Tool Demonstration: Testing JSON Web Services Using JSONGEN

Ignacio Ballesteros Luis Eduardo Bueso de Barrio
Lars-Åke Fredlund Julio Mariño
Escuela Técnica Superior de Ingenieros Informáticos, Universidad Politécnica de Madrid

This article describes a tool, jsongen, which permits testing behavioural aspects of web services that communicate using the JSON data format. Provided a characterisation of the JSON data as a JSON schema, the jsongen tool will: (i) automatically derive a QuickCheck (the property-based testing tool) generator which can generate an infinite number of random JSON values (according to the probability distribution implemented by the generator) that validate against the schema, and (ii) provides a generic QuickCheck state machine which is capable of following the (hyper)links documented in the JSON schema, to automatically explore the web service. The default behaviour of the state machine can be easily customized to include web service specific checks. The approach is demonstrated by applying it to the task of testing a simplified web service for banking.

1 Introduction

A large number of web services use the JSON (JavaScript Object Notation) [14] language for exchanging data between a server and its clients. Similar to the role of XML schemas for data in XML, JSON schemas provide typing for JSON formatted data. Although the JSON schema language has not been universally accepted, a fair number of tools are starting to appear that check conformance of JSON data against such schemes [7], revealing a growing interest in JSON based web services that also justifies providing a better support for testing them.

In this article we explain the design of the open source tool, *jsongen* [9], which provided a JSON schema describing the shape of JSON data, can automatically derive a QuickCheck [1] generator which produces random JSON data values which conform to that schema, and moreover derives a generic QuickCheck state machine which can, using the facility to automatically generate JSON data, explore the (web service) links documented in the JSON schema. The jsongen tool is implemented in the Erlang programming language, as can be noticed in code fragments in this article where the default behaviour of the jsongen tool is modified in order to be able to test accurately more complicated web service behaviours.

The most recent article on the jsongen tool was published in 2014 [4]. Since then we have focused on implementing new functionalities (e.g., support for HTTP headers, and handling application errors) and other improvements to enable the testing of more complex web services. In this tool demonstration article we focus not on explaining the foundations of the tool, as was done in previous articles, but rather, using a running example, we explain how to test a typical web service application using jsongen.

Related Work Tools that work with the JSON schema format are starting to appear, primarily of course parsing and validation tools. Tools similar to jsongen that address the generation of JSON data from schemas include Schematic Ipsum [6], json-schema-faker [8], and the json-schema-random tool [13]. The main feature of Schematic Ipsum is its capability of generating “more readable” random JSON data by using Wikipedia for string generation. However, the tool can handle only a small subset of JSON schemas compared to jsongen. The jsongen tool permits embedding quickcheck generators inside JSON schemas, thus also affording the possibility to generate non-random data. This can be used to, for example, provide a set of username and password pairs which are effective for a login procedure.

The json-schema-random tool, like jsongen, also generates instances of JSON data from schemas, but supports just a small subset of schemas (e.g. lacking support for references, and logical connectives).

For previous work on the automated and model-based testing of web services, the reader is referred to [5, 3, 2, 12] which mostly focus on the problem of test case generation from a semantic description of the service, often expressed in some extension of WSDL (WSDL-S, SWRL, or WSDL+OCL) and relying on XML schemas for data definition. Property-based testing and Erlang are already used in [5, 11]. Comparing our work with the article by Lampropoulos and Sagonas [11], for instance, the main difference is that in this work JSON is adressed instead of WSDL (and JSON schema instead of XML Schemas), and, more importantly, their work addresses only the problem of checking that type information is correct upon static web service invocation, whereas jsongen addresses the task of *exploring web services*. This permits to, in principle, check automatically and without requiring additional programming, dynamic and RESTful web services which generate web service links (URIs) upon invocation. In this respect, JSON schemas with links are much more expressive than WSDL descriptions.

The article [12] focuses on generating test properties from WSDL and XSD, and in contrast with our work addresses static web services only. However, the main focus of that work is an interesting one, i.e., how to effectively refactor test properties when a web service changes over time, addressing the classical but hard problem of how to keep specification up-to-date when implementations undergo change. In [10] the same authors address a different problem, the testing of RESTful web services. Clearly, when testing web services that conform to the RESTful paradigm, one should test the principal properties of RESTful services such as e.g. the idempotency of `GET` methods. Moreover, one should not have to explicitly specify what such standard properties mean in the test specification, but rather just declare a web service as being RESTful. The article [10] begins the process of providing good test support for checking RESTful specification, but, as the authors state, the approach is still not sufficiently automated. In jsongen, the proper support for checking pure RESTful web services (e.g., checking idempotency of `GET`) is an item for future work.

