Automating Property-based Testing of Evolving Web Services

Huiqing Li   Simon Thompson
Pablo Lamela Seijas

School of Computing, University of Kent, UK
{H.Li,S.J.Thompson,P.Lamela-Seijas}@kent.ac.uk

Miguel Ángel Francisco

Interoud Innovation, A Coruña,Spain
miguel.francisco@interoud.com

Abstract

Web services are the most widely used service technology that drives the Service-Oriented Computing (SOC) paradigm. As a result, effective testing of web services is getting increasingly important. In this paper, we present a framework and toolset for testing web services and for evolving test code in sync with the evolution of web services. Our approach to testing web services is based on the Erlang [5, 9] programming language and QuviQ QuickCheck, a property-based testing tool written in Erlang, and our support for test code evolution is added to Wrangler, the Erlang refactoring tool.

The key components of our system include the automatic generation of initial test code, the inference of web service interface changes between versions, the provision of a number of domain specific refactorings and the automatic generation of refactoring scripts for evolving the test code. Our framework provides users with a powerful and expressive web service testing framework, while minimising users' effort in creating, maintaining and evolving the test model. The framework presented in this paper can be used by both web service providers and consumers, and can be used to test web services written in whatever language; the approach advocated here could also be adopted in other property-based testing frameworks and refactoring tools.

Keywords   Web Service, Property-based Testing, QuickCheck, Wrangler, API Evolution, Erlang, WSDL

1.   Introduction

Service-Oriented Computing (SOC) is a computing paradigm that uses services as the basic building blocks to support the rapid and low-cost development of distributed applications even in heterogeneous environments. Web services (WS) are nowadays the most widely used service technology that drives the SOC paradigm. However testing web services is more challenging than testing traditional software due to the complexity of web service technologies and the limitations that are caused by the SOC environment [22].

Over the last decade, a variety of techniques have been proposed for testing different aspects of web services. For example, a recent survey [22] of existing web service testing methods lists dozens of tools utilising these methods. The testing methods covered by the survey include unit-testing, model-based testing, formal verification, fault-based testing, partition testing and contract-based testing. Most of these testing frameworks speed up the testing process to some extent, but are also limited in some way or another. For example, unit testing approaches reduce the cost of testing by automated test data generation and test execution, but they lack automated test oracle generation; model-based testing approaches have the advantages of automating the test case generation process and the ability to analyse the quality of web services statically, but a model based on a WSDL (Web Services Description Language) specification might not represent the complete behaviour of a web service due to the lack of behavioural information that it contains.

Web services, like any other applications and libraries, change their APIs. API changes affect not only client applications that use the API, but also the code that tests the API. The process of designing an API is itself an evolving processes, and the details of an API could go through a number of changes before being finalised. Hence the test code needs to evolve along with the API, a process which is generally done manually. With some fully-automated testing tools, it might be possible to re-generate the complete test code from scratch; for testing frameworks that allow users to incorporate further information about the expected behaviour into the test code, this is not always an option.

In this paper, we present a property-based approach to testing web services, and a refactoring approach to evolving test code when a web service API changes. The framework is based on the Erlang programming language, and is illustrated with a case study based at the company Interoud Innovation.

Property-based testing (PBT) provides a powerful, high-level, approach to testing; rather than focusing on individual test cases, in PBT this behaviour is specified by *properties*, expressed in a logical form. For example, a function without side effects might be specified by means of the full input/output relation using a universal quantification over all the inputs; a stateful system will be described by means of model, which is an extended finite state machine. The system is then tested by checking whether it has the required properties for randomly generated data, which may be inputs to functions, sequences of API calls to the stateful system, or other representations of test cases.

Property-based testing is gaining popularity, especially in the community of functional programming languages. For Haskell, there is the original QuickCheck tool [11] developed by Koen Claessen and John Hughes in 2000; for Erlang there are Quick-Check [15] commercialised by QuviQ, and PropEr [20], an open source tool inspired by QuickCheck.

Erlang QuickCheck is the tool of choice for our property-base testing framework. In particular, we utilise its support for abstract state machines through the *eqc_statem* behaviour. With *eqc_statem*, the user defines an abstract model of the system under test (SUT), including an abstraction of the state of the SUT itself. This model

includes an initial abstract state in which test cases begin, as well as describing how each command changes that state. The state is used by QuickCheck both during test case generation and during test execution. For each command, a number of things can be described: *preconditions* to decide whether or not to include a candidate command in test cases; *postconditions* to check that the value returned by the command executed is correct; a description of the changes on the abstract state as a result of command execution; and how to generate an appropriate function call to appear next in a test case.

We have enhanced QuickCheck with tools that automate those aspects of the problem that are tedious and cumbersome to write manually. In particular, our testing framework includes tools to automatically create data generators according to XSD (XML Schema Definition) schema and WSDL specification, connector functions for invoking web services from test, generic WS properties, and the initial code implementing an *eqc_statem* behaviour. The initial test code is runnable on its own, and can be used for response testing purpose. However with the user adding an abstract model into it, operation-specific test oracles can be specified, which is particularly useful for testing stateful web services

The tool that lies at the heart of our support for test code evolution is Wrangler [17]. Wrangler, developed at the University of Kent by the some of the authors, is a *user-extensible* refactoring and code inspection tool for Erlang programs. Apart from providing a set of built-in refactorings, Wrangler allows users to define refactorings, code inspections, and general program transformations for themselves to suit their particular needs. These are defined using a template- and rule-based program transformation and analysis framework built into Wrangler. Wrangler also provides an embedded domain-specific language (DSL) [18] for describing composite refactorings: that is, refactorings that are built up from a number of primitive refactorings. The DSL gives a powerful and easy-to-use framework that allows users to script their own composite refactorings in order to carry out large-scale batch refactorings in an efficient way.

Due to the particular nature of the refactorings needed for evolving *eqc_statem*-based test models, we have extended Wrangler with a number of domain-specific refactorings, such as *add/remove an operation*, *rename an operation*, *add/remove an argument to/from an operation*. The refactorings are defined using Wrangler's support for user-extensibility. Our experience demonstrate the usefulness for making refactoring tools user-extensible.

To facilitate the evolution of test code, our framework is also able to compare two different versions of the WSDL description of a web service, and automatically infer the changes made to the web service API. Based on the result, the tool generates a composite refactoring script that can be applied to the test code in batch mode. In summary, the main contributions of this paper are:

- A property-based testing framework with a toolset that fully automates the initial runnable test code generation, https://github.com/RefactoringTools/WSToolkit.

- A library of QuickCheck data generators for generating data with restrictions.

- A tool for the automatic inference of web service interface changes.

- A refactoring approach for evolving test code in sync with the evolution of web services.

- An industrial case study describing how the framework is used in practice.

