Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Αυτόματος Έλεγχος Υπηρεσιών Διαδικτύου με Χρήση Ιδιοτήτων

Διπλωματική Εργασία

του

Λεωνίδα Λαμπρόπουλου

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Ιούλιος 2012

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Αυτόματος Έλεγχος Υπηρεσιών Διαδικτύου με Χρήση Ιδιοτήτων

Διπλωματική Εργασία

του

Λεωνίδα Λαμπρόπουλου

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την $10^{η}$ Ιουλίου, 2012.

....	....	....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2012

....
Λεωνίδας Λαμπρόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Καθώς οι υπηρεσίες διαδικτύου (Web Services) αρχίζουν και αποτελούν ολοένα και βασικότερα τμήματα μοντέρνων διαδικτυακών συστημάτων λογισμικού, η ύπαρξη αυτόματων και εύχρηστων αλλά ταυτόχρονα και εκφραστικών προγραμμάτων ελέγχου για υπηρεσίες διαδικτύου καθίσταται όλο και σημαντικότερη. Η διπλωματική αυτή στοχεύει στον πλήρως αυτοματοποιημένο έλεγχο υπηρεσιών διαδικτύου: Ιδανικά, ο χρήστης απλά περιγράφει ιδιότητες που οι υπηρεσίες πρέπει να ικανοποιούν, με τη μορφή σχέσεων εισόδου-εξόδου, και το σύστημά μας αναλαμβάνει τα υπόλοιπα. Σε αυτή τη διπλωματική περιγράφουμε αναλυτικά όλα τα επιμέρους τμήματα του εργαλείου που φτιάξαμε: Πώς οι προδιαγραφές (WSDL) μιας υπηρεσίας διαδικτύου χρησιμοποιούνται για να παραχθούν με αυτόματο τρόπο γεννήτριες συντακτικά ορθών τυχαίων δεδομένων και ιδιότητες, οι οποίες μπορούν να δοθούν στο PropEr, ένα εργαλείο ελέγχου μέσω ιδιοτήτων, ώστε να κληθούν οι μέθοδοι της υπηρεσίας διαδικτύου και να ελεγχθεί η απόκρισή τους. Παρόλο που η διαδικασία είναι πλήρως αυτοματοποιημένη, το εργαλείο δίνει τη δυνατότητα στο χρήστη να αλλάξει το παραγόμενο αρχείο ελέγχου που περιέχει τις ιδιότητες και τις γεννήτριες ώστε να έχει μεγαλύτερο έλεγχο στην όλη διαδικασία και να μπορέσει να ελέγξει πιο στοχευμένα την λειτουργικότητα της υπηρεσίας διαδικτύου.

Λέξεις Κλειδιά

Erlang, Υπηρεσίες Διαδικτύου, Αυτόματος Έλεγχος, Έλεγχος μέσω Ιδιοτήτων, Έλεγχος βάσει προδιαγραφών, PropEr, WSDL

Abstract

With web services already being key ingredients of modern web systems, automatic and easy-to-use but at the same time powerful and expressive testing frameworks for web services are increasingly important. Our work aims at fully automatic testing of web services: ideally the user only specifies properties that the web service is expected to satisfy, in the form of input-output relations, and the system handles all the rest. In this thesis we present in detail all the components which form this system: how the WSDL specification of a web service is used to automatically create test case generators and properties that can be fed to PropEr, a property-based testing tool, to create structurally valid random test cases for its operations and check its responses. Although the process is fully automatic, our tool optionally allows the user to easily modify its output to either add semantic information to the generators or write properties that test for more involved functionality of the web services.

Keywords

Erlang, PropEr, Web Services, Testing, Property-based Testing, WSDL-based Testing, Automatic Testing

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τους γονείς μου και την αδερφή μου για την υποστήριξη που μου έχουν προσφέρει στηρίζοντας όλες μου τις επιλογές.

Χρωστάω ένα εξίσου μεγάλο ευχαριστώ στον επιβλέποντα καθηγητή μου, Κωστή Σαγώνα, γιατί η καθοδήγησή του και η αγάπη για τις Γλώσσες Προγραμματισμού που μου μετέδωσε με έκαναν αυτό που είμαι σήμερα.

Θα ήθελα επίσης να ευχαριστήσω όλους τους καθηγητές που με βοήθησαν στην πενταετή πορεία μου στη σχολή διαμορφώνοντας τα ενδιαφέροντά μου, και ιδιαίτερα τον Νίκο Παπασπύρου και την Ντόρα Βαρβαρίγου για τη βοήθεια που μου προσέφεραν και το ιδιαίτερο ενδιαφέρον που έδειξαν προς το πρόσωπό μου.

Τέλος, θέλω να ευχαριστήσω όλους τους φίλους μου που μου έχουν σταθεί τα τελευταία χρόνια σε καλές και κακές στιγμές.

Λεωνίδας Λαμπρόπουλος

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Web services are becoming essential components of modern web systems, while numerous tools to aid in their design and creation have been developed. At the same time, testing web services is an extremely slow and painful process, mainly due to that overly verbose nature of XML SOAP messages that makes writing test cases by hand an impractical option. Many of the existing tools help in speeding up the process, up to a point, but when web service functionality becomes involved, they fail to assist the tester in testing the web services in an easy and straightforward manner.

One approach that could be used to make the testing of involved web services easier is property-based testing (PBT). The idea of PBT is to express the properties that a program must satisfy in the form of input-output relations, and present the general structure of valid input messages in the form of generators while letting the system handle the creation of progressively more complex test cases in an attempt to find a counter-example for the property. Property-based testing is gaining popularity, especially in the community of functional programming languages where tools such as QuickCheck (for Haskell and Erlang) or PropEr (for Erlang) exist. When trying to apply property based testing to web services, the user faces a similar problem with manual testing approaches: the test case generators are cumbersome to write manually. This is where our tool comes in.

In this thesis, we present a tool designed to aid in property based testing of web services. By parsing the WSDL specification of the property, we can extract all the information needed to randomly create test data to provide as arguments in invoking web service operations, an idea that has already been thoroughly exploited by other researchers whose work is presented in a later in this thesis, in Chapter 5. At the same time, we can find information about the return types of such operations for further verification. Our tool uses all of the aforementioned information to fully automate black-box type testing of web services, by parsing the WSDL specification to provide generators and properties, feeding them to PropEr for execution.

Skipping ahead, in Listings 1.1 and 1.2 we show two small examples of using our tool to perform fully-automated testing on a web service that, at the time of this writing (June 2012), can be accessed freely on the web.

In both examples, what our tool did was to read the WSDL specification of a web service and based on the types and operations that were specified there, create a file with test case generators and some properties. In Listing 1.1 the tool used an automatically

```
1> proper_ws:type_check("http://www.webservicex.net/ConvertCooking.asmx?WSDL").
Testing property: prop_ChangeCookingUnit_responds
..... (100 dots) .....
OK: Passed 100 test(s).
true
```

Listing 1.1: Motivational response testing example

```
1> proper_ws:type_check("http://www.webservicex.net/ConvertCooking.asmx?WSDL").
Testing property: prop_ChangeCookingUnit_is_well_typed
..... (100 dots) .....
OK: Passed 100 test(s).
true
```

Listing 1.2: Motivational type checking example

created property named `prop_ChangeCookingUnit_responds` to invoke the web service operation "ChangeCookingUnit" 100 times and ensure that the web service responds with valid SOAP messages for all these tests. In Listing 1.2 our tool used a more complex property `prop_ChangeCookingUnit_is_well_typed`. This property, uses the automatically created generators to invoke the operation "ChangeCookingUnit" and parses its response to verify that its type is what is described in the WSDL specification. After compiling and loading this file, our tool performed 100 random tests on the same web service to check the result types. The web service responded with well-typed SOAP messages on all these tests as well.

Even though the above level of test automation is a great addition to the existing tools by providing means of validating the response of web services, our tool is designed for much more than automatically type checking web services. The file created by our tool is actually an Erlang source file, with properties and generators written as functions (and macros) in Erlang. Therefore, the user can easily modify generators to add semantic information to the created test data — for example by restricting a string valued field to a handful of meaningful names — or write his own properties using the ones provided as guides, to test for more involved functionality of web services. Using the full power and expressiveness of the Erlang programming language, a tester can describe arbitrarily complex input-output relations in order to test for more involved functionality of a web service without any knowledge and access to its source code.

The rest of this diploma thesis is organized as follows. In Chapter 2 we give a brief overview of property based testing and PropEr, we include basic information about Web Services and discuss the difficulties that arise when attempting to test them. We also describe a couple of Erlang tools, besides PropEr, that we used in our implementation. Chapter 3 is the main chapter of the thesis where we present our design decisions and the implementation of the new property based testing tool for web services. We describe in depth how the types in the WSDL specification are transformed to an intermediate Erlang representation and then converted to actual code. We also show some examples of our tool being used to find errors in web services both in automated and semi-automated ways. Then, we attempt to show how the tool can be used to test a stateful web system in Chapter 4. In Chapter 5 we present related work in this area. Finally, in Chapter 6 we present some concluding remarks and discuss possibilities of future work.

Chapter 2

Background

2.1 Property Based Testing and PropEr

Testing of software products is one of the key steps in the development life cycle. Manual testing has proved to be inefficient, time-consuming and accounts for an unreasonably large portion of an average project's cost. Many attempts to automate testing have appeared over the years, and one approach that seems to be subject to automation in a natural way was unit testing. In unit testing, where a software system individual components are tested as black box "units". The main process in unit testing can be broken into three steps, as described in the first PropEr publication [12]:

- Acquire a valid input
- Invoke the component with that input and capture its response
- Validate the response according to some input-output specification

Even though in most traditional unit testing methods only the second step is automated and the user is responsible for steps one and three for each individual test, property based testing introduces a new way for input generation and output validation. It has shown a great deal of promise in functional languages like Haskell and Erlang where the unit components are pure functions. To test such functions, the user describes the general structure of the inputs and a relation between input and output that must hold for all valid inputs, and the property based testing (PBT) tool handles all the rest. Specifically, a PBT tool generates random input that conforms to the aforementioned general structure, calls the function with that input as argument, captures the response and checks that the user specified property holds by examining the output. Should the property not hold, the PBT tries to isolate the reason of the fault between the random "noise" in the input using a process called shrinking.

PropEr is a property based testing tool for Erlang. Since its capabilities in testing pure functions and stateful systems have been described in full in various other publications [12, 11, 1], we will present here its parts that are directly related to this thesis along with some examples of their use.

2.1.1 Properties

Properties lie at the core of every PBT tool. In PropEr, a user can write properties using the full power of the Erlang language. Since properties specify input-output relations, the simplest of properties can be viewed as Boolean expressions containing variables and functions calls. The PropEr way of writing properties is wrapping such Boolean expressions with the following macro:

?FORALL(Vars, Generators, Prop) This macro receives three arguments. The first field (`Vars`) must be a variable name, a list of variables or a tuple of variables. In each case, the second field (`Generators`) contains the PropEr type, called a PropEr generator, for each of the listed variables in the same structure. Finally, the last field of the macro is a Boolean expression containing the variables in `Vars` as free variables. An example of a ?FORALL macro in practice can be found in Listing 2.2.