Article Organization Section 2 briefly describes the JSON data format, and the JSON schemas, followed by a short introduction to QuickCheck. For completeness, Section 3 discusses the automatic derivation of QuickCheck generators from JSON schema definitions. Section 4 introduces the running example in this article: a RESTful web service implementing a banking application. Section 5 describes the support for automatically exploring JSON web services using a generic QuickCheck state machine which follows (hyper) links that can be embedded in JSON schemas. The overall approach is illustrated in Section 6, by testing the conformance of the banking application JSON based web service against a service description specified using a JSON schema. Finally, in Section 7, we summarise the results and outline directions for future work.

2 Preliminaries

2.1 JSON and JSON Schemas

JSON is a simple readable data format providing a few basic value constructors: strings (`"hello"`), numbers (`2`), a null value (`null`), and booleans (`true`). The complex JSON values are arrays, e.g., `[" hello",2, null, true]`, and objects consisting of key-value pairs:

```
{"firstName": "John", "lastName": "Smith", "age": 25}
```

JSON Schemas – Types for JSON Data JSON schemas provide a typing mechanism for JSON data, permitting to characterise and name subsets of JSON values. JSON schemas are represented as objects, with a number of keys having special meanings. The definition of JSON schemas is available at [15]. Given a JSON value, a JSON schema is said to *validate* the value if the value conforms to the schema. For example, the JSON value `"hola"` is validated by the schema {`"type" :"string"`}, and the integer `2` is validated by {`"type" :"integer"`}. Note that a value is normally validated by many schemas; in particular, the empty schema {} validates any JSON value. An arbitrary array is validated by the JSON schema {`"type":"array"`} while the following JSON schema just validates arrays of integers:

```
{"type" : "array", "items": {"type" : "integer"}}
```

The JSON object above defining the data of a person is validated by the following JSON schema:

```
{
  "type" : "object", "required" : ["firstName", "lastName"],
  "properties" : {
    "firstName" : {"type" : "string"},
    "lastName" : {"type" : "string"},
    "age" : {"type" : "integer"}  }
}
```

Informally, the schema validates any object that has keys `"firstName"`, `"lastName"`, and, optionally, `" age"`. The `"firstName"` and `"lastName"` values must be strings, and the `"age"` value must be an integer.

2.2 Quviq QuickCheck

The basic functionality of Quviq QuickCheck [1] is simple: when supplied with an Erlang data term that encodes a boolean property, which may contain universally quantified variables, QuickCheck generates a random instantiation of the variables, and checks that the resulting boolean property is true. This procedure is by default repeated at most 100 times. If for some instantiation the property returns false, or a runtime exception occurs, an error has been found and testing terminates.

State Machines for Testing QuickCheck provides a state machine based library for testing APIs with side effects. The goal of the library is to enable a tester to easily generate random sequences of sensible calls to the API, and to decide if the execution of such a call sequence was successful (i.e., the API returned the expected results) or not. Technically, the tester, with help from the library, builds a state machine that serves as a model of the behaviour of the API, which is used both to: (i) generate individual tests which consists of sequences of commands, and (ii) check that the test execution results are correct.

3 Generation of JSON Data

To test a JSON based web service using QuickCheck test data must be generated. The approach taken in this paper is to use a tool, jsongen, which provided a description of the format of data as a JSON schema, automatically derives a QuickCheck generator. This QuickCheck generator can generate random JSON values matching the given JSON schema. In the following we illustrate the derivation process by considering a few key JSON schema constructs.

Basic Types The translation of many JSON schema constructs to QuickCheck generators is straightforward since the QuickCheck `eqc_gen` library provides similar mechanisms. For instance, a JSON integer is translatable to a QuickCheck generator of Erlang integers using the QuickCheck generators `int/0`, `nat/0` and `choose/2`. The JSON numbers, booleans and null values are treated similarly.

Strings The JSON strings can be constrained by specifying their minimum or maximum lengths (`min`- and `maxLength`), or by specifying a regular expression in the format of ECMA 262 (JavaScript) regular expressions; the jsongen tool is capable of translating such regular expressions into generators. The challenges involved in generating valid JSON strings from such string constraints are mainly to do with the mixing of global and local constraints. Consider for instance the following string specification:

```
{
  "type" : "string",
  "pattern" : "^[A-Z][a-z]{0,4}\.[a-z]{1,7}$",
  "minLength" : 10, "maxLength" : 15
}
```

Intuitively the schema defines the shape of strings to be composed of a first part beginning with an uppercase letter and followed by between 0 and 4 lowercase letters, a dot ".", and a second part having between 1 and 7 lowercase letters. Moreover, as a global constraint the total number of characters in the string must be between 10 and 15.