The rest of the paper is organised as follows. Section 2 introduces WSDL and the key existing tools used in this framework, and Section 3 introduces how the framework is used in practice in an indus-

trial case study. Section 4 describes the architecture of our testing and evolution framework. Section 5 presents the components of the testing framework, and Section 6 presents the components of the test code evolution framework. A summary and discussion of the contributions of the paper are included in Section 7. Related work is discussed in Section 8 and Section 9 concludes the paper.

2. Existing Tools and Technologies

In this section, we give a short introduction to WSDL and the key existing tools that are used in our framework.

2.1 Web Services Description Language (WSDL)

WSDL [4] is an XML-based interface description language that is used for describing the functionality offered by a web service. A WSDL description is a document written in XML. The document provides a full description of how the service can be called, what parameters it expects, what data structures it returns, and the address or connection point to the web service, etc.

Every WSDL specification contains (or references) an XSD schema [3] inside. The XSD schema describes the data types of messages exchanged by the web service methods. The types of these messages are divided in two categories: simple and complex types. A simple type can be either a primitive data type, such as float, integer, string, etc., an aggregate of the primitive data type, such as list and union, or a restricted version of it, like an enumeration, a string conforming to a pattern or range-constrained integers. A complex type on the other hand is derived based on other types, either simple or complex. Usually, complex types are created by forming element aggregates: sequences, all or choices [3].

2.2 Erlsom

Erlsom [1] is a set of Erlang functions to parse (and generate) XML documents. It supports various modes of operation: as a SAX parser, as a simple DOM-like parser, or as a data binder. As a *data binder*, Erlsom parses XML documents that are associated with an XSD schema. It checks whether the XML document conforms to the schema, and translates the document to an Erlang structure that is based on the types defined in the schema. The internal Erlang representation generated by Erlsom is also the internal representation used by our framework.

The current Erlsom implementation ignores all the restrictions on simple types. However, restrictions on simple types are crucial for writing good-quality data generators, we have therefore extended Erlsom with another pass, which extracts information about simple types from the XSD schema and inserts it into the internal representation generated by Erlsom.

2.3 Property-based testing with QuickCheck

QuickCheck [15] supports random testing of Erlang, and also of C programs, through a foreign function interface. Properties of the programs are stated in a subset of first-order logic, embedded in Erlang syntax. QuickCheck verifies these properties for collections of Erlang data values generated randomly, with user guidance in defining the generators where necessary. When a counterexample is found, QuickCheck tries to generate a simpler – and thus more comprehensible – counterexample, in a constructive manner; this process is called *shrinking*.

When testing state-based systems it makes sense to build an abstract model of the system, and to use this model to drive the testing of real system. The abstract state machine can be implemented as a client module of the pre-defined QuickCheck behaviour *eqc_statem*. To do so, the user needs to define a number of callback functions:

- `state_state()` – returns the initial model state.

- `precondition(S, C)` – returns `true` if the symbolic function call `C` can be performed in the symbolic state `S`. This is used to decide whether or not to include a candidate command in test cases.

- `postcondition(S, C, R)` – checks the postcondition of symbolic call `C`, executed in dynamic state `S`, with result `R` during test execution.

- `next_state(S, R, C)` – the state transition function of the abstract state machine. During test generation, it computes the symbolic state after symbolic call `C`, performed in symbolic state `S`, with result `R`; during test execution, the same function is used to compute the next *dynamic* state.

- `command(S)` – generate a candidate symbolic function call to appear next in a test case, if the symbolic state is `S`. Test sequences are generated by using `command(S)` repeatedly.

2.4 Refactoring with Wrangler

Wrangler [17, 18] is an interactive refactoring tool, implemented in Erlang, and is integrated with (X)Emacs and with Eclipse. It is downloadable from `https://github.com/RefactoringTools/Wrangler`. One of the features that distinguish Wrangler from most other refactoring tools is its user-extensibility:

- Wrangler provides a high-level *template- and rule-based API* [17], so that users can write refactorings, or general program transformations, that meet their own needs in a concise and intuitive way without having to understand the underlying AST representation and other implementation details.

- Wrangler is built with an embedded, domain-specific language for describing composite refactorings: refactorings that are composed from a number of elementary refactorings. The DSL [18] gives a powerful and easy-to-use framework that allows users to script their own reusable composite refactorings in order to carry out large-scale batch refactorings.

User-defined refactorings can be invoked via the Emacs interface to Wrangler, in exactly the same way as built-in refactorings, and so their results can be previewed and undone.

3. Industrial Case Study at Interoud Innovation

The framework presented here is used by Interoud Innovation (`http://www.interoud.com/`) to test their web services. Interoud Innovation is a company based in Spain, whose main activity is the development and commercialisation of software products for media distribution, mainly IPTV and OTT for telecommunications service providers and hospitality environments.

Interoud testing workflow. The design of the framework was influenced by the way Interoud Innovation test web services in practice using property-based testing. Thus, the steps they follow to test a web service consist of implementing a QuickCheck state machine, in which each API operation to test is a command of the state machine, and a WS connector module that allows the operations of the web service to be invoked from Erlang.

The Interoud web service for which this approach is being used has 363 operations, from which approximately 160 are invoked by POST and 203 by GET. All the operations return data in XML. The existing QuickCheck state machine implemented by Interoud tests 98 operations of this web service (around 27% of the total). This implementation requires a hand-crafted WS connection module with approximately 1K lines of code.

The benefits of using the framework to create a QuickCheck state machine. With the use of this framework, both the WS connector module and a skeleton of the QuickCheck state machine

are automatically generated from a description of the web service to test, specifically, from a WSDL specification.

As Interoud Innovation already has WSDL descriptions for their web services, no extra work is required to use these new tools. The automation provided by this framework therefore saves a significant amount of time in the creation of the test code, as well as minimising the introduction of errors in the implementation of that test code, in particular in the WS connector module.

In fact, the use of this new framework allowed Interoud's testers with no effort at all to add the operations of the web service described in the WSDL to the Quickcheck state machine, and hence they will be invoked in every test execution.

Evolution of Interoud's web services. Interoud's web services evolve very rapidly. For example, in September and October 2013, ten new operations were added to the web service, and fifteen existing operations were modified, many of them by extending their functionality with new parameters. Every time a developer changes the web service, the test code needs to be changed as well.

Before using the framework described in this paper, all the changes in the test code had to be done manually. Now, it is possible to use Wrangler with auto-generated refactoring scripts to help testers to change their test code according to the changes made to the web service. This new approach makes it easier to update the test code, because it automates many of the changes to be applied to the test code to align it with the new version of the web service.

Conclusions. Despite this automation, testers still have to modify the QuickCheck state machine manually, specifically by adding semantic information related to the new or modified API operations. However, even though some manual work is required, Interoud's testers find this framework useful for two reasons. Firstly, it is possible to have a test module that invokes all the operations of the web service simply by providing a WSDL description of the service. And secondly, the auto-generated refactoring scripts help to change the test code without overlooking parts that might be affected by the changes in the WSDL, therefore making the process more robust.