2.1.2 Generators

PropEr generators are ways of describing the types of variables for use in properties. Simple generators closely resemble Erlang types and type names. The ones we will be using appear in Table 2.1

Generator	Represented term
integer()	all integers
integer(<LO>, <HI>)	integers in the range <LO> - <HI>
range(<LO>,<HI>)	integers in tha range <LO> - <HI>
float()	all floats
float(<LO>, <HI>)	floats in the range <LO> - <HI>
{T1,T2,...,TN}	tuples of size N with elements of type T1,T2,...,TN
list(Type)	arbitrary length lists of type Type
vector(<Len>, Type)	fixed length lists of type Type with length <Len>
union([T1,T2,...,TN])	instances of T1,T2,...,TN chosen with equal probability
<Term>	the actual Erlang term <Term>

Table 2.1: Basic PropEr Generators

In addition to the basic types described in table 2.1, PropEr also gives us the option to combine different generators in order to produce more complex types. The PropEr way of doing this is with the following macro:

?LET(Vars, Generators, In) The first two fields are identical with the ones used in the ?FORALL macro. The difference is that the third field is not a Boolean expression but an expression containing variables from Vars that is (or a evaluates to) a type.

To better understand the use of a ?LET macro, consider the example in Listing 2.1 of a generator that produces lists of integer where the list has length in a given range.

This basically uses a randomly generated instance created by the range generator to populate the first argument of a vector generator.

```
ranged_int_list(Lo,Hi)->
  ?LET(Length, range(Lo,Hi), vector(Length, integer())).
```

Listing 2.1: Integer list generator example

2.1.3 Example of Use

To demonstrate the power of a property based testing tool we will borrow an example of the first PropEr publication. Assume we want to create a `delete/2` function that should take an integer X and a list L as arguments and return the list with the occurrences of X in it deleted. Its implementation is shown in Listing 2.2.

```
-module(mylists).
-export([delete/2]).
-include_lib("proper/include/proper.hrl").

%--------------------------------------------------------------------------------

delete(X, L) ->
  delete(X, L, []).

delete(_, [], Acc) ->
  lists:reverse(Acc);
delete(X, [X|Rest], Acc) ->
  lists:reverse(Acc) ++ Rest;
delete(X, [Y|Rest], Acc) ->
  delete(X, Rest, [Y|Acc]).

%--------------------------------------------------------------------------------

prop_delete_removes_every_x () ->
  ?FORALL({X,L}, {integer(),list(integer ())},
          not lists:member(X, delete(X,L))).
```

Listing 2.2: Delete implementation and property

The property is basically a Boolean expression saying that X is not a member of the list that results from the call to delete with X as its first argument, wrapped in a PropEr ?FORALL macro. To test our implementation with this property, we only need to run PropEr as shown in Listing 2.3.

In Listing 2.3 we see that PropEr run 64 successful tests before discovering a test case that does not satisfy the property. This example, also shows the need for shrinking. The first list that failed our property is a relatively large list of random integers that does not appear to have any special structure. After the shrinking, we are left with a simple list with two copies of the same element. This quickly reveals the fault in our implementation to be mishandling of duplicate values, a fact that could be hidden amongst random noise without shrinking the failed testcase.

```
1> proper:quickcheck(mylists:prop_delete_removes_every_x()).
.................................................................!
Failed: After 65 test(s).
{4,[12,-5,0,-15,50,29,10,4,-10,-42,5,-15,-30,24,41,-19,93,-6,8,4,51,16,5,-45,8]}

Shrinking ............(12 time(s))
{0,[0,0]}
false
```

Listing 2.3: Testing the delete property

2.2 Web Services

Over the past ten years, web systems designers have started to direct their attention towards service-oriented architectures in order to tackle the problem of integrating highly heterogeneous systems. By supporting the interoperability of inherently different systems developed under disparate platforms through the use of widely established standards, web services appear as the natural solution; a solution being adopted and pushed by numerous companies including Microsoft, IBM, Sun and Amazon.

During this time period, many definitions of what a web service is have appeared. One of the earliest definitions, given by the IBM Web Services architecture team as early as September 2000 defines them as “A collection of functions that are packaged as a single entity and published to the network for use by other programs”, focusing on the operational aspect. A Microsoft researcher, focusing on a developers point of view, described web services as “a very general model for building applications that can be implemented for any operating system that supports communication over the Internet and represent black-box functionality that can be reused without worrying about how the service is implemented”. A final and more concise definition was given by the main international standards organization, World Wide Web Consortium (W3C). According to W3C web services are defined as “a software system designed to support interoperable machine-to-machine interaction over a network”. All these definitions capture the essence of a web service as software components providing interfaces to other software components for network communication.

In order for this communication to take place in a truly cross-platform, unrestricted manner a neutral specification and description language has been adopted: XML. Using XML a number of standards has been developed and established to define data transfer, method invocation and publishing protocols. The most widely used standards are SOAP, WSDL and UDDI respectively. In the following few subsections we briefly present the above technologies and introduces key concepts used in the rest of this thesis.

2.2.1 XML

XML stands for Extensible Markup Language. It was designed by W3C to describe data, representing them in a flexible and customizable way while at the same time maintaining both human- and machine-readability. An XML document forms a tree-like structure with nodes and leaves that represent information. Listing 2.4 shows the general structure of an XML document and also how data is represented.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<thesis type="diploma thesis">
  <title>
    Automatic Random Testing of Function Properties from Specifications
  </title>
  <author>
    Manolis Papadakis
  </author>
  <year>
    2010
  </year>
  <school>
    National Technical University of Athens
  </school>
</thesis>
```

Listing 2.4: A simple XML document

An XML document optionally begins with a single line - the prologue - that can contain a declaration of the XML version used, encoding information and other document related information. In this case, the prologue specifies that this document conforms to the 1.0 XML specification and that the ISO-8859-1 character set is being used.

Every XML document, regardless of the existence of a prologue, must contain a single root element, that is the parent of all other elements in the document. In our example this root element is a `thesis` element that contains four children. Each child contains some extra information regarding the entity represented by the XML document, in this case a thesis. Finally, this example shows another way to store information via attributes. The starting tag of the `thesis` node contains an attribute declaring that the type of the thesis to be "diploma thesis".

An XML document with a correct syntax like the above is called "Well Formed". In addition to syntactical correctness however, one can define a list of legal elements that describe the structure of an accepted document. At first, this was done with Document Type Definitions (DTD), however this is gradually being replaced by XML Schema Definitions (XSD), which we will present in a following section.

2.2.2 SOAP

Due to the numerous advantages of XML, from being inherently platform independent because of its textual nature to providing the ability to create your own tags since it is an extensible language, it is the chosen format for web service communication. In order to achieve this independence for every web service, the Simple Object Access Protocol (SOAP) was developed.

SOAP is a protocol specification that uses XML in a standardized message format enabling machine-to-machine communication without any prior knowledge of the other machine and its architecture. The main part of this message format is called the envelope and it is an XML document. The envelope is the root node of the XML document that must contain a body, which contains all the information that will reach the final recipient. In the web services case, the body differs from request to response messages. A request body contains information needed to invoke a web service operation, like name, uri and parameters, while a response body contains information about the response value of the web service. In addition to the body, a SOAP message can contain a header node describing application specific information or a fault node indicating error messages.

To show an example of SOAP based communication with a web service, we use the example contained in the W3Schools tutorials [17]. Listings 2.5 and 2.6 are indicative examples of request and response SOAP messages.

```
<?xml version=”1.0”?>
<soap:Envelope
xmlns:soap=”http://www.w3.org/2001/12/soap-envelope”
soap:encodingStyle=”http://www.w3.org/2001/12/soap-encoding”>

<soap:Body xmlns:m=”http://www.example.org/stock”>
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Listing 2.5: A request SOAP message

We see that in order to invoke an operation of a web service called `GetStockPrice` that takes a single parameter `StockName`, we wrap the parameter ("IBM") in a StockName node, make it a child of the operation name node and then include it in the body of a SOAP envelope.

```
<?xml version=”1.0”?>
<soap:Envelope
xmlns:soap=”http://www.w3.org/2001/12/soap-envelope”
soap:encodingStyle=”http://www.w3.org/2001/12/soap-encoding”>

<soap:Body xmlns:m=”http://www.example.org/stock”>
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Listing 2.6: A response SOAP message

To process the response of such a web service we must look inside the response envelope body for our result. Here the result lies in a `Price` node, representing the actual price of the stock whose name we provided in the request message.

2.2.3 WSDL

The Web Service Definition Language (WSDL) is essentially an XML document with a predefined grammar describing a web service and mainly how to access the operations it exposes [6]. A WSDL document uses the following elements to describe a web service:

Types A data type definition container. WSDL can define types using a type system like XSD, which we will explain more in the following section.

Operation A description of a method the web service exposes. These operations are the main points of access of a web service.

Message A message element contains information about the data that can be passed in a single remote procedure call (RPC).

Port Type A port type is the element of a WSDL specification that associates operations with their respective input and output messages.

Binding A binding element describes the data format specification of a single port type.

Port A port element defines an endpoint combining binding elements to the unique resource identifiers (URI's) where they can be found.

Service The service element of a WSDL specification is simply a list of endpoints.

Using all of the above elements in an XML document, one can fully describe a web service, the methods that can be invoked, their respective locations and associated message and data types. In order to be able to generate structurally valid SOAP messages and invoke a web service operation, we must conform to the type system described in the specification. For our framework, we used the most commonly used WSDL type system which is also a W3C Recommendation, the XML Schema. In the following Section we explain some of its basic notions that we will refer to in later sections that describe our tool's implementation.

2.2.4 XSD

An XML Schema Definition (XSD) is basically an XML document that describes the legal structure of another XML document [8, 13]. In our case, inside a WSDL specification, an XSD describes the data types of the input and output messages for a web service's operations. Almost every type described in an XML Schema, with the exception of the `anyType` which is the top type in every XSD type system, is a restriction or an extension of another type. Primitive, commonly used datatypes such as integers and strings are already implemented to avoid rewriting boilerplate code and they can be used by a user to define custom simple or complex types for his web service. The rest of this section will be devoted to make the distinction between simple and complex types and show how they can be created by a user.

A simple type definition is always a restriction (the XSD equivalent of a subtype) upon the `anySimpleType` datatype. This datatype is the root of the hierarchy of type definitions for all simple types and can be described by any name that is a sequence of characters and can be mapped to practically any set of values. Every restriction on a simple type yields a new type that limits the value or the lexical space of its parent. The restrictions on a simple type are called facets and describe how the value space is restrained. For example, any real numbers that have a finite base 10 representation form the `decimal` type. This type is derived from the anySimpleType by limiting the lexical choice to strings that conform to the following pattern:

$$(+|-)?([0-9]+(.[0-9]*)?|.[0-9]+)$$

which is a regular expression that describes all known finite base 10 fractional or not numbers. In a similar fashion, this type can be further reduced to an integer (no fractional part) or have its value space constrained by a limit (e.g. a maximum value).

Complex type definitions are basically ways to group other types (simple or complex) together. There exist a great deal of ways to form these groupings of child elements, but the most used one is to include the child elements with their types inside the complex type declaration along with an indicator showing how these elements are grouped. There are three indicators in an XSD schema which are described below:

all This indicator shows that the child elements contained must all appear exactly once, but in any order. This is not used a lot in favor of the sequence indicator.

sequence This is the most widely used indicator. All the children elements must appear in the specified order, while duplicates are allowed with respect to attributes in the Schema.

choice This is another indicator that declares that exactly one of the child elements can be used to form this super type.

We close this section with an example showing an example of (a fraction of) an XSD schema that could be used to validate the sample XML document in Section 2.2.1.

