4: **if** $\neg cmd_i.post(model_i, sut)$ **then** ▷ check post condition
5: **return** *Fail*
6: **end if**
7: **end for**
8: **return** *Pass*

Algorithm 2 shows how such a generated test case can be executed. The test case is the input of this algorithm together with a *spec*, and the result is a verdict. (Note that shrinking is omitted in this simplified algorithm.) In the first step, the initial SUT is built by the *initialActual* function of the *spec*. After that we loop over the *testSequence*. Next, the command is executed on the SUT with *runActual*, which results in a modified SUT (Line 3). The postcondition of the command is applied to compare the SUT with the stored model of the *testSequence*. If it is false, then the test failed. Otherwise, the execution continues and if the loop is finished, then the postconditions of all commands were satisfied and a pass-verdict is returned.

Algorithm 3 Incident specification *Spec*.

```
Input:
    SUT class for the connection to the SUT,
    Model class
1: function initialActual
2:     return new SUT()                                  ▷ create new SUT instance
3: end function
4: function initialModel
5:     return new Model()
6: end function
7: cmds ← {new IncidentCreateTask(), new IncidentEditTask(),
                                                  new IncidentCloseTask()}
8: function next(model)
9:     return Gen.Elements(cmds)                         ▷ chose one element
10: end function
```

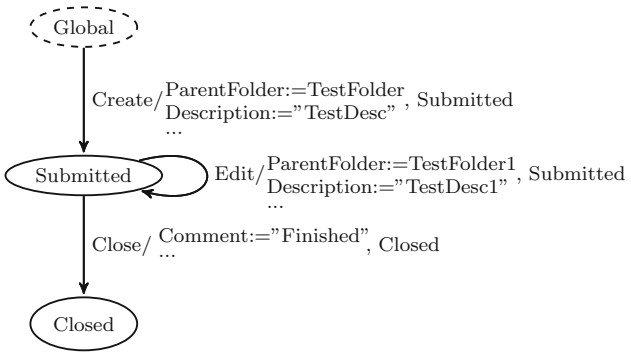

Fig. 4 EFSM of the incident manager

4.2 Example of model-based testing with FsCheck

In this subsection, we show how FsCheck can be applied for model-based testing. A simple example of an incident manager taken from our industrial case study shall serve to demonstrate how the necessary interface implementations have to be realised.

FsCheck modelling In order to use FsCheck for model-based testing, we need a specification class that implements an ICommandGenerator interface and contains the following elements:

- SUT definition (which is called Actual by FsCheck)
- Model definition
- Initial state of the SUT and the model
- Generator for the next command given the current state of the model
- Commands combining preconditions, postconditions and the transition execution semantics of the SUT and the model

Details about the structure of such specifications were already presented in Sect. 4.1. Now we give a more concrete example for FsCheck on an object-oriented level. Algorithm 3 outlines an example specification. In order to implement the interface for FsCheck, we need the mentioned elements. The class of the SUT is basically a wrapper that provides methods for the execution of all tasks and a method to retrieve the current state of one incident object of the SUT. An incident object is an element of the application domain. For example, it could be a bug report. It has a number of attributes (form data), which are stored in the database. In this example, we assume that the attributes are set statically in the wrapper class of the SUT. In Sect. 6, we will see, how form data can be generated automatically for these attributes.

Figure 4 illustrates the state machine of one incident object. Initially, the machine is in a global state. The IncidentCreateTask (abbreviated as Create) creates and opens a new incident object, which can be edited and closed with the corresponding tasks. The transitions are labelled as follows: input i, an optional guard g/assignment operations op, and an output o. The assignment operations of this EFSM assign values to the attribute variables, and the output indicates the target state of a transition. The initial global state has a special meaning: tasks of the global state, i.e. IncidentCreateTask, are globally enabled in all states. Hence, it is possible in every state to create new incident objects. However, to simplify the discussion, we assume that the state machine only represents a currently opened incident object. Generally, in an object-oriented system comprising several objects, we need functionality to switch between active objects. This functionality is discussed in Sect. 5.2.

Algorithm 4 IncidentCreateTask command.

```
1: function pre(model)
2:     return True
3: end function
4: function post(sut, model)
5:     return sut.State = model.State
6: end function
7: function runModel(model)
8:     model.doStep("IncidentCreateTask")
9:     return model
10: end function
11: function runActual(sut)
12:     sut.doStep("IncidentCreateTask")
13:     return sut
14: end function
15: function toString
16:     return "IncidentCreateTask"
17: end function
```

The initial states of the model and SUT are set by creating new objects (Algorithm 3: Lines 2 and 5). The generator in the *next* function selects one element of a command set randomly, which can be accomplished with the default *Elements* generator of FsCheck (Line 9).

In the standard PBT approach, all command classes need to be defined manually as shown in Algorithm 4. The classes need to define how the transitions should be executed on the model and SUT and what postcondition should hold after the execution. In this simple example, the execution of the model only changes the state, later we will also see how we handle form data. For example, the state-changing function of an IncidentCreateTask is defined as follows:

$$model.doStep(\text{“}IncidentCreateTask''\text{)} =_{df} model.s := \text{“}Submitted''$$

Note that in contrast to the previous abstract definition, the postcondition here checks whether the state of the SUT matches the state of the model. Moreover, a *toString* method can be used to display various information of the command and optionally a precondition can be defined. The classes for the IncidentEditTask and the IncidentCloseTask command are similar to this class and are, therefore, omitted.

For a large model with many transitions, it is not practical that all commands have a separate class. Therefore, it makes sense to implement this definition in a more generic way for all possible transitions and to automate the process as far as possible.

Command generation and execution The tool FsCheck generates test cases according to Algorithm 1 with the difference that the specification is provided in an object-oriented style as shown in Algorithm 3. After a test case is generated, it is executed on the SUT and the state of the SUT is compared with the stored model state after each command execution as explained in Algorithm 2.

In order to start testing in FsCheck, the specification has to be converted into a property. This is achieved with the *toProperty*() method of FsCheck. The property can then be tested by calling the QuickCheck() method or also with the help of unit testing frameworks:

```
new Spec().toProperty().QuickCheck();
```

By default, 100 test cases will be generated and executed, but this number can be configured. Listing 2 shows two example sequences that were produced by FsCheck for the incident specification. It can be seen that the sequences have quite different lengths, because FsCheck generates them randomly with a variety of lengths. Moreover, FsCheck classifies the sequences according to their lengths, which can be seen in the last line of the listing. These classifications can be helpful to find out that a certain generator only considers trivial cases. Each of these generated tasks in the command sequences requires form data for the attributes, which also needs to be generated. Listing 3 shows example form data for some attributes that was generated randomly for the IncidentCreateTask. Note that this randomly generated strings form a kind of robustness test in order to check that the SUT can process non-standard input.

5 Application-specific extensions to the method

In this section, we present some application-specific extensions, which were needed for the web-service application that we evaluated. We show the translation of business-rule models into EFSMs, and we demonstrate how we introduced functionality to switch between application objects and REMs.

```
0:
[IncidentCreateTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentCreateTask;IncidentEditTask]

1:
[IncidentCreateTask;IncidentEditTask;IncidentEditTask;
IncidentCloseTask;IncidentCreateTask;IncidentEditTask;
IncidentCloseTask;IncidentCreateTask;IncidentCreateTask;
IncidentEditTask;IncidentCreateTask;IncidentCloseTask;
IncidentCreateTask;IncidentEditTask;IncidentCreateTask;
IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentCreateTask;IncidentEditTask;IncidentCloseTask;
IncidentCreateTask;IncidentEditTask;IncidentEditTask;
IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentCreateTask;IncidentCreateTask;
IncidentCloseTask;IncidentCreateTask;IncidentCreateTask;
IncidentEditTask;IncidentEditTask;IncidentCreateTask;
IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentEditTask;IncidentCloseTask;IncidentCreateTask;
IncidentCreateTask;IncidentCloseTask;IncidentCreateTask;
IncidentCreateTask]
Ok, passed 2 tests, 50
```

Listing 2 Generated command sequences

5.1 Translating business-rule models into EFSMs

In the following, we show how we translate our business-rule models into EFSMs. In order to do this, we introduce a function *translate* which takes an REM as described in Sect. 2.2 as input and converts it into an EFSM. Figure 5 illustrates this process. It can be seen that the translate function is a composition of several sub-functions that convert the individual parts of the REM. First, we translate the states of the REM by applying the helper function *names_of*, which returns the name of all elements of a set of tasks, states or attributes. The name is a unique identifier for each element of these sets. The same helper function is used to translate the attributes, tasks and states to variable, input and output sets. The initial state of the EFSM is set to the constant “Global”. The translation of the transitions is more complex and is performed with the *buildTrans* function. For this translation, we take the states, the tasks and the attributes of the REM as arguments, because we need their fields *possibleTasks* and *possibleNextStates* to form the transitions of the EFSM: the states of the REM form the source states, the tasks that are enabled in these states (*possibleTasks*) represent the inputs, and their possible next states define the target states. Note, there can be multiple possible next states for one task in a specific state. In reality, one of these states is selected by the user, which represents an additional external input. For example, in some REMs an AdminEdit task can be performed, where a user can select the next state of an object, like *Created*,

```
IncidentCreateTask:
—— ParentFolder = IncidentTestFolder1
—— Description = /
B-cfkMNn3—›eA —sb!—R—9/{@bXzF—fb4›LB}SY3—›r i—!-p-x-f
—— CommitNote = HC—Gz_p;
...
```