4. System Architecture

In this section, we give an overview of our testing and evolution framework architecture. More implementation details are presented in Sections 5 and 6.

Figure 1 shows the framework for testing web services using QuickCheck. Given the WSDL description and the associated XSD schema of a web service, we compile the XSD schema first, and this gives us the intermediate representation of the data types of messages exchanged by the web service operations in the format of Erlang records. We call this intermediate representation the *data model* of this web service.

The next step is to parse the WSDL file itself according to the data model generated from the WSDL schema. This checks if the WSDL document conforms to the WSDL standard, and translates the WSDL document to an Erlang structure that is based on the types defined in the WSDL schema. From this Erlang representation together with the data model derived, we are able to derive the operations supported by the web service, the input and output of each operation and their types, the binding method, location information, etc.

The data model and information about the operations are then used to generate three components that form the initial QuickCheck test model. The three components generated are:

- The *data generators* for generating the input data of web service operations.
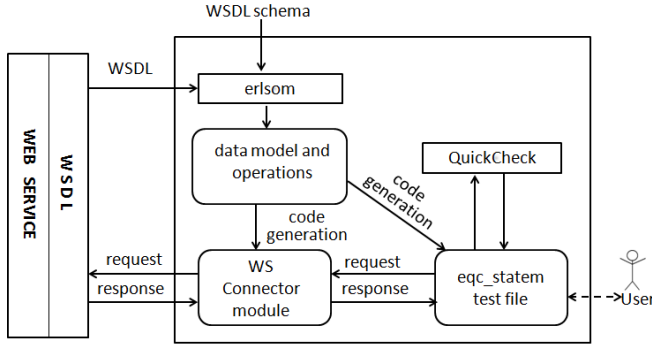
Figure 1. The QuickCheck Testing Model of Web Services

- The *web service connector module* that contains the wrapper functions, one per operation, for invoking web services from test code written in Erlang, and generic test oracles that check the results returned from the web service conforms to the requirements of the data model.

- The initial eqc_statem *module*. At this stage, the *eqc_statem* has an empty abstract state; however it has the template code for the callback functions, and another simple layer of wrapper functions for invoking functions from the connector module and processing the result returned. So the test module generated is compilable and runnable on its own. Without an abstract model defined, only properties defined in the collector module are checked. This module provides the platform for the user to further develop the test model.
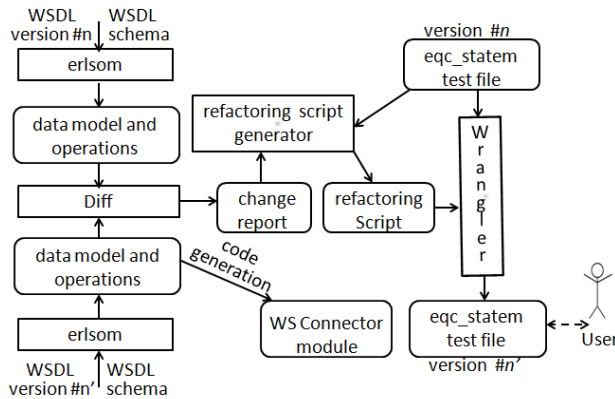

Figure 2. The Test Code Evolution Model

Figure 2 shows the framework for test code evolution. The WSDL specifications of two versions of a web service are parsed and analysed in the same way as in the testing framework, the results are then fed to a 'diff' algorithm, which derives the changes made to both the data types and the web service operations. The 'diff' result is then presented to the user as a change report, but also used by the tool to generate a composite refactoring script using Wrangler's DSL. The refactoring script generator takes both the change report and the current *eqc_statem* implementation as input, and returns the composite refactoring as an Erlang module implementing Wrangler's `gen_composite_refac` behaviour.

The user could apply the composite refactoring script as it is, or apply the refactoring commands in the script one by one manually.

When an abstract model is defined, the changes made to a web service API may also affect the logic of the abstract model. Updating the logic of the abstract model is beyond this paper's scope. It is also possible that a user might refine some of the data generators generated automatically by the tool so that the abstract state data is used. Our tool respects user's code, and does not change it without user's permission. For this reason, the evolution of data generators is handled in a slight different way, more details about this is described in Section 6. Since the WS connector module in general untouched by the user, we re-generate this module instead of refactoring it.

5. Automating Property-based Testing

This section discusses the implementation of the various components to automate the property-based testing using QuickCheck.

5.1 Automatic Creation of Data Generators

QuickCheck comes with a library of data generators. It supports not only the random generation of values of an Erlang built-in type, such as `integer`, `float`, `char`, `boolean`, etc, but also the generation of values of compound data types. It allows tuples and lists containing generators to be used as generators for values of the same form. For example, {`int()`, `bool()`} is a generator that generates random pairs of integer and booleans.

QuickCheck also provides a collection of macros for controlling the data generation process. For example the macro `?SUCHTHAT(X, G, P)` generates values `X` from `G` such that the condition `P` is true; the macro `?SIZED(Size, G)` is used to control the size of the generated data, etc. These macros are useful for generating data that satisfy certain constraints, but are rather limited when generating test data for web services, due to the various restrictions/facets that a user can specify for data types. Take the `string` type as an example, the restrictions that can be used on `string` values include: `enumeration`, `length`, `maxLength`, `minLength` and `pattern`, as shown by the examples in Fig 3. A naive use of the `?SUCHTHAT(X, G, P)` macro to generate values satisfying certain constraints could easily lead QuickCheck to fail with an error message like: `"?SUCHTHAT failed to find a value within 100 attempts."`.

However, as QuickCheck allows a user to compose complex data generators from simpler ones, we were able to define a collection of data generators especially for generating primitive data types with restrictions. These new generators are defined in a module named `gen_lib`. For instance, the following three data generators have been defined for generating `string` values:

- `string()` – generates a printable string of random length.

- `string(Min, Max)` – generates a printable string of length between `Min` and `Max`.

- `string(Pattern)` – generates a printable string that satisfies the pattern constraint.