```
<element name=”thesis”>
  <complexType>
    <sequence>
      <element name=”title” type=”string”/>
      <element name=”author” type=”string”/>
      <element name=”year” type=”year”/>
      <element name=”school” type=”string/>
    </sequence>
  </complexType>
</element>

<simpleType name=”school”>
  <restriction base=”integer”>
    <minInclusive value=”1837”>
  </restriction>
</simpleType>
```

Listing 2.7: An example of an XML Schema Definition

This listing, although it does not show the exact syntax of a valid XML Schema, is pretty close to one that could be used to describe the thesis document omitting only namespace information. In this listing we describe a complex type named thesis that contains four child elements that appear exactly once. Three of them are of string type, while the year element is a restriction on integers imposing a minimum value limit.

2.3 Web Service Testing

The increased attention web services have received has yielded a great number of tools that aid in creating new and combining existing web services with little effort and time. At the same time, however, little progress has been made in creating an effective and mature tool that correctly tackles the issue of testing web services, largely because of the inherent difficulty of the area.

First and foremost, the fact that service oriented architecture results in distributed, loosely coupled systems with reusable components from various sources, while viewed as a great advantage over more conventional techniques when it comes to creating a scalable and extensible system, is a major drawback when trying to maintain a consistent Quality of Service across various components. Since multiple components are combined in a single, large system, the integrator must be able to validate the individual services and before proceeding to test the entire application to reduce the introduced opportunities for failure.

In addition, most conventional testing techniques cannot apply to services. Testing at the graphical user level is rendered useless without additional support to monitor and validate messages between various intermediaries in order to locate and isolate problems. In addition, since services are practically just interfaces for the integrator (and the user), white box testing that used information extracted from the source code is impossible. Mutation testing is also not an option, since without access to the source code, there is little room in seeding it with errors.

Finally, in traditional applications, testing is done in relatively static environments, whereas web service applications live in a dynamic environment, subject to continuous change.

However, manual testing remains a tedious, time-consuming task that accounts for a high portion of the cost in software production. Therefore, despite the challenges posed in testing web services, researches in recent years have strived to propose a number of different approaches for the automation of testing of both functional and non-functional requirements.

2.3.1 Non Functional Testing

The term non functional testing of web services refers to the process of ensuring all the non functional requirements (NFR) of a web service. The following list provides some examples of NFRs.

Accessibility Verify the ability of anyone (user, integrator, provider) to access the application depending on access privileges.

Availability Verify that the web service has a high uptime, usually conforming to a service level agreement (SLA).

Performance Verify that system characteristics like response time and throughput are in accordance with the requirements.

Load Verify that the application can process a number of concurrent transactions at the same time.

The above NFRs introduce a more or less natural way of automated testing, and are amongst the few characteristics of web services which existing tools can successfully test usually by means of load and stress testing. However there exist other, equally important, NFRs like scalability that are not subject to testing.

2.3.2 Functional Testing

Functional testing refers to the process of ensuring that a web service does what it is supposed to do. The unit testing approach when applied to web services can be automated in the same natural way, where each component now is a single web service operation. The process of testing such an operation can be regarded as practically identical to testing unit components in traditional software. Basically, it is the same three-step process:

- Produce a structurally valid input for the operation
- Invoke the web service and capture its response
- Ensure that the response conforms to some input-output specification

Similar to when testing traditional software, the second step is the one that has already been fully automated. Complex test suites, like SoapUI, can simplify the life of testers by introducing easy ways to enter data and capture the web service output. However, manual data input and response verification still occurs.

One main difference in the field of web service testing, is that researchers realized early on that the first step can also be automated using only information provided by the web service, namely its WSDL specification. Without leaving the limits of black-box testing, one can create an arbitrary number of structurally valid inputs of web service operations. A number of tools have been proposed that follow this idea, and are described in length in Chapter 5. However, these tools leave it to the tester to verify each output, requiring still a significant amount of time to finish the testing. This problem of validating the response of a software component, also referred to as the “oracle problem”, is amongst the most challenging when it comes to black-box testing.

More recently another idea has emerged that pushes black-box testing of web services to its limit [20]. Since WSDL specifications contain information both about input and output messages in the contained XML Schema Definitions, one can use that information to typecheck the output of the web service, providing a small kind of automated oracle. This approach uses property based testing, as described in the previous section, to fully automate the three-step testing process without any user input.

While this kind of testing shows much promise, it is still limited by the view of web services as black boxes. As was quoted by a product manager of Compuware Corp named Mark Eshelby, “No matter how easy it is to invoke WSDL, if you don’t know what the object was supposed to do, I don’t believe you can test it”. He was advocating in favor of developers doing component level, white-box testing instead of users or integrators. This is where our tool comes in.

Using ideas from property based testing, we have created a tool that supports fully automatic black box type testing as described above. However, the true power of our tool does not lie in simply automating black box testing. Our tool provides the necessary support to the testers to write their own properties regarding a web service, without need to concern themselves with writing the generators, invoking the operation or parsing the output. Our tool provides the generators for the input messages and the functions to call the WSDL specification, deconstructs the output to its smallest units and all this in a out-of-the-box compilable Erlang source file. The methods and techniques we used to achieve this result is the main object of this thesis.

2.4 Some Erlang Tools

2.4.1 xmerl

Xmerl is an XML parser, included in the Erlang/OTP distribution [18]. It transforms any valid XML to a (rather verbose) Erlang structure containing all the information contained in the original XML document. In our framework, xmerl is used to parse the XSD Schema of the WSDL specification into an Erlang structure, in order to extract the typing information needed to create PropEr generators.

2.4.2 Yaws and Erlsom

Yaws is one of the most widely used Erlang HTTP web servers [19]. Yaws uses an XML parser called Erlsom to handle the encoding and decoding of SOAP messages, a parser module faster and more user friendly than the xmerl module of the Erlang distribution, imposing however a few additional limitations, such as requiring most data to be converted to strings. In our framework, Yaws is used at two different times: in the beginning, in order to extract all the supported SOAP operations from the WSDL specification, and during the actual testing phase, as an intermediary between PropEr and the web service, wrapping the data generated by PropEr in a valid SOAP structure, invoking a web service operation with the formed SOAP message, retrieving the result and returning it in the form of an Erlang tuple to PropEr for further analysis.

Outside of our framework, we also used Yaws to create and host web services in Erlang, which resulted in the examples of Chapter 3.

Chapter 3

PropEr WS Design and Implementation

In this chapter we will present the design and implementation of a tool that can handle automated property based testing of web services. First, we will briefly describe the framework as a whole and give an overview of the process of testing. Then, we will get into detail about the implementation and the techniques used in creating our tool.

3.1 PropEr WS Architecture

Figure 3.1 shows the architecture of our testing framework. Given a URI, the testing starts by obtaining the WSDL specification of the web service. This specification is then fed into two different Erlang tools, Yaws and xmerl, briefly described in Chapter 2. Using xmerl we extract all the type information associated with the WSDL specification, while using Yaws we extract needed information for all supported (SOAP) operations. These two pieces of information are then used to create a testing file (the default is `proper_ws_autogen`) with Erlang code that contains PropEr generators and properties ready for use. Then, as we will see, the user can (optionally) modify this auto-generated file to add his own properties

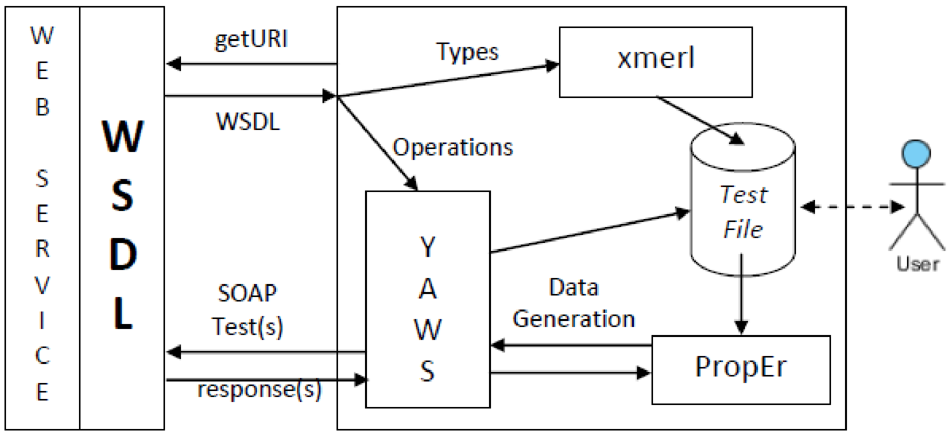

Figure 3.1: System architecture of our property-based testing framework.

or refine the generators. The testing file is then given as input to PropEr, which generates random test cases, invokes the web service (using Yaws as a SOAP wrapper) and then analyzes the result.

3.2 Intermediate Representation

In order to be able to generate PropEr generators and type-checking properties, we introduced an intermediate representation (IR) of the types described in the WSDL specification as Erlang tuples. This representation was created so that it holds all type information needed to create both the generators and the properties, while at the same time it allows for easier manipulation of all constraining facets of the XSD schema.

3.2.1 Simple Types

The first step towards creating the intermediate representation is mapping the primitive datatypes of an XSD schema into Erlang tuples. Table 3.1 shows this mapping for some of the most used basic types describing the format of the type that is expected by Yaws. Yaws expects most of the types as strings, with the exception of integers and booleans. An atom `erlsom_string` means the simple type must be converted to a string before it is used to invoke the web service. Similarly, `erlsom_bool` and `erlsom_int` mean the simple type must stay as a `boolean` or a `integer` respectively.

Simple Type	Erlang Intermediate Representation
boolean	`{erlsom_bool, bool, []}`
float	`{erlsom_string, float, {inf, inf}}`
double	`{erlsom_string, float, {inf, inf}}`
integer	`{erlsom_string, integer, {inf, inf}}`
nonPositiveInteger	`{erlsom_string, integer, {inf, 0}}`
negativeInteger	`{erlsom_string, integer, {inf, -1}}`
long	`{erlsom_string, integer, {-1 bsl 63, 1 bsl 63 -1}}`
int	`{erlsom_int, integer, {-1 bsl 31, 1 bsl 31 - 1}}`
short	`{erlsom_string, integer, {-1 bsl 15, 1 bsl 15 - 1}}`
byte	`{erlsom_string, integer, {-1 bsl 7, 1 bsl 7 - 1}}`
nonNegativeInteger	`{erlsom_string, integer, {0, inf}}`
positiveInteger	`{erlsom_string, integer, {1, inf}}`
unsignedLong	`{erlsom_string, integer, {0, 1 bsl 64 - 1}}`
unsignedInt	`{erlsom_string, integer, {0, 1 bsl 32 - 1}}`
unsignedShort	`{erlsom_string, integer, {0, 1 bsl 16 - 1}}`
unsignedByte	`{erlsom_string, integer, {0, 1 bsl 8 - 1}}`
string	`{list, {{range, 0, inf}, {erlsom_int, integer, {32, 127}}}}`

Table 3.1: Basic XSD primitive types mapped to our Erlang Intermediate Representation

The second element of the intermediate representation show in Table 3.1 is an atom describing the base type of the simple type, without regard for range constraints: integer, boolean, float or list. This atom will be used later on both to create PropEr generators and to create type-checking properties.

The third element of the tuple contains extra information about the simple type. For integers and floats, this extra information is basically range constraints: the minimum and maximum possible type of the value. For a list, we store more information in this representation: the possible values of its length and also the intermediate representation of the type of the lists' elements. In the above table we see that strings are represented as arbitrary length lists containing integers in the range 32 – 127, since strings in Erlang are implemented as integer lists. The added range constraint is used to ensure that the list contains only printable ASCII characters.

With the above representation, we can easily handle all facet constraints for a simple type in the XSD schema. Given a tuple and a range or length constraining facet — such as `minInclusive` or `maxLength` — we can construct a new tuple with the respective element altered. Therefore starting with the tuples described above for the primitive types, we can go through the list of constraining facets and get a final tuple of the same form that represents a type satisfying them. There is one specific facet that yields a different kind of tuple: `enumeration`. When such a facet is encountered, we replace the tuple with one entirely new one of the form: `{elements, Options}`, meaning that we can only use one of the `Options` as a valid argument.

3.2.2 Complex Types

Another important issue that needs addressing is combining simple data types into complex aggregates. First, we extract the name of the type from the WSDL specification. The name is an Erlang tuple. Its first element is an atom that represents the actual name of the type, the second element is a list of atoms that describes the path from the schema node of the WSDL specification down to the complex type, and the final element holds namespace information about the complex type. After creating the type's name tuple, we proceed to parse the elements that form the complex type and return their respective types in order in a list. Table 3.2 shows how to create a complex type's representation based on the name, the generators list and also the indicator that describes how to combine the generators.

Indicator	Erlang Intermediate Representation
all	`{TypeName, {tuple, Generators}}`
sequence	`{TypeName, {tuple, Generators}}`
choice	`{TypeName, {union, Generators}}`

Table 3.2: XSD indicators mapped to our Erlang Intermediate Representation

A choice indicator dictates that only one of the inner generators will be used as an argument. An all and a sequence indicator dictate that all of the inner generators must be used. More precisely, a sequence indicator requires all the inner generators in order, while an all indicator allows for mixed order in the SOAP message. However, we treat both in the same way, since Yaws does not yet allow for arbitrary ordering in the inner elements.

3.2.3 Example

In this section we will introduce now an example of a self-created web service that will also be used in later sections to demonstrate the use of our tool. Listing 3.1 shows the XSD of this web service.

