Brief Overview of the PropEr Extension for Web Service Testing

1 Introduction

We describe an extension of PropEr to accomodate property based testing of web services. This extension receives the URL of a Web Service (or more accurately the url of the Web Service's WSDL specification), parses the specification and creates an output file containing generators, calling functions and sample properties for all the input messages and types required by the Web Service to be invoked. The output file can be automatically compiled and used for response testing of web services with random, yet structurally valid testcases. In addition, the automatically created PropEr generators and properties can be easily modified, allowing the user to write properties the web service should have (? Edw dn m aresei to have), utilizing the full power of Erlang and PropEr.

2 Prerequisites

To use the extension required are an Erlang distribution (with xmerl), Proper and Yaws. To run some of the examples below, Eclipse and Tomcat have been used, but any means of creating a Web Service should be an option.

2.1 PropEr

Include here:
-What property based testing is
-What PropEr is
-Small Example?

2.2 WSDL

WSDL is the leading specification for web services in XML format, describing the web service in full: Operations, Input and output message types, locations, etc. Every WSDL specification contains (or references) an XSD schema inside, describing the types of the messages needed to invoke the web service.

2.3 Yaws

Yaws is the most widely used Erlang HTTP webserver. We basically use only its soap library to handle the interaction between our erlang code and the web services, wrapping our types and generated data in valid soap structure, communicating with the web service, and returning the result. Yaws uses an XML parser called Erlsom to handle its soap message encoding/decoding, a parser module faster and more user friendly than the xmerl module of the Erlang distribution, imposing however a few additional limitations.

3 Overview

The main idea behind the extension is to use PropEr to do property based testing on web services. This extension helps the user by creating generators, sample properties and call functions so that the user focuses on the important task - writing the property.

3.1 Generators

The most important part of the extension is automatically creating PropEr generators for each input message type of the operations supported. To that end, we parse the XSD schema of the WSDL specification using the xmerl module and create the most generic generators for the types described, in the format Yaws and Erlsom expect their arguments. While parsing the XSD schema, we use a small tuple format as an intermediate representation of the types of the Web Service, so that the resulting generators can be directly mapped from these tuples while being able to give the generators unique and descriptive names. These tuples are never encountered by the user, yet are extremely helpful in holding all the necessary information (and only that) to create the PropEr generators and overcome some important limitations that would otherwise be imposed by Erlsom. (? Thelei kati allo edw prin apo to paradeigma ?)

For our example we will use an existing free web service for converting cooking units from one form to another. This web service is hosted in webservicex.net and its WSDL specification can be found at "http://www.webservicex.net/ConvertCooking.asmx?WSDL". Here we show the schema part of the specification which describes the type of the soap message to be sent:

```
<s:schema elementFormDefault="qualified"
  targetNamespace="http://www.webserviceX.NET/">
  <s:element name="ChangeCookingUnit">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="1" maxOccurs="1"
          name="CookingValue" type="s:double"/>
        <s:element minOccurs="1" maxOccurs="1"
          name="fromCookingUnit" type="tns:Cookings"/>
        <s:element minOccurs="1" maxOccurs="1"
```

```
            name="toCookingUnit" type="tns:Cookings"/>
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:simpleType name="Cookings">
      <s:restriction base="s:string">
        <s:enumeration value="drop"/>
        <s:enumeration value="dash"/>
        <s:enumeration value="pinch"/>
        <s:enumeration value="milliliterPerCC"/>
        <s:enumeration value="deciliter"/>
        <s:enumeration value="coffeeSpoon"/>
        <s:enumeration value="fluidDram"/>
        <s:enumeration value="teaspoonUS"/>
        <s:enumeration value="teaspoonUK"/>
        <s:enumeration value="tablespoonUS"/>
        <s:enumeration value="tablespoonUK"/>
        <s:enumeration value="fluidOunceUS"/>
        <s:enumeration value="fluidOunceUK"/>
        <s:enumeration value="cupUS"/>
        <s:enumeration value="cubicInch"/>
        <s:enumeration value="gillUS"/>
        <s:enumeration value="gillUK"/>
        <s:enumeration value="pintUS"/>
        <s:enumeration value="pintUK"/>
        <s:enumeration value="quartUS"/>
        <s:enumeration value="liter"/>
        <s:enumeration value="gallonUS"/>
        <s:enumeration value="gallonUK"/>
        <s:enumeration value="TwoPointFiveCan"/>
        <s:enumeration value="TenCan"/>
      </s:restriction>
    </s:simpleType>
    <s:element name="double" type="s:double"/>
</s:schema>
```