Listing 3 Generated form data for a task

$$translate : REM \rightarrow EFSM$$
$$translate(rem(attributes, states, tasks)) =_{df} (names\_of(states), "Global",$$
$$names\_of(attributes), names\_of(tasks), names\_of(states), buildTrans(states, tasks, attributes))$$

$$buildTrans : AllStates \times AllTasks \times AllAttributes \rightarrow Transition\_set$$
$$buildTrans(states, tasks, attributes) =_{df} \{(s, i, true, op, s') \mid \exists\ state(sname, possibleTasks) \in states\ .\ (s = sname \wedge$$
$$\exists\ task(tname, tattributes, nextStates) \in tasks\ .\ (tname \in possibleTasks \wedge s' \in nextStates \wedge$$
$$i = tname +\!+ optionalSuffix(nextStates, s') \wedge$$
$$op = (a_1 := v_1, \ldots, a_n := v_n, output) \wedge a_i \in tattributes \wedge (a_i, v_i) \in translate(attributes) \wedge output = s'))\}$$

$$translate : AllAttributes \rightarrow (Variable \times Generator)\_set$$
$$translate(attributes) =_{df} \{(id, gen) \mid attr(id, type) \in attributes \wedge gen = Gen(type) \vee$$
$$attr(id, type, par) \in attributes \wedge gen = Gen(type, par)\}$$

$$names\_of : AllTasks \mid AllStates \mid AllAttributes \rightarrow Name$$
$$names\_of(xs) =_{df} \{x.name \mid x \in xs\}$$

$$optionalSuffix : Name* \times Name \rightarrow Name$$
$$optionalSuffix(possibleNextStates, nextState) =_{df} \begin{cases} "" & \text{if } card(possibleNextStates) = 1 \\ nextState & \text{if } card(possibleNextStates) > 1 \end{cases}$$

Fig. 5 Translation of a rule-engine model to an EFSM by translating the attributes, tasks and states of the rule-engine model to the 6-tuple representing an EFSM

Available or *Deleted*, explicitly from a drop-down menu. Since this selected state forms part of the input, we add the name of the state to the task name to form the input. The function *optionalSuffix* appends $(+\!+)$ the next state s' to the input (task name), if more than one possible next state exists ($cardinality > 1$). In the following, we skip this detail in order to simplify our graphical representations. Furthermore, we translate the required attributes of the tasks in a separate function, which returns a set of pairs of variables and generators (id, gen) for the types of the attributes and their optional parameter. These pairs and an output assignment are used to form the operation sequences of the transition.

5.2 Switching between REM objects

In this subsection, we explain how we extended our models to enable the switching between multiple application objects. Our business-rule models only consider the behaviour of an object of a specific type and not the behaviour of a set of objects. Our application includes a number of transitions that create new application objects, and a user can switch between these objects before a task is started. Our original implementation only considered the currently active object, which is automatically changed when a new object is created. In order to also support the switch functionality, we extended our models. We changed the original variable set V of the EFSM to $V = V_{attr} \cup activeObj \cup stateMap$, where $V_{attr} : Obj.Id \rightarrow (Variable \rightarrow Val)$ are the variables for our attributes. They are now represented as maps, so that different values can be stored for the different objects. The *activeObj* is a variable that marks the currently opened object and it contains an identifier and a state. The variable $stateMap : Obj.Id \rightarrow State$ was added to map object identifiers to object states in order to keep track of the current states of all objects. Moreover, we added additional transitions for selecting an object. These transitions are only enabled when there are at least two objects available, and they are chosen randomly in the same way as other transitions. The decision, which object should be selected, is also performed by a generator. Figure 6 illustrates the select functionality with the additional transitions, which is repented as a hierarchical state machine [13]. Note that transitions that create objects are always active, i.e. they are enabled in all states. Hence, in the following display of EFSMs, we skip their source state.

In the same way as we added transitions to switch between objects of an REM, we also added transitions to switch between different REMs within a module. Figure 7 demonstrates how a switch between multiple REMs of the Test Order Manager can be accomplished with SelectREM transitions. In this example, we have three EFSMs for the different REMs of this module, which are explained in more detail in Sect. 7.2. Each EFSM in this figure also has the select functionality to switch between objects of the REM as shown in Fig. 6. On top of the switch of REMs, also a switch of modules is possible as demonstrated in Fig. 8. We did not implement the switch functionality at this level yet, because we wanted to test the modules separately, but it would be straight forward and very similar to the switch functionality of REMs within a module. In this figure, we have three modules: Test Order Manager, Test Equipment Manager and Test Factory Scheduler. The first two modules are both models we used for case studies in Sect. 7. The Test Factory Scheduler