```
<wsdl:types>
  <s:schema elementFormDefault=”qualified” targetNamespace=”http://foo/”>
    <s:element name=”MakeOrder”>
      <s:complexType>
        <s:sequence>
          <s:element minOccurs=”1” maxOccurs=”unbounded”
                     name=”Orders” type=”tns:SingleOrder” />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:complexType name=”SingleOrder”>
      <s:sequence>
        <s:element name=”Title” type=”tns:BookName”/>
        <s:element name=”Amount” type=”s:int”/>
      </s:sequence>
    </s:complexType>
    <s:simpleType name=”BookName”>
      <s:restriction base=”s:string”>
        <s:enumeration value=”Programming Erlang”/>
        <s:enumeration value=”Concurrent Programming in Erlang”/>
        <s:enumeration value=”Learn You Some Erlang for Great Good”/>
        <s:enumeration value=”Software for a Concurrent World”/>
        <s:enumeration value=”Erlang Programming”/>
        <s:enumeration value=”Thinking in Erlang”/>
        <s:enumeration value=”Functions + Messages + Concurrency = Erlang”/>
      </s:restriction>
    </s:simpleType>
    <s:element name=”MakeOrderResponse”>
      <s:complexType>
        <s:sequence>
          <s:element name=”MakeOrderResult” type=”s:double” />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</wsdl:types>
```

Listing 3.1: An XSD example

In this XSD schema there exist two elements, a complex and a simple type. The simple type `BookName` is a simple enumeration of string possibilities describing a book title. The complex type `SingleOrder` is a sequence of two inner elements, a `Title` of type `BookName` and an `Amount` of type integer. Since no minOccurs or maxOccurs attributes are present, exactly one of each type is expected to form the sequence. This complex type represents a single order of a number of copies of a book. The first element, `MakeOrder` is a non-empty unbounded list of `Orders` of type `SingleOrder`. This represents the entire order, different books in different amounts. Finally, the last element `MakeOrderResponse` is a sequence of a single element of type double, representing the web service's response: the total price of the order.

When our tool is used on the XSD schema of Listing 3.1, we get the intermediate Erlang tuples shown in Listing 3.2.

```
[{{’MakeOrderResponse’,[],’http://foo/’},
  {{’_xmerl_no_name_’,[anonymous,’MakeOrderResponse’],’’},
   {tuple,
       [{{’MakeOrderResult’,
              [anonymous,’MakeOrderResponse’],
              ’http://foo/’},
          {erlsom_string,float,{inf,inf}}}]}}},
 {{’BookName’,[],’http://foo/’},
  {elements,
      [”Programming Erlang”,”Concurrent Programming in Erlang”,
       ”Learn You Some Erlang for Great Good”,
       ”Software for a Concurrent World”,”Erlang Programming”,
       ”Thinking in Erlang”,
       ”Functions + Messages + Concurrency = Erlang”]}},
 {{’MakeOrder’,[],’http://foo/’},
  {{’_xmerl_no_name_’,[anonymous,’MakeOrder’],’’},
   {tuple,
       [{{’Orders’,
              [anonymous,’MakeOrder’],
              ’http://foo/’},
          {list,
              {{range,1,unbounded},
               {toplevel,
                   {simple_or_complex_Type,
                       {’SingleOrder’,[],
                           ’http://foo/’}}}}}}]}}},
 {{’SingleOrder’,[],’http://foo/’},
  {tuple,
      [{{’Title’,[’SingleOrder’],’http://foo/’},
        {toplevel,
            {simple_or_complex_Type,
                {’BookName’,[],’http://foo/’}}}},
       {{’Amount’,[’SingleOrder’],’http://foo/’},
        {erlsom_int,integer,{-2147483648,2147483647}}}]}}]
```

Listing 3.2: Intermediate representation for example of Listing 3.1

3.3 Creating PropEr Generators

The back-end of our implementation is responsible for converting the intermediate representation to either a PropEr generator or a property. In this section we look at the details of converting the IR Erlang tuples to actual Erlang code.

The first step in our conversion is figuring out exactly which of the tuples we have constructed we need to output in the form of generators, so that no extraneous or unused code is included in the output file. To that end, we get all the input message types of the supported web service operations, which are toplevel (or imported) element nodes in the XSD schema. We add their representation to a list of tuples that need handling and proceed to the actual creation of the generators. Every time we encounter a `toplevel` tuple we add it to the list of the tuples that still require handling, and recursively go through this list taking care not to handle the same tuple twice. Since the number of tuples is finite this process reaches a fixed point when all the necessary generators have been successfully created.

The intermediate representation for simple types can be easily mapped to Erlang code. In Table 3.3 we see this mapping for the three primitive types:

Erlang Intermediate Representation	Code
`{_, integer, {inf, inf}}`	`integer()`
`{_, integer, {Min, Max}}`	`integer(Min, Max)`
`{_, float, {inf, inf}}`	`float()`
`{_, float, {Min, Max}}`	`float(Min, Max)`
`{_, bool, _}`	`union([true, false])`

Table 3.3: Intermediate Representations of simple types mapped to Erlang Code

The first argument is used to determine whether or not to wrap the resulting code in a ?LET macro like this:

```
?LET(Gen, Code, %TYPE%_to_list(Gen))
```

which converts the generated instance to an Erlang string.

When a toplevel tuple is encountered, the generator that is created is simply a function call to the actual generator that will be (or has already been) created at some time during our toplevel element handling.

Finally, to create generators for list tuples, we create the generator code for the list element type and then wrap this code in a ?LET macro to create a vector of a specified range.

Complex types, however, are an entirely more complex matter. To use Yaws in order to invoke the web services — and also for better code readability — we populate the record structures provided by erlsom in an .hrl (Erlang header) file when parsing the WSDL specification. To be able to use these records however, we must strictly comply to the erlsom naming conventions for sequence and choice complex types.

Sequences are relatively easy to handle. We iterate through the list of the inner IR tuples, output their code as appropriately named generators and use these generators to populate each field of the record inside a ?LET macro. To create a choice generator on the other hand we first create generators for each one of the possibilities, wrap this generator in a ?LET macro that populates an erlsom-named record, and use all these wrapped macros in another generator along with a `union` PropEr generator to choose one and populate the `choice` field of yet another record.

The above process will be better understood viewing the examples that follow. In Listing 3.3 we show the generators produced for the example web service described in Section 3.2.3. The Erlang records used in that above code excerpt exist in an erlsom-created .hrl file, that can be seen in Listing 3.4.

The Listing 3.5 depicts an XSD file which describes a simple choice between an integer and a double. The code generated by our tool can be found in Listing 3.6.

We can see that our tool created a generator for each of the two types (`generate_Choice_foo/0` and `generate_Choice_bar/0`), then each of these generators was used inside another

```
generate_MakeOrder_Orders() ->
  ?LET(
    Len,
    range(1,inf),
    vector(Len, generate_SingleOrder())
        ).

generate_MakeOrder() ->
  ?LET(
    Pr_MakeOrder_Orders,
    generate_MakeOrder_Orders(),
    [#'p:MakeOrder'{'Orders' = Pr_MakeOrder_Orders}]
  ).

generate_SingleOrder_Title() ->
  generate_BookName().

generate_SingleOrder_Amount() ->
  integer(-2147483648,2147483647).

generate_SingleOrder() ->
  ?LET(
    {Pr_SingleOrder_Title, Pr_SingleOrder_Amount},
    {generate_SingleOrder_Title(), generate_SingleOrder_Amount()},
    [#'p:SingleOrder'{
        'Title' = Pr_SingleOrder_Title,
        'Amount' = Pr_SingleOrder_Amount
      }
    ]
  ).

generate_BookName()->
  elements(["Programming Erlang","Concurrent Programming in Erlang",
 "Learn You Some Erlang for Great Good","Software for a Concurrent World",
 "Erlang Programming","Thinking in Erlang",
 "Functions + Messages + Concurrency = Erlang"]).
```

Listing 3.3: PropEr generators for example in Listing 3.1

```erlang
%% HRL file generated by ERLSOM
%%
%% It is possible to change the name of the record fields.
%%
%% It is possible to add default values, but be aware that these will
%% only be used when *writing* an xml document.

