A DSL for Web Services Automatic Test Data Generation

Macías López [1] Henrique Ferreiro [1] Laura M. Castro [1]
Thomas Arts [2]

[1]MADS Research Group
University of A Coruña (Spain)

[2]Quviq AB (Sweden)

Nijmegen. August 30, 2013

Outline

1. Introduction
2. The problem
3. The solution: DSL
4. Using the DSL for testing web services
5. Conclusions and future work

Introduction

One of the main problems of testing: **data generation**.

- Quantity: writing significant and enough test cases.
- Quality: the data has to be complex enough.

Property-based testing (PBT)

Describe properties of the software. **QuickCheck**: can automatically derive different test cases from properties.

Introduction

We need **data generators**:

- To describe input data.
- To derive significant test cases capable of exposing non-trivial software errors.

Example

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

eqc_gen:sample(int_list_gt(5)).

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

The problem

The problem

Generating input data for **testing web services** must:

1. Take into account some **constraints**. They are usually defined in a **WSDL** file. Examples:
   - Optional information.
   - Number of occurrences of certain information.
   - Patterns for content.
2. Take these constraints into account **when generating** the data, not afterwards.
   - They are not data-type inherent.
   - Generating first and filter out later could even not generate any value:

```
eqc_gen:sample(int_list_gt(100)).
```

```
** exception exit:
  {"SUCHTHAT failed to find a value
                        within 100 attempts.",[]}
```

The problem

WSDL description

```
 ...
<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>
...
<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> ...
```

The problem

WSDL description

```
 ...
<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>
...
<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> ...
```

The problem

WSDL description

```
 ...
<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>
...
<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> ...
```

The problem

WSDL description

```
 ...
<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>
...
<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> ...
```

The problem

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> </rooms>
```

The problem

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> </rooms>
```

The problem

Possible implementation

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

The problem

Possible implementation

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


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

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

The problem

Possible implementation

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


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

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


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

The problem

Possible implementation

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


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

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


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


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

The problem

Possible implementation: sampling

eqc_gen:sample(createRoomParams())

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

This data formatted into XML would feed the previous `createRoom` web service.

The problem

Possible implementation: problems

But even the previous implementation presents concerning issues...

The problem

Possible implementation: problems

But even the previous implementation presents concerning issues...

- No difference between **empty** and **non-present** description. Both are represented using the empty list.

Example 1

Test data 1:

```
<room>
  <roomId>12-5-6-7</roomId>
</room>
```

Test data 2:

```
<room>
  <roomId>12-5-6-7</roomId>
  <description></description>
</room>
```

- Some tags **can be empty** but should **never be absent**.

Example 2

Valid data: `<rooms></rooms>`

The problem

There is a **non-exhaustive list of WSDL element constraints** that can describe web services:

- length of elements (`length`, `minLenght`, `maxLength`), not only strings but also lists of items
- patterns (regular expressions)
- whitespace handling (`preserve`, `replace`, or `collapse` tabs, spaces, carriage returns...)
- Order constraints, like `sequence`, that specifies that the child elements must appear in a specific order
- Already explained occurrence constraints like `maxOccurs` and `minOccurs`.

The solution: DSL

The DSL will define a **set of combinators** which can be used together with basic data type generators provided by QuickCheck.

Previous WSDL

```
<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>
```

The solution: DSL

The DSL will define a **set of combinators** which can be used together with basic data type generators provided by QuickCheck.

Previous WSDL

```
<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>
```

In our DSL

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

The solution: DSL

Writing data generators in our DSL for web services testing involves:

1. Using **combinators** which resemble WSDL elements $\rightarrow$ **AST**

2. Wrapping these combinators in `wsdlType/1` $\rightarrow$ **data generator**

The solution: DSL

1. Using combinators to build the AST

General form → `{Tag, Attributes, Content}`

- `Tag` → atom

```
{int, [], 5}
{string, [], ``ifl2013''}
{sequence, [], []}
{roomId, [], ``5-5-0-10''}
```

The solution: DSL

1. Using combinators to build the AST

General form → `{Tag, Attributes, Content}`

- `Attributes` → list of key-value pairs for constraints
  - `maxOccurs`, `minOccurs`, `length` ...

```
{string, [{minOccurs,0},
          {maxOccurs,1}],``ifl2013''}
{string, [{length,8}], gen}
```

The solution: DSL

1. Using combinators to build the AST

General form → `{Tag, Attributes, Content}`

- `Content` → 3 different *flavours*
  1. For **basic types** (integers, strings...): **literal** or a **special atom**

```
{int, [], 5}
{string, [{length, 8}], gen}
```

