Automatic Generation of Test Models for Web Services Using WSDL and OCL

Macías López[1], Henrique Ferreiro[1],
Miguel A. Francisco[2], and Laura M. Castro[1]

[1] MADS Group, University of A Coruña, Spain
{macias.lopez,henrique.ferreiro,laura.castro}@madsgroup.org
[2] Interoud Innovation S.L, Spain
miguel.francisco@interoud.com

Abstract. Web services are a very popular solution to integrate components when building a software system, or to allow communication between a system and third-party users, providing a flexible, reusable mechanism to access its functionalities.

To ensure these properties though, intensive testing of web services is a key activity: we need to verify their behaviour and ensure their quality as much as possible, as efficiently as possible. In practise, the compromise between effort and cost leads too often to smaller and less exhaustive testing than it would be desirable.

In this paper we present a framework to test web services based on their WSDL specification and certain constraints written in OCL, following a black-box approach and using property-based testing. This combination of strategies allows us to face the problem of generating good quality test suites and test cases by automatically deriving those from the web service formal description. To illustrate the use of our framework, we present an industrial case study: a distributed system which serves media contents to customers' TV screens.

Keywords: Property-Based Testing, Web Services, WSDL, OCL.

1 Introduction

The need to provide access to different kind of systems across the web has become critical. The usual way to do it is through web services, which aim to provide a means for interaction among software systems, or systems and final users over the network. There are multiple ways for describing these interactions, one commonly used being WSDL (Web Services Description Language) [5], an XML-based language to specify the operations offered by a web service. The WSDL standard operates at the syntactic level and does not represent the requirements or operational constraints of the web service. Thus, in order to add semantic information to web service description, the WSDL description must be completed. A number of choices have been proposed to do this, such as WSDL-S (Web Services Semantics) [6], SWRL (Semantic Web Rule Language) [4], or OCL (Object Constraint Language) [2].

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 483–490, 2013.

To ensure the quality of a web service [13], we need to guarantee that the operations work as their specification require, this is, that the semantic information is not violated. Based on previous work [16] which used UML descriptions together with OCL properties to perform automatic testing of software components, we propose to apply property-based testing (PBT) [12] to perform automatic testing of web services. When using PBT, testers have to write properties that the system under test (SUT) needs to satisfy, rather than specific test cases. From the properties description, tools can produce the specific test cases automatically. Using this technique, we have a black-box model which describes the functional properties of the SUT and use it for testing purposes. In particular, given a WSDL description of a web service and its OCL semantic definition, we generate the model instead of writing it manually.

2 Property-Based Testing and QuickCheck

As an alternative to manually producing tests from a high-level natural-language specification, or writing a formal model to describe a system or component, PBT uses declarative statements to specify properties that the software needs to satisfy according to its specification. Using this approach, test cases can then be generated from those properties, a process that can be automated, allowing to run many tests for each written property.

In our work, we have used QuickCheck, a PBT tool that automates generation, execution and evaluation of test cases. This allows us to run lots of tests with very little effort, checking whether the defined properties hold or not.

For testing complex systems, however, isolated properties are not expressive nor powerful enough. Instead of sequences of independent test cases, we want to test sequences of calls which modify the *state* of the service, checking that some conditions hold before and after each interaction, and that the global state of the service remains coherent with its expected behaviour call after call.

3 Test Approach: From WSDL+OCL to Properties

The requirements of a system represent the needs that it must fulfil, and they are usually specified in an abstract way, without technical details. As we want to use PBT to test the web services behind the WSDL specification, we need to have the appropriate properties which describe the requirements of the SUT. To do that, we get information both from the WSDL and the OCL constraints, and the combination of both allows us to automatically build our test model, composed by properties. Depending on the requirements of the web service, the test model can differ: for stateless web services, universally quantified properties are generated; for stateful ones, the requirements are modelled into a state machine. Either from the properties or from the state machine modelling the web service, QuickCheck derives the specific test cases, and then, using an HTTP

adapter (generated from the WSDL specification), we feed the SUT. Thus, using our framework, the testers do not need to know any specific details about web services implementation languages.

In addition, if the same API is preserved, different implementations of a web service can be tested with the same test properties. The general architecture of our framework is shown in Figure 1.

