A DSL for Web Services Automatic Test Data Generation

Macías López   Henrique Ferreiro
Laura M. Castro

Department of Computer Science
University of A Coruña, Spain
{macias.lopez,hferreiro,lcastro}@udc.es

Thomas Arts

Quviq AB
Sweden
thomas.arts@quviq.com

Abstract

Testing web services requires generating wellformed XML data compliant to a WSDL specification. The knowledge on developing QuickCheck data generators for such data is a barrier to the use of QuickCheck by non-experts. The lack of property-based approaches for testing web services makes testing expensive and in practice faults slip through by insufficient test coverage.

We present a domain specific language which reuses the essential part of the WSDL syntax making it straightforward to express WSDL types as QuickCheck generators.

Making it easier to write these data generators in a language mimicking WSDL allows a larger audience to benefit from the advantages of property-based testing.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms reliability, verification

Keywords web services, QuickCheck, WSDL

1. Introduction

One of the main problems of testing is data generation. This problem has two sides: quantity and quality. On the one hand, testing usually involves writing specific test cases, including test data. For testing to be thorough, the number of test cases needs to be significant and so is the effort to write them. On the other hand, for testing to be effective, the data has to be complex enough to expose non-trivial software errors.

Property-based testing (PBT [6]) is a testing approach that can tackle the two sides of this problem. Rather than manually writing individual test cases, PBT requires developers to describe properties of the software. Then, PBT tools such as Erlang QuickCheck [2] can automatically derive hundreds of different test cases from those properties.

While properties describe the functional behaviour of the software, data generators are written by developers to describe the input data. These generators are the key to the significant derivation of test cases capable of exposing non-trivial software errors. Although QuickCheck provides generators for basic types (int, char, lists...), building complex data generators, possibly parametrised by dynamic conditions, is not easy for QuickCheck non-experts.

As an example, if a function requires a list of integers greater than a given integer `N`, a possible way to write a QuickCheck data generator is this:

```
int_list_gt(N) ->
  eqc_gen:list(?SUCHTHAT(M, eqc_gen:int(), M>N)).
```

Sampling this generator `eqc_gen:sample(int_list_gt(5))` will return specific data instances such as:

```
[]
[14]
[16,11]
[18,11,19,12]
[11,18,16,18,7,12]
...
```

However, these kind of simple instances, and consequently such simple generators, may not be enough to represent the required complexity of the data. In particular, generating input data for testing web services frequently requires the ability to observe a set of constraints over that data defined as part of the description of the web service. This description can be provided in a format such as WSDL [19], or just as part of the web service documentation (i.e. a PDF or HTML document).

As an example, an existing web service for a multimedia system which manages the different rooms in which video streaming devices are located within a household, could handle XML-formatted data as:

```
<rooms>
  <room>
    <roomId>95926-69-43-227</roomId>
    <description>Living room</description>
  </room>
  <room>
    <roomId>926-9-43-90</roomId>
  </room>
  <room>
    <roomId>6227-39-5-6920</roomId>
    <description>Kid's room</description>
  </room>
<rooms>
```

To produce such data, a number of constraints have to be implemented in the generators, like:

- Optional information. Web services frequently include fields (*tags*, if in XML format) which are not required for proper operation, such as the `description` in the example above. However, if provided, optional information must be properly processed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
IFL'13 August 28-30th, Nijmegen
Copyright © 2013 ACM [to be supplied]...$15.00

- Number of occurrences. Web services frequently indicate a number of occurrences of information that is expected for proper operation. For instance, a total number of `rooms` between 1 and 5.
- Patterns for content. Web services frequently expect data to fulfil certain format for proper operation, such as the four-number code for `roomId` in the example above.

These constraints are common to many kinds of data, regardless of its particular type, so the fact that developers need to take care of them every time they need to write a data generator for a web service is highly inefficient and error prone.

But more importantly, this naïve approach to data generator writing tends to *first generate* data that is *later on filtered out* according to its required constraints. This makes it rather easy that data ends up being discarded, thus spoiling generation efforts, even if only we want a list of integers greater than 100:

```
> eqc_gen:sample(test_eqc:int_list_gt(100)).
** exception exit:
   {"?SUCHTHAT failed to find a value
                            within 100 attempts.",[]}
```

Rather than putting a lot of effort in generating data that may be no longer valid when we check the constraints it needs to fulfil, we need to proceed the other way around: taking the constraints into account *when generating* the data, not afterwards.