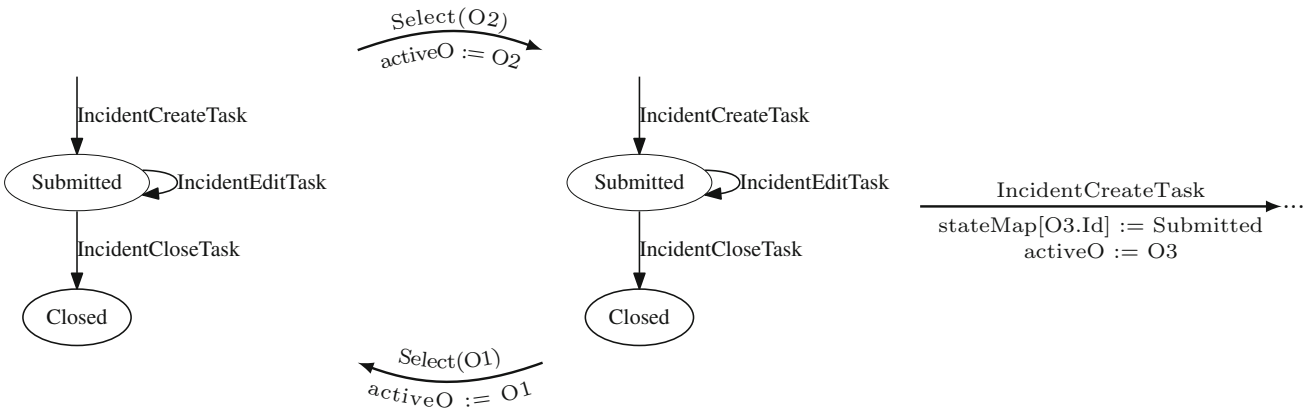

Fig. 6 Switching between objects of the incident object class

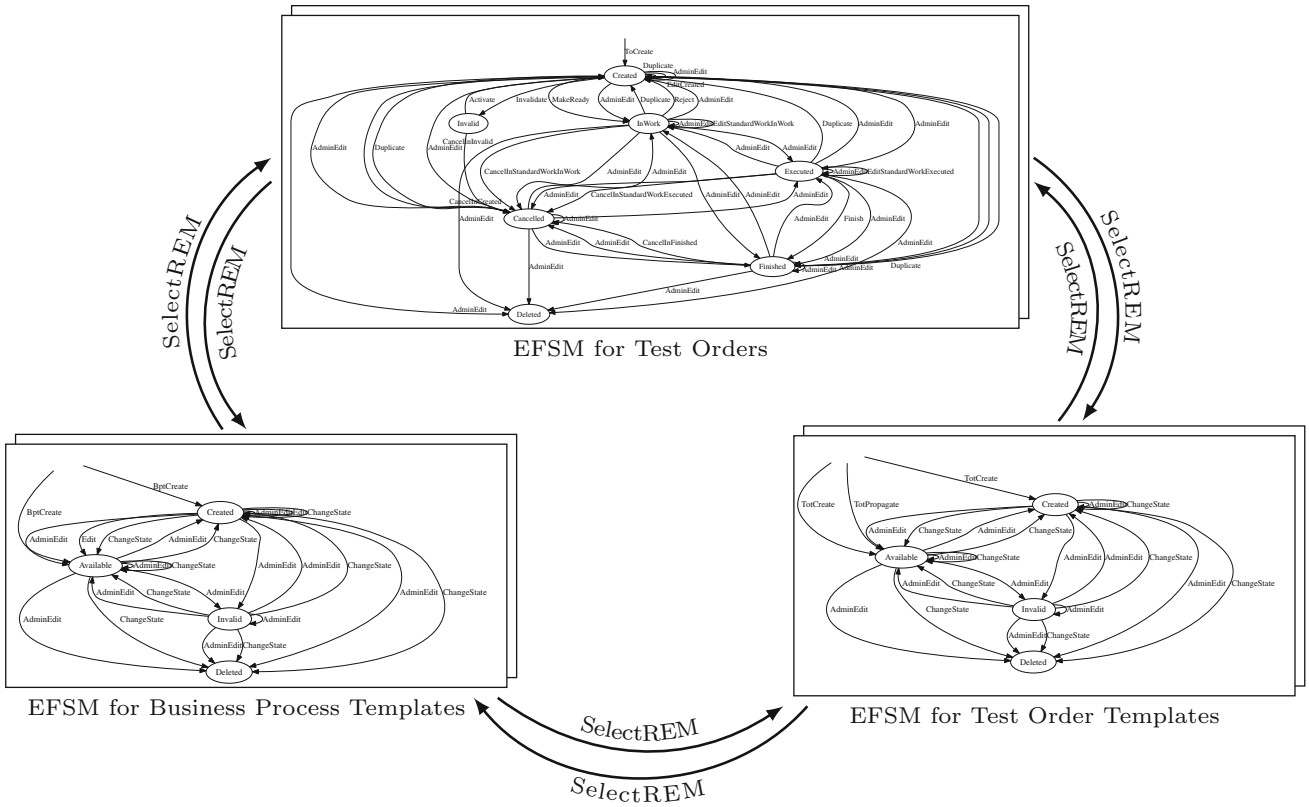

Fig. 7 Switching between rule-engine models inside the Test Order Manager module. For details about the EFSMs see Figs. 10, 11 and 12

module is for scheduling of test orders on the test beds. It can automatically consider availability of resources like test beds and equipment. Test orders can be assigned according to priorities and to meet certain deadlines.

6 Architecture and implementation

In this section, we discuss the architecture and the relevant implementation details of our test case generator.

6.1 Singleton rule-engine models

As explained in Sect. 1, we parse the business-rule models from XML files and translate them into an EFSM. It should be noted that our model representation does not strictly follow the EFSM definition, because we used optimised data structures for the application of FsCheck. However, the semantics of our model combined with FsCheck corresponds to an EFSM. Hence, while the translation to EFSM (in Fig. 5) provides the abstract syntax and the formal semantics of our

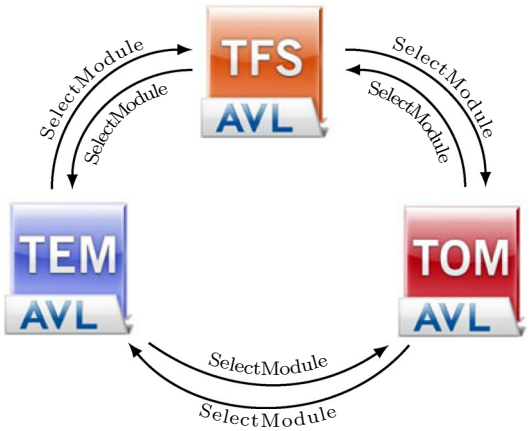

Fig. 8 Switching between modules: Test Order Manager (TOM), Test Factory Scheduler (TFS), Test Equipment Manager (TEM)

models, now we focus on the concrete model implementation in the object-oriented context of FsCheck. Our model is encoded as an object tree, i.e. an abstract syntax tree that serves as input to FsCheck as part of the Spec shown in Definition 4. The class diagram for this object structure is shown in Fig. 9. It can be seen that the model consists of an attribute dictionary (i.e. a map), an initial state, a current state, a list of states and a dictionary of tasks. The transition relation is represented by a class called *Task* that contains hash sets for the possible source and target states of a task, a name and a flag, which indicates that the state should not change after the execution of the task. Furthermore, the class includes a dictionary for the required attributes. The attributes represent the form data of a web-service operation. All attributes have a common base class, which has fields like name and data type. The derived classes for specific data types extend this base class by adding possible constraints and a custom generator for the data type that respects these constraints. For example, an integer attribute class can have constraints for the minimum and maximum value and the generator chooses a number between these boundaries or an arbitrary number if no constraints are given. We have implemented attribute classes for simple data types, like enumerations, doubles, dates and times, but we also support more complex data types:

– *Reference attributes* a reference to another object of the SUT can also be an attribute for a task. The possible options for this object are given by a query, which represents a search string for the database. The interface for

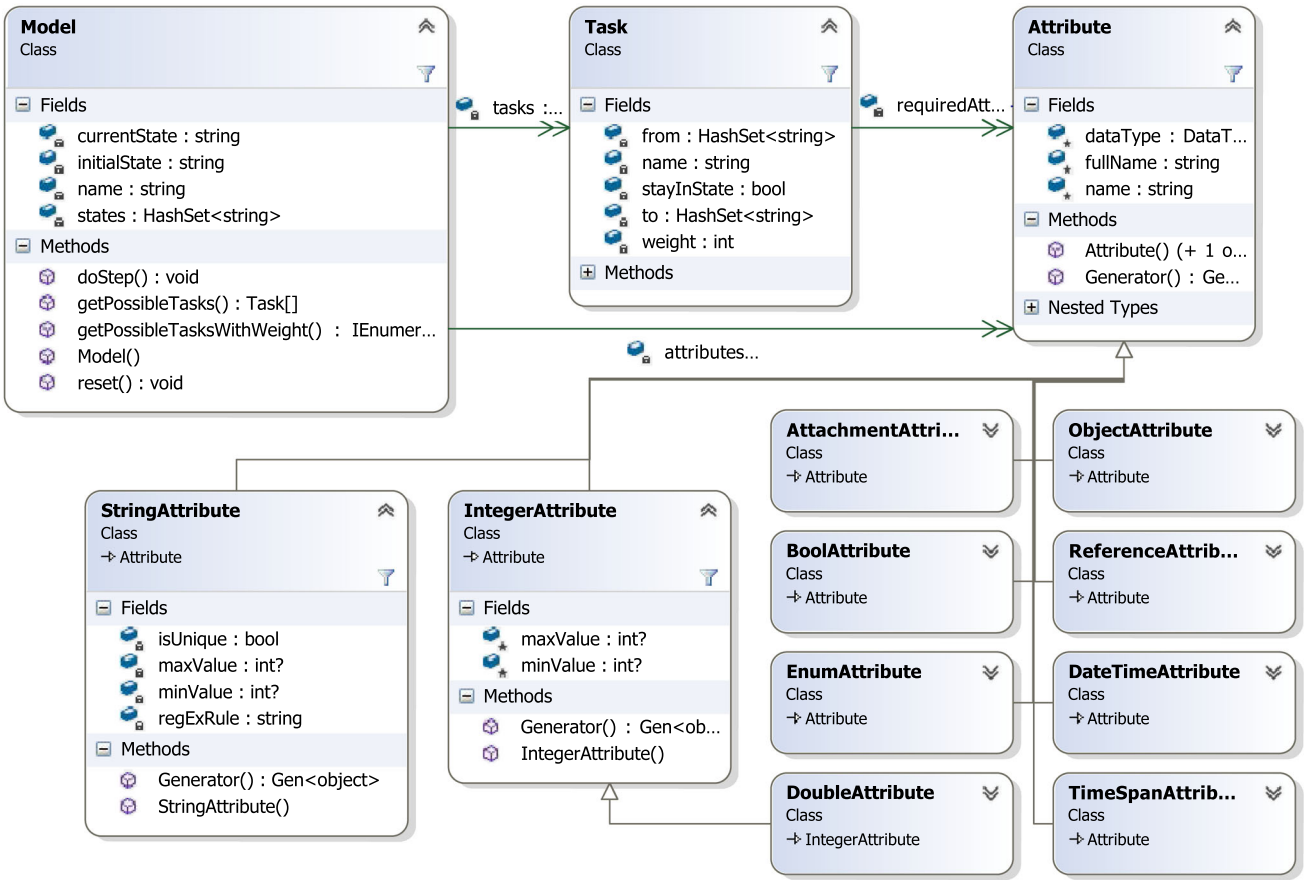

Fig. 9 Class diagram for a model, which is parsed from XML and serves as input to FsCheck as part of the Spec

Algorithm 5 Attribute data generation.

```
Input:
    Attributes: an array of attribute instances
Output:
    attributeData: a generator for maps (Attribute.Name ↦ Val)
1: function GenerateData(Attributes)
2:     for each attr ∈ Attributes do
3:         genArray.Add(attr.Generator())                ▷ fill with Attr. Gen.
4:     end for
5:     return Gen.Sequence(genArray).Select(Values → (
6:         for i ∈ {1, ..., length_of(Attributes)} do -
7:             attributeData[Attributes[i].Name] ← Values[i]
8:         end for
9:         return attributeData))
10: end function
```

the SUT provides a method to get results for a valid query and an element generator chooses one of the results randomly. This generator was already explained in Sect. 4.2.

