Property-based Testing with FsCheck by Deriving Properties from Business Rule Models

Bernhard K. Aichernig and Richard Schumi
Institute of Software Technology, Graz University of Technology, Austria
{aichernig,rschumi}@ist.tugraz.at

Abstract—**Previous work has demonstrated that property-based testing can successfully be applied to web services. For example, it has been shown that JSON schemas can be used to automatically derive test-case generators for web forms. This paper presents a test-case generation approach for web services that takes business rule models 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. Most of the work in this field applies property-based testing 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.**

I. 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 [1]. Properties can range from simple algebraic equations to complex state machine models. Like in all model-based testing techniques the properties serve as a source for test-case generation as well as test oracles. It is a well-known testing practice in functional programming, but nowadays we see a growth of applications outside its traditional domain. Examples include the automated testing of automotive software [2] and web-services [3]. In this work we will focus on the latter.

Many web services store configuration settings in XML files. Some web services also store work flow details and user access rules in XML business rule models [4], [5]. These XML definitions can be seen as an abstract specification of the service behaviour, which can be used to verify whether the service complies to this specified behaviour [6]. 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 an object-oriented way with C#. We use C#, because it is the application language of our case study and because the system-under-test (SUT) has an object-oriented architecture.

The process of our approach is illustrated in Figure 1. The first step is to translate the XML 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)[7]. 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).

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.

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

López et al. presented a domain-specific language (DSL) for automatic test data generation with QuickCheck, which is easy to use for non-experts [8]. 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 of 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 [3] 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 if no error occurred.

A similar approach was presented by Li et al. [9]. 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 makes it easy to adapt the 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 to our work is that their models have to be made manually by the user and that their

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

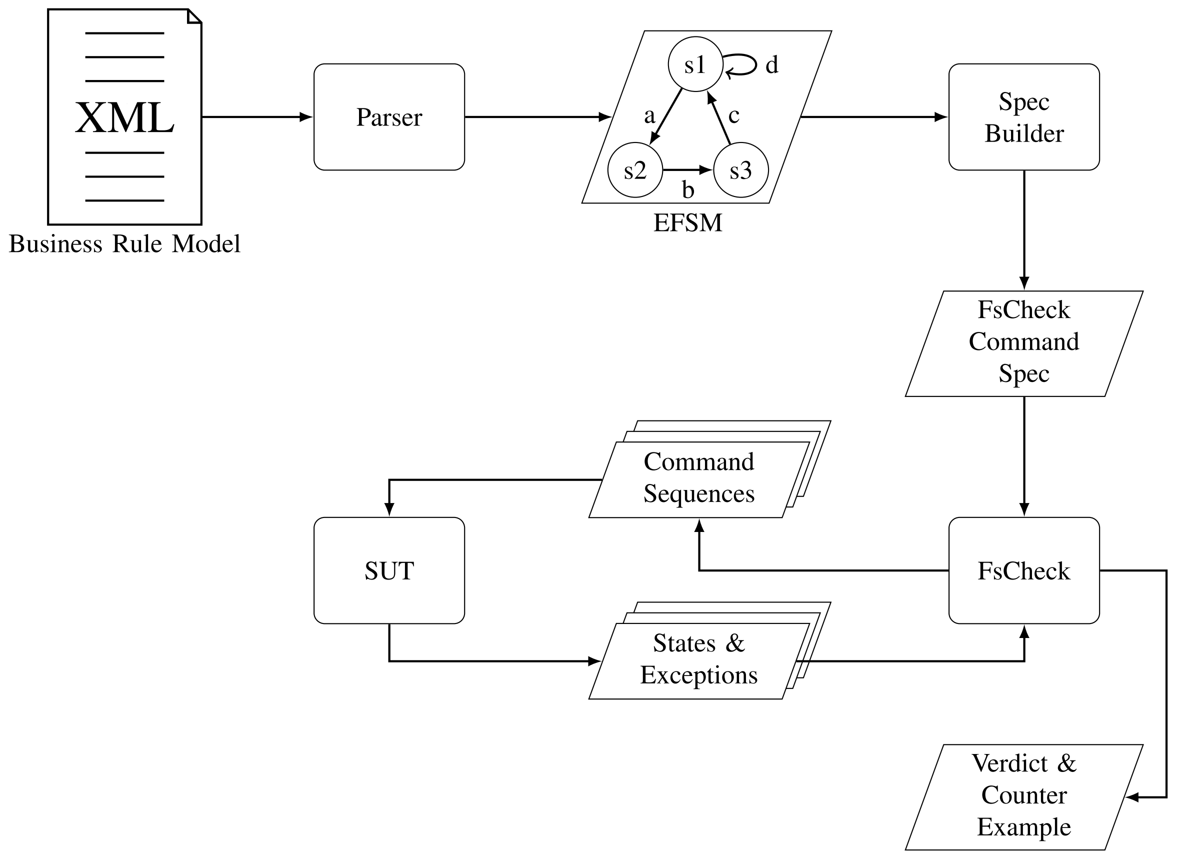


Fig. 1. Overview of the Steps for the FsCheck Command Sequence Generation for Business Rule Models

focus lies only on evolving web services.