Taking the global constraints into account, it would be possible to calculate finer local constraints, e.g., that the first range (e.g. `{0,4}`) can be refined to `{1,4}` and that the second range can be refined to `{4,7}`. Moreover, the choice of a concrete bound for the first range has to guide the choice of the second bound (or the other way around). As a limitation, the current version of jsongen does not implement such calculations, but rather implements a two-step procedure whereby first a string is generated using the regular expression, and is then filtered using the global constraints. This can of course lead to inefficient (slow) data generation.

Objects The schemas for objects define which key-value pairs must be present (`required`), whether additional properties are permitted (`additionalProperties`), and type (schema) information for each property in the object provided using a `properties` declaration or, interestingly, using regular expressions to characterise permitted properties keys using the `patternProperties` property.

As a result of the interaction between the object schema properties, the validation rules are somewhat complex, and hence too are the rules for value generation. As an implementation illustration we consider the case where a schema specifies required properties, and specifies pattern properties, but forbids additional properties. An example of such a schema is depicted below.

```
{
 "type" : "object",
 "required" : ["firstName","lastName","age"],
 "additionalProperties" : false,
 "properties" : {
  "firstName" : {"type" : "string"},
  "lastName" : {"type" : "string"},
  "age" : {"type" : "integer"}
 }
 "patternProperties" : { "^car[1-9]$" : {"enum" : ["honda", "bmw"]} }
}
```

Any value generated from such a schema is required to have properties `firstName`, `lastName` and `age` and may have properties `carN` where `N` is a digit. The generator automatically derived from the schema by jsongen consequently first generates a random natural number greater or equal to 3 (the minimum number of properties) corresponding to the number of properties in the generated object, and proceeds to generate the required properties using the associated schema definitions. If additional properties are to be generated (the generated natural number is greater than 3), and since all the properties enumerated (in `properties`) are also required, the `patternProperties` specification is used to generate the additional properties, taking care to ensure that generated properties are unique.

As an example, the following is a valid object that might be generated by the QuickCheck generator automatically derived from the above schema:

```
 {
   "firstName": "John",
   "lastName": "Smith",
   "age": 2,
   "car7" : "bmw",
   "car2" : "honda"
 }
```

4 An Example Web Service: the Bank

The example we are going to use to illustrate the functionality of jsongen is a small web API for a banking application; the source code of the example is available at `https://github.com/ignaciobll/WebMachineApiExamples.git`. The banking API supports the following functionalities: (i) creating users, (ii) each user can create new bank accounts, and (iii) accounts have various operations associated with them. A `get` operation will return a JSON term with the information of the corresponding account. The `deposit` and `withdraw` operations introduce money into an account, and remove money from it. Both operations return a JSON term as the result body (representing the new state of the account) or an error message if something went wrong. Finally we have the `transfer` operation, that removes money from an account and moves it to another account. The result body of this operation is similar to the body returned by the `deposit` and `withdraw` operations. Apart from the (optional) result body, the operations return one of the following HTTP status codes:

- `200`, meaning that the operation did not modify any resource of the API, and that the operation was successful. In our example it is used in the only operation which does not modify the state of the resource, i.e., `get`.
- `201`: this status code is used when the operation is performed successfully, and when the state of the resource was modified.

- `304`: this status code should be coupled with an empty result body, indicating that something went wrong when processing the operation.
- `404`, also known as “not found” – this status code is sent in the response when the resource we are trying to access does not exist.

Table 1 shows the available operations, the corresponding URI (uniform resource identifier) “route”, and the possible return codes when invoking the operation.

HTTP Method	Route	Status Code
POST	`/auth/new`	`201, 304`
POST	`/bank/new`	`201, 304`
GET	`/bank/account`	`200, 404`
POST	`/bank/account/deposit`	`201, 404`
POST	`/bank/account/withdraw`	`201, 304, 404`
POST	`/bank/account/transfer`	`201, 304, 404`

Table 1: API operations

5 Using Links to Automatically Explore JSON Web Services

In addition to generating JSON data from JSON schemas, the jsongen tool is capable of *exploring* a web service, that is, systematically making calls to the web service guided by a JSON schema description. This functionality is implemented by a QuickCheck state machine, which, by default, principally checks that the (JSON) values returned by a call to a web service operation are indeed instances of the return type defined in the JSON schema. However, as we shall see, the underlying state machine can easily be customized to also check behavioural aspects of the web service under test.

Note that jsongen can be used both for testing traditional (non-REST) JSON-based web services, as well as for testing pure RESTful services that communicate using JSON.

In Section 5.1 we will first describe the means of annotating JSON schemas with details regarding access to web services; the bank example introduced in Section 4 will be used as a running example. Note that the open source repository `https://github.com/ignaciobll/WebMachineApiExamples.git` contains both the source code of an implementation of the bank example web service, and JSON schemas useful as inputs to the testing process. Next, Section 5.2 provides an overview of the design of the QuickCheck state machine, including how the resulting state machine can be specialized to test properties which cannot be expressed by a JSON schema. Finally, Section 6 describes the results of testing the simple bank example.