-record('p:MakeOrder', {anyAttribs, 'Orders'}).
-record('p:MakeOrderResponse', {anyAttribs, 'MakeOrderResult'}).
-record('p:SingleOrder', {anyAttribs, 'Title', 'Amount'}).
-record('soap:Body', {anyAttribs, choice}).
-record('soap:Envelope', {anyAttribs, 'Header', 'Body', choice}).
-record('soap:Fault', {anyAttribs, 'faultcode', 'faultstring',
                               'faultactor', 'detail'}).
-record('soap:Header', {anyAttribs, choice}).
-record('soap:detail', {anyAttribs, choice}).
```

Listing 3.4: Auto-generated Erlang Library file

```xml
<s:schema elementFormDefault="qualified" targetNamespace="http://foo/">
  <s:element name="Choice">
    <s:complexType>
      <s:choice>
        <s:element name="foo" type="s:int"/>
        <s:element name="bar" type="s:double"/>
      </s:choice>
    </s:complexType>
  </s:element>
</s:schema>
```

Listing 3.5: An XSD example containing a simple choice

```

generate_Choice_foo() ->
  integer(-2147483648,2147483647).

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

generate_choice_Choice_foo() ->
  ?LET(
    Pr_Choice_foo,
    generate_Choice_foo(),
    #'p:Choice-foo'{'foo'=Pr_Choice_foo}
  ).

generate_choice_Choice_bar() ->
  ?LET(
    Pr_Choice_bar,
    generate_Choice_bar(),
    #'p:Choice-bar'{'bar'=Pr_Choice_bar}
  ).

generate_Choice() ->
  ?LET(
    Choice,
    union([
      generate_choice_Choice_bar(),
      generate_choice_Choice_foo()
    ]),
    #'p:Choice'{choice=Choice}
  ).
```

Listing 3.6: Generators for example shown in Listing 3.5

wrapper to populate an appropriately named (NameSpace:ElementName-FieldName) Erlang record and finally the generator for the actual toplevel element `generate_Choice/0` that uses the wrapper generators inside a union PropEr generator.

3.4 Creating Type-Checking Properties

Our tool creates two kinds of properties automatically: one that checks that a single operation responds with valid SOAP messages for various random inputs and one that checks the types of the elements inside these SOAP messages.

The response properties are created entirely without requiring information from the intermediate representation. All that is required is the name of the operation for which the property is created. Knowing this name we simply output a code stub based on the one shown in Listing 3.7.

```
prop_" ++ Op_name ++ "_responds() ->
  ?FORALL(Args, " ++ "generate_" ++ Op_name ++ "(),
          case call_" ++ Op_name ++ "(Args) of
            {ok, _Attribs, [#'soap:Fault'{}]} -> false;
            {ok, _Attribs, _Result_record} -> true;
            _ -> false
          end).
```

Listing 3.7: Stub for response testing

All this property does is use the generator that is created for the operation to generate random arguments, use these arguments to invoke the web service and finally pattern match on the results returned. If the resulting tuple is not tagged by ok then the system concludes that some error has occurred and the property fails. Otherwise we check the record returned by Yaws: if it contains a `soap:Fault` record, the property fails. In any other case, we (conservatively) pass the testcase and let PropEr proceed to create even more complex structurally valid testcases.

The automatically created type-checking properties on the other hand are an entirely different matter. The process of creating them is largely similar to the creation of the PropEr generators. We use the exact same technique to reach a fixed point and determine exactly which IR tuples need to be converted to actual Erlang code. We take extra care in creating user-readable type checkers, because the functions that check the code also show to the user how to deconstruct the answer records to their elements. Since our main goal is property-based testing of web services, the user should only be concerned with writing the property and not with implementation specific details regarding SOAP, Yaws, etc.

We introduced a few invariants to successfully handle all cases without a lot of code reuse. First, each type-checking function receives a single argument named `X`. This argument is assumed to be a simple type in the form returned by the web service, extracted from the erlsom records and is afterwards converted to a correctly typed Erlang variable. During this process, if an exception is raised — meaning `X` is incorrectly typed — the property fails. Finally, in order to achieve a common way to handle range checks for numeric values, we create Erlang code that returns a string `true` if both bounds are infinite or comparisons between `Value` and the range limits contained in the intermediate representation.

To demonstrate how simple types are type checked we include the type-checking functions created by our tool for the example in Section 3.2.3.

```erlang
typecheck_MakeOrderResult(X) ->
  try erlang:list_to_float(X) of
    _Value ->
      true
  catch _:_ ->
    false
  end.
```

Listing 3.8: Bottom Level Typechecking Code

```erlang
typecheck_deleteReturn(X) ->
  X =:= undefined orelse
  (is_list(X) andalso
  lists:all(
    fun (X) ->
      is_integer(X) andalso
      begin
        Value = X,
        2147483647 >= Value andalso Value >= -2147483648
      end
    end,
    X)).
```

Listing 3.9: Bottom Level Typechecking Code for Lists

In Listing 3.8 we see how we typecheck a float. Mainly we attempt to convert its string representation (since erlsom handles all floats as strings). If this succeeds we check that this value is a number and (if needed) we check the range constraints. An example of a type cheking function when the argument is a list is shown in Listing 3.9 When type checking lists, we check that the variable X is actually a list with the same built in function is_list and then we use a predicate similar to this inside a `lists:all/2` higher order function.

Complex types are handled differently. The function headers that do the checking still receive a single argument, but this time we unwrap the record to its elements “the Erlang way” and typecheck each field using the functions created before. The Listing 3.10 shows the checker for the actual element of our example.

```erlang
check_MakeOrderResponse(
  #'p:MakeOrderResponse'{
    'MakeOrderResult' = Pr_MakeOrderResult
      }
  ) ->
    typecheck_MakeOrderResult(Pr_MakeOrderResult).
```

Listing 3.10: Top Level TypeChecking Code

The final stage in creating the type checking property is to output the actual property in a way similar to the response property. The only difference is that we need not only the operations' input message type, but also the output one as shown in Listing 3.11.

```
prop_OPERATION_NAME_is_well_typed() ->
  ?FORALL(Args, ” ++ ”generate_” ++ Op_in ++ ”(),
          case call_OPERATION_NAME(Args) of
            {ok, _Attribs, [#’soap:Fault’{}]} ->
              false;
            {ok, _Attribs, [Result_record]} ->
              check_OPERATION_RETURN_NAME(Result_record);
            _ -> false
          end).
```

Listing 3.11: Type Checking Property

In Listing 3.11 we can see that the difference between checking return type correctnes and response testing lies in the second case clause of the pattern matching where we don't always return true, but delegate the check to an the already created type checking function.

3.5 Using PropEr WS

In this Section, we demonstrate how our tool can be used to test web services both with and without user input.

3.5.1 Fully Automatic Response Testing

Our tool can automatically handle two types of testing without any user input: response testing and type testing. Using the XSD schema described in Section 3.2.3, we create two different (deliberately) faulty implementations of web services.

Firstly, the “response” web service, whose code is shown in Listing 3.12.

The fault in this implementation lies not only in the fact that we “forgot” the final enumeration value, but that we include no error handling when trying to extract an item's price. Trying to test this web service with our tool we get the output that can be seen in Listing 3.13.

The first command is a call to our tool to generate a test file for the web service we feed as an argument. Since no other arguments are given, the default output file is used `proper_ws_autogen.erl`. In the second command we compile the output file, and in the last one we use PropEr to check the response property created by our tool. As expected, the web service crashes after a few tests, and PropEr manages to isolate the error in a single test ordering 0 copies of the unhandled book title. The output of Listing 3.13 outputs strings as integer lists. Even though in this case we would like a string representation, PropEr has no way of distinguishing between a string and an integer list when it comes to printing them - therefore, it conservatively prints everything as an integer list, to account for examples such as the one in Listing 2.3, where we would like to avoid a list like `[0,0]` be printed as a string. Listing 3.14 shows the real structure of the failed testcases, taken simply by copy-pasting the structure in the Erlang shell.

```erlang
-module(response).

-export([handler/4]).

-include("response.hrl").

-define(AVAILABLE_BOOKS,
   [{"Programming Erlang", 1.00},
    {"Concurrent Programming in Erlang", 0.42},
    {"Learn You Some Erlang for Great Good", 1.42},
    {"Software for a Concurrent World", 2.42},
    {"Erlang Programming", 3.00},
    {"Thinking in Erlang", 3.42}]).

handler(_Header, [#'p:MakeOrder'{'Orders'=Orders}], _Action, _SessionValue) ->
  {ok, undefined, [get_response(Orders)]}.

get_price(Title) ->
  {Title, Price} = lists:keyfind(Title, 1, ?AVAILABLE_BOOKS),
  Price.

get_response(Orders) ->
  TotalPrice = lists:foldl(
    fun (#'p:SingleOrder'{'Title'=Title,'Amount'=Amount}, Acc) ->
      Acc + Amount * get_price(Title)
    end, 0.0, Orders),
  Result = float_to_list(TotalPrice),
  #'p:MakeOrderResponse'{anyAttribs = [], 'MakeOrderResult'=Result}.
```

Listing 3.12: Implementation of a simple web service

```
1> proper_ws:generate("file://tmp/response.wsdl").
ok
2> c(proper_ws_autogen).
{ok,proper_ws_autogen}
3> proper:quickcheck(proper_ws_autogen:prop_MakeOrder_responds()).
....!
Failed: After 5 test(s).
[{'p:MakeOrder',undefined,[[{'p:SingleOrder',undefined,[67,111,110,99,117,114,
114,101,110,116,32,80,114,111,103,114,97,109,109,105,110,103,32,105,110,32,69,
114,108,97,110,103],0}],[{'p:SingleOrder',undefined,[67,111,110,99,117,114,114,
101,110,116,32,80,114,111,103,114,97,109,109,105,110,103,32,105,110,32,69,114,
108,97,110,103],1}],[{'p:SingleOrder',undefined,[70,117,110,99,116,105,111,110,
115,32,43,32,77,101,115,115,97,103,101,115,32,43,32,67,111,110,99,117,114,114,
101,110,99,121,32,61,32,69,114,108,97,110,103],0}],[{'p:SingleOrder',undefined,
[76,101,97,114,110,32,89,111,117,32,83,111,109,101,32,69,114,108,97,110,103,32,
102,111,114,32,71,114,101,97,116,32,71,111,111,100],-1}]]}]