  2. For **complex types** (sequence, choice, tags...): **list of elements**

```
{sequence, [],
  [{roomId, [], ``5-5-0-10''},
   {description, [], ``myRoom''}]}
```

  3. **Nested AST**

```
{rooms, [],
  {room, [],
   {sequence, [], ... } } }
```

The solution: DSL

1. Using combinators to build the AST

Combinators

```
wsdlType(tag("room",
               [tag("roomId","8124-24-39"),
                minOccurs(0,maxOccurs(1,tag("description",
                                            string()))))])).
```

Internal AST

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

The solution: DSL

1. Using combinators to build the AST

Combinators

```
wsdlType(tag("room",
               [tag("roomId","8124-24-39"),
                minOccurs(0,maxOccurs(1,tag("description",
                                            string()))])).
```

Internal AST

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

The solution: DSL

1. Using combinators to build the AST

Combinators

```
wsdlType(tag("room",
                [tag("roomId","8124-24-39"),
                 minOccurs(0,maxOccurs(1,tag("description",
                                             string()))])).
```

Internal AST

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

The solution: DSL

1. Using combinators to build the AST

Combinators

```
wsdlType(tag("room",
                [tag("roomId","8124-24-39"),
                 minOccurs(0,maxOccurs(1,tag("description",
                                              string()))))])).
```

Internal AST

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

The solution: DSL

1. Using combinators to build the AST

Combinators

```
wsdlType(tag("room",
                [tag("roomId","8124-24-39"),
                 minOccurs(0,maxOccurs(1,tag("description",
                                             string()))))])).
```

Internal AST

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

The solution: DSL

2. Wrapping in `wsdlType/1` function

1. **Consistency checks** on the existing constraints.

2. **Transformation of the AST into a data generator:**
   - Applied at every place in the AST where the special atom **gen** is used.

Needed to overcome the previous problem

- Get constraints before generation
- Check their coherence

The solution: DSL

2.1. Consistency checks

- **Coherence** between them: `minLengh=3` and `maxLength=5`
- Application to **suitable data types**: `length` or `preserve` (whitespaces) to strings.
- Recursive on the `Content`.

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
 {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

AST

```
{description, [{minOccurs,0},{maxOccurs,1}],
      {string, [{minLength,5},{maxLength,8}], gen}}
```

Transformation

```
generate({string, Attributes, gen}) ->
  MinL = proplists:get_value(minLength, Attributes),
  MaxL = proplists:get_value(maxLength, Attributes),
  Gen = ascii_char(),
  ?LET(N, choose(MinL, MaxL),
    escaped_string(vector(N, Gen)));

generate({Tag, Attributes, Content}) ->
  {MinO, MaxO, Attrs} = get_occurs(Attributes),
  Gen = explore({Tag, Attrs, Content}),
  ?LET(N, choose(MinO, MaxO), vector(N, Gen)).

explore({Tag, Attrs, Content}) ->
  C = generate(Content),
  {Tag, Attrs, C}.
```

The solution: DSL

2.2. Transformation of the AST into a data generator

Our final generator

```
{description, [],
  {?LET(N, choose(0,1),
    vector(N, ?LET(M, choose(5,8), vector(M, ascii_char()))))}}
```

Sampling

```
{description,[],[]}
{description,[],["  zY)y"]}
{description,[],["’ T) A"]}
{description,[],[]}
{description,[],[]}
{description,[],["d^qLe4"]}
{description,[],["rym1 t "]}
{description,[],[]}
{description,[],["#2‘c4t"]}
{description,[],["lN9 Jc "]}
{description,[],[" 4  X"]}
ok
```

Using the DSL for testing web services

Using our DSL

- The example used in this talk, `createRoomParams` is a web service offered by **VoDKATV**.
- System to manage streaming devices in different locations.

Even we are in a very early stage to effectively test all operations in a WSDL file...

- The web server **crashed** when the `description` field was feeded with *strange* characters like `ä`, revealing data of the internals of the application.
- This behaviour had never been triggered using manually-written test cases.

Conclusions

Our DSL overcomes **2 important pitfalls** in the definition of data generators for web service data:

- The need to **express constraints** on data that are **not type-dependent**.
- The need to take constraints into account **at generation time**.

Our DSL includes:

- A **set of combinators** to apply constraints on all kinds of data.
- A common wrapper to **transform user-defined generators into QuickCheck data generators**.

More importantly

Enabling mechanism for non-expert QuickCheck users to define their own web service data generators.

Future work

- Improve the DSL to address **web semantics** in order to automate the testing proccess.

- Integrate the DSL with other tools in PROWESS project for **automatic derivation of QuickCheck state machines** from WSDL descriptions.

Questions?

Audience ! thanks

macias.lopez@udc.es