Frelund et al. [10] 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.

Benac Earle et al. [11] 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 hyper links, 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 made 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 real case study.

The most similar work to ours was presented by Francisco et al. [12]. 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 structure, 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, we could not find any 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.

Consequently, the contribution of our work can be summarised as follows. The main contribution is a new approach that uses XML business rule models in the form of EFSMs as input for PBT. Another contribution is the application and evaluation of our approach in an industrial case study. Furthermore, our work seems to be one of the first publications about FsCheck and its C# version.

The rest of the paper is structured as follows. First, Section II will explain the basics of PBT, FsCheck and business rule models. Next, in Section III we show a small example of model-based testing with FsCheck. Then, in Section IV we describe details about the structure and implementation of our approach. Section V shows an industrial case study where our approach was applied to a real world application. Finally, the work is concluded in Section VI.

II. Background

A. Property-based Testing

Property-based testing (PBT) is an online testing technique. In contrast to off-line testing, tests are not first stored and then executed, but directly executed while they are generated. Properties are expressed as functions, which contain code that should be tested. When the function runs through without exception, then the property passed, otherwise a counter example is returned. The simplest properties are functions with a Boolean return value that should be true, when the function runs as expected. The 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 smaller failing counterexamples. The search method can be specified individually for different data types [13], [14], [15].

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

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

A PBT tool will generate a series of random lists xs, execute the reverse function and evaluate the property.

PBT can also be applied for models in the form of extended finite state machines (EFSMs) [16]. An EFSM can formally be described as a 6-tuple (S, s_0, V, I, O, T)
S is a finite set of states,
$s_0 \in S$ is an initial state,
V is a finite set of variables,
I is a finite set of inputs,
O is a finite set of outputs,
T is a finite set of transitions, $t \in T$ can be described as a 5-tuple (s_s, i, g, op, s_t),
s_s is the source state,
i is an input,
g is a guard,
op is a sequence of output and assignment operations,
s_t is the target state [16].

In order to use such an EFSM for PBT the permitted transition sequences have to be defined with preconditions and also the effect of each transition has to be defined with the postconditions. Preconditions, postconditions and the execution semantics of transition are encapsulated in so-called commands $Cmds$. The property of an EFSM is that for each permitted path on the model, the postcondition of each transition respectively command of the path must hold. In order to verify this property a PBT tool produces random transition sequences and checks the postconditions after each transition. Formally a property for an EFSM can be defined as follows.

$$\begin{aligned}&\forall s \in S, i \in I, cmd \in Cmds :\\ &cmd.pre(i, s) \implies cmd.post(cmd.runActual(i, s),\\ &\qquad\qquad\qquad\qquad cmd.runModel(i, s))\end{aligned}$$

$$\begin{aligned}cmd.runActual &: S \times I \to S \times O\\ cmd.runModel &: S \times I \to S \times O\end{aligned}$$

The Boolean function $cmd.pre$ is the precondition. It defines the valid inputs and states of a command. The post condition $cmd.post$ relates the new states and the outputs of the SUT and the model after the execution of the command on both the SUT $cmd.runActual$ and the model $cmd.runModel$.

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 [17].

The first PBT tool was QuickCheck [1] for Haskell. There are many other tools that are based on QuickCheck, for example, ScalaCheck[2] or Hypothesis[3] for Python. For our approach we work with FsCheck.

B. 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 small limitation of the current version is that it does not consider preconditions when shrinking command sequences. However, this feature is included in an experimental release. With FsCheck, properties can be defined both in a functional programming style with F# and object-oriented 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.