5.1 Addressing Web Services in JSON Schemas

The core JSON schema specification[1] simply provides a language for describing types of JSON data. However, an additional draft specification (*JSON Hyper-Schema*[2]) provides two additional major features: specification of additional non-JSON based “multimedia” data, and “link” objects which document hyperlinks between instances (or resources). In the following we focus on the links exclusively.

[1] `http://tools.ietf.org/html/draft-zyp-json-schema-04`

[2] `http://tools.ietf.org/html/draft-luff-json-hyper-schema-00`

In essence, a link describes how one can calculate the URI of another instance, which operations (PUT, GET, etc.) are supported by the instance, what parameters the operations expect, and what data the operations return.

To illustrate the JSON schema language, we use the bank web service. First of all we must decide which is our first operation. For the bank web service it is the operation of creating a new user. The schema for this operation is shown in Figure 1. The attributes used in this schema have the following

```
{
  "links": [ {
    "rel": "new_user",
    "href": "http://localhost:8080/auth/new",
    "method": "POST",
     "title": "new_user",
    "schema": {
      "type": "object",
      "required": ["user", "pass"],
      "properties":{
        "user": {"quickcheck": "bank_generators:user_generator"},
        "pass": {"quickcheck": "bank_generators:password_generator"}
      },
      "additionalProperties": false
    },
    "targetSchema":{"$ref":"new_user_response.jsch#", "error":[304]}
  } ]
}
```

Figure 1: JSON Schema definition for the `new_user` operation.

meaning:

links is a list with the specification of the different operations.

rel defines a name for the operation.

href is the URI that jsongen will use to invoke the operation.

method defines the type of HTTP header we use for our request (GET, PUT, POST, etc.).

title just defines the title jsongen will use to print the results of the tests.

schema defines the type of arguments expected by the operation. The jsongen tool can, if needed, automatically generate values of this type to perform syntactically correct operation calls.

targetSchema defines the type of values returned by the operation. Jsongen uses this declaration to check that the actual web service implementation returns well-typed results.

The value of the “targetSchema” attribute, for instance, contains a reference to the file that contains the JSON schema that defines, or validates, the body, but also an “error” attribute. This “error” attribute provides the ability to define error codes for responses that do not have a body (*304* in the schema). The value of the error attribute is a list, permitting to define multiple status codes for an operation. Note that sending the user and the password in the body of the response, as is done in Figure 1 (properties `user` and `pass`), is not considered a good web service design practice. In Section 5.1.1 we will explain how to instead use authentication headers for user authentification.

```
{
  "type": "object",
  "required": ["user", "pass"],
  "properties": { "user": {"type": "integer"},
                  "pass": {"type": "string"} },
  "adittionalProperties": false,
  "links": [ {
    "rel": "new_account",
    "href": "http://localhost:8080/bank/new",
    "method": "POST",
    "schema": { "type": "object", "additionalProperties": false },
    "title": "new_account",
    "targetSchema": { "$ref": "new\_account\_response.jsch#" },
    "isRelative": true
  } ]
}
```

Figure 2: JSON schema ”new_user_response.jsch”

Figure 2 defines the type of return values for a successful “new account” operation call. If we examine the schema carefully, we find a new attribute which was not used in the first schema, the “links” attribute. This attribute “unlocks” a list of new operations when the response is valid, in this schema just the operation “new account”. Next, in Figure 3, we define which operations can be called on accounts.

In the schema in Figure 3 we can use information from the previous response in order to generate new operations. For example, we can see that the `get` operation, has an “href” that depends on the “account” attribute in the body of the previous response. Another interesting item is that we use the same response schema `response.jsch` for all the operations. Let us examine the response schema (Figure 4) in further detail.

The response schema validates two different kinds of responses. The first one is the result of a correct operation. The second is the result of accessing a resource which is not found in the API.

5.1.1 Header Generation

The jsongen tool supports HTTP headers generation and their inclusion in a web service operation call, as a recent improvement. In our example we are going to use this feature to add authentication to our requests. That is, when an account is created, all the operations that will be performed on this account will have an HTTP header with the information of the user attempting to call the operation. We can also do negative testing to ensure that authentication is required, e.g., that when a user which has no permissions for an account but tries to perform some operation on it, a “not permitted” error is returned. In this paper we will focus on the positive testing aspect only, but note that is not a limitation of jsongen itself, as it is possible to, for example, (i) specify JSON schemas which larger (more permissive) types than the implementation permits, and (ii) specify that such typing violations should result in web service operation failures.