Shrinking .....(5 time(s))
[{'p:MakeOrder',undefined,[[{'p:SingleOrder',undefined,[70,117,110,99,116,105,
111,110,115,32,43,32,77,101,115,115,97,103,101,115,32,43,32,67,111,110,99,117,
114,114,101,110,99,121,32,61,32,69,114,108,97,110,103],0}]]}]
false
```

Listing 3.13: Using PropErWS on the code of Listing 3.12

```
1> proper_ws:generate(”file://tmp/response.wsdl”).
ok
2> c(proper_ws_autogen).
{ok,proper_ws_autogen}
3> proper:quickcheck(proper_ws_autogen:prop_MakeOrder_responds()).
....!
Failed: After 5 test(s).
[{'p:MakeOrder',undefined,
                [[{'p:SingleOrder',undefined,
                                   ”Concurrent Programming in Erlang”,0}],
                 [{'p:SingleOrder',undefined,
                                   ”Concurrent Programming in Erlang”,1}],
                 [{'p:SingleOrder',undefined,
                                   ”Functions + Messages + Concurrency = Erlang”,
                                   0}],
                 [{'p:SingleOrder',undefined,
                                   ”Learn You Some Erlang for Great Good”,
                                   -1}]]}]

Shrinking .....(5 time(s))
[{'p:MakeOrder',undefined,
                [[{'p:SingleOrder',undefined,
                                   ”Functions + Messages + Concurrency = Erlang”,
                                   0}]]}]
false
```

Listing 3.14: Real failing testcase structure

Henceforth, we will convert the integer lists to strings where needed, without a specific mention of this fact.

3.5.2 Fully Automatic Type Checking

In addition to automatic response testing we can successfully type check a web service's responses fully automatically. To demonstrate this use of our tool, we created another faulty web service: “semantic”. Its implementation is shown in Listing 3.15.

In this implementation, we also “forget” the final book, however now we include some sort of error handling attempting to return an error string when a book is not found. Since yaws expects most arguments in string form, it will not detect our error and allow our web service to respond with a string instead of a double representing the price!

Our tool can be used to find this kind of error, as we can see in Listing 3.16.

As we can see the web service successfully passes 100 tests when tested with the automatic response property, however is quickly discover to be faulty when we use the type-checking property.

3.5.3 Property Based Testing

As mentioned before, the real power of our tool is providing all the necessary support for a user to do property based testing on a web service. To demonstrate how easy it is to use

```
-module(semantic).

-export([handler/4]).

-include("semantic.hrl").

-define(AVAILABLE_BOOKS,
   [{"Programming Erlang", 1.00},
    {"Concurrent Programming in Erlang", 0.42},
    {"Learn You Some Erlang for Great Good", 1.42},
    {"Software for a Concurrent World", 2.42},
    {"Erlang Programming", 3.00},
    {"Thinking in Erlang", 3.42}]).

handler(_Header, [#'p:MakeOrder'{'Orders'=Orders}], _Action, _SessionValue) ->
  {ok, undefined, [get_response(Orders)]}.

get_new_acc(_, {error, _Reason} = Error) ->
  Error;
get_new_acc(#'p:SingleOrder'{'Title'=Title, 'Amount'=Amount}, {ok, AccPrice}) ->
  case lists:keyfind(Title, 1, ?AVAILABLE_BOOKS) of
    {Title, Price} ->
      AccFloat = list_to_float(AccPrice),
      {ok, float_to_list(AccFloat + Price * Amount)};
    false ->
      {error, "Book Not Found"}
  end.

get_response(Orders) ->
  {_, Result} = lists:foldl(
    fun (Order, Acc) -> get_new_acc(Order, Acc) end, {ok, "0.0"}, Orders),
  #'p:MakeOrderResponse'{anyAttribs = [], 'MakeOrderResult'=Result}.
```

Listing 3.15: A web service with error handling

```
1> proper_ws:generate(”file://tmp/semantic.wsdl”).
ok
2> c(proper_ws_autogen).
{ok,proper_ws_autogen}
3> proper:quickcheck(proper_ws_autogen:prop_MakeOrder_responds()).
....(100 dots)....
OK: Passed 100 test(s).
true
4> proper:quickcheck(proper_ws_autogen:prop_MakeOrder_is_well_typed()).
....!
Failed: After 5 test(s).
An exception was raised: error:badarg.
Stacktrace: [{erlang,list_to_float,[”Book Not Found”],[]}].
[{'p:MakeOrder',undefined,
                [[{'p:SingleOrder',
                      undefined,
                      ”Functions + Messages + Concurrency = Erlang”,
                      -6}],
                 [{'p:SingleOrder',
                      undefined,
                      ”Learn You Some Erlang for Great Good”,
                      -1}],
                 [{'p:SingleOrder',undefined,”Thinking in Erlang”,1}],
                 [{'p:SingleOrder',
                      undefined,
                      ”Learn You Some Erlang for Great Good”,
                      -1}],
                 [{'p:SingleOrder',undefined,”Programming Erlang”,0}],
                 [{'p:SingleOrder',
                      undefined,
                      ”Functions + Messages + Concurrency = Erlang”,
                      0}],
                 [{'p:SingleOrder',undefined,”Programming Erlang”,0}]]}]



Shrinking ..(2 time(s))
[{'p:MakeOrder',undefined,
                [[{'p:SingleOrder',
                      undefined,
                      ”Functions + Messages + Concurrency = Erlang”,
                      0}]]}]
false
```

Listing 3.16: Using PropErWS on 3.15

the output of our tool we translated the delete example of Listing 2.2 to a web service.

To translate it, we created a WSDL file whose contained XSD schema is shown in Listing 3.17, and implemented the web service as shown in Listing 3.18. We can see that the code that handles the main functionality has stayed exactly the same, and we just wrap the arguments and results appropriately to handle the conversion to a web service.

```
<schema elementFormDefault="qualified" targetNamespace="http://tests"
        xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="delete">
    <complexType>
     <sequence>
       <element name="list" type="xsd:int"
                minOccurs="1" maxOccurs="unbounded"/>
       <element name="x" type="xsd:int"/>
     </sequence>
    </complexType>
   </element>
   <element name="deleteResponse">
    <complexType>
     <sequence>
       <element name="deleteReturn" type="xsd:int"
                minOccurs="0" maxOccurs="unbounded"/>
     </sequence>
    </complexType>
   </element>
  </schema>
```

Listing 3.17: XSD file of delete example

Afterwards, we used our tool to create generators and properties for testing this service. In Section 2.1.3 we needed to create a ?FORALL property and add a `lists:member` predicate to form a boolean expression. To demonstrate how easy it is to use the created file for property based testing, we modified the typechecking property and functions in place for direct comparison.

The modified parts of the file is shown in Listing 3.19. We can see the following changes:

- In the `typecheck_deleteReturn/1` function we added an argument (Elem) representing the element we wanted to delete and replaced the entire `lists:all/2` predicate with the `lists:member/2` predicate.
- In the `check_deleteResponse/1` function we just forward the extra argument needed.
- In the property we change the name, use the generator to copy-paste the record that `generate_delete/0` creates so that we can extract the 'x' variable and finally feed the check function with it.

That's all! Obviously, we could also use the typechecking property as a stub to create numerous other properties. To see property based testing of web services in action, we just compile the modified file and run the property. The output of the Erlang shell is shown in Listing 3.20.

```
-module(myDelete).
-export([handler/4]).

-include("myDelete.hrl").

handler(_Header,
        [#'p:delete'{list = List, x = X}],
        _Action,
        _SessionValue) ->
  {ok, undefined, get_response(List, X)}.

delete(X, L) ->
  delete(X, L, []).

delete(_, [], Acc) ->
  lists:reverse(Acc);
delete(X, [X|Rest], Acc) ->
  lists:reverse(Acc) ++ Rest;
delete(X, [Y|Rest], Acc) ->
  delete(X, Rest, [Y|Acc]).

get_response(List, X) ->
  [#'p:deleteResponse'{anyAttribs = [], deleteReturn = delete(X, List)}].
```

Listing 3.18: Implementation of delete example as a web service

```
%typecheck_deleteReturn(X) ->
typecheck_deleteReturn(Elem,X) ->
  X =:= undefined orelse
  (is_list(X) andalso
    not lists:member(Elem,X)).
%  lists:all(
%    fun (X) ->
%  is_integer(X) andalso
%  begin
%    Value = X,
%   2147483647 >= Value andalso Value >= -2147483648
%  end
%  end, X)).

%check_deleteResponse(
check_deleteResponse(X,
  #'p:deleteResponse'{
    'deleteReturn' = Pr_deleteReturn
      }
  ) ->
        %typecheck_deleteReturn(Pr_deleteReturn).
        typecheck_deleteReturn(X, Pr_deleteReturn).

%prop_delete_is_well_typed() ->
prop_delete_removes_every_x() ->
%  ?FORALL(Args, generate_delete(),
  ?FORALL([#'p:delete'{'x' = Pr_delete_x}]=Args, generate_delete(),
          case call_delete(Args) of
            {ok, _Attribs, [#'soap:Fault'{}]} ->
              false;
            {ok, _Attribs, [Result_record]} ->
              %check_deleteResponse(Result_record);
              check_deleteResponse(Pr_delete_x, Result_record);
            _ -> false
          end).
```

Listing 3.19: Modified parts of output file

```
1> c(proper_ws_autogen).
{ok,proper_ws_autogen}
2> proper:quickcheck(proper_ws_autogen:prop_delete_removes_every_x()).
....!
Failed: After 5 test(s).
[{'p:delete',undefined,[-5,0,-1,0,1,1,-2,1,-1],0}]

Shrinking .....(5 time(s))
[{'p:delete',undefined,[0,0],0}]
false
```

Listing 3.20: Testing the web service

Chapter 4

Towards Property-Based Testing of Stateful Web Services

One of the most powerful features of PropEr is its ability to test stateful systems. Some great tutorials on the subject were written by Eirini Arvaniti [1] and are located on the PropEr website [14]. In this chapter we will present an application of our tool in testing a stateful web service. Firstly, we will present the implementation of the actual web service in Java. Then we will show how our tool's output can be modified to add state information. Finally, we will actually use our modified Erlang code to test the service.

4.1 Web Service Implementation

The basic web service example we developed for this chapter is a simple authentication service. The idea is we have a basic pool of usernames with their associated passwords, and upon successful login we provide an authentication token. For simplicity purposes, we don't encrypt our data and information when communicating, since we just want to examine the potential in property based testing of web systems.

Our web service provides the following four operations:

login This method requires two strings, a name and a password, and upon successful authentication returns an integer token. Its implementation can be found in Listing 4.1, while the XSD types associated can be viewed in Listing 4.2.

authenticate This method receives an integer and returns a boolean value that is true if this integer represents an actual authentication token. Implementation and XSD types can be found in Listings 4.3 and 4.4.

logout This operation receives an authentication token and invalidates it. Returns a boolean value according to the success of the operation. The source code associated with this operation is contained in Listings 4.5 and 4.6.

getUsername Finally, this operation returns the username associated with a specific authentication token or an empty string otherwise. The related implementation details can be viewed in Listings 4.7 and 4.8.