In short, the above XSD schema describes one element ("ChangeCookingUnit") whose type is a complex one, a sequence (mapped as a list in the Yaws soap library) consisting of exactly 3 elements (the attributes min- and maxOccurs define that). These three ordered elements are: "CookingValue" with type double, and two more elements "fromCookingUnit" and "toCookingUnit" with type "tns:Cookings", which is described below the element to be a restriction of the generic string type, with only the values described in the enumeration tags as acceptable.

The resulting generators created by the PropEr extension are:

```
generate_ChangeCookingUnit_1_CookingValue() ->
  ?LET(Gen, float(), float_to_list(Gen)).

generate_ChangeCookingUnit_1_fromCookingUnit_Cookings() ->
  elements(["drop","dash","pinch","milliliterPerCC","deciliter",
  "coffeeSpoon", "fluidDram","teaspoonUS","teaspoonUK","tablespoonUS",
  "tablespoonUK", "fluidOunceUS","fluidOunceUK","cupUS","cubicInch",
  "gillUS","gillUK","pintUS","pintUK","quartUS","liter","gallonUS",
  "gallonUK","TwoPointFiveCan","TenCan"]).

generate_ChangeCookingUnit_1_toCookingUnit_Cookings() ->
  elements(["drop","dash","pinch","milliliterPerCC","deciliter",
  "coffeeSpoon", "fluidDram","teaspoonUS","teaspoonUK","tablespoonUS",
  "tablespoonUK", "fluidOunceUS","fluidOunceUK","cupUS","cubicInch",
  "gillUS","gillUK","pintUS","pintUK","quartUS","liter","gallonUS",
  "gallonUK","TwoPointFiveCan","TenCan"]).

generate_ChangeCookingUnit_1() ->
  ?LET(
    {Pr_ChangeCookingUnit_1_CookingValue,
     Pr_ChangeCookingUnit_1_fromCookingUnit_Cookings,
     Pr_ChangeCookingUnit_1_toCookingUnit_Cookings},
    {generate_ChangeCookingUnit_1_CookingValue(),
     generate_ChangeCookingUnit_1_fromCookingUnit_Cookings(),
     generate_ChangeCookingUnit_1_toCookingUnit_Cookings()},
    [Pr_ChangeCookingUnit_1_CookingValue,
     Pr_ChangeCookingUnit_1_fromCookingUnit_Cookings,
     Pr_ChangeCookingUnit_1_toCookingUnit_Cookings]
  ).
```

The extension has created a single generator for the main element named "generate_ChangeCookingUnit_1()" which recursively calls 3 more generators for the 3 elements described above, and using the instances created to form a 3-element list.

One thing worth noticing is that instead of simply using a float() generator for the double field of the Web Service, we use a wrapper that converts it to a string. This is done because of erlsom limitations. Specifically, erlsom treats some types defined in the XSD schemas natively while turns everything else to a string. This leads to irregularities such as using a long instead of an int in the schema to require a string instead of an integer (without considering the range). This sort of irregularity is taken into account by the extension which creates the string wrappers wherever necessary.