First we add a new attribute to the JSON Schema definition called “headers”, which contains the necessary information for its generation. This attribute will be a list with the information needed to generate the headers. This attribute can reference normal Erlang functions and/or *QuickCheck* generators that should be loaded before the testing process starts. Next we will proceed to test that our bank example

```
{
  "type": "object",
  "required": ["account", "balance"],
  "status": 201,
  "properties": {
    "account": { "type": "string" },
    "balance": { "type": "integer" }
  },
  "additionalProperties": false,
  "links": [
  {
    "rel": "get",
    "href": "http://localhost:8080/bank/{account}",
    "title": "get",
    "method": "GET",
    "targetSchema": { "$ref": "response.jsch#" }
  },
  {
    "rel": "deposit",
    "href": "http://localhost:8080/bank/{account}/deposit",
    "title": "deposit",
    "method": "POST",
    "schema": {
      "type": "object",
      "required": ["quantity"],
      "properties": {
        "quantity": { "type": "integer" }
      },
      "additionalProperties" : false
    },
    "targetSchema": { "$ref": "response.jsch#" }
  },
  {
    "rel": "withdraw",
    "href": "http://localhost:8080/bank/{account}/withdraw",
    "title": "withdraw",
    "method": "POST",
    "schema": {
      "type": "object",
      "required": ["quantity"],
      "properties": {
        "quantity": { "type": "integer" }
      },
      "additionalProperties" : false
    },
    "targetSchema": {
      "$ref": "response.jsch#",
      "error": [304]
    }
  },

  (...)
```

Figure 3: JSON schema ”new_account_response.jsch”.

```
{
  "oneOf": [
    {
      "type": "object", "required": ["account", "balance"],
      "properties": {
        "account":{"type":"string"}, "balance":{"type": "integer"}
      },
      "additionalProperties": false
    },
    {
      "type": "object", "status": 404, "required": ["error"],
      "properties": { "error": {"enum": ["Not found"]} },
      "additionalProperties": false
    }
  ]
}
```

Figure 4: JSON Schema definition for the response.

correctly implements authentication support with HTTP authentication headers. First of all we will define the JSON schema below:

```
"headers": [ {
  "quickcheck": "bank_generators:gen_auth_header",
  "params": ["{user}", "{account}"]
} ]
```

We can see how to write a schema definition for the authentication header generation. In this example there are two attributes: (i) "params" which defines a list with the parameters of the function generator, and (ii) "quickcheck" which defines the Erlang module and function of the generator, an example function is shown below.

```
gen_auth_header([User, Password]) ->
    {"Authorization", "Basic " ++
        base64:encode_to_string(<<User/binary, ":", Password/binary>>)}.
```

The above function declares a number of header parameters to the erlang tool, `httpc`, which is responsible for invoking the web service call (using these parameters).

Such schemas can be embedded inside a "links" attribute, as shown in Figure 5. By default, when an operation uses some headers generator, the operations that are "linked" from that operation (e.g., new_account from new_user) will use the same headers as the linking operation. This default behaviour was designed for the case when we use some kind of authentication in the headers, and we need the next operations to be also authenticated.

5.2 A Generic QuickCheck State Machine for Web Services Interaction

The jsongen tool provides a generic QuickCheck state machine, implemented by the Erlang module `js_links_machine` (henceforth abbreviated `jslm`), which given a set of JSON schemas, with link definitions, uses the information provided by the links to interact with the web service under test.

```
{
  "type": "object",
  "required": ["user", "pass"],
  "properties": { "user": { "type": "integer" },
                  "pass": { "type": "string" } },
  "additionalProperties": false,
  "links": [ {
    "rel": "new_account",
    "href": "http://localhost:8080/bank/new",
    "method": "POST",
    "headers": [ {
        "quickcheck": "bank_generators:gen_auth_header",
        "params": ["{user}", "{account}"]
    } ],
    "schema": {
      "type": "object",
      "additionalProperties": false
    },
    "title": "new_account",
    "targetSchema": { "$ref": "new_account_response.jsch#" },
    "isRelative": true
  } ]
}
```

Figure 5: JSON Schema definition for the “new_account” operation with the authentication header generation.

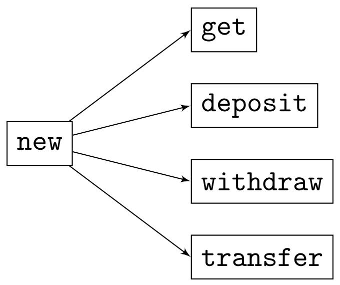

Figure 6: Discovery graph of requests.