```java
public int login(String name, String password){
  Pair<String,String> p = new Pair<String,String>(name,password);
  for(Pair<String,String> pass : passwords){
    if (pass.first.equals(p.first) && pass.second.equals(p.second)){
      Random r = new Random();
      int id = r.nextInt(10000);
      while(authenticate(id)) id = r.nextInt(10000);
      logged.add(new Pair<String, Integer>(name, id));
      return id;
    }
  }
  return -1;
}
```

Listing 4.1: Login source code

```xml
<element name=”login”>
 <complexType>
  <sequence>
   <element name=”name” type=”xsd:string”/>
   <element name=”password” type=”xsd:string”/>
  </sequence>
 </complexType>
</element>
<element name=”loginResponse”>
 <complexType>
  <sequence>
   <element name=”loginReturn” type=”xsd:int”/>
  </sequence>
 </complexType>
</element>
```

Listing 4.2: Login XSD types

```java
public boolean authenticate(int id){
  return !getUsername(id).equals(””);
}
```

Listing 4.3: Authenticate source code

```
<element name=”authenticate”>
  <complexType>
   <sequence>
    <element name=”id” type=”xsd:int”/>
   </sequence>
  </complexType>
 </element>
 <element name=”authenticateResponse”>
  <complexType>
   <sequence>
    <element name=”authenticateReturn” type=”xsd:boolean”/>
   </sequence>
  </complexType>
 </element>
```

Listing 4.4: Authenticate XSD types

```
public boolean logout(int id){
  String user = getUsername(id);
  if (user.equals(””)) {
    return false;
  }
  for (Pair<String,Integer> p : logged){
    if (p.getFirst().equals(user)){
      logged.remove(p);
      return true;
    }
  }
  return false;
}
```

Listing 4.5: Logout source code

```
<element name=”logout”>
  <complexType>
   <sequence>
    <element name=”id” type=”xsd:int”/>
   </sequence>
  </complexType>
 </element>
 <element name=”logoutResponse”>
  <complexType>
   <sequence>
    <element name=”logoutReturn” type=”xsd:boolean”/>
   </sequence>
  </complexType>
 </element>
```

Listing 4.6: Logout XSD types

```java
public String getUsername(int id){
  for(Pair<String,Integer> p : logged){
    if (p.getSecond() == id){
      return p.getFirst();
    }
  }
  return ””;
}
```

Listing 4.7: GetUsername source code

```xml
<element name=”getUsername”>
 <complexType>
  <sequence>
   <element name=”id” type=”xsd:int”/>
  </sequence>
 </complexType>
</element>
<element name=”getUsernameResponse”>
 <complexType>
  <sequence>
   <element name=”getUsernameReturn” type=”xsd:string”/>
  </sequence>
 </complexType>
</element>
```

Listing 4.8: GetUsername XSD types

4.2 Stateful Testing with PropEr

In order to test the web service described in the previous section, we will need to use the stateful testing part of PropEr. The idea behind testing a stateful system using PropEr is to create a model of the system under test, in order to simulate the state that is not directly accessible from the API of the system [9, 7]. Then, we generate test cases in the form of *symbolic* API calls, which facilitates easier shrinking and also allows for repeatable testcases. Since the PropEr terminology is vast, we will present here more parts of PropEr not described in Chapter 2 that we will be needing in later sections.

Symbolic commands

{call, Module, Fun, Args} This tuple represents a symbolic function call. When the generated symbolic testcases are executed, any such symbolic call is converted to an actual call to the function `Fun` in module `Module` with `Args` as arguments.

{set, var, N, Command} This is a symbolic *command*. This binds the result of `Command`, which must be a symbolic function call, to the symbolic variable `{var, N}` where `N` is a unique integer identifier for the symbolic variable. The generated testcases are actually sequences of this kind of commands, binding each result of an API call to a different symbolic variable that can be used in the following commands.

More PropEr macros

?TRAPEXIT(Prop) This macro makes sure that if the contained property spawns and links to a process that crashes, PropEr will treat it as a test failure instead of crashing as well.

?WHENFAIL(Action,Prop) This macro executes `Action` when the property `Prop` fails. It is very useful for debugging purposes.

Model callback functions

To define a state PropErly, a module must implement and export some functions that fully describe the state and success conditions.

initial_state() Executed everytime the generated command sequence is executed to produce the initial state.

command(State) Receives the symbolic state as an argument and generates a symbolic call.

precondition(State,Call) Allows a command to be executed based on the current symbolic state or forces a new call to be chosen using the `command/1` generator.

postcondition(State,Call,Result) Specifies a condition that should hold regarding the `Result` based on the dynamic `State` after evaluating the symbolic `Call`.

next_state(State,Result,Call) Using this function, the next state is created after a successful `Call`. Since this function is called both when generating testcases and when executing them, the state and result handling must work both in symbolic and dynamic content.

More PropEr functions

In order to evaluate symbolic command sequences PropEr we can use the following function:

run_commands(Module, Cmds) Evaluates a command sequence as described above based on the callback functions described in `Module`. The result is a tuple of the form `{History, State, Result}`.

- `History` contains a list of commands that where run without raising an exception.
- `State` contains the state when execution stopped.
- `Result` contains the outcome of the execution. An `ok` atom represents success.

4.3 Defining a PropEr State Model

In this section we will show how to describe an abstract state machine for the web service we presented earlier. To keep things simple, we will do *positive testing* of the web service, meaning we will only test our functionality with valid inputs. In addition, we will only use in our tests the three primary operations of the web service (excluding `getUsername`).

First we define our modules behaviour and export the related functions as shown in Listing 4.9. Then we need to implement all those functions we just exported.

```
-behaviour(proper_statem).
-export([command/1,initial_state/0,next_state/3,precondition/2,postcondition/3]).
```

Listing 4.9: Setting the behaviour of our module

To implement the commands function we initially just list the different calls to web service operations with their respective generators. This is shown in Listing 4.10.

```
command(_S) ->
  oneof([{call, ?MODULE, call_login, generate_login()},
         {call, ?MODULE, call_logout, generate_logout()},
         {call, ?MODULE, call_authenticate, generate_authenticate()}]).
```

Listing 4.10: Implementation of `command/1`

Afterwards, we need to define our state which is shown in Listing 4.11. We can describe our state as a list of authentication tokens, representing logged in clients. Therefore, we define our initial state to be an empty list. Then, we need to describe the state transitions for each of the three different commands. The simplest one is the authentication call, since it changes nothing. Therefore, the associated `next_state/3` call will just return the State unchanged. To define the next state after a `login` operation, we must take extra care with the symbolic nature during test case generation. Our initial approach would be to add the result of the web service call to our list. However, since our result is not the integer authentication token directly, but it is wrapped inside complex structures by Yaws, we need to extract this information. Taking a look at how `check_loginResponse` and `typecheck_loginReturn` work, we can copy paste the related records from the automatically generated records until we reach our desired integer. The function `extract_id` is the resulting function that returns the integer token based on the web service call. To add it to our state, we add a symbolic call to this function to avoid raising an exception during testcase generation. Finally, the state after a logout will be to delete the id from the state. Here, we need to extract the argument, simply by copy-pasting the structure of the generator `generate_logout`.

```

%% State
initial_state() -> [].

%% State update
next_state(S, V, {call, _, call_login, _}) ->
  [{call, ?MODULE, extract_id, [V]} | S];

next_state(S, _V, {call, _, call_authenticate, _}) -> S;

next_state(S, _V, {call, _, call_logout, [#'p:logout'{'id' = Pr_logout_id}]}) ->
    lists:delete(Pr_logout_id, S).

extract_id({ok, undefined,
               [#'p:loginResponse'{'loginReturn' = Pr_loginReturn}]}) ->
  Pr_loginReturn.
```

Listing 4.11: State callback functions

Next, we need to take care of pre- and post-conditions. This is shown in Listing 4.12. Since we don't need to exclude any commands, the precondition is always `true`. A postcondition for a `call_login` command would be to ensure that the resulting authentication token is not a duplicate. To that end we use the `extract_id` function from Listing 4.11. For the `call_authenticate` function, since we are using only valid, registered id's we expect the resulting value to always be true. The same is the case with `call_logout`. The rest of the implementation details are related to extracting the boolean values from the records returned by Yaws.

The next step will be to make sure we only use valid integer tokens in our calls. To that end we must modify the generators. The generator modifications are shown in Listing 4.13. To generate a valid token, we just need to select one from the state using an elements PropEr generator. This leads to the `generate_userID` generator that is used in `generate_logout` and `generate_authenticate` generators subsequently. To use these generators we must slightly modify the command generators to avoid exceptions during the early stages of execution as proposed by Eirini Arvaniti in her tutorials [14]. Last, but not least, we need to ensure that we only use valid passwords. This is the first time we lose the notion of

```
%% Preconditions
precondition(_,_) -> true.

%% Postconditions

%When logging in a NEW id is returned
postcondition(S, {call, _, call_login, _}, V) ->
  ID = extract_id(V),
  not lists:member(ID, S);

% We use valid passwords so we always authenticate
postcondition(_S, {call, _, call_authenticate, _}, V) ->
  {ok, undefined, [#'p:authenticateResponse'{'authenticateReturn' = Bool}]} = V,
  Bool;

postcondition(_S, {call, _, call_logout, _}, V) ->
  {ok, undefined, [#'p:logoutResponse'{'logoutReturn' = Bool}]} = V,
  Bool.
```

Listing 4.12: Preconditions and Postconditions

black box testing, since we must know which the valid passwords would be. We could try random username and password combination using PropEr until we find a correct combination, but that is not its intended use... Therefore, we define a `?PASSWORDS` macro and use it in the `generate_login` generator.

Finally, we need to write our property. This is where we lose *all* ability to do black box testing for this stateful web service. Before running each sequence of commands, we must ensure in some way (e.g. via another web service call on some auxiliary web service) that the state is the same for all executions and shrinking. In our implementation we use a `reset_state` function that handles all the necessary state restorations. The final property is shown in Listing 4.14 and uses both the `?TRAPEXIT` and the `?WHENFAIL` macros discussed earlier.

4.4 Testing the Web Service

After all these modifications, we are ready to test the stateful system. Using an Erlang Shell we run the property and get the output shown in Listing 4.15.

This Listing shows that our property failed after 28 tests, having created a rather large and complex command sequence. The shrinking process allows us to recognize the actual cause of the failure. Logging in twice with the same account, logging out with the second and then attempting to authenticate the first account yields an error in our authenticate postcondition - meaning the token does not authenticate. Upon careful examination of our Java Code we locate the problem: As shown in Listing 4.5 we delete the first occurrence of the username when logging out, rather than the occurrence of the integer token. We fix that mistake with the code shown in Listing 4.16 and run the property again as shown in Listing 4.17. Our web service passes 1000 correct tests!