In this paper, we show how we overcome these two issues by defining a DSL which developers can use instead of writing their own data generators. We have designed this DSL taking inspiration from WSDL elements (constraints and keywords), to lower the entry barrier from web service developers even further.

Thus, the contributions of this work are the following:

- The definition of a DSL to express web services data.
- An enabling mechanism for non-expert QuickCheck users to define their own web service data generators.
- We provide a target for automatic testing of web services.

2. The problem of data generation

The advantages of the use of PBT [12] are highly dependent on the definition of good generators for data [7, 11, 16]. As we mentioned in the introduction, writing data generators that produce data taking into account constraints which are not data-type inherent is challenging. Besides, these kind of constraints are indeed applicable to different kinds of data types.

Take for instance our previous example of room information that could be sent to a web service for multimedia management. This example is borrowed from a real web service [9], with the excerpt of WSDL definition shown in Fig. 1.

This defines an operation to create rooms which takes as parameter a user-defined *complex type*, `createRoomParams`. This type consists of a `sequence`, which represents a fixed-length list of the elements it contains: a `roomId` (a string), and a `description`, which can be present or not, as reflected by the use of the constraints `minOccurs=0`, `maxOccurs=1`. These WSDL occurrence modifiers can be applied to any WSDL `element` and, if not present, the convention is `minOccurs=1`, `maxOccurs=1`.

For testing this functionality, we would like to generate data like the one shown in Fig. 2.

For generating such data, we could write a set of generators that produce a list of rooms:

```
createRoomParams() ->
  eqc_gen:list(room()).
```

```
...
<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:documentation>
    It creates a new household with the specified values.
    </wsdl:documentation>

    <wsdl:input element="msg:createRoomParams"/>
    <wsdl:output element="msg:createRoomResponse"/>
</wsdl:operation>
...
<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>
...
```

Figure 1. Samples of test data

```
Test data 1: <rooms></rooms>

Test data 2: <rooms>
               <room>
                 <roomId>5-5-0-10</roomId>
                 <description>za</description>
               </room>
             </rooms>

Test data 3: <rooms>
               <room>
                 <roomId>8-1-1-9</roomId>
               </room>
             </rooms>

Test data 4: <rooms>
               <room>
                 <roomId>13-5-10-12</roomId>
                 <description>rtmsj</description>
               </room>
               <room>
                 <roomId>12-5-6-7</roomId>
               </room>
               <room>
                 <roomId>8-2-6-11</roomId>
                 <description>ydi</description>
               </room>
             </rooms>
```

Figure 2. Samples of test data

each of which has an identifier and –optionally- a description, so we write a generic `optional` utility function to be used with any generator `G`:

```
room() ->
  {roomId(), optional(description())}.

optional(G) ->
  eqc_gen:oneof([[], G]).
```

To comply with the web service specification, room identifiers are four-number codes:

```
roomId() ->
  {eqc_gen:nat(),
   eqc_gen:nat(),
   eqc_gen:nat(),
   eqc_gen:nat()}.
```

and descriptions are just strings of printable characters:

```
description() ->
  eqc_gen:non_empty(eqc_gen:list(printable_char())).
```

An `eqc_gen:sample(rooms())` produces output like

```
[]
[ { { 5,5,0,10},    "za"},
  { {8,1,1,9},        []}  ]
[ { {12,5, 6, 7},     []},
  { {13,5,10,12},"rtmsj"},
  { { 8,2, 6,11},  "ydi"}  ]
...
```

which formatted into XML would look like the test samples in Fig. 2, any of which could be used to feed the web services functionalities that receive a list of rooms (i.e. create rooms, delete rooms, etc.), and specifically the `createRoom` service in Fig. 1.

Even thought the generated data we have just shown may seem suitable for testing purposes, it presents a series of concerning issues.

First, take into account we cannot tell the difference between an *empty* description and a *non-present* description. Both are represented using the empty list (`[]`). With the previous generators, we cannot generate these two different representations:

```
Test data 1:  <room>
                <roomId>12-5-6-7</roomId>
              </room>

Test data 2:  <room>
                <roomId>12-5-6-7</roomId>
                <description></description>
              </room>
```

Figure 3. Non-differentiable test data

which, with testing in mind, constitute two different and equally relevant test cases. Both test cases should end up dealing with the same data (i.e. a room with certain ID and empty description), but we want to test that indeed this is the case.