3.2 Response Property

The Proper extension also creates a function for each web service operation that invokes this operation with parameterized arguments and a small property to test that the Service always responds for random testcases without returning malformed XML or a Soap Fault for random (possibly corner) cases. For the Cooking service we described above there is a single soap operation supported, which is defined in the following WSDL specification exerpt:

```
<wsdl:portType name="CookingUnitSoap">
  <wsdl:operation name="ChangeCookingUnit">
    <wsdl:input message="tns:ChangeCookingUnitSoapIn"/>
    <wsdl:output message="tns:ChangeCookingUnitSoapOut"/>
  </wsdl:operation>
</wsdl:portType>
```

The code created to invoke this operation and the respective property is the following:

```
call_ChangeCookingUnit(Arguments) ->
  inets:start(),
  Wsdl = yaws_soap_lib:initModel(?WSDL_URL),
  yaws_soap_lib:call(Wsdl, "ChangeCookingUnit", Arguments).

call_ChangeCookingUnit(WSDL, Arguments) ->
  yaws_soap_lib:call(WSDL, "ChangeCookingUnit", Arguments).

prop_ChangeCookingUnit_responds() ->
  ?FORALL(Args, generate_ChangeCookingUnit_1(),
    begin
      Result = call_ChangeCookingUnit(Args),
      case Result of
        {ok, _Attribs, [#'soap:Fault'{}]}) -> false;
        {ok, _Attribs, _Result_record} -> true;
        _ -> false
      end
    end).
```

The first function simply calls the operation assuming nothing about the state of the connection (it tries to start a connection, (possibly re-)parses the WSDL specification and then invokes the operation, while the second function takes an already parsed WSDL specification structure and invokes the operation directly, which is far more efficient. Finally, the "responds" property simply creates the arguments using the previously created generator ("generate_ChangeCookingUnit_1()") and does a pattern matching on the resulting tuple after calling the web service. If this resulting tuple is a 'soap:Fault' record (or the resulting tuple is an{error, Reason} tuple) then the service failed to respond correctly (possibly raised an exception described in the record), otherwise we

assume (conservatively) that the web service responded correctly.

3.3 Entire generated code

For the resulting file to be directly compilable, we include headers, defines, imports, etc. The resulting output file for the cooking service is:

```
-module(proper_output).

-include_lib("proper/include/proper.hrl").
-include("proper_output.hrl").

-define(PREFIX, "properns").
-define(WSDL_URL, "http://www.webservicex.net/ConvertCooking.asmx?WSDL").

-export([call_ChangeCookingUnit/1, call_ChangeCookingUnit/2]).
-export([answer_ChangeCookingUnit/1]).

generate_ChangeCookingUnit_1_CookingValue() ->
  ?LET(Gen, float(), float_to_list(Gen)).

generate_ChangeCookingUnit_1_fromCookingUnit_Cookings() ->
  elements(["drop","dash","pinch","milliliterPerCC","deciliter",
  "coffeeSpoon", "fluidDram","teaspoonUS","teaspoonUK","tablespoonUS",
  "tablespoonUK", "fluidOunceUS","fluidOunceUK","cupUS","cubicInch",
  "gillUS","gillUK","pintUS","pintUK","quartUS","liter","gallonUS",
  "gallonUK","TwoPointFiveCan","TenCan"]).

generate_ChangeCookingUnit_1_toCookingUnit_Cookings() ->
  elements(["drop","dash","pinch","milliliterPerCC","deciliter",
  "coffeeSpoon", "fluidDram","teaspoonUS","teaspoonUK","tablespoonUS",
  "tablespoonUK", "fluidOunceUS","fluidOunceUK","cupUS","cubicInch",
  "gillUS","gillUK","pintUS","pintUK","quartUS","liter","gallonUS",
  "gallonUK","TwoPointFiveCan","TenCan"]).

generate_ChangeCookingUnit_1() ->
  ?LET(
    {Pr_ChangeCookingUnit_1_CookingValue,
     Pr_ChangeCookingUnit_1_fromCookingUnit_Cookings,
     Pr_ChangeCookingUnit_1_toCookingUnit_Cookings},
    {generate_ChangeCookingUnit_1_CookingValue(),
     generate_ChangeCookingUnit_1_fromCookingUnit_Cookings(),
     generate_ChangeCookingUnit_1_toCookingUnit_Cookings()},
    [Pr_ChangeCookingUnit_1_CookingValue,
```

```
      Pr_ChangeCookingUnit_1_fromCookingUnit_Cookings,
      Pr_ChangeCookingUnit_1_toCookingUnit_Cookings]
  ).

call_ChangeCookingUnit(Arguments) ->
  inets:start(),
  Wsdl = yaws_soap_lib:initModel(?WSDL_URL),
  yaws_soap_lib:call(Wsdl, "ChangeCookingUnit", Arguments).

call_ChangeCookingUnit(WSDL, Arguments) ->
  yaws_soap_lib:call(WSDL, "ChangeCookingUnit", Arguments).

prop_ChangeCookingUnit_responds() ->
  ?FORALL(Args, generate_ChangeCookingUnit_1(),
    begin
      Result = call_ChangeCookingUnit(Args),
      case Result of
        {ok, _Attribs, [Result_record]}
            when is_record(Result_record, 'soap:Fault') -> false;
        {ok, _Attribs, _Result_record} -> true;
        _ -> false
      end
    end).

answer_ChangeCookingUnit({ok, _, [Answer_record]}) -> Answer_record.

% ---------- End of Auto Generated Code ----------
```