As an example, the generator `string(MinLen, MaxLen)` is defined as:

```
string(MinLen, MaxLen) ->
  ?LET(N, eqc_gen:choose(MinLen, MaxLen),
    lists:foldl(fun(_X, Acc)->
              ?LET(C, eqc_gen:choose(32, 127),
                   [C|Acc])
            end, [],lists:seq(1,N))).
```

where `choose(M, N)` is a QuickCheck built-in generator that generates a number in the range `M` and `N`; `?LET(Pat,G1,G2)` is a pre-defined QuickCheck macro that generates a value from gener-
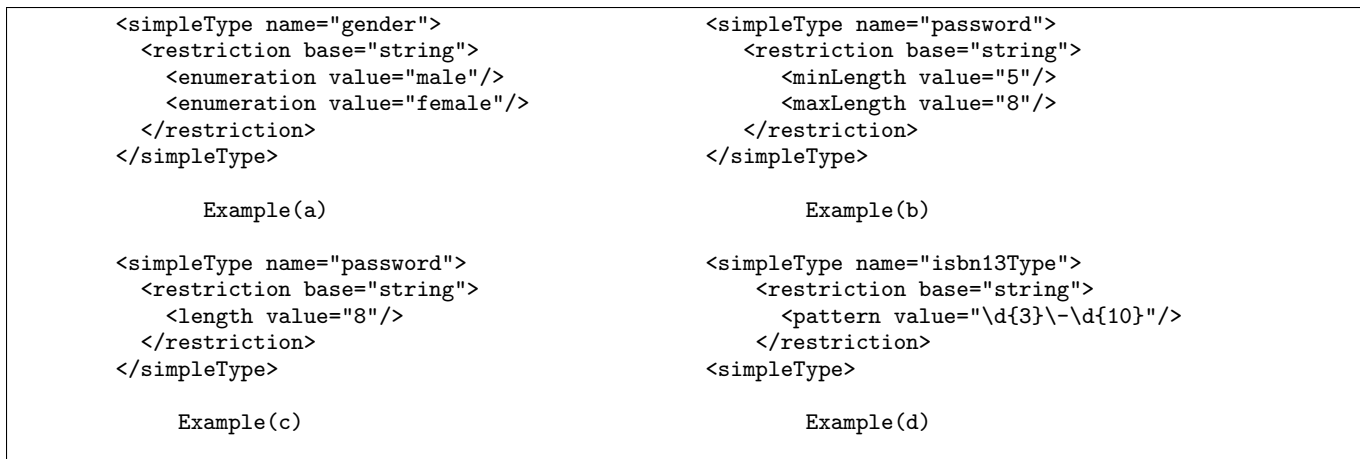
```
<simpleType name="gender">               <simpleType name="password">
  <restriction base="string">             <restriction base="string">
    <enumeration value="male"/>             <minLength value="5"/>
    <enumeration value="female"/>           <maxLength value="8"/>
  </restriction>                           </restriction>
</simpleType>                             </simpleType>

       Example(a)                                Example(b)

<simpleType name="password">             <simpleType name="isbn13Type">
  <restriction base="string">             <restriction base="string">
    <length value="8"/>                     <pattern value="\d{3}\-\d{10}"/>
  </restriction>                           </restriction>
</simpleType>                             <simpleType>

      Example(c)                                 Example(d)
```

Figure 3. Example restrictions on `string` values

ator `G1`, binds it to `Pat`, then generates a value from `G2`, which may refer to the variables bound in `Pat`.

The generator `string(Pattern)` is defined in a similar way, apart from that the regular expression representing the pattern is parsed into an internal representation, and that data generators are composed according to the structure of the regular expression pattern. As an example, below is a sampling output of generating `isbn13Type` (see example (d) in Fig 3) values.

```
Eshell V5.10.2  (abort with ^G)
1>eqc_gen:sample(gen_lib:string("\d{3}\-\d{10}")).
"557-0879218041"
"060-6994306535"
"188-2111291597"
"298-2437633024"
"800-3253986877"
"980-9306669387"
"338-8755797024"
"265-0014560327"
"456-9514747557"
"818-8831798457"
"145-3571075325"
ok
```

`String` values with `enumeration` constraints can be expressed with QuickCheck's built-in generator `oneof`, which generates a value using a randomly chosen element of the list of generators. Hence, a data generator that generates `gender` (see example (a) in Fig 3) values can be written as: `eqc_gen:oneof(["male", "female"])`.

With QuickCheck's data generator library and our library for generating primitive data values with restrictions, it is straightforward to map the data model generated from the schema to QuickCheck data generators in most cases. Similar work has been done by Lampropoulos and Sagonas to automatically generate data generators from WSDL specification using PropEr [16], however their tool does not handle the pattern constraining facet on integers/strings, and some date formats.

5.2 Automatic Creation of the WS Connector Module

The web service connector module defines a collection of functions that are used by QuickCheck to invoke web service operations. There is a connector function defined for each web service operation. Given the large number of operations provided by a web service, and the common code structure of these function definitions,

tool support for the automatic generation of this module proved valuable.

Figure 4 shows an example connector function for invoking the `GetWeather` operation from the `GlobalWeather` web service hosted at: `http://www.webservicex.net/globalweather.asmx`. This function takes the data values generated by the data generators, encodes them into the format accepted by the service using `generate_get_params`, sends the request to the service, and processes the response from the web server according to the XSD schema to ensure that no exception has been raised and the data returned conforms to the data model specified in the XSD schema. The name of the file containing the XSD schema is defined as a macro in the connector module, hence does not appear in the definitions of connector functions. While the example shown here is for an HTTP API, the same idea applies to SOAP APIs.

5.3 Automatic Creation of the *eqc_statem* Behaviour

A number of callback functions, together with data generators and some utility functions need to be defined in order to implement an *eqc_statem* test module. Callback functions such as `precondition`, `postcondition` and `next_state` in general require a function clause for each operation to be tested. While our tool cannot automate the creation of code specifying the logic of the abstract model in these callback functions, it can automate the generation of the remaining part, and that is exactly what it does. Automatic generation of the initial test module also helps to maintain the consistency between the WSDL description and the test code in terms of naming of identifiers and the textual order of function clauses handling operations.

An an example, Figure 5 shows the test module generated from the WSDL specification (`http://www.webservicex.net/globalweather.asmx?WSDL`) of the `GlobalWeather` web service. Two operations are supported: `GetCitiesByCounty` and `GetWeather`. The test module generated is already compilable and runnable for response testing purpose as shown below.

```
1> eqc:quickcheck(weather_test:prop_state_machine()).
.....................................................
.....................................................
OK, passed 100 tests
```

As one may have noticed, the `state` record defining the abstract state representation is an empty tuple, the state transitions after a WS operation leave the current state unchanged as defined in the `next_state` function, and the only post-condition checked after a

```
get_weather(CityName, CountryName)->
  GetParams = generate_get_params(`GetWeather', [CityName, CountryName]),
  Url = add_get_params(?BASE_URL++"/GetWeather",GetParams),
  http_request(`GET', Url,
                  fun(Data) ->
                      process_response(`GetWeatherResponse', Data)
                  end).
```

Figure 4. An example connector function

WS operation is that no error has been raised and that the response data confirms to the data model. The abstract state representation, state transitions as well as various condition checks are exactly where the user's knowledge should be added.

As a design decision, we let a wrapper function take the tuple of its input elements as the parameter, unless no input is needed. Another option is to let each input element as a separate parameter, but grouping input elements into a tuple has the advantage of allowing dependency between the generators of input elements that are related to each other.