More importantly, there are cases in which this inability to tell the difference may coerce our capability of expressing relevant situations. In contrast with the previous example, in which we want to allow both the presence of an empty description and the absence of a description at all, the list of rooms *can be empty*, but should *never be absent*. In other words, when invoking functionalities such as create rooms or delete rooms, this is valid input data:

```
<rooms></rooms>
```

but the absence of data is not a valid input.

This behaviour, which we can link to the use of the occurrence contraints in the WSDL definition of the web service, is not limited to this example. A non-exhaustive list of WSDL element constraints that can be used to describe web services is:

- length of elements (`length`, `minLenght`, `maxLength`), not only strings but also lists of items
- patterns (regular expressions) of strings, dates, numbers
- `enumerations`, which define a list of acceptable values for any basic type
- whitespace handling (`preserve`, `replace`, or `collapse` tabs, spaces, carriage returns. . .)

There are other indicators which can be applied to a set of WSDL elements to define order restrictions, such as:

- `all`: specifies that the child elements can appear in any order, and that each child element must occur only once
- `choice`: specifies that either one child element or another can occur
- `sequence`: specifies that the child elements must appear in a specific order

or the already mentioned occurrence restrictions (`maxOccurs`, `minOccurs`).

Our DSL defines a set of constructs which can be used together with basic data type generators provided by tools like Quviq QuickCheck in order to generate data which needs to comply with those kinds of contraints. These constructs will be used in a similar way to other utility functions such as `eqc_gen:non_empty`, `eqc_gen:choose`, or `eqc_gen:oneof` provided by QuickCheck, or the `optional` showed earlier.

Thus, taking as example the `createRoomParams` operation at the beginning of this section (cf. Fig. 1), our main goal is to define a `wsdlType/1` function that generates valid data according to the WSDL specification. We have used a WSDL-inspired syntax.

```
createRoomParams() ->
  wsdlType(tag("room",
               [tag("roomId",string()),
                minOccurs(0,maxOccurs(1,tag("description",
                                            string())
                         ))
               ])
          ).
```

Figure 4. `CreateRoomParams` data generator

3. DSL for data generation

We designed our DSL as a deep embedding [8] of a WSDL combinator language in Erlang. This has allowed us to easily integrate with Erlang QuickCheck [1].

From the point of view of a QuickCheck user, writing data generators using our DSL means:

1. Using a number of combinators which closely resemble WSDL elements and constraint modifiers. This will create an internal data structure representing the desired data type.
2. Wrapping these combinators in a `wsdlType/1` function call. This function will transform the aforementioned internal structure into a data generator.

Internally, the DSL is structured using an opaque abstract syntax tree (`AST`) which represents either basic or complex types, and a list of attributes which add restrictions to them:

```
AST ::= Empty | Int | String | .. |
        Sequence | Union | List |
        {Tag, Attributes, AST}
```

[1] In the rest of this paper, we refer to Erlang QuickCheck or Quviq QuickCheck simply as QuickCheck.

Every constructor in the AST is represented by the general form {Tag, Attrs, Content}. In the case of predefined WSDL types, such as basic types (integers, strings. . .) or complex types (sequences. . .), we use a specialised version where the Tag is an atom with its textual representation.

The Attributes are key-value pairs representing WSDL predefined constraint modifiers (minOccurs, maxLength, etc.), or the attributes of a generic XML element.

The Content of each data description can be the empty list, so that a suitable QuickCheck generator is used in a second phase, or else a literal which represents the identity generator.

For example, the integer 5 is represented in our AST as follows:

```
{int, [], 5}.
```

A generator for 8-char strings is represented as:

```
{string, [{length, 8}], []}.
```

And one of our room examples is represented as:

```
{room, [],
 {sequence, [],
   [{roomId, [], "8124-24-39"},
    {description, [{minOccurs,0},{maxOccurs,1}],
       {string, [], []}}]}}.
```

We show here how the tag/2 function is used to create a XML element by providing its name and contents (cf. Fig. 4). The content of a room is a sequence of two elements: a roomId and a description. They are similarly built, being relevant the second one in which we use the minOccurs/2 and maxOccurs/2 combinators to make it an optional element. Lastly, string/0 creates an empty string element, which will be identified as a place where an string generator is needed.