The generic state machine is defined using the relatively new QuickCheck libraries `eqc_dynamic_cluster` and `eqc_component`. However, the APIs of these libraries are very similar to the normal `eqc_statem` state machine library. The main difference is that these new libraries enable a test case to be derived, step-by-step *during* test case execution, whereas the normal QuickCheck state machine library generates the whole test case *prior* to test case execution. As jsongen explores, in principle, web services with whose behaviour may not be fully known, the functionality of the new QuickCheck libraries is a good match.

The default actions of the generic state machine, to be described below, can be tailored by either modifying the set of JSON schemas, or, by providing a web service specific Erlang module which can modify the behaviour of the generic state machine, using a well-defined set of callbacks from the generic state machine to the Erlang module. This is similar to how many Erlang "components" are implemented, e.g., the generic server design pattern provided by the `gen_server` library.

State Machine State The basic state of the derived state machine is a set of URIs, represented by links, which have been revealed during testing. Initially, the only known links are the ones that represent URIs that do not depend on the object in which they reside (henceforth called "static links"). Such static links are, for instance, the `new_user` link (in Figure 1), but *not* `get`, `deposit`, `withdraw` or `transfer` link (in Figure 3), since to calculate its URI an account object is needed. The latter type of link will be referred to as a "dynamic link".

Thus, a state machine state is defined by the following Erlang record:

```
-record(state,{static_links,dynamic_links,private_state}).
```

The `private_state` represents state information that can be modified, if needed, by the web service specific Erlang module.

We can use "static" and "dynamic links" to automatically explore JSON services. With every `targetSchema` that we follow, new links can be added to the collection. This can be useful to create a dependency in our command sequence. For example, any test case for the "bank" service must first create an *account* and then start to manage *transfers* (if the account creation was successful). This can be seen as a discovery graph in Figure 6.

Generating Commands Next, we show the part of the code in `jslm` that implements the `command` function, which given a state, returns a QuickCheck generator which is called by the state machine library to compute the next command to execute. The function generates only a single type of command: an order to follow a link, represented by a call to the function `follow_link`.

```
command(State) ->
  Alternatives =
    [
     {call, ?MODULE, follow_link, [Link,gen_http_request(Link)]}
     || Link <-
          sets:to_list
          (sets:union
             (State#state.static_links,
              State#state.links)),
        link_permitted(State,Link)
    ],
  eqc_gen:oneof(Alternatives).
```

The code randomly selects any of the static or dynamic links in the machine state, when this is permitted by the function `link_permitted`, which by default always returns true. However, a more restrictive implementation of the function can be provided in the web service specific module. The function `gen_http_request` (not shown) constructs a http request using the link argument: first an URI (uniform resource identifier) is computed using the `href` attribute of the link, then the argument (e.g., a body for a PUT request) to the http request is generated as an instance of the schema in the link `schema` attribute (if present). Optionally, it is possible to define a special header for the request. This header can be defined inside a "link object".

Determining the Result of Executing a Command The `postcondition` function inspects the return value of a command, and judges whether the result was satisfactory or not. In our case this corresponds to inspecting the value returned from interacting with the web service. The generic `postcondition` function in `jslm` checks two things: (i) that the web service returned a result, and that HTTP result code from invoking the web service is admitted, and (ii) that the returned value (if in JSON format) is validated by (conforms to) the schema defined by the `targetSchema` property in the link definition. Admitted status codes can be defined on the JSON schema as seen in Figure 4 in the `"status":404` key-value pair. If no `"status"` key is provided, 200 is used as default.

The `postcondition` function first checks if the HTTP response has body. A response without a body may be the result of an error, and this error can be considered in the JSON schema as shown in Figure 3 inside the `targetSchema` object of the `withdraw` link. If the HTTP response has a body, it is validated that both the result code and the body, are correct.The validation is done by the "json-schema-validator" Java library (available at `https://github.com/java-json-tools/json-schema-validator`).

Computing the Next Test State The next test state of the state machine defined by the `jslm` module simply adds any dynamic new links discovered due to the invocation of a HTTP request, as shown in Figure 6. As an illustrative example, suppose that URI defined in the static link `new_user` (in Figure 1) is accessed successfully, a JSON object of an account is returned in the response body. Then, for this object, the `next_state` function adds the links found on the `new_user_response` (Figure 2) for this object response.