[2]https://www.scalacheck.org
[3]https://pypi.python.org/pypi/hypothesis

```
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" Enabled="true"
20          Required="true" />
21         <Attribute Name="Description" Enabled="true"
22          Required="true" />
23          ...
24       </DynamicAttributesInfo>
25       <RequiredUserRoles>
26         <Role>IncidentSubmitter</Role>
27          ...
28       </RequiredUserRoles>
29       <PossibleNextStates>
30         <State Name="Submitted" NoteRequired="false" />
31       </PossibleNextStates>
32     </Task>
33      ...
34   </AllTasks>
35   <AllStates>
36     <State Name="Submitted">
37       <PossibleTasks>
38         <Task>IncidentEditTask</Task>
39         <Task>IncidentCloseTask</Task>
40       </PossibleTasks>
41     </State>
42      ...
43   </AllStates>
44 </RuleEngineModel>
```

Listing 1. Simplified XML Representation of a Business Rule Model

Furthermore, FsCheck has extensions for the unit testing tools NUnit[4] and Xunit[5]. These extensions allow the definition of properties in a simple and convenient way like normal unit tests.

C. 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 [18], [19], [20]. 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 work flow details can be included, although there is a separate technology for this, called work flow engine [21], [22]. The web service application of our case study has a custom development 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 many other rule definitions, like for example the business process execution language (BPEL) [23].

Our rule engine models can be seen as EFSMs. Listing 1 shows a simplified version of the XML file of one rule engine model that was used as basis for the example in Section III. It can be seen that these models are structured as follows. The main components are:

- attribute definitions with data types and constraints, which represent the variables of the EFSM (Lines 3 to 15)
- tasks, which represent transitions with enabled and required attributes and required user roles and possible next states (Lines 17 to 34)
- states with possible tasks (Lines 35 to 43)

Optionally the models can 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

Each of the business rule models describes one object of our application. The objects are stored in a data base and a list of them can be selected by a query, which make a search for objects with specific attributes. The application has a number of modules, which include multiple objects that are used in composition. A task represents (1) opening a page with many form fields respectively attributes, (2) entering form data in these fields and (3) submitting the form.

Ideally, all customer dependent business logic should be encoded in the business rule models. Testing these models would only test the interpreter. In practice the programmers often manipulate the source code to integrate new features and during the evolution the source code deviates from the model. Hence, it makes sense to verify if the SUT still conforms to the model. We perform this verification in the postconditions of commands, which serve as test oracle, and check if the state of the model matches the state of the SUT. Furthermore, these automatic tests are a good basis for further load and stress testing.

III. Example of Model-based Testing with FsCheck

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

A. 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)

[4]http://www.nunit.org
[5]https://xunit.github.io

```
1  public class Spec : ICommandGenerator<SUT, Model>
2  {
3    public SUT InitialActual{get{return new SUT();}}
4    public Model InitialModel{get{return new Model();}}
5
6    public Gen<Command<SUT, Model>> Next(Model m){
7      return Gen.Elements(new Command<SUT, Model>[] {
8        new IncidentCreateTask(),
9        new IncidentEditTask(),
10       new IncidentCloseTask()});
11   }
12
13   private class IncidentCloseTask : BaseCommand{
14     public override bool Pre(Model m){
15       return m.State == "Submitted";
16     }
17     public override Property Post(SUT s, Model m){
18       return (s.State == m.State).ToProperty();
19     }
20     public override SUT RunActual(SUT s){
21       s.IncidentCloseTask(); return s;
22     }
23     public override Model RunModel(Model m){
24       m.IncidentCloseTask(); return m;
25     }
26     public override string ToString(){
27       return "IncidentCloseTask";
28     }
29   }
30   ...
31 }
```

Listing 2. FsCheck Specification for the Incident Example

- 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

Listing 2 shows an example of a specification for FsCheck. The specification in this example implements an ICommand-Generator interface, which takes the model and the SUT as generic type parameter. The class of the SUT basically is 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 data base. In this example we assume that the attributes are set statically in the wrapper class of the SUT. In Section IV we will see, how form data can be generated automatically for these attributes.

The state machine in Figure 2 represents the states and tasks of one incident object. 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 would need functionality to switch between objects. Hence, in a more realistic model, an additional select transition should be added, which can open other (incident) objects in all states.

In this example an IncidentCreateTask is possible globally in all states. It creates and opens a new incident object, which can be edited and closed with the corresponding tasks.

The initial states of the model and SUT are set via calling their constructor methods (Lines 3 and 4). The generator for the next state selects one element of a command array randomly, which can be accomplished with a default elements generator from FsCheck (Lines 6 to 11).

For this standard approach all command classes need to be defined manually (Lines 13 to 29). The classes need to define how the transitions should be executed on the model and SUT and what postcondition should hold after the execution. Moreover, a ToString() method can be used to display various information of the command and optionally a precondition can be defined, which has to hold so that the transition is allowed. The classes for the IncidentCreateTask and the IncidentEditTask are similar to the IncidentCloseTask and therefore omitted in Listing 2.