```
-define(PASSWORDS, [{"Lemonidas", "foo"}, {"Kostis", "42"}, {"gearg", "100"}]).

%% Custom id generator
generate_userID(S) ->
  elements(S).

generate_login() ->?LET(
    {Pr_login_name, Pr_login_password},
    elements(?PASSWORDS),
    [#'p:login'{
        'name' = Pr_login_name,
        'password' = Pr_login_password
      }
    ]
  ).

generate_logout(S) ->?LET(
    Pr_logout_id,
    generate_userID(S),
    [#'p:logout'{'id' = Pr_logout_id}]
  ).

generate_authenticate(S) ->?LET(
    Pr_authenticate_id,
    generate_userID(S),
    [#'p:authenticate'{'id' = Pr_authenticate_id}]
  ).

command(S) ->
  Logged = (S =/= []),
  oneof([{call, ?MODULE, call_login, generate_login()}] ++
        [{call, ?MODULE, call_logout, generate_logout(S)}
         || Logged] ++
        [{call, ?MODULE, call_authenticate, generate_authenticate(S)}
         || Logged]).
```

Listing 4.13: Generator Modifications

```
prop_login_service_works_fine() ->
    ?FORALL(Cmds, commands(?MODULE),
            ?TRAPEXIT(
                begin
                    reset_state(),
                    {History,State,Result} = run_commands(?MODULE, Cmds),
                    ?WHENFAIL(io:format("History: ~p\nState: ~p\nResult: ~p\n",
                                        [History,State,Result]),
                              Result =:= ok)
                end)).
```

Listing 4.14: Property for Login Service

```
proper:quickcheck(login_statem:prop_login_service_works_fine()).
...........................!
Failed: After 28 test(s).
[{set,{var,1},
      {call,login_statem,call_login,
            [{'p:login',undefined,"Kostis","42"}]}},
 {set,{var,2},
      {call,login_statem,call_login,
            [{'p:login',undefined,"Lemonidas","foo"}]}},
 {set,{var,3},
      {call,login_statem,call_login,
            [{'p:login',undefined,"gearg","100"}]}},
 {set,{var,4},
      {call,login_statem,call_authenticate,
            [{'p:authenticate',undefined,
                              {call,login_statem,extract_id,[{var,3}]}}]}},
 {set,{var,5},
      {call,login_statem,call_login,
            [{'p:login',undefined,"Lemonidas","foo"}]}},
 {set,{var,6},
      {call,login_statem,call_logout,
            [{'p:logout',undefined,
                        {call,login_statem,extract_id,[{var,5}]}}]}},
 {set,{var,7},
      {call,login_statem,call_authenticate,
            [{'p:authenticate',undefined,
                              {call,login_statem,extract_id,[{var,2}]}}]}},
 {set,{var,8},
      {call,login_statem,call_login,
            [{'p:login',undefined,"Kostis","42"}]}},
 {set,{var,9},
      {call,login_statem,call_logout,
            [{'p:logout',undefined,
                        {call,login_statem,extract_id,[{var,2}]}}]}}]
History: [...]
State: [208,8442,9485]
Result: {postcondition,false}

Shrinking ....(4 time(s))
[{set,{var,2},
      {call,login_statem,call_login,
            [{'p:login',undefined,"Lemonidas","foo"}]}},
 {set,{var,5},
      {call,login_statem,call_login,
            [{'p:login',undefined,"Lemonidas","foo"}]}},
 {set,{var,6},
      {call,login_statem,call_logout,
            [{'p:logout',undefined,
                        {call,login_statem,extract_id,[{var,5}]}}]}},
 {set,{var,7},
      {call,login_statem,call_authenticate,
            [{'p:authenticate',undefined,
                              {call,login_statem,extract_id,[{var,2}]}}]}}]
History: [{[],{ok,undefined,[{'p:loginResponse',[],3407}]}},
          {[3407],{ok,undefined,[{'p:loginResponse',[],2193}]}},
          {[2193,3407],{ok,undefined,[{'p:logoutResponse',[],true}]}},
          {[3407],{ok,undefined,[{'p:authenticateResponse',[],false}]}}]
State: [3407]
Result: {postcondition,false}
false
```

Listing 4.15: Testing the Property of Listing 4.14

```java
public boolean logout(int id){
  for (Pair<String,Integer> p : logged){
    if (p.getSecond() == id){
      logged.remove(p);
      return true;
    }
  }
  return false;
}
```

Listing 4.16: Correct Implementation of Logout

```
1> proper:quickcheck(login_statem:prop_login_service_works_fine(), 1000).
... (1000 dots) ...
OK: Passed 1000 test(s).
true
```

Listing 4.17: Re-Testing the Property

Chapter 5

Related Work

Research in testing web services has seen a lot of growth in the past few years. A variety of tools has emerged handling disparate aspects of testing, from functional to integration and regression testing; cf. a survey on the subject [5]. Our tool can handle automatic functional testing with structurally valid test cases created based on the WSDL specification of a web service. Prior research work using a similar idea includes the work of Bartolini *et al.* [2] and of Ma *et al.* [10]. Most existing tools have expanded on the idea of generating XML messages based on a static analysis of the WSDL specification. Most of them, however, lack in the aspect of validating the results of the web service and just present them to the user for inspection.

Amongst the existing frameworks and tools in the area, there are a few that stand out. Most notably, SoapUI [16], one of the most complete testing frameworks that can handle semi-automated functional testing, amongst other things. This tool however does not automatically generate sample test cases, just aids the user in doing so. Other tools, such as WSDLTest [15] are similar to ours in generation strategy, yet user input happens for every script if modifications and assertions (for output validation) are needed. Our tool, creates generators that allow random test case creation, while any modification by the user to refine the generators needs to take place only once and will be valid for all the SOAP messages generated. Another category of tools is the one that contains WS-Taxi [3]. This is one of the first tools to have been created based on the idea of WSDL-based testing. WS-Taxi was first outlined in 2007 [4], but as stated in the related papers, while it provides automatic data generation, the tool lacks a test oracle. Finally, there is a couple of papers and a tool that use Haskell's QuickCheck to do automatic test case generation. The idea of Zhang *et al.* [21] is largely similar to our own: use a property-based testing tool to create generators that allow for automatic testing. The tool that spawned from this effort, monadWS [20], contains a promising comparison with SoapUI and SoapTest, yet also does not utilize the power of QuickCheck for deeper validation of the results returned by the web service.

All in all, what makes our tool stand out from the rest is that it was designed for use with a property based testing tools with the power of PropEr. Our tool handles automatic test data generation as efficiently and automatically as many other available tools, yet its design will allow for faster and powerful testing, using properties to automatically validate an arbitrary number of progressively more complex test cases.

Chapter 6

Concluding Remarks and Future Work

This thesis described the technical and design issues and the implementation of a new property based testing tool for web services. After introducing all the necessary background information for web services and PropEr, we thoroughly presented implementation details of this new property based testing tool and examples of its use. To evaluate our tool's capabilities we applied our tool to type check more than 40 web services freely available on the net. Our tool was able to succesfully test all but one of them, finding no type errors. The reason that we were unable to test that single web service was that it contained a special (German) character in its WSDL specification, which Yaws did not support yielding an error.

All in all, we consider the work as *successful.* We managed to create a tool that addresses many of the difficulties in testing web services and provides significant support for the tester to speed up the testing process.

Even though our tool has been able to succesfully handle all WSDL specifications that could be handled by Yaws, some of the XSD language constructs (such as patterns) have yet to be integrated into our tool. Providing completeness to our tool is definitely one of the primary directions for future work.

The primary future goal, however, is to support stateful testing of web services. PropEr provides this ability, thesis of Eirini Arvaniti [1] and was shown in Chapter 4, but it is still unclear how to effectively automate or aid the user in applying property based testing techniques of stateful systems for web services.

Bibliography

[1] E. Arvaniti. Automated Random Model-Based Testing of Stateful Systems. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, July 2011. `http://artemis.cslab.ntua.gr/el_thesis/artemis.ntua.ece/DT2011-0142/DT2011-0142.pdf`.

[2] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Towards automated wsdl-based testing of web services. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, ICSOC '08, pages 524–529, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Ws-taxi: A wsdl-based testing tool for web services. In *International Conference on Software Testing, Verification, and Validation*, pages 326–335, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[4] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Automatic test data generation for xml schema-based partition testing. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pages 4–, Washington, DC, USA, 2007. IEEE Computer Society.

[5] G. Canfora and M. Di Penta. Service-oriented architectures testing: A survey. In A. De Lucia and F. Ferrucci, editors, *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 78–105. Springer, Berlin, Heidelberg, 2009.

[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, Mar. 2001.

[7] I. K. El-Far and J. A. Whittaker. Model-based software testing. Encyclopedia on Software Engineering. 2001.

[8] S. Gao, C. M. Sperberg-McQueen, and H. S. Thompson. W3c xml schema definition language (xsd) 1.1 part 1: Structures, Jan. 2012.

[9] A. P. M. Utting and B. Legeard. A taxonomy of model-based testing. Working paper. 2006.

[10] C. Ma, C. Du, T. Zhang, F. Hu, and X. Cai. Wsdl-based automated test data generation for web service. In *International Conference on Computer Science and Software Engineering*, pages 731–737, Dec. 2008.

[11] M. Papadakis. Automatic Random Testing of Function Properties from Specifications. Diploma thesis, National Technical University of Athens, School of Electrical

and Computer Engineering, October 2010. `http://artemis.cslab.ntua.gr/el_thesis/artemis.ntua.ece/DT2010-0295/DT2010-0295.pdf`.

[12] M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*, pages 39–50, New York, NY, Sept. 2011. ACM Press.

[13] D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson. W3c xml schema definition language (xsd) 1.1 part 2: Datatypes, Jan. 2012.

[14] Proper website.

[15] H. M. Sneed and S. Huang. Wsdltest - a tool for testing web services. In *Eighth IEEE International Symposium on Web Site Evolution, WSE '06*, pages 14–21, Sept. 2006.

[16] soapUI: The swiss-army knife of testing.

[17] W3c tutorials.

[18] xmerl reference manual.

[19] Yaws: yet another web server.

[20] Y. Zhang, W. Fu, and C. Nie. monadws: a monad-based testing tool for web services. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 111–112, New York, NY, USA, 2011. ACM.

[21] Y. Zhang, W. Fu, and J. Qian. Automatic testing of web services in haskell platform. 2010.