Thus, the exploration of the web service by the generic state machine is driven by acquiring more dynamic links. A test user can steer this process by omitting, or including, additional dynamic links in the schema definitions. In Figure 7 we can see how new "dynamic links" are discovered for our bank example. Every circular node is a response validated against its schema. Inside this schema there are defined some links to follow. These links are the rectangular nodes. This way we can discover every

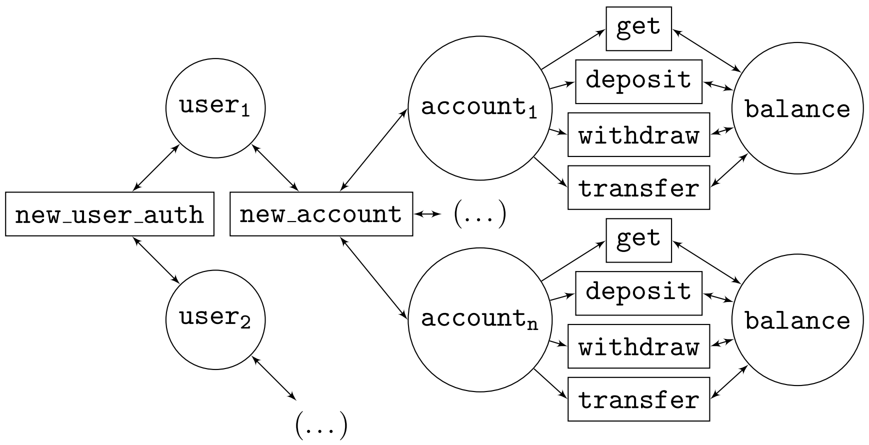

Figure 7: Explored states of the generic state machine.

"dynamic link" between every response. In this case, `new_user_auth` is a "static link" that every time that is called, generates a `user` object.

With every `new_user_auth` operation a new `user` is created. After that, the link for creating an account is discovered, so each user created can now create *n* accounts. The same process appears when a request to any account is made, but in this case, only a `balance` is returned and no new "dynamic links" are added to the state machine.

5.3 Defining a Web Service Specific Module

It is possible to tailor the generic state machine in two ways: (i) by modifying the JSON schemas, or (ii) by adding a web service specific Erlang module, which can modify the default behaviour of the generic state machine. The latter possibility is useful to, for instance, include behaviour checks that cannot be implemented in JSON schemas. The functions that can be tailored by the web service specific Erlang module include `initial_state`, `link_permitted`, `next_state` and `postcondition`, which were explained in the previous Section.

A Specific Module for the Bank Web Service JSON schemas can be used to check if a request was successful and whether returned values correspond to the types expected. This is useful to check if the operations are correctly typed, but cannot in general ensure that the web service behaves correctly. For instance, for the `transfer` operation which moves money between two accounts, it is important to ensure that the operation correctly updates the balances of both accounts. Such service properties can easily be checked using a web service specific module.

Clearly the private state (recall the definition of that state in Section 5.2) has to keep track of users and their accounts. Then, after calling a `transfer` operation, we will receive the new balance of the order (as part of the returned JSON object). Then, in the web service specific Erlang module, we can check that the returned balance is correct.

Another way of using this web service specific Erlang module might be to filter which links should be followed. For example, we can control the number of requests generated for each operation. In our bank service, for example, we would like to limit the number of accounts created for each test. By keeping track in the private state of the number of created users, the `link_permitted` function can prohibit the generation of new accounts if a limit has been reached.

6 Experimental Results: Testing the Bank Web Service

In this Section we show the results of testing the bank web service using jsongen. We have already seen the schema definitions for this service. With these schemas we can begin to test the bank service. The main function provided by the `jslm` module is `run_statem(Module, SchemaFiles, Options)`, with the three arguments: (1) the name of the optional web service specific Erlang module (as discussed in previous sections) which modifies the general state machine, (2) a set of JSON schema files providing the initial *static* links, and list of options:

- Default user and password, if they are needed to access the web service API.
- Timeout, if we want to specify the timing to wait for a response.
- Debugging options like `show_http_timing`, `simulation_mode`, `show_http_result` and `show_uri`.

In the following we will test the bank service by invoking the function variant `run_statem(SchemaFiles)` which omits the specific module, and the options.

When called, the `run_statem` function generates command sequences accessing the specified web service, executes them, and judges whether the execution was a success. Thus, to test the bank service we invoke the command below, which results in an error.

```
$ erl
1> jslm:run_statem(["new_user"]).
**************************************************
ERROR [postcondition error] [WRONG BODY]
the JSON value "account"
did not validate against the schema
{
  "type": "object",
  "required": ["account", "balance"],
  "properties": {
    "account": { "type": "string" },
    "balance": { "type": "integer"}
  },
  "additionalProperties": false
}
**************************************************
```

The error indicates that the bank web service returned a response which does not validate with the given schema. In this case, our implementation was returning an integer in the account attribute instead of a string, so the validation process failed. After fixing this and re-running our tests we obtain:

```
$ erl
1> jslm:run_statem(["new_user"]).
**************************************************
ERROR [postcondition error] [WRONG HTTP STATUS CODE]
for http call
http://127.0.0.1/bank/12/deposit using POST body={"quantity":1552}
the HTTP response code was: 404 but expected: 200
**************************************************
```

Here the result code was 404 instead of 200. This could be because the account creation is not working properly. After fixing it we finally get an API that perfectly matches our schema specification:

```
$ erl
1> jslm:run_statem(["new_user"]).
.......................................................
OK, passed 100 tests
PASSED

Link statistics:
-------------------
new: 1658 calls (55.54810593%)
get: 346 calls (8.246731478%)
deposit: 420 calls (14.07978545%)
withdraw: 342 calls (11.46496815%)
transfer: 217 calls (7.274555816%)
```

When the implementation passes all tests, jsongen shows some statistics. We can see the number of `new`, `get`, `deposit`, `withdraw` and `transfer` operations that jsongen called during the tests.