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

B. FsCheck Command Generation and Execution

FsCheck takes a specification as shown in Listing 2 and generates commands that consider the preconditions and start at the initial state. A single test case is a sequence of commands, but the command generator can also create trivial sequences with zero or one command. After a command is generated it is directly executed on the SUT. Hence, FsCheck performs online testing.

In order to start testing, we have to make a property from the specification. This can be done with the ToProperty() method, which is provided by FsCheck. The property can then for example be tested by calling the QuickCheck() method or also with the help of unit testing frameworks.

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

Per default 100 test cases will be generated and executed, but this number can be configured. Listing 3 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.

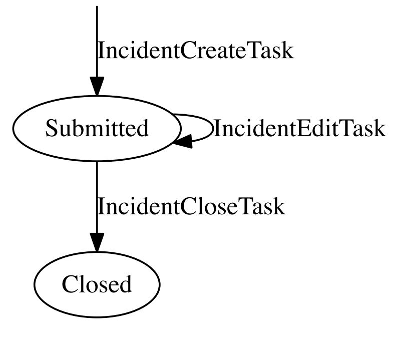


Fig. 2. Incident Example State Machine

```
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% short sequences (1-6 commands)
```

Listing 3. Generated Command Sequences for the Incident Example

IV. Architecture and Implementation

In this section we show how our approach makes it possible to use rule engine models to automatically derive FsCheck specifications, which are used to generate test cases.

As already explained in Section I we parse the business rule models from XML files and create an EFSM. This task is done by a parser that we have implemented for our custom business rule models. The result is a parsed model represented as an object structure, i.e. an abstract syntax tree. The class diagram for this object structure is presented in Figure 3. It can be seen that the model consists of an attribute dictionary, an initial state, a current state, a list of states and a dictionary of transitions. The transitions are represented by a separate class, that contains hash sets for the possible source and target states, a name and a flag, which indicates that the state should not change after this transition. 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 data base. The interface for 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 Section III.
- Object attributes: an object attribute can group together multiple attributes in a struct or in 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 name with this type. The generation of the actual file is added to a wrapper class of the SUT, because the file should also be deleted after the test execution.

The object representation of the model is also used for the interface specifications for FsCheck. For example, preconditions for the restriction of the transitions 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. Listing 4 shows how attribute data can be generated. First a list of generators is created by iterating over the required attributes and adding the generators to the list (Lines 2 to 5). This list is then given to a sequence generator as input, which creates an array of values for all the generators in the list. In order to store them in a dictionary, we use the select function of the generator (Line 6). This function takes an anonymous function, which takes the values as input and returns an object that should be created by the generator. In our case the returned object is a dictionary with attribute names as keys and the data as values (Lines 7 to 13).

This attribute data generation is required for the command generation, which is shown in Listing 5. It can be seen that an array of possible transitions 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 transitions and with the selectMany function we process the chosen transition (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. Therefore, selectMany makes it possible to nest generators and also to pass the generated value to the inside generator.

If the chosen transition can lead to multiple next states, then we also choose a next state with an element generator (Line 5). Otherwise, it is possible that the transition should keep the current state (Lines 10 to 12). In both cases we apply the attribute data generator, which was shown in Listing 4 (Lines 6 and 13). With the generated data we create a DynamicCommand object, which takes the transition, the model, the attribute data and the next state as arguments for the constructor. In order to fulfil the interface requirements for FsCheck, we have to cast this object to its base class before we can return it for the generator (Lines 7 and 14).

The DynamicCommand class is shown in Listing 6. This class can handle the execution of all transitions of the parsed model, because the definition is made in a generic way, which works for all transitions. It can be seen that the class has the transition, the model, attribute data and the next