In addition to the headers, the only other thing created by the extension is a function called "answer_ChangeCookingUnit" which shows how to extract the Answer (record) that is returned by yaws.

Finally, the extension also uses erlsom to output a .hrl file that describes the records used for the responses of the Service. In our case:

```
-record('properns:ChangeCookingUnit',
        {anyAttribs, 'CookingValue', 'fromCookingUnit', 'toCookingUnit'}).
-record('properns:ChangeCookingUnitResponse',
        {anyAttribs, 'ChangeCookingUnitResult'}).
-record('properns:double', {anyAttribs, 'double'}).
-record('soap:Body', {anyAttribs, choice}).
-record('soap:Envelope', {anyAttribs, 'Header', 'Body', choice}).
-record('soap:Fault', {anyAttribs, 'faultcode', 'faultstring', 'faultactor',
-record('soap:Header', {anyAttribs, choice}).
-record('soap:detail', {anyAttribs, choice}).
```

We will see how these can be used in later sections.

4 Response Testing

(? Den ekana akoma to change gia na kanei compile apeftheias ston kwdika gia response testing ?)

The first kind of testing, that works without any user additions is response testing. Basically, the output file created by the extension contains a property that invokes an operation of the web service with random inputs and expects an answer for each different input. This can basically check if a web service crashes for a specific input or similar unwanted behaviors.

Let's see how to use the extension on the cooking service:

```
Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4]
[async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9  (abort with ^G)
1> wsdl_handler:generate("http://www.webservicex.net/
ConvertCooking.asmx?WSDL").
ok
2> c(proper_output).
{ok,proper_output}
3> proper:quickcheck(
        proper_output:prop_ChangeCookingUnit_responds()).
....(100 dots) .....
OK: Passed 100 test(s).
true
```

As we can see, the Cooking web service was invoked 100 times with random arguments and returned a correctly formed result each time.

Now for an example of a service that crashes, we created our own web service using eclipse and tomcat. This example will also be used later on and is based on the first PropEr publication by Manos Papadakis, about a faulty delete function.