- *Object attributes* an object attribute can group together multiple attributes in a struct or a list. The generator for this type recursively calls the generators of included types, which can be object attributes again.
- *Attachment attributes* some tasks require files of certain file types. The generator for this attribute chooses one of the possible file types and generates a random file name. The generation of the actual file is added to the wrapper class of the SUT, because the file should also be deleted after the test execution.
- *String attributes* a string attribute may include restrictions like a minimum/maximum length or a regular expression. In order to generate strings that match these regular expressions, we apply a .NET port of the Xeger library.[4] This library can generate text that matches a given regular expression.

The object representation of the model is also used for the interface specifications for FsCheck. For example, preconditions for the restriction of the tasks are automatically created by the model class. The generator for the next command also includes information of the model to generate commands with possible next states and attribute data.

Algorithm 5 shows how attribute data can be generated. First, an array of generators is created by iterating over the attributes and adding the generators to the array. This array is then given to a sequence generator as input, which creates an array of values for all the generators in the array (Line 5). In order to store them in a map, we use the select function of the generator. This function takes an anonymous function, which takes the values as input and returns an object that should be created by the generator. It can be applied to convert a generator of certain type A to a generator of a different type B by processing the generating values of the first generator.

[4] https://code.google.com/archive/p/xeger

Algorithm 6 Next: generates a *Cmd* for a given model.

```
Input:
    model: model instance that incorporates the state
Output:
    gen: a generator for commands
1: function spec.next(model)
2:     ts ← model.getPossibleTasks()                    ▷ possible tasks
3:     return Gen.Elements(ts).selectMany(t →
4:         Gen.Elements(t.PossibleNextStates()).selectMany(s →
5:             GenerateData(t.requiredAttributes).select(data →
6:                 return new DynamicCmd(t, data, s)))
7: end function
```

Hence, the select function has the following signature:

$$GenA.select : (A \rightarrow B) \rightarrow GenB$$

In our case, we build a map generator from a sequence generator in order to enable to generation of maps with attribute names as keys and the data as values (Lines 6 to 9). This attribute data generation is required for the command generation, which is shown in Algorithm 6. First, an array of possible tasks is created in the model class, which considers the preconditions for this creation (Line 2). An element generator is used to choose one of these tasks, and with the selectMany function we process the chosen task (Line 3). The selectMany function is similar to the select function. It can be applied to a generator and requires an anonymous function as argument. This anonymous function takes a value of the generator as input and has to return a new generator.

$$GenA.selectMany : (A \rightarrow GenB) \rightarrow GenB$$

Therefore, selectMany makes it possible to nest generators and also to pass the generated value to the inside generator.

A chosen task can lead to multiple next states; hence, we also choose a next state with an element generator (Line 4). Then, the attribute data generation of Algorithm 5 is applied. With the generated data, we create a DynamicCommand object which takes the task, the model, the attribute data and the next state as arguments for the constructor (Line 6).

The outline of the DynamicCommand class is shown in Algorithm 7. This generic command class can handle the execution of all tasks of the parsed model. The class has the task, the model, attribute data and the next state as constructor

Algorithm 7 DynamicCmd: generic *Cmd* definition.

```
Input: t : task instance, data : map (Attribute.Name ↦ generated value), s : nextState
1: function post(sut, model)
2:     return sut.State = model.State
3: end function
4: function runModel(model)
5:     model.doStep(t, s);
6:     return model
7: end function
8: function runActual(sut)
9:     sut.DefaultTask(t, data, s);
10:    return sut
11: end function
```

arguments and as fields. They are required for the execution of the transition. Running a transition on the model is realised with a function of the model class (Line 5). The execution on the SUT in *runActual* calls the wrapper class of the SUT (Line 9). This wrapper class uses reflection to call the actual methods on the SUT, and it also sets the attributes. In the postcondition, we check whether the after-state of the model is equal to the after-state of the SUT. Note that our SUT allows an explicit observation of the state. This information can be accessed via the wrapper class of the SUT, which allowed us to easily compare the state of the model and the SUT in the postcondition. In other SUTs, this information might not be accessible, e.g. only the output might be comparable.

7 Evaluation

Our approach was developed for a web-service application as discussed in Sect. 2. This system is a composition of various modules. We performed an evaluation of two representative modules of the application, the Test Order Manager and the Test Equipment Manager. Moreover, we also tested other small modules like the Incident Manager, which was shown in the example of Sect. 4.2, but the major part was the Test Order Manager. The goal of the evaluation was to analyse the applicability and bug-finding ability of our method for industrial use cases. The human effort of the evaluation was primarily the implementation of the parsing process of the business-rule models and the connection to the SUT. The human effort for the testing process itself was insignificant, due to the high automation. We found several issues in the systems under test that are listed in the following sections.

7.1 Settings

We performed our experiments in a virtual machine with Windows Server 2008, 4 GB RAM and one CPU on a Macbook Pro (late 2013 version) with 8 GB RAM and a 2.6 GHz Intel Core i5. The system was running TFMS 1.7, and we applied FsCheck 2.4 as PBT tool.

First, we ran the default test settings of FsCheck, which produces 100 test cases with an average length of 51. When an issue has been found, we repeated the whole test process until no more issues were detected.

Additionally, we performed test runs with an increasing number of test cases and a fixed length of ten in order to evaluate the coverage on the models. This ensured that all relevant parts of the SUT were covered. Furthermore, this coverage analysis helped in regression testing. When fixing a detected issue, it gave us confidence that the randomly regenerated test sequences were covering the affected parts of the SUT. The results of this evaluation are presented in the following sections.

7.2 Case study test order manager

The Test Order Manager module controls individual work steps and preparations for automotive test orders and corresponding templates. A test order is a composition of multiple processes that are necessary for a test sequence at an automotive test field. It consists of both organisational and technical processes, which are defined in templates. Organisational processes, e.g. accepting a test order, are defined in a Business Process Template, and a Test Order Template defines technical processes, like preparing a test cell on the test bed. For a Test Order we need to select both, a Business Process