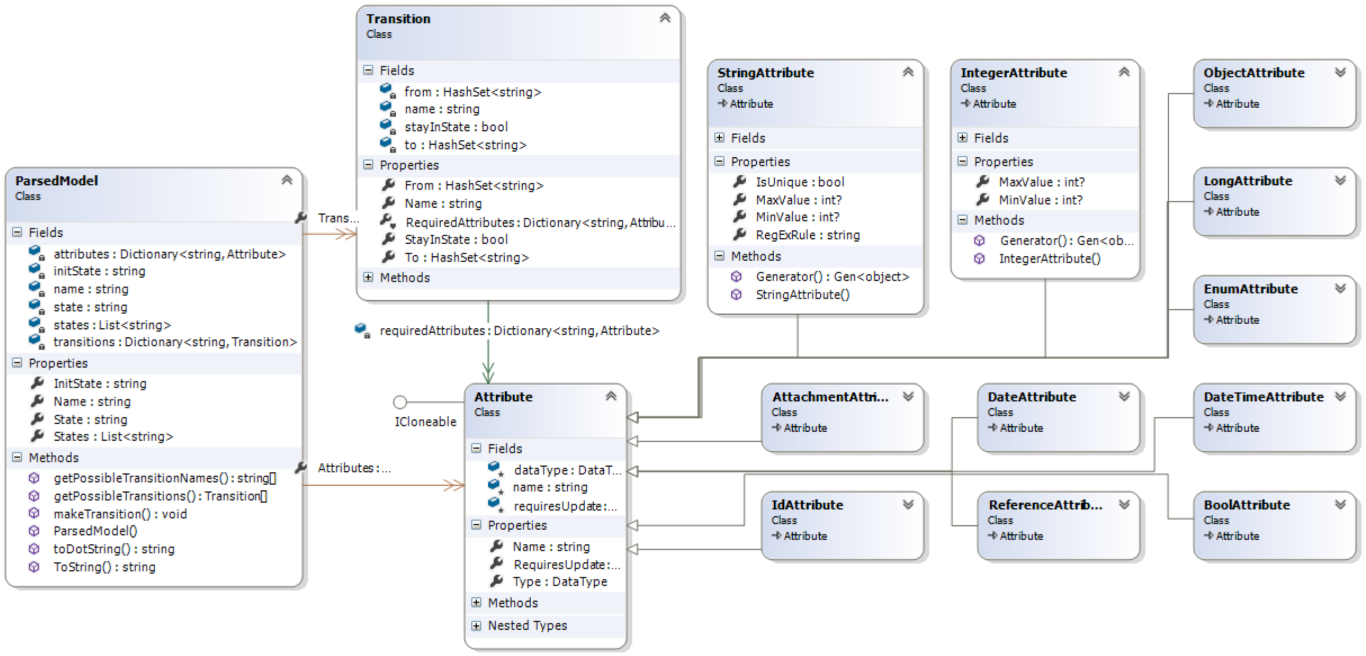


Fig. 3. Class Diagram for a Parsed Model

```
Gen<Dictionary<string,object>> generateData(Attribute[] attributes) {
  List<Gen<object>> genList = new List<Gen<object>>();
  foreach (Attribute a in attributes){
    genList.Add(a.Generator());
  }
  return Gen.Sequence(genList.ToArray()).Select(values => {
    var attributeData = new Dictionary<string, object>();
    int i = 0;
    foreach (object value in values){
      string name = attributes[i++].Name;
      attributeData.Add(name, value);
    }
    return attributeData;
  });
}
```

Listing 4. Attribute Data Generator for Automatically Filling Web Service Forms with Data

```
public Gen<Command<SUT, ParsedModel>> Next(ParsedModel m){
  Transition[] options = m.getPossibleTransitions();
  return Gen.Elements(options).SelectMany(t => {
    if (t.hasMultipleNextStates()){
      return Gen.Elements(t.To.ToArray()).SelectMany(nextState =>
        generateData(t.RequiredAttributes.Values.ToArray()).Select(data =>
            (Command<SUT, ParsedModel>) new DynamicCommand(t, m, data, nextState)));
    }else{
      string nextState = null;
      if (t.StayInState){
          nextState = m.State;
      }
      return generateData(t.RequiredAttributes.Values.ToArray()).Select(data =>
        (Command<SUT, ParsedModel>) new DynamicCommand(t, m, data, nextState)));
    }
  });
}
```

Listing 5. Next State Generator for Automatically Generating Web-Service Requests

```
1  public class DynamicCommand: Command<SUT,ParsedModel>
2  {
3    Transition t;
4    ParsedModel m;
5    Dictionary<string,object> attributeData;
6    string nextState = null;
7
8    public DynamicCommand(Transition t, ParsedModel m,
9     Dictionary<string,object> data, string nextState){
10     this.t = t;
11     this.m = m;
12     this.nextState = nextState;
13     this.attributeData = data;
14   }
15   public override ParsedModel RunModel(ParsedModel m)
16   {
17    m.makeTransition(t.Name, nextState);
18     return m;
19   }
20   public override SUT RunActual(SUT s){
21     sut.DefaultTask(m, t, attributeData, nextState);
22     return s;
23   }
24   public override Property Post(SUT s,ParsedModel m){
25     return (s.State == m.State).ToProperty();
26   }
27   public override string ToString(){
28     return t.Name;
29   }
30 }
```

Listing 6. Generic Command Definition

state as member variables and constructor arguments. They are required for the execution of the transition. Running a transition on the model is done with a function of the model class (Line 17). The execution on the SUT, which is done in RunActual, calls the wrapper class of the SUT (Line 21). This wrapper class uses reflections to call the actual methods on the SUT and it also sets the attribute data. In the postcondition we check if the state of the model is equal to the state of the SUT.