For instance, with an empty abstract model representation, country names and city names are generated as random strings or the literal `none` (when the name is optional), this results in the fact that nearly all of the country/city names generated are semantically invalid, and in most cases the web service either returns `<string>Data Not Found</string>` as the response or the complete data set when no input is provided. However, this could be improved once an abstract state representation has been defined. Suppose we redefine the `state` record as:

```
-record(state,
        {country_cities=[]::[{country(), [city()]}]}).

-type country()::string().
-type city()::string().
```

then we could rewrite the data generators as:

```
gen_country_name(S)->
   Countries = [Country||{Country, _City}
                            <-S#state.country_cites].
   eqc_gen:oneof([none|Countries]).

gen_get_weather(S)->
   ?LET({Country, Cities},
       eqc:oneof(S#state.country_cities),
       {eqc:oneof([none|Cities]),
        eqc:oneof([none,Country])}).
```

The `next_state` function can also be refined so that the abstract state is updated after each WS operation; `pre_condition` can be enriched to ensure only realistic test cases are generated, and operation-specific test oracles can be added to `post_condition`.

6. Automating Test Code Evolution

This section describes the implementation of the components to support test code evolution.

6.1 Automatic Inference of Web Service Interface Changes

The automatic inference of web service API changes is based on the Levenshtein distance algorithm [2] implemented in Erlang. The algorithm returns the minimum number of changes (insert, delete or substitution) required to change from one sequence to another. In our case, a sequence could be either a sequence of

API operations or a sequence of operation input/output elements. An API operation consists of the operation name, input elements with type information, response data with type as well the binding method; whereas an input/output element consists of the element name and its type.

The algorithm is first applied to the old and new sequences of WS operations, which are derived from the old and new versions of WSDL specifications. This step returns a list of WS operations (or operation pairs in case of substitution) each tagged with `unchanged`, `insert`, `delete`, or `substitute`. The result is further processed to

- replace `substitution` with `delete` and `insert`, i.e. {substitution, A, B} is replaced with {delete, A}, {insert B};

- remove superfluous insertions/deletions, i.e. the deletion and insertion of the same API. This could happen due to the re-ordering of operation specifications in the WSDL specification. In our case, we do not require that the definitions of function clauses follow the same order in which the descriptions of operations are arranged.

- merge the deletion and insertion of two operations that only differ in the operation name as `rename`.

- merge the deletion and insertion of two operations that have the same name and binding method as `input_change`, `output_change` or `input_output_change` depending on where the differences are.

For operations with input/output change, the same algorithm is used to infer the changes required to derive the new input/output sequence from the old, but since the order may matter (e.g. when the elements are tagged with `sequence` in the WSDL specification) in this case, we do not remove superfluous insertions/deletions.

The report returned is to be inspected, and corrected if needed, by the user. Correction is needed in cases that substantial changes have been made to an operation, i.e. both renaming of operation and input/output changes.

As an example, Fig 6 shows the 'diff' report, with some manually added comments, generated during a case study. The report says that two new operations, `FindAllRooms` and `DeleteDevice`, have been added, and the interface of operation `FindDevices` has been changed with three new elements added to the input, two added to the response data structure and some type changes to existing input elements.

To be concise, the type information associated with each input/response element is represented in the format of Erlang data types derived from the internal representation of the XSD schema. In Erlang, a complex data type is represented as a record, and a simple data type is represented as a type synonym using the `-type` declaration or an Erlang built-in type. There are certain restrictions on simple types, such as patterns, that cannot be encoded into Erlang data types in a straightforward way, and in this case we put the restrictions as comments to the base data type. As a by-product of

```
-module(weather_test).

-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").
-define(SUT, weather_sut).
-export([prop_state_machine/0]).  %% Prop
-export([initial_state/0, command/1, precondition/2,
         postcondition/3, next_state/3]).  %%eqc_callbacks.
-record(state, {}).

prop_state_machine() ->
  ?SETUP(fun setup/0,
    ?FORALL(Cmds, commands(?MODULE),
      begin
        {_H, _S, Res} = run_commands(?MODULE, Cmds),
        Res==ok
      end)).

initial_state()-> #state{}. %% abstract state representation.

command(S)-> oneof([
  {call, ?MODULE, get_weather, [gen_get_weather(S)]},
  {call, ?MODULE, get_cities_by_country, [gen_get_cities_by_country(S)]}]).

precondition(_S, {call,?MODULE,get_weather,[{_CityName,_CountryName}]})->
  true;
precondition(_S, {call,?MODULE,get_cities_by_country,[{_CountryName}]})->
  true.

postcondition(_S, {call, ?MODULE, get_weather,
                   [{_CityName, _CountryName}]}, Result)->
  Result == ok;
postcondition(_S, {call, ?MODULE, get_cities_by_country,
                   [{_CountryName}]}, Result)->
  Result == ok

next_state(S,_R,{call,?MODULE,get_weather,[{_CityName,_CountryName}]})->S;
next_state(S,_R,{call,?MODULE,get_cities_by_country,[{_CountryName}]})->S.

%% wrapper functions.
get_weather({CityName,CountryName})->?SUT:get_weather(CityName,CountryName).

get_cities_by_country({CountryName})->?SUT:get_cities_by_country(CountryName).

%% Data generators.
gen_get_weather(S)->{gen_city_name(S), gen_country_name(S)}.

gen_city_name(_S)->eqc_gen:oneof([none, gen_lib:string()]).

gen_country_name(_S)->eqc_gen:oneof([none, gen_lib:string()]).

gen_get_cities_by_country(_S)->{gen_country_name(_S)}.

setup() ->  %%utility functions.
  inets:start(),
  fun teardown/0.

teardown() -> inets:stop().
```

Figure 5. The automatically created test module for `GlobalWeather`

the change interface tool, we are able to generate the complete set of Erlang type definitions of a web service data model, and write them into a separate Erlang `.hrl` file. The Erlang representation of data types are not actually used by the test code, but it provides another way to view the data model of a web service. As an example, Fig 7 shows the definitions of some of the data types used in Fig 6.

```
[{api_added,
  {"FindAllRooms",[],      %% operation name
   [{room,[#room{}]},      %% input
    {errors,#errors{}}],   %% response
   "GET"}},                %% binding method
 {api_input_output_changed,
  %% orginal API interface
  {"FindDevices",
   [{startIndex,integer()},
    {count,integer()}],
   [{device,[#device{}]},
    {errors,#errors{}}],
   "GET"},
  %% new API interface
  {"FindDevices",
   [{startIndex,pos_integer()},
    {count,pos_integer()},
    {sortBy,none|string()},
    {order,none|orderType()},
    {'query',none|string()}],
   [{device,[#device{}]},
    {errors,#errors{}},
    {existsMore,boolean()},
    {countTotal,integer()}],
   "GET"},
  %% changes to input
  {[{input_type_changed,
     {startIndex,integer()},
     {startIndex,pos_integer()}},
    {input_type_changed,
     {count,integer()},
     {count,pos_integer()}},
    {input_added,{sortBy,none|string()}},
    {input_added,{order,none|orderType()}},
    {input_added,{'query',none|string()}}],
   %% changes to respsonse
   [{unchanged,{device,[#device{}]}},
    {unchanged,{errors,#errors{}}},
    {output_added,{existsMore,boolean()}},
    {output_added,{countTotal,integer()}}]}},
 {api_added,
  {"DeleteDevice",
   [{deviceId,none|nonempty_list(integer())}],
   [{device,[#deletedDevice{}]},
    {errors,#errors{}}],
   "GET"}}]}
```