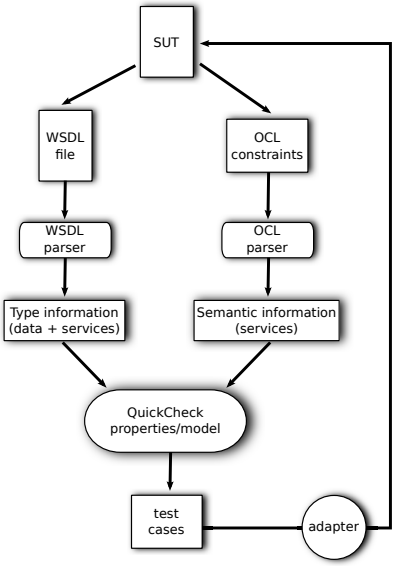

Fig. 1. Proposed testing architecture

Firstly, we need to retrieve information from the WSDL file, so a WSDL parser has been developed. We decided to implement our own because we need to integrate the semantic information provided as OCL constraints, writing it as an easy-to-manage structure to transform into properties. For instance, from the WSDL for a calculator we need: the name of the web service from the `service` tag; the name of each operation from the `operation` tag, contained in `interface`; the name of `input` and `output` tags for each operation; the `types` referred by each input and output elements; the `endpoint` and the `operation` from the `binding` tag to get the URL; and the `modelReference` attribute referring the OCL file.

Then, we have to parse the referred OCL file and check if there is semantic information associated to any of the operations retrieved in the second item before. As in the previous case, we found several tools to parse OCL files, but in all cases the parsing functionality has to be executed associating a UML model to the OCL. This led us to develop our own OCL parser, taking advantage from the work made by the OCLNL project [3]: a labelled BNF grammar [15] for OCL. This grammar was fed to the BNF compiler (BNFC) [1] to produce the abstract syntax tree, lexer and parser which we used.

Finally, when the required information from the WSDL and OCL file is retrieved, it is time to build the properties for testing.

3.1 Stateless web Services

Stateless services or systems do not have an internal state that affects the outcome of a sequence of calls to their API, so the response returned by a specific call is independent of the specific moment when it is executed. In this case, the name of the operation to be tested and the type of its result and arguments is parsed from the WSDL file; in turn, the test oracle is built out of the constraints specified in the OCL file.

For the calculator example, we could generate the following property:

```
prop_pow() ->
  ?FORALL({A, B}, {ocl_gen:int(), ocl_gen:nat()},
      mathUtils:pow(A, B) == ocl_seq:iterate(fun (I, Acc) -> Acc * A end, 1, ocl_seq:new(1, B))).
```

QuickCheck can run this property and produce specific test cases; this way, instead of specifying input data manually, data generators are used to generate data of the corresponding data type. This approach leads to a significant improvement over traditional tests [21] and most of the research in the state of the art [8–11, 17, 19, 22, 24], since instead of specific values, we define types, ranges, and conditions that the input data has to meet, which are then produced automatically instead of manually listed. So, for each pair of an integer and a natural number that is generated, they are used by the HTTP adapter to make a call to the web service under test, getting the URL from the WSDL file. The value returned by the web service is finally checked in the property body.

With QuickCheck we can not only generate a large amount of specific test cases derived from properties and executed against the real SUT. Another very interesting QuickCheck feature is that, when a failing test case is found, the tool automatically *shrinks* it to the smallest equivalent counterexample it can find, making it easier to understand the reason of the failing case [23], and thus improving also the debugging process.

3.2 Stateful Web Services

In opposition to stateless components, in which each action is independent of each other, many systems have a behaviour that depends on which actions were previously performed. In order to test these systems, the internal state has to be taken into account in the test process. QuickCheck has support for testing this kind of systems by using state machines. Instead of specifying general properties, the state machine behaviour is specified by defining an initial state and a state transition function. Additionally preconditions and postconditions are used to verify state-related properties in each step. The generated tests cases consist in random sequences of state transitions where, at each state, both pre- and postconditions are checked [7]. Our case study, explained in the next section, falls in this category of stateful web services.

4 Case Study: VoDKATV

VoDKATV is an IPTV/OTT middleware that provides end-users access to different services on a TV screen, tablet, smartphone, PC, etc., allowing an advanced multi-screen media experience. Architecturally, it is a distributed system composed by several components, which are integrated through web services.

Among other things, VoDKATV stores information about the users and devices that can access the system. Devices are identified by a MAC address, and they are associated to a household (*room*, in VoDKATV nomenclature). Thus, when a new user is registered, a new household must be created and the devices of that user must be registered to that household. This particular subset of VoDKATV functionalities is offered by one single administration web service, which we have chosen as case study. The web service offers, among others, operations to create, modify, update and delete households and devices.