A nice feature, which is also supported by FsCheck is command generation with different frequencies. This feature makes it possible to test certain problematic tasks more frequently and also to simulate user behaviour. Normally commands are generated randomly with a uniform distribution, but we added the possibility to generate commands according to certain probability distributions, which can be specified by the probability mass function respectively weights for the transitions. Listing 7 shows how this can be done. We just create a WeightAndValue object for each transition, which takes a weight and a generator as parameter. In this case the weight is a member variable of the transition and we use a constant generator, which simply generates a transition object. With the FsCheck generator Gen.Frequency one transitions is selected according to the weights. The remaining part of the command generation is the same as in Listing 5 and was therefore omitted.

```
var wv = new List<WeightAndValue<Gen<Transition>>>();
foreach(Transition t in m.getPossibleTransitions()){
   wv.Add(new WeightAndValue<Gen<Transition>>
      (t.Weight,Gen.Constant(t)));
}
return Gen.Frequency(wv).SelectMany(t => ...
```

Listing 7. Command Generation with Frequencies

V. Industrial Case Study

Our approach was developed for a web service application which was provided by our industrial partner AVL[6]. This application has a client-server architecture. It originates from the automotive domain and is called Testfactory Management Suite (TFMS)[7].

This system makes it possible to manage test field data, activities, resources, information and work flows. The test fields can, for example, test power trains and engines of cars. 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 many modules. Each of them provides different functionalities, like the management of test equipment and test standards.

Our case study was primarily applied to one module of the TFMS, which is called the Test Order Manager. This module controls individual work steps and preparations for test orders. A test order is a composition of multiple steps that are necessary for an automotive test sequence at a test field. We also tested other small modules like the Incident Manager, which was shown in the example of Section III, but the major part was the Test Order Manager. A state machine of a test order is shown in Figure 4. 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 work-flow respectively life cycle of a test order. The number of possible transitions between these states 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 can be seen in Figure 5. 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 performs the communication with the web services on the server. This framework offers functions for executing tasks and for retrieving data, like the state of a domain object. This state is compared in the postcondition to the state of the model.

The case study revealed the following problems and bugs:

- 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.
- 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