Figure 6. An example web service change report

The final report is used by the framework to generate a refactoring script for evolving the test code. However one should note that not all the changes can be represented as refactorings. For example, the effect of changing an operation response data structure depends on how the data is processed in the abstract model, and is hard to be described as a refactoring, we hence leave this for the user to handle while the tool only points out the changes.

```
-record(room,
        {roomId::string(),
         description::none|string()
        }).

-type orderType()::'ascending'| 'decending'.

-record(error,
        {code::string(),
         params::none|#errorParams{},
         description::string()
        }).

-record(errors,
        {error::nonempty_list(#error{})
        }).
```

Figure 7. Some example Erlang representation of data types

6.2 Domain-specific Refactorings

Our manual study of the change log of a few web services led to a collection of refactorings, or program transformations, that could typically be applied to the test code. While the refactorings identified here only concern API interface changes, Wrangler's existing collection of built-in refactorings, both general-purpose and QuickChick-specific, can equally be applied to the test code in particular to support the evolution of the abstract model. Some of the refactorings identified, such as *renaming of function name*, *swapping function arguments* and *move functions from one module to another*, are already supported by Wrangler, but many of them are more domain-specific, hence not included in Wrangler's set of built-in refactorings. Nevertheless, with Wrangler's API and DSL support for user-extensibility, we were able to define and add these new refactorings to Wrangler without much hassle.

Our study also revealed the different characteristics between these domain-specific refactorings and traditional general-purpose refactorings. Unlike in traditional refactorings, where most of the changes are caused due to a *define-use* relation, the refactorings defined for evolving *eqc_statem* test code involve changes that are loosely related to each other due to the structure of the code, and changes to handle the coding idioms, such as symbolic function calls, used by QuickCheck.

Take the *add an input element to an operation* refactoring as an example, one might think that this should be similar to the general *add a parameter to a function* refactoring, however, it is not quite the case. The general purpose *add a parameter to a function* refactoring implemented in Wrangler first adds the new parameter to the function definition, then adds the atom `undefined` as a placeholder to the use-sites of this function, since the actual argument to be supplied cannot be decided by the refactoring tool in general; whereas for the domain-specific refactoring, we use domain knowledge to decide what the new argument should be, and where an argument should be added, etc.

Suppose a new parameter `PostCode` of base type `string` has been added as the first input element to the `GetWeather` operation of the `GlobalWeather` web service, we then need to refactor the test module shown in Figure 5 to reflect this change. Fig 8 shows the expected code after the refactoring where the changes are underlined. Since we use a tuple to group together the elements of a complex type, we actually need to add an element to the tuple of input elements. We could initiate this refactoring from the wrapper function `get_weather`, and add `PostCode` as the first element of the tuple argument, then the same variable name is added

to the call to the `get_weather` function defined in the connector module. Since the connector module is re-generated automatically, the use-sites of connector functions with interface changes outside the connector module have to be refactored separately as here. The new parameter also needs to be added to the use-sites of the wrapper function. Wrapper functions are used by callback functions `precondition`, `postcondition` and `next_state`; however what is different here is that the wrapper functions are used in the parameter parts of the three function definitions through a symbolic function call. Without domain knowledge, a refactoring tool would not be able to detect those change places.

Adding a new element to the input of a WS operation also affects the definition of existing data generators. However, since the definitions of data generators are likely to be refined by the user, and the code structure could become hard to analyse, we try not to refactor the definition of existing data generators. Instead, our tool generates a new file containing all the new data generators and the user could examine this file and move the generators needed to the test file using the *move function from one module to another* refactoring, or manually refactor some existing data generators to accommodate the changes. A data generator is treated as new if it is not defined, or its definition is different from the none derived from the old WSDL specification.

As to the implementation, most refactorings consist of a set of transformation rules to be applied to the AST representation of the test code in a particular way. Some refactorings also need code analysis to collect certain information about the code under refactoring, in which case Wrangler's templated-based information collection macros can be used.

A transformation rule defines a basic step in the transformation of a program; it involves recognising a program fragment to transform and constructing a new program fragment to replace the old one. In Wrangler, a transformation rule is denoted by a macro `?RULE` with the format of:

```
?RULE(Template, NewCode, Cond),
```

where `Template` is a template representing the kind of code fragment to search for; `Cond` is an Erlang expression that evaluates to `true` or `false`; and `NewCode` is an Erlang expression that returns the new code fragment in the format of a string. All the meta-variables/atoms declared in `Template` are visible to `NewCode` and `Cond`, and therefore can be referenced in defining them; furthermore, it is also possible for `NewCode` to define its own meta-variables to represent code fragments.

For instance, the code in Fig 9 defines a transformation rule that adds a new element `NewArg` at the `Nth` position of the input tuple of operation `Op` in the symbolic call to this operation. Variables ending with '`@`' are meta-variables that match a subtree in the AST, and variables ending with '`@@`' are *list meta-variables* that match a sequence of elements of the same sort. `_This@` denotes the complete subtree that pattern matches the template, which is denoted by the macro `?T`.

Apart from *add an input element to an operation*, other *eqc_statem*-specific refactorings added into Wrangler include:

Add an operation. This refactoring is applied when a new API operation has been added to the web service. It adds a new function clause to each of these three functions: `precondition/2`, `postcondition/3` and `next_state/3`, adds an element to the list of command generators used by the function `commands/1`, also adds a new wrapper function for invoking the connector function defined in the connector module.

Remove an operation. This refactoring is applied when an operation has been removed from the web service. It removes the function clauses handing this API operation from `precondition/2`,

```
rule(Op, NewArg, Nth) ->
  ?RULE(?T("{call, M@, F@, [{Args@@}]}"),
    begin
      Args1@@=lists:sublist(Args@@, Nth-1),
      Args2@@=lists:nthtail(Nth-1, Args@@),
      ?TO_AST("{call, M@, F@, [{Args1@@,"++
                              NewArg++
                              ", Args2@@}]}")
    end,
  ?PP(F@)==Op andalso
    api_refac:is_pattern(_This@)).
```

Figure 9. A transformation rule

`postcondition/3` and `next_state/3`, removes the command generator for this operation, as well as the wrapper function.

Rename an operation parameter. This is a general variable renaming refactoring, but needs to be applied to multiple functions/function clauses in the test code in order to keep the consistency of names across the test code. A composite refactoring that generates a collection of renaming refactorings has been implemented for this purpose. This composite refactoring dynamically search the test code for symbolic calls to the operation whose parameter has been renamed, and perform a variable renaming refactoring for each candidate found.