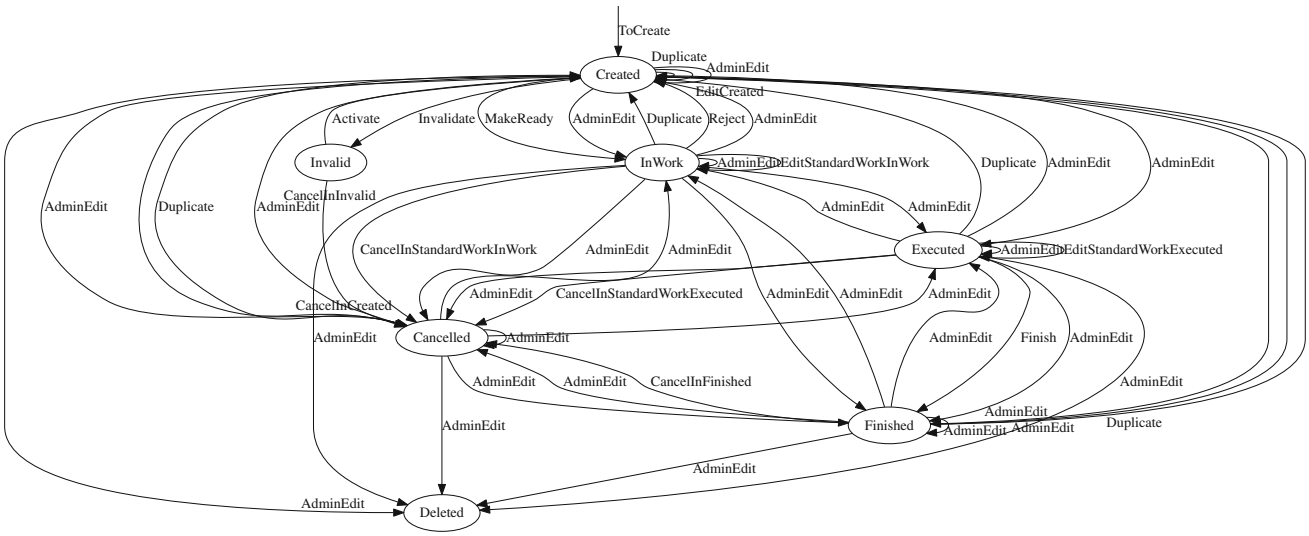

Fig. 10 EFSM for test orders

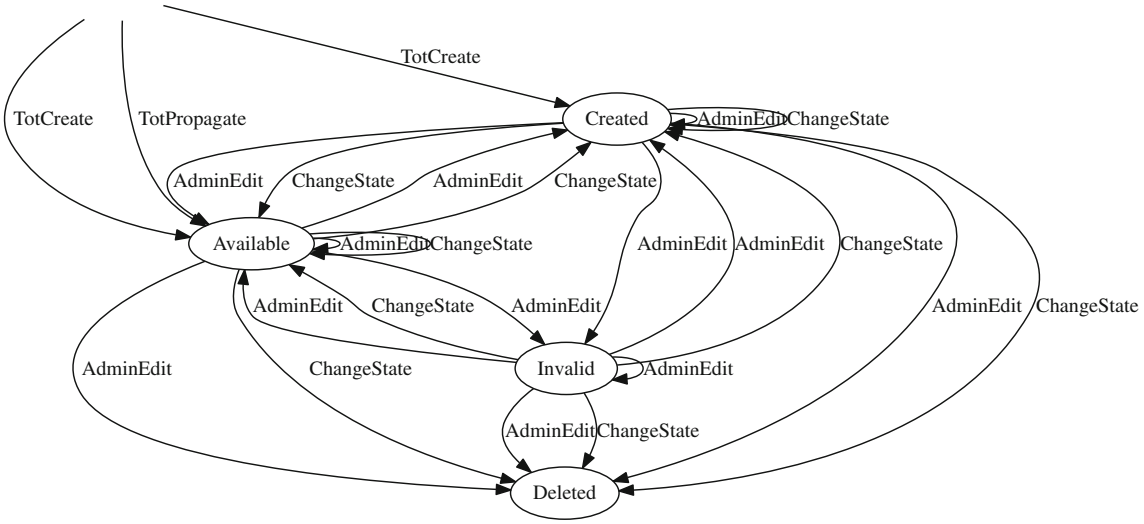

Fig. 11 EFSM for Test Order Templates

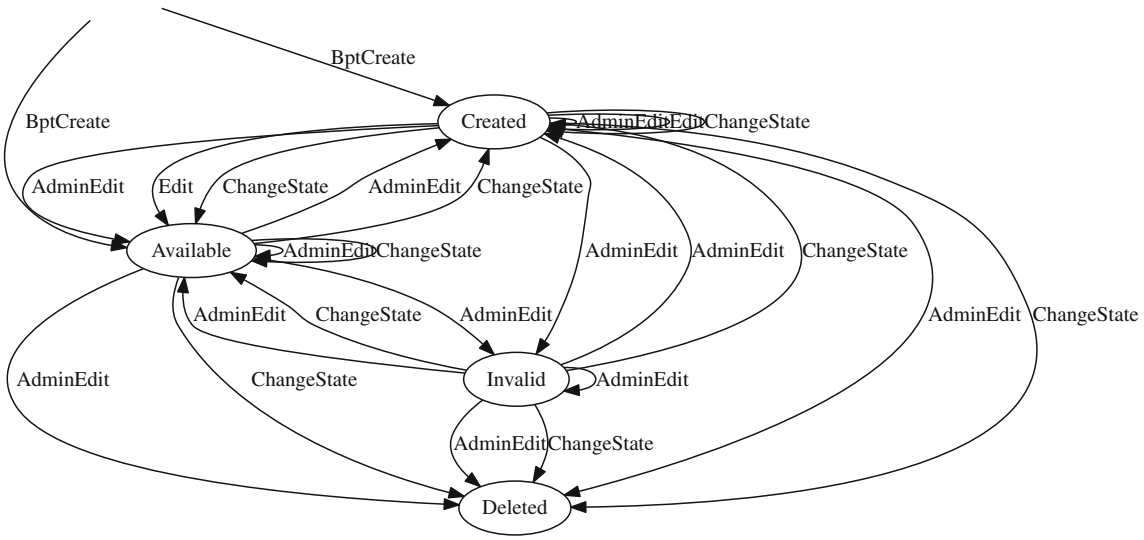

Fig. 12 EFSM for Business Process Templates

Template and a Test Order Template. Test Orders, Business Process Templates and Test Order Templates can be managed individually and they have separate REMs as shown in Fig. 7. A state machine of a test order is shown in Fig. 10. (The REMs of Business Process Templates and Test Order Templates are similar, but they have less states and transitions. They are illustrated in Figs. 11 and 12.) The figure displays only states and transitions, because there are too many attributes to show them. It can be seen that the model contains a number of states for the workflow, respectively, life cycle of a test order.

Table 1 displays the size of the models within the Test Order Manager module. It shows the number of states, tasks, transitions and attributes. It can be seen that the number of possible transitions is high. Therefore, our automated approach makes sense, because otherwise the test of all these transitions would be impractical, especially, since the transitions are not simple actions in this case study. Each transition represents the opening of a page, entering data for form fields and saving the page. One example page of an AdminEdit task

Table 1 Number of states, tasks, transitions and attributes of the REMs within the Test Order Manager

Model	States	Tasks	Transitions	Attributes
Test Order	8	16	49	15
Business Process Templates	5	4	25	58
Test Order Templates	5	4	24	53
Test Order Manager	18	24	98	126

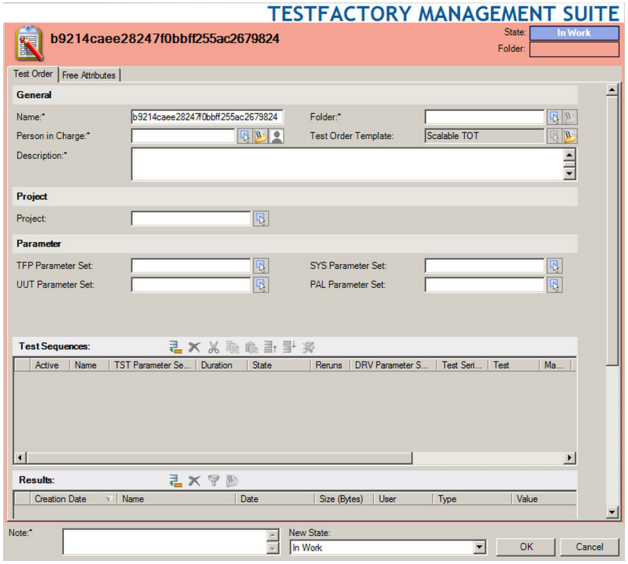

Fig. 13 TFMS form for the AdminEdit task

is shown in Fig. 13. This page is part of the graphical user interface of a client application that connects to web services on a server. It contains a number of form fields and tables that require generated data.

For the case study, AVL provided us a test framework that was specifically developed for the SUT and performs the communication with the web services on the server. It basically represents an abstraction of the graphical user interface and is intended to facilitate the testing effort. A tester should not need to know any web-service details in order to run tests. Hence, the framework offers functions, which perform web-service requests in the background, in order to execute the required steps of the test cases. The framework is written in C# and has interfaces for modules that provide functions for login/logout, executing tasks, opening domain objects, retrieving data and so on. We call these functions, e.g. to start tasks (representing the opening of forms), to set attributes (of form data) and to save forms.

The case study revealed the following problems and bugs:

1. There was a bug in the original testing framework that was provided by our industrial partner. The expected state after a task execution was sometimes wrong, because in some cases an old version of the object was used by the testing framework.
2. Another issue we detected concerns our test case generation method. In some rare cases, the business models do not contain enough information. For example, there were reference attributes that could not be changed to a different subtype after an object was created. The query for these attributes needed an additional restriction for the subtype. This information was correctly implemented in the code, but is missing in the rule-engine model. Hence, the tool reported a bug that in fact was not a bug. It is rather a limitation of the approach of relying on the business-rule models as primary source for the test case generation.

Table 2 Average number of commands needed for finding the issues of the Test Order Manager

Issue	Number of *Cmd*s
1	–
2	1.4
3	23.8
4	9.4

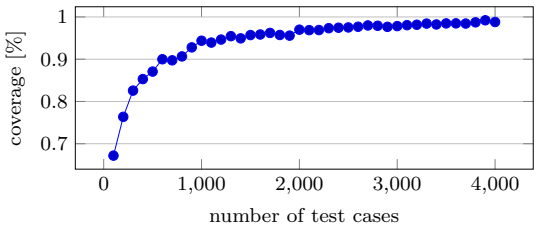

Fig. 14 Test Order Manager: transition coverage for increasing number of test cases (with test cases of length ten)

The following bugs were found in hidden tasks that were not enabled in the user interface. These tasks remained in the business-rule models, and they would cause problems when they were enabled again. Therefore, we also tested them.