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

[7]https://www.avl.com/-/avl-testfactory-management-suite-tfms

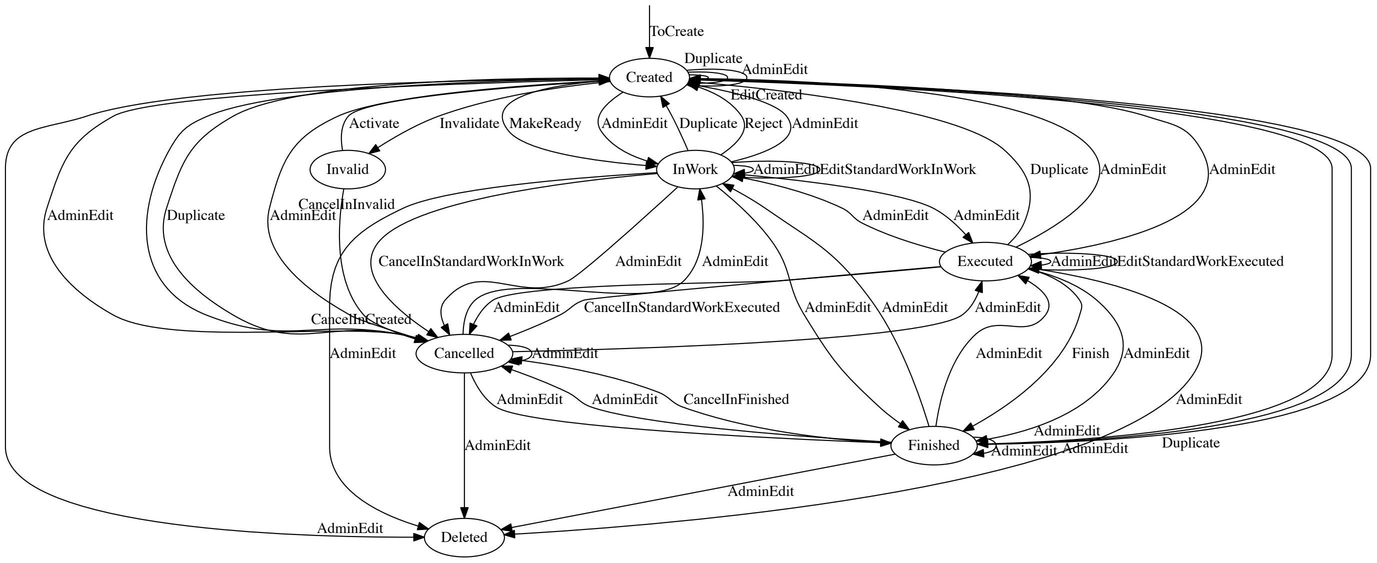
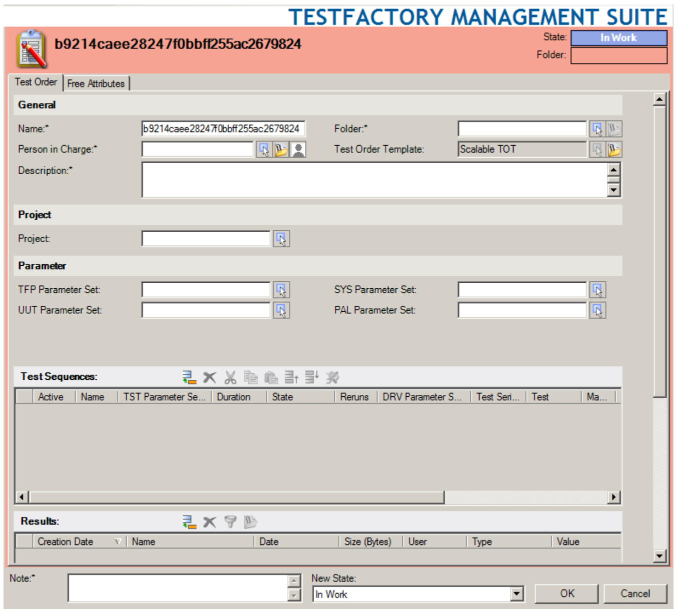


Fig. 4. Test Order Manager Model

the subtype. This information was correctly implemented in the code, but is missing in the rule engine. Hence, the tool reported a bug that in fact was not. It is rather a limitation of the approach of relying on the business rule models as primary source for the test case generation.

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.

- 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.


TESTFACTORY MANAGEMENT SUITE

b9214caee28247f0bbff255ac2679824

State: In Work
Folder:

Test Order | Free Attributes

General

Name:* b9214caee28247f0bbff255ac2679824
Folder:*
Person in Charge:*
Test Order Template: Scalable TOT
Description:*

Project

Project:

Parameter

TFP Parameter Set:
SYS Parameter Set:
UUT Parameter Set:
PAL Parameter Set:

Test Sequences:

Active	Name	TST Parameter Se...	Duration	State	Reruns	DRV Parameter S...	Test Seri...	Test	Ma...

Results:

Creation Date	Name	Date	Size (Bytes)	User	Type	Value

Note:*
New State: In Work
OK
Cancel

Fig. 5. TFMS Form for the AdminEdit Task

- 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.

VI. Conclusion

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

First, we showed how model-based testing can be implemented with FsCheck by a manual specification with a small example. Then we discussed how our approach works in detail. It takes XML files with the business rule models as input and converts them into 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 an industrial case study, which was applied to a web service application from AVL, a testfactory management suite. This case study revealed four issues that needed to be fixed. Two of them concern the testing framework, two the system-under-test. This demonstrates that the approach is effective in finding bugs. 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.

ACKNOWLEDGEMENT

The research leading to these results was funded by 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 Elisabeth Jöbstl, Florian Lorber, Martin Tappler and the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. Furthermore, we would like to thank the FsCheck community and Kurt Schelfthout for providing FsCheck and support for this tool.

REFERENCES

[1] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. [Online]. Available: http://doi.acm.org/10.1145/351240.351266

[2] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing AUTOSAR software with QuickCheck," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, April 2015, pp. 1–4.

[3] L. Lampropoulos and K. F. Sagonas, "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*, ser. EPTCS, J. Silva and F. Tiezzi, Eds., vol. 98, 2012, pp. 3–16. [Online]. Available: http://dblp.uni-trier.de/db/series/eptcs/eptcs98.html#abs-1210-6110

[4] M. A. Cibrán and B. Verheecke, "Dynamic business rules for web service composition," in *In Proc. of the 2nd Dynamic Aspects Workshop (DAW05) in conjunction with AOSD*, 2005.