The operation used to create a new household is specified in WSDL as:

```
<wsdl:operation name="CreateRoom"
    pattern="http://www.w3.org/ns/wsdl/in-out"
    style="http://www.w3.org/ns/wsdl/style/iri" wsdlx:safe="true">
    <wsdl:input element="msg:createRoomParams"/>
    <wsdl:output element="msg:createRoomResponse"/>
</wsdl:operation>
```

where `createRoomParams` specifies the parameters received by the web service (`roomId` and `description`):

```
<xsd:element name="createRoomParams">
    <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="roomId" type="xsd:string" />
          <xsd:element name="description" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

and `createRoomResponse` is the response returned by the web service:

```
<xsd:element name="createRoomResponse">
    <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="roomId" type="xsd:string" />
          <xsd:element name="description" type="xsd:string" minOccurs="0" maxOccurs="1"/>
          <xsd:element name="error" type="tns:error" minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:complexType name="error">
    <xsd:sequence>
        <xsd:element name="code" type="xsd:string" />
        <xsd:element name="params" type="tns:errorParams" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="description" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="errorParams">
    <xsd:sequence>
      <xsd:element name="param" type="tns:errorParam" minOccurs="1" maxOccurs="unbound"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="errorParam">
    <xsd:attribute name="name" type="xsd:string" />
    <xsd:attribute name="value" type="xsd:string" />
</xsd:complexType>
```

Our approach requires to specify the behaviour of the web service so that the test cases can be generated automatically. The specification of the `CreateRoom` operation is:

- if the specified household identifier (`roomId`) is empty, the web service must return a `required` error;
- if the specified household identifier (`roomId`) already exists in the VoDKATV system, the web service must return a `duplicated` error;
- otherwise, the household must be created, and its identifier (`roomId`) and description (`description`) must be returned by the web service.

This specification is written using OCL pre- and postconditions. For instance, the specification for the `CreateRoom` operation can be written in OCL with the following code, where `state_rooms` represents the internal test state:

```
context VoDKATVInterface::CreateRoom(roomId:String, description:String): CreateRoomResponse
post CreateRoom:
  if ((roomId = '') or (roomId = null)) then
    (self.state_rooms = self.state_rooms@pre and
        result.errors->size() = 1 and
        result.errors->at(0).code = 'required')
  else if (self.state_rooms->select(room | room.roomId = roomId)->notEmpty()) then
    (self.state_rooms = self.state_rooms@pre and
        result.errors->size() = 1 and
        result.errors->at(0).code = 'duplicated')
     else self.state_rooms = self.state_rooms@pre->including(
           Tuple {
             roomId:String = roomId,
             description:String = description
           }) and
               result.roomId = roomId and
               result.description = description
     endif
  endif
```

This OCL specification, together with the WSDL, is used by our framework to generate QuickCheck properties. To do that, we use the same approach described in [16], but using WSDL and OCL to generate QuickCheck code. In addition, during test execution, the newly generated QuickCheck model uses the HTTP adapter, which is also generated by our tool from the WSDL. Thus, when an operation is executed, the corresponding web service operation will be invoked, and the result is analysed by the corresponding postcondition. For example, this is part of the code generated to check that a `required` error is returned when the household identifier is empty:

```
postcondition(PreState, AfterState,
              {call, vodkaTV, createRoom, [RoomId, Description]}, Response) ->
  case RoomId of
    "" ->
      ocl_seq:eq(AfterState#state_rooms, PreState#state_rooms)
          andalso ocl_string:eq(ocl_datatypes:get_property(code, Response), "required")
  end;
```

where `ocl_seq`, etc. are ancillary modules that implement utility functions.

Therefore, as a result, we have a QuickCheck test model automatically generated by our tool from the WSDL and the OCL constraints. This test model checks that the web service described by the WSDL satisfies the constraints specified with OCL.

4.1 Analysis of Results

QuickCheck generates specific test cases from the generated test model, i.e., random sequences of commands with random parameter values that satisfy the preconditions. As a second step, QuickCheck executes these commands, invoking the corresponding operations of the web service, and checks if the SUT fulfils the postconditions.

Although we have not found any errors in the web service used as case study (which, considering the system has been in production for a number of years, was to be expected), we have introduced a number of errors to empirically verify the effectiveness of our methodology. We have real error reports of VODKATV as source of inspiration, thus demonstrating that all of them were exposed immediately using the generated QuickCheck model and proposed test architecture. Besides, thanks to QuickCheck shrinking capabilities, the counterexamples found were qualified, when shown to the developers who fixed the error corresponding reports, as very valuable, had it been in place when they had to diagnose them.

5 Conclusions and Future Work

In this paper we have presented a test framework to build test models for web services using a PBT tool, where semantics are added to WSDL using OCL constraints. Using this black-box approach, properties are automatically generated from one WSDL specification, and specific test cases are automatically generated and executed. Our framework can generate properties for both stateless and stateful web services, using declarative statements in the first case and state machines models in the second. In all cases, the test model produced by the framework can be used as an updated specification of the SUT with the shape of an executable model.

One of the main advantages of our approach is the use a PBT tool like QuickCheck to generate and run the test cases, because it automatically generates complex testing sequences which stress-test the real system in a more objective and efficient way than any human tester could [14, 18, 20]. We remove the need to think of specific test cases, rather the general behavioural properties. Another important aspect of QuickCheck is its shrinking and counterexample capabilities, a very valuable asset to fault debugging.

We have used a standard specification language, OCL, so testers need not learn a specification language to write test cases, because properties are automatically generated from the OCL specification. Another advantage of using PBT instead of hand-written tests is that properties are independent of the implementation details of the SUT. This means that an evolving code base does not force rewriting the test model, maintaining an intact, updated and executable specification of the SUT.

As future work, we should be able to trace back the conditions that have failed when QuickCheck generates a counterexample, showing the specific piece of OCL code that has produced the error. Furthermore, nowadays a particular kind of web services is most popular: RESTful web services; we plan to extend our framework to adapt specifically to the intrinsic properties of these web services.

References

1. BNFC, `http://bnfc.digitalgrammars.com/`
2. Object Constraint Language (OCL), `http://www.omg.org/spec/OCL/2.3.1/`

3. OCLNL, http://www.key-project.org/oclnl/
4. Semantic Web Rule Language (SWRL), http://www.w3.org/Submission/SWRL/
5. Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl/
6. Web Services Semantics (WSDL-S), http://www.w3.org/Submission/WSDL-S/
7. Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing telecoms software with Quviq QuickCheck. In: ACM SIGPLAN Workshop on Erlang., pp. 2–10 (2006)
8. Askarunisa, A., Abirami, A., Mohan, S.: A test case reduction method for semantic based web services. In: International Conference on Computing, Communication and Networking Technologies, pp. 1–7 (2010)
9. Bai, X., Lee, S., Tsai, W., Chen, Y.: Ontology-based test modeling and partition testing of web services. In: IEEE International Conference on Web Services, pp. 465–472 (2008)
10. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: WS-TAXI: A WSDL-based testing tool for web services. In: International Conference on Software Testing, Verification, and Validation, pp. 326–335 (2009)
11. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 141–150 (2009)
12. Derrick, J., Walkinshaw, N., Arts, T., Benac Earle, C., Cesarini, F., Fredlund, L.-A., Gulias, V., Hughes, J., Thompson, S.: Property-based testing - the ProTest project. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 250–271. Springer, Heidelberg (2010)
13. Emmerich, W.: Managing web service quality. In: International Workshop on Software Engineering and Middleware, pp. 1–1 (2006)
14. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. SIGSOFT Software Engineering Notes 22(4), 74–80 (1997)
15. Forsberg, M., Ranta, A.: Labelled BNF: a highlevel formalism for defining well-behaved programming languages. Estonian Academy of Sciences: Physics and Mathematics 52, 356–377 (2003)
16. Francisco, M.A., Castro, L.M.: Automatic generation of test models and properties from UML models with OCL constraints. In: International Workshop on OCL and Textual Modelling, pp. 49–54 (2012)
17. Lampropoulos, L., Sagonas, K.F.: Automatic WSDL-guided test case generation for proper testing of web services. In: International Workshop on Automated Specification and Verification of Web Systems, vol. 98, pp. 3–16 (2012)
18. Mouchawrab, S., Briand, L.C., Labiche, Y., Di Penta, M.: Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. IEEE Transactions Software Engineering 37(2), 161–187 (2011)
19. Noikajana, S., Suwannasart, T.: An improved test case generation method for web service testing from WSDL-S and OCL with pair-wise testing technique. In: International Computer Software and Applications Conference, pp. 115–123 (2009)
20. Farrell-Vinay, P.: Managing Software Testing. Auerbach Publishers (2008)
21. Petrenko, A.: Why automata models are sexy for testers (Invited talk). In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 26–26. Springer, Heidelberg (2007)
22. Timm, J., Gannod, G.: Specifying semantic web service compositions using UML and OCL. In: IEEE International Conference on Web Services, pp. 521–528 (2007)
23. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. 28(2), 183–200 (2002)
24. Zheng, Y., Zhou, J., Krause, P.: An automatic test case generation framework for web services. Journal of Software 2(3), 64–77 (2007)