3. There were tasks that first resulted in an exception, which stated that certain attributes are missing. However, when the attributes were set, it resulted in an exception that said that the attributes are not enabled.
4. There was a problem with a task that had a next state in the model, which was not permitted by the SUT. Furthermore, the error message of the SUT was wrong in this case. It should list possible next states. However, the list did not contain states, but tasks.

Table 2 presents the average number of commands that were needed to find the issues. The average was computed over five test runs. Note that for the first issue no data is available, as the bug in the testing framework was fixed in an early state of the evaluation. The data show that Issue 3 is especially hard to find as it requires on average of more than 23 input commands until its detection.

We monitored the coverage of our tests on the model in order to obtain confidence that we tested enough. The states and tasks of the model were covered with a small number of tests and are, therefore, omitted. The transition coverage for an increasing number of test cases is illustrated in Fig. 14. The test case length is fixed to ten, and the coverage is given as the average percentage of the transitions that are visited during 100 test runs with the same number of samples. Due to

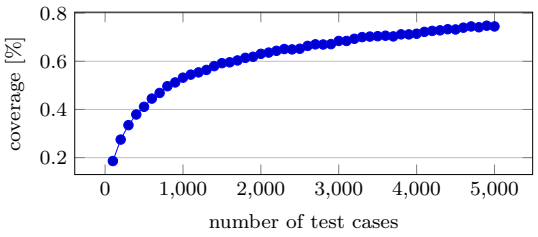

Fig. 15 Test Order Manager: transition-pair coverage for increasing number of test cases (with test cases of length ten)

the high number of transitions, we need about 4,000 test cases in order to obtain an average transition coverage that is close to 100%. We performed the same evaluation for transition-pair coverage, which is also called 1-switch coverage after Chow [7]. For this coverage criterion, we evaluate how many sequences of two consecutive transitions are observable with our test cases. The results are shown in Fig. 15. Transition-pair coverage requires even more test cases, e.g. 5,000 test cases only produce an average transition-pair coverage of 75%.

7.3 Case study test equipment manager

Similar to the Test Order Manager, we performed a case study for the Test Equipment Manager module. The main function of this module is the administration of all equipment that is relevant for the test field, like test beds, measurement devices, sensors, actuators and various input/output modules. All these test equipment can be created, edited, calibrated and maintained with the Test Equipment Manager. A hierarchy of test equipment types is used to classify the test equipment. Test configurations, which are compositions of different test equipment, can also be administrated, and also the connection of devices via channels can be controlled with this module.

Figure 16 illustrates the main REMs and the complexity of the Test Equipment Manager. It can be seen that the EFSM for test equipment has many transitions for maintenance and administration purposes. Most of the state names are self-explanatory. The state *Invalid* is for an object that was copied and has to be adapted. *Mounted* is a state that means that the equipment was installed in the test field. The EFSM for test equipment types is smaller but similar, because it does not contain maintenance operations. Details about the behaviour of the REMs are omitted, because they are too specific for the SUT and not relevant for this work. The size of the module and its REMs is summarised in Table 3, which shows the number of states, task, transitions and attributes. In contrast to the Test Order Manager, we only have two REMs and the module is not as complex.

We found a number of issues which are listed below. It should be noted that the case study was performed with test rule-engine files, which are not used by actual customers and which were not inspected as intensively as productive rule-engine files. However, if productive rule engines would contain these kinds of issues, then our tests could also find them. The following two issues could be found with strings by utilising our string generators, which support the generation of strings with regular expressions.

1. Inconsistency regarding the use of tab characters in names could be found. It was never planned that the object names should support tabs. On some occasions, these characters were replaced with blanks, but not consistently. Blanks were still saved in the database and only replaced, when they were sent to the graphical user interface. Therefore, two entries could be created that were indistinguishable, because both a name containing a tab and a blank were presented in the same way by the SUT.
2. Another problem found was that the regular expressions for several names in our REMs were insufficient. We assume that these regular expressions were designed to prevent certain special characters, and no blanks should be allowed at the end and at the beginning. However, the regular expressions were written so that they allowed all non-white space characters at the beginning and at the end of the string, even characters that are not allowed in the middle of the string. We could observe this issue when we tested the copy functionality, which duplicates an object and appends an underline and a number to its name. When certain special characters were at the end of the string, then the name was not valid any more, after a copy operation. This was, because the special character moved from the end to the middle of the string, where they were not allowed due to the regular expressions.

Further issues could be found concerning misconfigurations in the REMs and unsupported functionality of the provided test framework.

3. An issue was found with required attributes. In a particular task, an attribute was required, but it could not be edited, because it was not enabled for this task. Therefore, it was not possible to complete this task, except the user returned to a previous task and edited the attribute there.
4. We found a task that was not supported by the test framework. The task could be triggered with the test framework, but resulted in an exception. In the graphical user interface, the task could be executed normally. Hence, we found a task that was not completely implemented in the test framework and could not be tested automatically, because without support of the test framework only a manual test via the graphical user interface was possible.

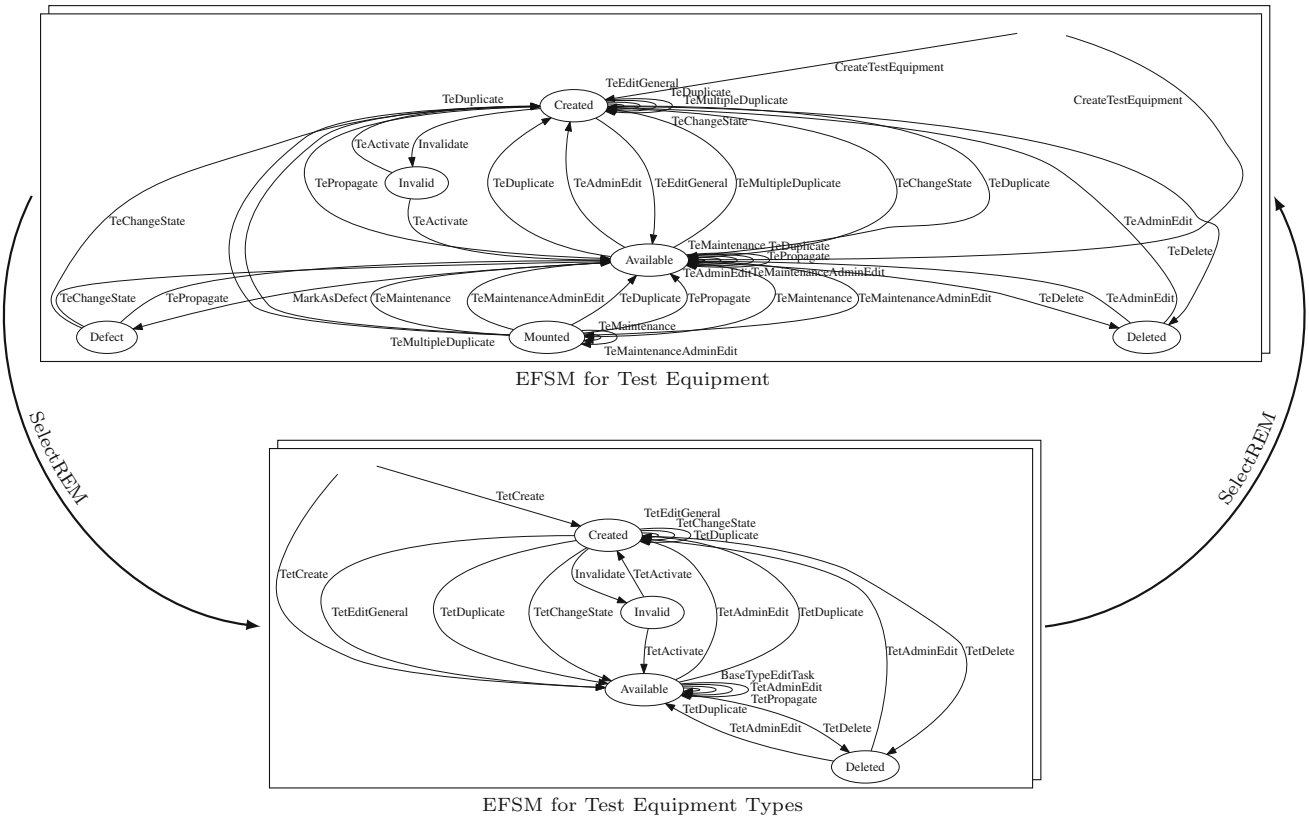

Fig. 16 EFSM for the rule-engine models of the Test Equipment Manager module

Table 3 Number of states, tasks, transitions and attributes of the REMs of the Test Equipment Manager

Model	States	Tasks	Transitions	Attributes
Test equipment type	5	10	21	43
Test equipment	7	13	39	23
Test equipment manager	12	23	60	66

Table 4 illustrates the average number of commands that were needed to find the issues. The numbers were computed in the same way as described for the Test Order Manager module in Sect. 7.2. The first issue was particularly hard to find. The reason is that the generator for strings does not generate a tab character very often, because it is only one of many options and the same string with a blank was also generated very rarely. We also monitored the average transition and transition-pair coverage for an increasing test case number, as already shown for the Test Order Manager module. The results are shown in Figs. 17 and 18. Due to the lower complexity of this module, only about 150 test cases are needed to reach a transition coverage that is close to 100% and about 2.500 test cases for transition-pair coverage.