We have a simple Java implementation of the web service:

```
public Class Delete {
    private String delete(String in, char c, StringBuffer acc){
        if (in.equals("")) {
            return acc.toString();
        }
        else if (in.charAt(0) == c) {
            return acc.toString().concat(in.substring(1));
        }
        else {
            return delete(in.substring(1), c, acc.append(in.charAt(0)));
```

```
        }
    }

    public String delete(String in, String c){
        return delete(in, c.charAt(0), new StringBuffer(""));
    }
}
```

We used this code to implement and publish a simple web service in tomcat. After using the PropEr extension to handle the WSDL specification of this web service, the output file was the following:

(?Den dokimasa akoma to ascii_list gia na to valw ki edw, opou vector(Len, integer(32,127)) theoroume string ?)

```
-module(proper_output).

-include_lib("proper/include/proper.hrl").
-include("proper_output.hrl").

-define(PREFIX, "properns").
-define(WSDL_URL, "http://localhost:8080/
    DeleteProject/services/Delete?WSDL").

-export([call_delete/1, call_delete/2]).
-export([answer_delete/1]).

generate_delete_1_in() ->
        ?LET(
                Len,
                range(0,inf),
                vector(Len, integer(32,127))
        ).

generate_delete_1_c() ->
        ?LET(
                Len,
                range(0,inf),
                vector(Len, integer(32,127))
        ).

generate_delete_1() ->
        ?LET(
                {Pr_delete_1_in, Pr_delete_1_c},
                {generate_delete_1_in(), generate_delete_1_c()},
                [Pr_delete_1_in, Pr_delete_1_c]
```

```
        ).

call_delete(Arguments) ->
  inets:start(),
  Wsdl = yaws_soap_lib:initModel(?WSDL_URL),
  yaws_soap_lib:call(Wsdl, "delete", Arguments).

call_delete(WSDL, Arguments) ->
  yaws_soap_lib:call(WSDL, "delete", Arguments).

prop_delete_responds() ->
  ?FORALL(Args, generate_delete_1(),
    begin
      Result = call_delete(Args),
      case Result of
        {ok, _Attribs, [Result_record]}
             when is_record(Result_record, 'soap:Fault') -> false;
        {ok, _Attribs, _Result_record} -> true;
        _ -> false
      end
    end).

answer_delete({ok, _, [Answer_record]}) -> Answer_record.

% ---------- End of Auto Generated Code ----------
```

Calling quickcheck with this property reveals a flaw in our implementation.

```
> proper:quickcheck(proper_output:prop_delete_responds()).
.!
Failed: After 2 test(s).
[[46],[]]

Shrinking .(1 time(s))
[[],[]]
false
```

In our implementation we assume that the string c which is supposed to contain at its first character the character that should be removed from the string, is not empty. We have two choices to fix it, either fix our implementation, or remove this testcase from the generator. To show how easy it is to change the generators created by the extension we choose the latter: we change the generator of c to:

```
generate_delete_1_c() ->
        ?LET(
                Len,
```

```
                range(1,1),
                vector(Len, integer(32,127))
        ).
```

We could change it to range(1,inf) or actually remove the ?LET macro, but we'd have to be carefull not to actually change the output of the generator to a char instead of a non-empty char list.

Now only valid testcases are created. Testing the response property again we get:

```
> proper:quickcheck(
        proper_output:prop_delete_responds()).
....(100 dots) .....
OK: Passed 100 test(s).
true
```

5 RESTfull Property Based Testing

Restfull web services are web services compliant to a few constraints, the main one having to do with state: Each request from any client must contain all the information needed by the service to handle the request. This makes property-based testing of these services relatively easy, since each test is self-contained and does not affect the outcome of other tests.

Let's see the delete example in more depth now.

The answer_result function shows how to extract the Answer record from the call response. This response is in the form of an erlsom-created record, described in the .hrl file created by the wsdl_handler. In the .hrl file we see a record:

```
-record('p:deleteResponse', {anyAttribs, 'deleteReturn'}).
```

This record is the record that contains the result (when there is no error).In the automatically generated code we can locate this function:

```
answer_delete({ok, _, [Answer_record]}) -> Answer_record.
```

We see that the 'delete response' is a record containing a 'deleteReturn' value. This 'deleteReturn' value is the boolean we seek; therefore we change the answer function to:

```
answer_delete({ok, _, [Answer_record]}) ->
  Answer_record#'p:deleteResponse'.'deleteReturn'.
```