Once we have built the AST, the wsdlType/1 wrapper involves a two-step process:

1. Consistency checks on the existing constraints, namely, the coherence between constraints (i.e. minLength = 3 and maxLength = 5) and application to suitable data types (i.e. length of a string).
2. Transformation of the AST into a data generator ready to be consumed by QuickCheck properties. This is applied to every place in the AST where no literal is used to describe the type.

This two-step process is needed in order to overcome the problem mentioned in Sec. 1: constraints must be checked when generating data, not afterwards. Also, constraints coherence should be guaranteed prior to any data generation attempt, in order to avoid wasting efforts.

3.1 CreateRoomParams detailed internal generation

In this section we show the detailed behaviour of our DSL for the CreateRoomParams example (Fig. 4).

First, we focus on a representative fragment of the combinators that we have used:

```
maxOccurs(1,tag("description",string()))
```

Combinators such as maxOccurs/2 above are used to apply the attributes they represent to the AST provided as second argument.

```
maxOccurs(N, {Tag, Attributes, Content}) ->
  {Tag, [{maxOccurs,N} | Attributes], Content}.
```

The function tag/2 builds a new AST using its arguments as tag and content, respectively. The function clause pattern-matches on the structure of the second argument to ensure that is AST-shaped:

```
...
tag(Name, {_,_,_} = C) ->
  {Name, [], C};
...
```

Our DSL provides different flavours of the tag combinator which allow the use of Content (as in the previous example), a list of Attributes (as a key-value list) or both of them. Another example of the use of tag in the createRoomParams/0 generator uses a list of ASTs as second argument, thus building a WSDL sequence:

```
...
tag(Name, L) when is_list(L) ->
  {Name, [], sequence(L)};
...
```

Lastly, functions like sequence/1 and string/0 are the ones used to build predefined WSDL types:

```
sequence(Elements) when is_list(Elements) ->
  {sequence, [], Elements}.

string() ->
  {string, [], ""}.

string(S) ->
  {string, [], S}.
```

Some of these functions for predefined WSDL types also take a literal in order to represent ASTs for fixed values, like string/1 above.

Once the AST is built, it is passed as parameter to the wsdlType/1 function. As we already mentioned, this function involves two steps:

```
wsdlType(AST) ->
  ConsistentAST = check_constraints(AST),
  generate(ConsistentAST).
```

The first step, embodied by the check_constraints/1 function, is a compatibility check on the values of the numerical Attributes referred to the same magnitude (e.g. min/max length, occurrences. . .), and a domain check of the values of the other kind of attributes (e.g. replace, preserve or collapse whitespaces in strings). Also, check_constraints is recursive on the third component of the AST, the Content, in order to explore the whole structure in depth.

Finally, after ensuring the AST includes consistent information, we use the generate function to, while preserving the AST structure, remove constraint attributes and replace the content at each level with a suitable generator. For instance, for string contents, a simplified version of the generation machinery is as follows:

```
...
generate({string, Attributes, ""}) ->
  Min = proplists:get_value(minLength, Attributes),
  Max = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(Min,Max),
    {string, escaped_string(vector(N, Gen))};
```

```
generate({string, _, S}) ->
  {string, S};
...
```

This snippet shows how we build a generator for strings (in this case only showing how to deal with the length attribute), and also how we use a string literal when one is provided.

For complex data types such as `sequence` or `union`, we return a list of generators for each member of the `sequence` or a choice (`eqc_gen:oneof/1`) among them in the case of `union`.

4. Using our DSL for testing web services

The main example we have used to conduct this paper, the web service offered by a multimedia system to manage streaming devices in different locations, is part of a real industrial system called 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. VoDKATV is a distributed system composed by several components, which are integrated through web services (cf. Fig. 5).