One of the problems found was the mutation of the state of the service outside the jsongen test suite. If we are keeping track of the internal state of the service inside the jsongen state machine, a request outside this machine that changes the state of the service may induce a failure in the `postcondition` of an operation.

7 Conclusions and Future Work

In this article we have provided support for the JSON Hyper-Schema draft specification in the jsongen tool. The means of support is through a generic QuickCheck state machine which given a set of JSON schemas with (hyper) links, explores the specified web service in a systematic manner. The exploration is tailorable by providing an Erlang module which redefines part of the behaviour of the generic state machine. The use of links to control exploration provides a clear and intuitive mechanism that users can use to tailor the exploration, adding new links to test further parts of the web service, or removing links, to focus testing on some essential functionality.

An item for future work is to extend the automatic checks done for pure RESTful web services. For instance, it should be possible to automatically check that a number of operations (e.g., GET) are idempotent.

Acknowledgment This work has been partially funded by the Spanish MINECO project TIN2012-39391-C04-02 *STRONGSOFT*, and the Madrid Regional Government grant S2013/ICE-2731 *N-GREENS*.

References

[1] Thomas Arts, John Hughes, Joakim Johansson & Ulf T. Wiger (2006): *Testing telecoms software with quviq QuickCheck*. In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, Portland, Oregon, USA, pp. 2–10.

[2] Xiaoying Bai, Wenli Dong, W.-T. Tsai & Yinong Chen (2005): *WSDL-based automatic test case generation for Web services testing*. In: *Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop*, pp. 207–212, doi:10.1109/SOSE.2005.43.

[3] Cesare Bartolini, Antonia Bertolino, Eda Marchetti & Andrea Polini (2009): *WS-TAXI: A WSDL-based Testing Tool for Web Services*. In: *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, IEEE Computer Society, Washington, DC, USA, pp. 326–335, doi:10.1109/ICST.2009.28.

[4] Clara Benac Earle, Lars-Åke Fredlund, Ángel Herranz-Nieva & Julio Mariño (2014): *Jsongen: a quickcheck based library for testing JSON web services*. In Laura M. Castro & Hans Svensson, editors: *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang, Gothenburg, Sweden, September 5, 2014*, ACM, pp. 33–41, doi:10.1145/2633448.2633454. Available at `http://doi.acm.org/10.1145/2633448.2633454`.

[5] Miguel A. Francisco, Macías López, Henrique Ferreiro & Laura M. Castro (2013): *Turning Web Services Descriptions into Quickcheck Models for Automatic Testing*. In: *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, ACM, New York, NY, USA, pp. 79–86, doi:10.1145/2505305.2505306.

[6] *Schematic Ipsum*. Available at `http://schematic-ipsum.herokuapp.com/`.

[7] *JSON Schema Erlang (jesse)*. Available at `https://github.com/klarna/jesse.git`.

[8] *JSON Schema Faker*. Available at `https://github.com/json-schema-faker/json-schema-faker`.

[9] *jsongen*. Available at `https://github.com/aherranz/jsongen.git`.

[10] Pablo Lamela Seijas, Huiqing Li & Simon Thompson (2013): *Towards Property-based Testing of RESTful Web Services*. In: *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, ACM, New York, NY, USA, pp. 77–78, doi:10.1145/2505305.2505317. Available at `http://doi.acm.org/10.1145/2505305.2505317`.

[11] Leonidas Lampropoulos & Konstantinos F. Sagonas (2012): *Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services*. In: *WWV'12*, pp. 3–16.

[12] Huiqing Li, Simon Thompson, Pablo Lamela Seijas & Miguel Angel Francisco (2014): *Automating Property-based Testing of Evolving Web Services*. In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, ACM, New York, NY, USA, pp. 169–180, doi:10.1145/2543728.2543741. Available at `http://doi.acm.org/10.1145/2543728.2543741`.

[13] *JSON Schema random*. Available at `https://github.com/andreineculau/json-schema-random.git`.

[14] *JavaScript Object Notation*. Available at `http://www.json.org/`.

[15] *JSON Schema*. Available at `json-schema.org`.