[5] F. Rosenberg and S. Dustdar, "Design and implementation of a service-oriented business rules broker." in *7th IEEE International Conference on E-Commerce Technology Workshops (CEC 2005)*. IEEE Computer Society, 2005, pp. 55–63. [Online]. Available: http://dblp.uni-trier.de/db/conf/wecwis/cecw2005.html#RosenbergD05a

[6] N. Milanovic and M. Malek, "Current solutions for web service composition." *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/internet/internet8.html#MilanovicM04

[7] K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proceedings of the 30th International Design Automation Conference*, ser. DAC '93. New York, NY, USA: ACM, 1993, pp. 86–91. [Online]. Available: http://doi.acm.org/10.1145/157485.164585

[8] L. M. López, H. Ferreiro, and T. Arts, "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.

[9] H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco, "Automating property-based testing of evolving web services," in *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '14. New York, NY, USA: ACM, 2014, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/2543728.2543741

[10] L. Fredlund, C. Benac Earle, A. Herranz, and J. Marino, "Property-based testing of JSON based web services," in *Web Services (ICWS), 2014 IEEE International Conference on*, June 2014, pp. 704–707.

[11] C. Benac Earle, L.-A. Fredlund, A. Herranz, and J. Mariño, "Jsongen: A QuickCheck based library for testing JSON web services," in *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '14. New York, NY, USA: ACM, 2014, pp. 33–41. [Online]. Available: http://doi.acm.org/10.1145/2633448.2633454

[12] M. A. Francisco, M. López, H. Ferreiro, and L. M. Castro, "Turning web services descriptions into QuickCheck models for automatic testing," in *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13. New York, NY, USA: ACM, 2013, pp. 79–86. [Online]. Available: http://doi.acm.org/10.1145/2505305.2505306

[13] M. Papadakis and K. Sagonas, "A PropEr integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '11. New York, NY, USA: ACM, 2011, pp. 39–50. [Online]. Available: http://doi.acm.org/10.1145/2034654.2034663

[14] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy SmallCheck: Automatic exhaustive testing for small values," in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, ser. Haskell '08. New York, NY, USA: ACM, 2008, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1411286.1411292

[15] J. Hughes, "QuickCheck testing for fun and profit," in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, M. Hanus, Ed. Springer Berlin Heidelberg, 2007, vol. 4354, pp. 1–32. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69611-7_1

[16] A. S. Kalaji, R. M. Hierons, and S. Swift, "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 Computer Society, 2009, pp. 230–239. [Online]. Available: http://dblp.uni-trier.de/db/conf/icst/icst2009.html#KalajiHS09

[17] Y. Wada and S. Kusakabe, "Performance evaluation of a testing framework using QuickCheck and Hadoop," *Journal of Information Processing*, vol. 20, no. 2, pp. 340–346, 2012. [Online]. Available: http://dx.doi.org/10.2197/ipsjjip.20.340

[18] B. Orriëns, J. Yang, and M. Papazoglou, "A framework for business rule driven service composition," in *Technologies for E-Services*, ser. Lecture Notes in Computer Science, B. Benatallah and M.-C. Shan, Eds. Springer Berlin Heidelberg, 2003, vol. 2819, pp. 14–27. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-39406-8_2

[19] R. G. Ross, *Principles of the Business Rule Approach*. Boston: Addison-Wesley, 2003. [Online]. Available: http://www.worldcat.org/search?qt=worldcat_org_all&q=0201788934

[20] H. Herbst, *Business Rule-oriented Conceptual Modeling*. Springer Science & Business Media, 1997. [Online]. Available: http://books.google.com/books?id=uIRul0nO8RYC&hl=de

[21] A. Charfi and M. Mezini, "Hybrid web service composition: Business processes meet business rules," in *Proceedings of the 2Nd International Conference on Service Oriented Computing*, ser. ICSOC '04. New York, NY, USA: ACM, 2004, pp. 30–38. [Online]. Available: http://doi.acm.org/10.1145/1035167.1035173

[22] F. Rosenberg and S. Dustdar, "Business rules integration in BPEL - a service-oriented approach," in *Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*, July 2005, pp. 476–479.

[23] J. Mendling, "Business process execution language for web services." *EMISA Forum*, vol. 26, no. 2, pp. 5–8, 2006. [Online]. Available: http://dblp.uni-trier.de/db/journals/emisa/emisa26.html#Mendling06