Finally we write our property in a similar form to the respond property:

```
prop_delete_removes_every_x() ->
  ?FORALL([_In, [C]]=Args, generate_delete_1(),
    begin
      R1 = call_delete(Args),
      Result = answer_delete(R1),
      not lists:member(C, Result)
  end).
```

Now checking this property with proper yields the following results:

```
> proper:quickcheck(proper_output:prop_delete_removes_every_x()).
.....................!
Failed: After 24 test(s).
[[92,62,59,97,51,79,76,86,96,75,38,76,51,57,125,37,121,53,57,66,
67,36,96,89,100,58,38,85,88,121,65,101,106,46,58,32,123,60,
117,34,106,42,110,127,40,51,75,124,34,94,115,76],[57]]

Shrinking ..............................(32 time(s))
[[32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,32,
32,32,32,32,32,32,32,32,33,32,32],[32]]
false
```

We see that we find after a while a counter-example, however the shrink does not work exactly as the non-web service case. This is mostly because of the different generators used to create these arguments, since the shrinking is now handled differently.

(? Afou to parapanw ginei ascii list logika tha allaksei kai to shrinking behavior tou, twra ofeiletai sto oti to me to ?LET wraper dn kanei shrink parapanw ?)

Now we show another example of testing a RestFul Web Service, to show some limitations of our implementation. Going back to the Cooking service, a simple property that should be valid in it would be if we convert something and then convert the answer back we should get the initial number.

Let's change the answer function and write this property on the output file of the extension:

```
answer_ChangeCookingUnit({ok, _, [Answer_record]}) ->
  Answer_record#'p:ChangeCookingUnitResponse'.'ChangeCookingUnitResult'.

prop_back_and_forth_is_equal() ->
  ?FORALL([Value, From, To] = Args, generate_ChangeCookingUnit_1(),
    begin
      Result = call_ChangeCookingUnit(Args),
      Conv_val = answer_ChangeCookingUnit(Result),
      Result2 = call_ChangeCookingUnit([Conv_val, To, From]),
      Initial = answer_ChangeCookingUnit(Result2),
      Value == Initial
    end).
```

Basically, we break down the arguments to the sub-elements, invoke the web service, reinvoke the web service with the answer and the other two arguments flipped and compare the result.

Testing this property we receive a (not unexpected) failure:

```
> proper:quickcheck(proper_output:prop_back_and_forth_is_equal(), 5).
!
```

```
Failed: After 1 test(s).
[[45,49,46,54,50,55,56,54,56,51,51,53,48,55,51,56,50,48,55,50,54,57,52,101,45

Shrinking ...(3 time(s))
[[48,46,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,48,101,43,48
false
```

Basically, it fails to produce the same result even if there is no change! A closer look with a few io:formats will reveal the problem : Floating point arithmetic. The floating point numbers created as arguments by proper are converted to Strings with some error in accuracy and the 2 subsequent calls include even more arithmetical errors. If we change the equality check to a small difference check (keeping in mind that erlsom represented those numbers as strings) we change the property to:

```
prop_back_and_forth_is_equal() ->
  ?FORALL([Value, From, To] = Args, generate_ChangeCookingUnit_1(),
    begin
      Result = call_ChangeCookingUnit(Args),
      Conv_val = answer_ChangeCookingUnit(Result),
      Result2 = call_ChangeCookingUnit([Conv_val, To, From]),
      Initial = answer_ChangeCookingUnit(Result2),
      F1 = list_to_float(Value),
      F2 = list_to_float(Initial),
      abs(F1 - F2) < 0.000001
    end).
```

And checking the property now:

```
8> c(proper_output).
{ok,proper_output}
9> proper:quickcheck(proper_output:prop_back_and_forth_is_equal()).
.....( 100 dots) .....
OK: Passed 100 test(s).
true
```