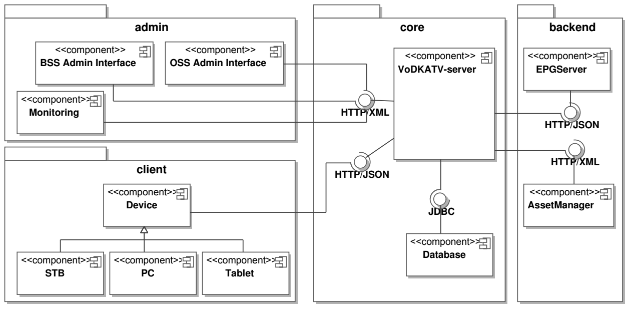

Figure 5. VoDKATV Architecture

The main part of the VoDKATV system is the *core* package, that contains the key components of the system. The core components use *backend* components to get information from external systems, for example the electronic program guide (EPG) of IPTV channels (provided by the EPGServer component) or multimedia contents to rent in a video on demand catalogue (provided by the AssetManager component). On the other hand, the core components are used by the *clients*, i.e., the applications that end-users use to access the system on a TV screen, smartphone or any other compatible device. Finally, the architecture also includes administration applications to manage and configure the system (*admin* package).

As a way to validate the capabilities of our DSL for testing real web services, we have used VoDKATV as case study. In particular, we have used the web service provided by the *VoDKATV-server* component which returns data in XML format. As it is shown in Fig. 5, this web service is used by *administration* components to configure the VoDKATV system, which includes room and device management functionalities.

All in all, given the number of web-service-based component integration scenarios present in this system, web service testing, and specifically data generation for web service testing, is very relevant to VoDKATV developers. Naturally, this style of integration is not unique to VoDKATV, and is also a very commonly used means of integration with third-party components, applications or systems [17].

Prior to the introduction of our DSL, the testing performed by VoDKATV developers on this web service consisted of a manually-written test suite of unit test cases. The data in those test cases had been manually produced as well, and inspection of the test cases revealed that the same data had been copied and pasted in different test cases.

VoDKATV developers were familiar with PBT, and actually used it in other parts of the system, written in Erlang. Difficulty to write proper generators was referred by them as one of the reasons not to engage in QuickCheck usage as far as web services were concerned, alongside with difficulties to model web service operations and behaviour, and time required to put the machinery in place to communicate PBT tools like QuickCheck with HTTP/XML-based interfaces.

Even though at these stage we yet to develop a whole methodology to effectively test all web services available operations in an automatic fashion using PBT, we have been able to use our DSL to produce QuickCheck generators and automatically build data to feed the web services, instead of the manually-written data in the test suite. In doing so, we were able to detect unspecified behaviour of the VoDKATV system when the rooms to be created contained *strange* characters (for instance, ä) as part of their descriptions. Being it an optional field, the rooms were created anyway, but the web service itself crashed badly and the web server returned a severe error message which revealed data of the internals of the application (i.e. complete error paths).

Such behaviour had never been triggered using the test cases in the manually-written test suite, because developers writing the test data would commonly used sensible, real-like strings using *common* characters to them. The problem was reported and diagnosed by VoDKATV developers as a case of bad handling of character encodings withing the server.

5. Related work

Most of the research carried out about web services testing focuses on methods to create good test suites, from a black-box perspective. Examples of this are [3], where authors use a technique called Orthogonal Array Testing (OAT); the pair-wise method defined in [13], which uses semantic information through WSDL-S [20] and OCL [15] to generate test cases; or a partition testing technique in [4].

Other proposals also build intermediate artefacts to help in the process, such as [21], which builds a finite automata using BPEL [14], or in [18] which uses UML and OCL. A complete framework to perform testing using a WSDL specification is explained in [5], where test cases are generated using a coverage criteria and some heuristics.

All the references above focus on the generation of test cases and give data generation little discussion. One different example is [10], where the authors present one fully automatic approach to test web services and also a PBT tool is used. Although they use a WSDL document to generate test cases, they only take into account the main type of the inputs/outputs (integers, strings…), leaving out the rest of the WSDL constraints.

In our review of the state of the art, we did not find any work which uses a DSL or some intermediate language to build test cases for web services testing in order to enhance test data quality. So, to the best of our knowledge, our work is a novel contribution to this research area.

6. Conclusions and future work

In this paper, we have described the definition of a DSL for writing QuickCheck data generators for web services. In order to lower the entry barrier for QuickCheck non-experts, we have taken inspiration from the WSDL elements themselves when designing the syntax of our DSL.

The use of the described DSL overcomes two important pitfalls in the definition of data generators for web service data:

- the need to express constraints on data that are not type-dependent, rather applicable to different kinds of data;
- the need to take constraints on data into account at generation time, rather than once the data has already been generated.