Remove an operation parameter. This refactoring is applied when an element has been removed from the inputs of an operation. The refactoring can also be initiated by removing the parameter from the wrapper function for this operation, but since the parameter is used in the call to the connector function, we have to force the removal of it. In the case that some of the parameters to be removed are used by the test model, this refactoring may result in code that does not compile, and we leave this for the user to fix.

These refactorings assume that the coding style used by the tool is preserved by the user. A different code style may necessitate changes in the refactoring implementation. For example, QuickCheck allows another way to implement the callback functions so that the pre-condition, post-condition and the state transition of an operation can be grouped into a single function definition. The current implementation of the refactorings does not work with this coding style yet.

6.3 Automatic Creation of Refactoring Scripts

When there are multiple API changes of a web service, a batch of refactorings may need to be applied to the test code. While it is possible to apply refactorings one by one, being able to apply refactorings in a batch mode is desirable, especially when the number of refactorings to be applied is large. Wrangler's DSL support for describing composite refactorings allows us to compose complex program transformations in order to carry out a large-scale batch refactoring in an efficient way.

With our test code evolution framework, we have gone one step further. Instead of requiring users to examine the interface changes and figure out what refactoring commands should be applied to the test code, our tool tries to analyse the changes and work out the refactoring commands needed automatically, and script them into a composite refactoring. In this way, we eliminate the need for a user to learn how to write a composite refactoring in Wrangler, but also allow the user to examine and change the refactoring commands before they are applied to the test code. The composite refactoring can be applied to the test code in an interactive way (indicated by the use of macro `interactive`), so that the user can still make

```erlang
-module(weather_test).

-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").
-define(SUT, weather_sut).
-export([prop_state_machine/0]).  %% Prop
-export([initial_state/0, command/1, precondition/2,
         postcondition/3, next_state/3]).  %%eqc_callbacks.
-record(state, ).

prop_state_machine() ->
  ?SETUP(fun setup/0,
    ?FORALL(Cmds, commands(?MODULE),
        begin
          _H, _S, Res = run_commands(?MODULE, Cmds),
          Res==ok
        end)).

initial_state()-> #state. %% abstract state representation.

command(S)-> oneof([
  call, ?MODULE, get_weather, [gen_get_weather(S)],
  call, ?MODULE, get_cities_by_country, [gen_get_cities_by_country(S)]]).

precondition(_S, {call,?MODULE,get_weather,[{_PostCode, _CityName,_CountryName}]})->
  true;
precondition(_S, {call,?MODULE,get_cities_by_country,[{_CountryName}]})->
  true.

postcondition(_S, {call, ?MODULE, get_weather,
                    [{_PostCode, _CityName, _CountryName}]}, Result)->
  Result == ok;
postcondition(_S, {call, ?MODULE, get_cities_by_country,
                    [{_CountryName}]}, Result)->
  Result == ok;

next_state(S,_R,{call,?MODULE,get_weather,[{_PostCode, _CityName,_CountryName}]})->S;
next_state(S,_R,{call,?MODULE,get_cities_by_country,[{_CountryName}]})->S.

%% wrapper functions.
get_weather({PostCode, CityName,CountryName})
      ->?SUT:get_weather(PostCode,CityName,CountryName).

get_cities_by_country({CountryName})->?SUT:get_cities_by_country(CountryName).

%% Data generators.
gen_get_weather(S)->{gen_post_code(S), gen_city_name(S), gen_country_name(S)}.

gen_post_code(_S) -> eqc_gen:oneof([none, gen_lib:string()]).

gen_city_name(_S)->eqc_gen:oneof([none, gen_lib:string()]).

gen_country_name(_S)->eqc_gen:oneof([none, gen_lib:string()]).

gen_get_cities_by_country(_S)->{gen_country_name(_S)}.

setup() ->  %%utility functions.
  inets:start(),
  fun teardown/0.

teardown() -> inets:stop().
```

Figure 8. Add parameter PostCode to the GetWeather operation

decisions as to whether to perform a particular refactoring or not during the actual execution of the refactoring script.

As an example, Fig 10 shows the composite refactoring generated from the change report shown in Fig 6. As it shows, five refactoring commands are generated, two for adding new operations, and three for adding parameters. The number used in the `refac_add_op_arg` commands indicates the position where the new parameter should be added. For instance '1' indicates that the new parameter should be added to the end of the parameter list, i.e. the first element in the tuple counting backwards.

7. Summary and Discussions

We have presented a framework for testing web services using property-based testing, in particular with the support of QuviQ QuickCheck, and for evolving test code to adopt changes made to web services between versions. With these tool support, we are able to automate the parts of the testing that are tedious to write manually, while let the user focus on defining the state-based test model, hence to improve the efficiency and productivity of testing.

Our property-based testing framework supports the fully automatic initial test code generation of web services. It verifies not only that the web service returns without raising an error message, but also that the data structure returned conforms to the data model specified in the schema. What is more important is that the use of abstract state machine gives the user the power to further develop the test model to improve the quality of the test in terms of both test case generation and test oracles, so that more serious testing can be done. This is especially important when the web service under test is a stateful web service, in which case the order in which the operations are executed may affect the result of the test, and the internal state has to be taken into account in the testing process.

Once an abstract model has been added to the test, the same test model can also be used to test other web services implementing the same WSDL specification.

The automatic inference of WSDL specification changes and the *eqc_statem*-specific refactorings help to reduce the effort of evolving test code in sync with web service evolution. Automatic refactoring reduces the chances of human errors being introduced. Our work also demonstrate the importance of making refactoring tools user-extensible, due to the fact that support for domain-specific refactorings in most general-purpose refactoring tools is very limited. Make refactoring tools user-extensible would potentially improve the adoption of refactoring tools in practice.

The framework presented is being used by Interoud Innovation (`http://interoud.com/`), a company based in Spain, to test their web service for managing streaming devices in difference locations. The design of the framework was also influenced by the way they test web services in practice using property-based testing.

8. Related Work

The two common test case generation methods used in web service testing are specification-based and model-based test data generation. Prior representing work on WSDL-based test case generation for web services testing includes the work of Bai et al. [7], Bartolini et al. [8], Lampropoulos and Sagonas [16], Zhang et al. [24], etc. Test data generation using WSDL definitions is limited to the datatypes due to the lack of behavioural information about the services. As a result, other alternative specifications that can provide more behaviour information have also been explored, such as contracted-based web service testing, where contracts define the conditions for a component to be accessed and the conditions that need to hold after the execution of methods of that component with the specified pre-condition. Some researchers proposed an extended WSDL that includes contract information for the service

and also a framework that uses this extended WSDL to test services [14, 21]. While our approach generates the initial data generators based on the static analysis of the WSDL specification, it is different from existing approaches in that it provides the infrastructure that allow users to refine the definition of data generators using the abstract state information. Our approach is different from contract-based testing in that pre- and post-conditions are specified in the test model, hence there is no need to extend the WSDL specification.