Table 4 Average number of commands needed for finding the issues of the Test Equipment Manager

Issue	Number of *Cmd*s
1	467.4
2	12.4
3	17.6
4	9

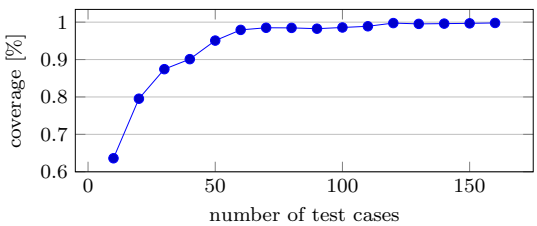

Fig. 17 Test Equipment Manager: transition coverage for increasing number of test cases (with test cases of length ten)

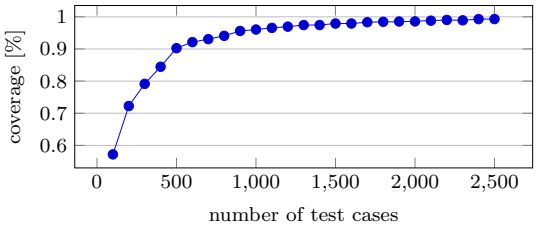

Fig. 18 Test Equipment Manager: transition-pair coverage for increasing number of test cases (with test cases of length ten)

7.4 Further result

Another bug was not directly found with our test cases, but during the extensions of our models with the select functionality as described in Sect. 5.2. In order to implement this functionality, we had to evaluate the behaviour of the SUT, because it is not described in the rule-engine models. During this evaluation, we could observe a bug that occurred when we tried to open a window for a module while a task was executed in an already open window. In this case, the window crashed and it was not possible to open a new window for the module until it was terminated via the task manager. Note, this bug was not directly found with our automated method, but was discovered by the authors during the model design. However, in other model-based testing approaches, it is also often the case that bugs are found in this phase. Hence, finding such bugs can be seen as a positive by-product of applying model-based testing in general.

8 Discussion

8.1 Limitations and threads to validity

The evaluation demonstrated that our method of using business-rule models for PBT is able to find bugs in the real system. Moreover, we showed that our randomly generated test cases are able to achieve a high transition coverage with an acceptable number of tests. A limitation of random testing is that certain coverage criteria cannot be guaranteed, and so important aspects of the SUT might not be tested. Hence, a more targeted test case generation strategy might be able to find more bugs. However, the random generation was especially helpful for the creation of complex form data, which was required for our SUT and it was able to cover most of the model with few tests. Hence, we did not evaluate other generation strategies. It should be noted that also the number of test cases is important for random testing. Depending on the size of the model, it is crucial to generate enough test cases in order to achieve an acceptable transition coverage.

Another limitation of our approach is that we rely on business rules as test models and oracles. In an ideal implementation, we could only test whether the business rules are interpreted correctly. However, for our SUT, we saw that there are a number of deviations of the SUT from the business rules. In other applications, this might not be the case. Furthermore, it should be noted that we are only able to find bugs that are caused from a deviation of the SUT from the business rules. A manually crafted model might be able to find more bugs due to a better oracle, but it is expensive to create a model manually.

A limitation of relying on the business rules can also be that the business rules might not contain enough information, e.g. not all data constraints that are present in the SUT might also be encoded in the business-rule models. In such cases, a manual intervention might be needed. This was already mentioned in Issue 2 of the Test Order Manager case study in Sect. 7.2.

An external thread to the validity of our method is that the random generation of a PBT tool might not be random enough. For example, there can be problems when the random generation is based on the system time or when multiple threads share a common random generator. In order to eliminate this thread, we analysed our generated command sequences for suspicious patterns and we made sure that the random generation functionality was implemented according to common practice.

An internal thread to the validity of our evaluation might be the research bias, which can come in different shapes. (1) We might have selected an SUT that has particular faults in order to support our approach. However, we did not select the SUT for our evaluation. It was given to us by our industrial partner AVL, and this was done before we had a particular testing method in mind. Hence, we had no influence over the choice of the SUT. (2) We could have found issues that are no real problems of the SUT. The fact that we had to present our findings to AVL and also that they had to confirm our found issues before we were allowed to publish them dissolves this thread. (3) We could have targeted our testing method towards specific bugs that were present in the SUT. This would limit the type of faults that can be found, but we did not know the bugs of the SUT beforehand. They were revealed by our evaluation. Hence, it was not possible for us to target our testing method towards specific known bugs of the SUT.

Another internal thread to the validity is that we only tested our method with a specific system. One could argue that a case study is not enough to evaluate the applicability or generality of our method. However, we did evaluate two modules of one big web-service application. These modules have different functionality and can be used independently. Therefore, we think that the evaluation of two modules is sufficiently representative for this application domain.

Algorithm 8 Command generation with frequencies.

```
Input:
  model: model instance that incorporates the state
Output:
  gen: a generator for commands
1: function spec.next(model)
2:   for each t ∈ model.getPossibleTasksWithWeight() do
3:     wv.add((t.Weight, Gen.Constant(t)))          ▷ fill array with
4:   end for                                        ▷ weight and generator pairs
5:   return Gen.Frequencies(wv).selectMany(t → ...)
6: end function
```

8.2 Future work

A nice feature, which is also supported by FsCheck, is the command generation with different frequencies. This feature makes it possible to test certain problematic tasks more frequently and also to simulate user behaviour. Usually commands are generated randomly with a uniform distribution, but it is also possible to generate commands according to certain probability distributions, which can be specified by the probability mass function, respectively, weights for the tasks.

Algorithm 8 shows how this can be done. We iterate over the possible tasks and fill the wv array with weight and generator pairs. In this case, the generator is a constant generator, which simply generates a task instance. With the FsCheck generator *Gen.Frequency*, one task is selected according to the weights.

$$Gen.Frequency : \mathcal{P}(\mathbb{R}_{>0} \times Gen) \rightarrow Distr(Gen)$$

The remaining part of the command generation is the same as in Algorithm 6 and is therefore omitted. Note that we did not apply this generation method with frequencies for the evaluation, because we had no data for the probabilities of tasks. However, we plan to utilise this feature in the future in order to simulate typical user behaviour.

Additional case studies would also be an option for future work. In order to analyse the generality of our method, it would make sense to test further applications that are driven by rule engines. Moreover, a comparison with other testing methods, like manual unit testing, would be an interesting option. This might help to assess the bug-finding performance of our approach.

Another potential topic for future work would be fuzzing. With our current method, we only test the behaviour of the SUT that is allowed by the business rules. However, it would also be important to test behaviour that is outside the scope of the business rules, i.e. invalid behaviour. This could be done by specifying generators that generate data that is not allowed by the business rules, e.g. tasks that are not enabled in a specific state, or form data that does not meet certain restrictions. By generating invalid data, we could check whether the error handling works as expected and also whether the business rules are applied correctly.

9 Conclusion

We have developed an automatic test case generation approach for business-rule models of a web-service application. The approach is based on property-based testing and written in C# with the tool FsCheck.

First, we presented our rule engine-driven web application under test. We introduced property-based testing, formalised its underlying concepts and algorithms and illustrated the tool FsCheck. Model-based testing with FsCheck was demonstrated with a small example of an incident management system. Then, we presented a formalisation of the translation of business-rule models to EFSMs. Next, we discussed how our approach works in detail. It takes XML files with the business-rule models as input and converts them into an EFSM in the form of an object representation that is used for FsCheck specifications and as model.

FsCheck can automatically derive command sequences from a specification, and it executes them directly on the SUT. Our approach also includes attribute data generation for simple and complex data types like objects, references or files. The attributes support a variety of constraints, which are also encoded in the XML business-rule models.

We evaluated our approach in two industrial case studies, which were applied to a web-service application from AVL, a testfactory management suite. The generated test cases were analysed by measuring their transition and transition-pair coverage. We found eight issues that were confirmed by AVL. This demonstrates that the approach is effective in finding bugs. The issues concerned the system-under-test, the testing framework and the business-rule models. The fact that not all bugs are due to the system-under-test is well known in the area of automated testing.