As we have shown, our DSL includes

- a set of combinators that allow developers to consistently apply constraints on all kinds of data;
- a common wrapper that interprets and transforms user-defined generators using these combinators into QuickCheck-ready data generators.

Consequently, we contribute not only the definition of a DSL to express web services data, but more importantly, an enabling mechanism for non-expert QuickCheck users to define their own web service data generators. Also, our DSL is a target for automatic testing of web services.

In fact, this work is part of a broader research. Having this DSL in place allows us to take the next step and use it as a target for additional tools designed to automate the testing process. Specifically, after addressing the syntax of web services, meaningful testing requires addressing web semantics as well. We intend to provide support for QuickCheck-flavoured ways to express these semantics, in a similar fashion to how we have mimicked WSDL constraints syntax in our DSL.

This research is developed in the context of the PROWESS research project [1], devoted to the improvement of web services testing by applying PBT in this field. As such, our DSL will be integrated with results from other project partners. For instance, Professor Thompson and his research group at the University of Kent are working on the automatic derivation of QuickCheck state machines from WSDL web service descriptions. Our common intention is to have their generation tool produce suitable data generators directly using our DSL.

Acknowledgments

This research has been partially funded by the European Framework Program, FP7-ICT-2011-8 Ref. 317820, and Xunta de Galicia CN2011/007, partially supported by FEDER funds.

References

[1] PROWESS Project (Property-based Testing for Web Services). `http://www.prowessproject.eu`, July, 2013.

[2] T. Arts and J. Hughes. Erlang/quickcheck. *Ninth International Erlang/OTP User Conference*, 2003.

[3] A. Askarunisa, A. Abirami, and S. Mohan. A test case reduction method for semantic based web services. 2010.

[4] X. Bai, S. Lee, W. Tsai, and Y. Chen. Ontology-based test modeling and partition testing of web services. pages 465–472, 2008.

[5] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Ws-taxi: A wsdl-based testing tool for web services. pages 326–335, 2009.

[6] J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-A. Fredlund, V. Gulias, J. Hughes, and S. Thompson. Property-based testing: the protest project. In *Proceedings of the 8th international conference on Formal methods for components and objects*, FMCO'09, pages 250–271, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17070-6, 978-3-642-17070-6.

[7] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267. URL `http://doi.acm.org/10.1145/263244.263267`.

[8] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[9] Interoud Innovation S.L. VoDKATV. `http://www.interoud.com`.

[10] L. Lampropoulos and K. F. Sagonas. Automatic wsdl-guided test case generation for proper testing of web services. In *WWV*, volume 98, 2012.

[11] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Trans. Softw. Eng.*, 37(2):161–187, Mar. 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.32. URL `http://dx.doi.org/10.1109/TSE.2010.32`.

[12] A. Nilsson, L. M. Castro, S. Rivas, and T. Arts. Assessing the effects of introducing a new software development process: a methodological description. *International Journal on Software Tools for Technology Transfer*, pages 1–16, 2013.

[13] S. Noikajana and T. Suwannasart. An improved test case generation method for web service testing from wsdl-s and ocl with pair-wise testing technique. volume 1, pages 115–123, 2009.

[14] OASIS Project. Business Process Execution Language (BPEL). `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

[15] OMG. Object Constraint Language (OCL). `http://www.omg.org/spec/OCL/2.3.1/`, 2012.

[16] P. Farrell-Vinay. *Managing Software Testing*. Auerbach Publishers, 2008.

[17] M. Stal. Web services: beyond component-based computing. *Commun. ACM*, 45(10):71–76, Oct. 2002. ISSN 0001-0782. doi: 10.1145/570907.570934. URL `http://doi.acm.org/10.1145/570907.570934`.

[18] J. Timm and G. Gannod. Specifying semantic web service compositions using uml and ocl. pages 521–528, 2007.

[19] W3C. Web Services Description Language (WSDL). `http://www.w3.org/TR/wsdl`, 2001.

[20] W3C. Web Services Semantics (WSDL-S). `http://www.w3.org/Submission/WSDL-S/`, 2005.

[21] Y. Zheng, J. Zhou, and P. Krause. An automatic test case generation framework for web services. *Journal of Software*, 2(3):64–77, 2007.