Test oracle generation is another major problem in web service testing. In [6], Atkinson et al. propose the use of a technique called *test sheet* in order to generate unit test cases and test oracles. Test sheets contain contract information which identifies the relation between the operations of a service, and the included relations define the effects of each operation from the clients perspective in order to help validation. Keckel and Lockmann [14] generate test oracle using pre-generated contracts which carry information such as pre- and post-conditions. In [23], Tsai et al. proposed an adaptive group testing technique that can test multiple web services that have the same business logic, internal states and input data; a voting mechanism is used in their approach to automate the generation of test oracle. Our approach automates the generation of generic test oracles that apply to all WS operations, such as verifying that the response data has a valid structure and is correctly typed, but also allows the specification of particular operation-related test oracles using the `post_condition` function associated with an operation.

Property-based testing of web services was first proposed by Zhang et al. In [24], the authors proposed a Haskell QuickCheck based framework for the automatic generation of test data and invocation of web services based on their WSDL specifications. Similar idea is also used by Lampropoulos and Sagonas [16] to carry out property-based testing using PropEr. Both approaches however suffer from the same limitations faced by specification-based approaches. Since PropEr also comes with support for state machines, and has a very similar user-interface to QuickCheck, the tools presented in this paper should work with PropEr as well with little modification.

There is much less work done to support the evolution of test code in sync with the evolution of a web service, although in the general context of software engineering, there have been developments of automated test-repair techniques, mostly focused on repairing unit tests [12] and GUI tests [10]. Our work on domain-specific refactorings and the automatic change inference reduces user's effort on evolving test code, and is also less error-prone than manual code manipulation and examination.

Finally this research was done within the context of the PROWESS project focusing on property-based testing of web services. Related work carried out in the project includes the domain specific language designed by López et. al [19] for writing QuickCheck data generators for web services using a syntax similar to the one used for defining WSDL elements, and the work for deriving QuickCheck models from a web service's WSDL description and its OCL (Object Constraint Language) semantic constraints [13].

9. Conclusions and Future Work

We have presented a framework for testing web services using stateful property-based testing, in particular with the support of QuickCheck *eqc_statem* library, and for evolving test code to adopt changes made to web services between versions. With tool support for the automatic generation of various components of the test code, the user only needs to focus on the part defining the logic of the abstract model. The framework presented can be used by both web service providers and consumers, and can be used to test web services written in whatever language; the approach advocated here

```
-module(refac_evolve_api).

-export([composite_refac/1, input_par_prompts/0, select_focus/1]).
-include_lib("wrangler/include/wrangler.hrl").
-behaviour(gen_composite_refac).

%% callback functions.
input_par_prompts() ->[].
select_focus(_Args) -> {ok, none}.

composite_refac(_Args=#args{current_file_name=File})
  ?interactive(
    [?refac_(refac_add_op,[File,"find_all_rooms",[],[File]]),
     ?refac_(refac_add_op_arg,[File,"find_devices",1,"SortBy",[File]]),
     ?refac_(refac_add_op_arg, [File,"find_devices",1,"Order",[File]]),
     ?refac_(refac_add_op_arg,[File,"find_devices",1,"Query",[File]]),
     ?refac_(refac_add_op, [File,"delete_device",["DeviceId"],[File]])]).
```

Figure 10. An example composite refactoring generated by the tool

could also be adopted in other property-based testing frameworks and refactoring tools.

One direction of our future work is to extend the framework to work with web service specifications written in other languages, such as JSON-based web services descriptions; another direction is to further facilitate the creation of the abstract model by learning from existing test cases. Integration with other tools developed within the PROWESS project within the framework is another goal of our work.

References

[1] Erlsom: An Erlang Libary for XML Parsing. http://sourceforge.net/projects/erlsom/.

[2] Levenshtein Distance. http://en.wikipedia.org/wiki/Levenshtein_distance.

[3] W3C XML Schema Definition Language (XSD). http://www.w3.org/TR/xmlschema11-1/, 2007.

[4] Web Services Description Language (WSDL) 2.0. http://www.w3.org/TR/wsdl20/, 2007.

[5] J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

[6] C. Atkinson, D. Brenner, G. Falcone, and M. Juhasz. Specifying high-assurance services. *IEEE Computer*, 41(8):64–71, 2008.

[7] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. Wsdl-based automatic test case generation for web services testing. In *Proceedings of the IEEE International Workshop*, SOSE'05, pages 215–220, Washington, DC, USA, 2005.

[8] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *Software Testing, Verification, and Validation, 2008 International Conference on*, Los Alamitos, CA, USA, 2009.

[9] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, Inc., 2009.

[10] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, New York, NY, USA, 2011.

[11] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.

[12] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *24th IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444, 2009.

[13] M. A. Francisco, M. López, H. Ferreiro, and L. M. Castro. Turning web services descriptions into quickcheck models for automatic testing. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, Erlang '13, pages 79–86, New York, NY, USA, 2013. ACM.

[14] R. Heckel and M. Lohmann. Towards contract-based testing of web services. *Electr. Notes Theor. Comput. Sci.*, 116:145–156, 2005.

[15] J. Hughes. QuickCheck testing for fun and profit. In *Ninth International Symposium on Practical Aspects of Declarative Languages(PADL)*, Berlin, Heidelberg, 2007.

[16] L. Lampropoulos and K. Sagonas. Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Service. In *The 8th International Workshop on Automated Specification and Verification of Web Systems*, Stockholm, Sweden, 2012.

[17] H. Li and S. Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK, 2011.

[18] H. Li and S. Thompson. A domain-specific language for scripting refactorings in erlang. In *15th International Conference on Fundamental Approaches to Software Engineering(FASE)*, pages 501–515, 2012.

[19] L. M. C. Macíias López, Henrique Ferreiro and T. Arts. A DSL for Web Services Automatic Test Data Generation. In *Draft Proceedings of the 25th International Symposium on Implementation and Application of Functional Languages*, 2013.

[20] E. A. Manolis Papadakis and K. Sagonas. PropEr: A QuickCheck-Inspired Property-Based Testing Tool for Erlang. http://proper.softlab.ntua.gr.

[21] H. Mei and L. Zhang. A framework for testing web services and its supporting tool. *IEEE Seventh International Symposium on Service-Oriented System Engineering*, pages 207–214, 2005.

[22] M. H. Mustafa Bozkurt and Y. Hassoun. Testing web services: A survey. Technical Report TR-10-01, Dept. of Computer Science, King's College London, 2010.

[23] W.-T. Tsai, Y. Chen, D. Zhang, and H. Huang. Voting multi-dimensional data with deviations for web services under group testing. *32nd International Conference on Distributed Computing Systems Workshops*, pages 65–71, 2005.

[24] Y. Zhang, W. Fu, and J. Qian. Automatic Testing of Web Services in Haskell Platform. *Journal of Computational Information Systems*, 2010.