One may wonder about the missing redundancy when generating the test models from the business rules. When a rule engine would be implemented optimally, then our approach would only test the interpreter of the business-rule models. However, in practice the programmers often change the source code without considering the rules. Hence, it makes sense to verify that the SUT still conforms to the model. Especially for custom rule-engine implementations and evolving applications, it is important to test this conformance. Therefore, we have developed an automated approach that verifies this conformance efficiently.

In future, we plan to apply these test cases in load testing, where the question of redundancy is irrelevant.

Acknowledgements Open access funding provided by Graz University of Technology. The research leading to these results has received

funding from the Austrian Research Promotion Agency (FFG), Project Number 845582, Trust via cost function-driven model-based test case generation for non-functional properties of systems of systems (TRUCONF). The authors would like to thank Florian Lorber and Martin Tappler for their helpful comments and suggestions to improve the quality of the paper. The authors would like to acknowledge the team at AVL, especially Elisabeth Jöbstl, Severin Kann, Manfred Uschan and Christoph Schwarz, who have provided us support regarding the web application under test and also valuable ideas and feedback. Furthermore, we are grateful to the FsCheck community and Kurt Schelfthout for providing FsCheck and support and to Silvio Marcovic for his help in the case studies.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M.: Business process management: a survey. In: Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, 26–27 June 2003, Proceedings, Springer, Lecture Notes in Computer Science, vol. 2678, pp. 1–12 (2003). https://doi.org/10.1007/3-540-44895-0_1
2. Aichernig, B.K., Schumi, R,: Property-based testing with FsCheck by deriving properties from business rule models. In: 2016 IEEE Ninth International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 13th Workshop on Advances in Model Based Testing (A-MOST 2016), IEEE, pp. 219–228 (2016). https://doi.org/10.1109/ICSTW.2016.24
3. Arts, T., Hughes, J., Johansson,J., Wiger, U.: Testing telecoms software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang (ERLANG'06), ACM, pp. 2–10 (2006). https://doi.org/10.1145/1159789.1159792
4. Arts, T., Hughes, J., Norell, U., Svensson, H.: Testing AUTOSAR software with QuickCheck. In: 2015 IEEE Eighth International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), IEEE, pp. 1–4 (2015). https://doi.org/10.1109/ICSTW.2015.7107466
5. Charfi, A., Mezini, M.: Hybrid web service composition: business processes meet business rules. In: Proceedings of the Second International Conference on Service Oriented Computing (ICSOC'04), ACM, pp. 30–38 (2004). https://doi.org/10.1145/1035167.1035173
6. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Proceedings of the 30th International Design Automation Conference (DAC'93), ACM, New York, NY, USA, pp. 86–91 (1993). https://doi.org/10.1145/157485.164585
7. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978). https://doi.org/10.1109/TSE.1978.231496
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM, pp. 268–279 (2000). https://doi.org/10.1145/351240.351266
9. Claessen, K., Hughes, J.: Testing monadic code with QuickCheck. SIGPLAN Not **37**(12), 47–59 (2002). https://doi.org/10.1145/636517.636527
10. Earle, C.B., Fredlund, L., Herranz-Nieva, Á., Mariño, J.: Jsongen: a QuickCheck based library for testing JSON web services. In: Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang (Erlang'14), ACM, pp. 33–41 (2014). https://doi.org/10.1145/2633448.2633454
11. Francisco, M.A., López, M., Ferreiro, H., Castro, L.M.: Turning web services descriptions into QuickCheck models for automatic testing. In: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang (Erlang'13), ACM, pp. 79–86 (2013). https://doi.org/10.1145/2505305.2505306
12. Fredlund, L., Benac Earle, C., Herranz, A., Marino, J.: Property-based testing of JSON based web services. In: 2014 IEEE International Conference on Web Services (ICWS), IEEE, pp. 704–707 (2014). https://doi.org/10.1109/ICWS.2014.110
13. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
14. Herbst, H.: Business Rule-oriented Conceptual Modeling. Physica, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59260-7
15. Hughes, J.: QuickCheck testing for fun and profit. In: Practical Aspects of Declarative Languages, LNCS, vol. 4354, Springer, pp. 1–32 (2007). https://doi.org/10.1007/978-3-540-69611-7_1
16. Hughes, J., Pierce, B., Arts, T., Norell, U.: Mysteries of DropBox: property-based testing of a distributed synchronization service. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE (2016). https://doi.org/10.1109/ICST.2016.37
17. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM). In: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation (ICST'09), IEEE, pp. 230–239 (2009). https://doi.org/10.1109/ICST.2009.29
18. Lampropoulos, L., Sagonas, K.F.: Automatic WSDL-guided test case generation for PropEr testing of web services. In: Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems, EPTCS, vol. 98, pp. 3–16 (2012). https://doi.org/10.4204/EPTCS.98.3
19. Li, H., Thompson, S., Lamela Seijas, P., Francisco, M.A.: Automating property-based testing of evolving web services. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14), ACM, pp. 169–180 (2014). https://doi.org/10.1145/2543728.2543741
20. López, LM., Ferreiro, H., Arts, T.: A DSL for web services automatic test data generation. In: Draft Proceedings of the 25th International Symposium on Implementation and Application of Functional Languages (2013)
21. Milanovic, N., Malek, M.: Current solutions for web service composition. IEEE Internet Comput. **8**(6), 51–59 (2004). https://doi.org/10.1109/MIC.2004.58
22. Nilsson, R.: ScalaCheck: The Definitive Guide. IT Pro, Artima Incorporated (2014)
23. Orriëns, B., Yang, J., Papazoglou, M.: A framework for business rule driven service composition. In: Technologies for E-Services: 4th International Workshop, TES 2003, LNCS, vol. 2819, Springer, pp. 14–27 (2003). https://doi.org/10.1007/978-3-540-39406-8_2
24. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang (Erlang'11), ACM, pp. 39–50 (2011). https://doi.org/10.1145/2034654.2034663
25. Rosenberg, F., Dustdar, S.: Business rules integration in BPEL-a service-oriented approach. In: 7th IEEE International Conference on E-Commerce Technology (CEC 2005), 19–22 July 2005, München, Germany, IEEE, pp. 476–479 (2005a). https://doi.org/10.1109/ICECT.2005.25
26. Rosenberg, F., Dustdar, S.: Business rules integration in BPEL-a service-oriented approach. In: Proceedings of the Seventh IEEE

International Conference on E-Commerce Technology (CEC05), IEEE, pp. 476–479 (2005b). https://doi.org/10.1109/ICECT.2005.25
27. Rosenberg, F., Dustdar, S.: Design and implementation of a service-oriented business rules broker. In: 7th IEEE International Conference on E-Commerce Technology Workshops (CEC 2005), IEEE, pp. 55–63 (2005c). https://doi.org/10.1109/CECW.2005.10
28. Ross, R.G.: Principles of the Business Rule Approach. Addison-Wesley Professional, Boston (2003). https://doi.org/10.1016/j.ijinfomgt.2003.12.007
29. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell'08), ACM, pp. 37–48 (2008). https://doi.org/10.1145/1411286.1411292
30. Svenningsson, R., Johansson, R., Arts, T., Norell, U.: Testing AUTOSAR basic software modules with QuickCheck. Adv. Math. Comput. Tools Metrol. Test. IX **84**, 391 (2012)
31. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**(5), 297–312 (2012). https://doi.org/10.1002/stvr.456
32. Wada, Y., Kusakabe, S.: Performance evaluation of a testing framework using QuickCheck and Hadoop. J. Inf. Process. **20**(2), 340–346 (2012). https://doi.org/10.2197/ipsjjip.20.340
33. Wagner, G., Antoniou, G., Tabet, S., Boley, H.: The abstract syntax of RuleML-towards a general web rule language framework. In: 2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2004), 20–24 Sept 2004, Beijing, China, IEEE Computer Society, pp. 628–631 (2004). https://doi.org/10.1109/WI.2004.134

Bernhard K. Aichernig is a tenured associate professor at Graz University of Technology, Austria. He investigates the foundations of software engineering for realising dependable computer-based systems. Bernhard is an expert in formal methods and testing. His research covers a variety of areas combining falsification, verification and abstraction techniques. Current topics include the Internet of Things, model learning and statistical model checking. Since 2006, he participated in four European projects. From 2004 to 2016, Bernhard served as a board member of Formal Methods Europe, the association that organises the Formal Methods symposia. From 2002 to 2006, he had a faculty position at the United Nations University in Macao S.A.R., China. Bernhard holds a habilitation in Practical Computer Science and Formal Methods, a doctorate, and a diploma engineer degree from Graz University of Technology.

Richard Schumi is a Ph.D. student at Graz University of Technology. His research interests are in model-based testing, property-based testing and statistical model checking. He works for his Ph.D. in a local project which is concerned with testing non-functional properties of systems of systems. Richard holds master and bachelor degrees in computer science from Graz University of Technology. He is also a qualified engineer in network and information technology.