Connectedness Testing of RESTful Web-Services

Sujit Kumar Chakrabarti
Philips Healthcare, Bangalore
sujit.chakrabarti@philips.com

Reswin Rodriquez
Manipal Institute of Technology - Manipal,
reswin.rodriquez@gmail.com

ABSTRACT

In the context of RESTful web-services, *connectedness* refers to the property wherein every resource in the web-service is reachable from the base resource by successive HTTP GET requests. Presence (or absence) of connectedness has practical implications and hence is an important property of RESTful web-services. In this article, we present an algorithm for testing the connectedness of RESTful web-services. Using a formal specification of the web-service, our algorithm tests the connectedness of a web-service automatically. We also discuss a formal notation we have developed to specify RESTful web-services. Using our notation, we wrote the formal specification for a prototype RESTful web-service which has been developed for internal use in our organisation. Using this specification we employed our method to conduct automated testing of the above web-service. Many functional defects apart from those related to connectedness were detected early during development. This demonstrated that both our specification notation and our testing method are promising innovations in the direction of specification and testing of RESTful web-services.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Algorithms, Languages

Keywords

Representational State Transfer, Specification based Testing, Testing, Web-service

1. INTRODUCTION

Web-services are becoming popular in designing distributed software solutions due to the flexibility and simplicity of the World Wide Web (WWW). An inherent aspect of the WWW is its resource-orientedness. This means that the architecture of the Web and the recommendation of HTTP [11] – which is the ubiquitous application level communication protocol of the WWW – encourages that entities visible on the web are exposed as resources with their own universal resource identifiers (URI) to be accessed through the uniform interface of HTTP. Roy Fielding, in his PhD thesis [12], coined the term *representational state transfer* (REST) which is an architectural style for networked applications which resemble the WWW in its essential features, i.e. stateless communication, uniform interface etc. In the past few years, the idea of REST has been explored widely in the design of web-services. Since web-services use WWW and HTTP, REST has been found the architectural style of choice for many of them. Web-services implemented following the RESTful architectural style, along with some additional engineering constraints, are known to have a *resource oriented architecture* (ROA) [19].

Testing web-services presents its own set of issues. They are centred around the fact that web-services are by definition distributed, headless (i.e. lacking UI) and are based on late data binding. Since WWW is often used as the communication channel, it often becomes an problem to isolate issues specific to the WWW from those of the web-service under test by testing. RESTful web-services share these testing issues with traditional SOA based web-services. An additional issue lies with testing their RESTfulness, i.e. to test if they fulfil the characteristics of a RESTful web-services, e.g. stateless communication, uniform interface, layered architecture, caching etc.

In this article, we present a method of testing RESTful web-services focusing on testing a specific characteristic of RESTful web-services, called *connectedness*. The method uses a formal specification of the web-service to generate the test cases. We present portions of this specification notation relevant to the topic of this discussion in this article.

Following are the technical contributions of this paper:

1. A formal definition of connectedness property, which forms the basis of the ensuing discussion
2. An algorithm for testing connectedness of RESTful web-services
3. A graphical formal notation for specification of RESTful web-services

Apart from the above, we introduce an XML based format to represent the specifications written in the above notation

ISEC'10, February 25–27, 2010, Mysuru, Karnataka, India.

in a persistent manner. We have also implemented the testing method as a part of an in-house prototype tool and have demonstrated the usefulness of the same by testing a RESTful web-service developed developed within our group using our tool.

The organisation of the paper is as follows: In section 2, we present the necessary background about connectedness property. In section 3, we present the specification notation for RESTful web-services which we use in our method. In section 4, we describe our connectedness testing method, which is the main topic of this article. We have developed a prototype implementing our testing algorithm. In section 5, we present our experience in using this tool to test a prototype RESTful web-service developed in-house in our organisation. In section 6, we discuss some related work. We conclude with a summary and future work in section 7 .

2. BACKGROUND

2.1 Representational State Transfer (REST)

A summary of characteristics of a RESTful web-service are the following:

- **Resources**: All entities are exposed as resources.
    1. **URI**: Every resource has a URI. One resource may have more than one URI, but each URI maps to only one resource.
    2. **Connected**: The resources in the web-service should be reachable by doing successive HTTP GETs starting with the base resource(s).

  As an example, in the WWW, every webpage is a resource. It has a URL (URI) belonging to itself. The pages may physically exist as files in the server end or may be generated through a computation by the server in response to an appropriate request made on the URI.

- **Protocol:**
    1. **client-server**: The responsibilities of subsystems is clearly defined in terms of producers (servers) and consumers (clients).
    2. **stateless communication**: Each HTTP request contains all the information required by the server to serve that request. Failure to serve the request leaves the system in the initial (valid) state. Repetitions of the same request does not change the final (valid) state.
    3. **cacheable**: The client and server are capable of avoiding unnecessary traffic by detecting if the state of the resource has altered since the last read, thereby using a cached copy of it.
    4. **layered**: The system is logically arranged as layers of clients and servers in such a way that members of a layer communicate only with those of a neighbouring one.

- **Uniform interface**: All operations happening within the system are performed using a uniform set of operations whose semantics are well-understood within the implementation.
    1. **CRUD**: **C**reate, **R**ead, **U**pdate and **D**elete is a typical set of operations allowed in a simple RESTful service. More sophisticated applications would typically allowed additional HTTP methods.

  Four of the methods that HTTP defines each corresponds to one of the above operations. They are: POST for create, GET for read, PUT for update and DELETE for delete. A RESTful web-service would adhere to the above standard. For example, the only HTTP method that should be used when a resource is getting created, is POST. Similarly, GET should be used only for reading the contents of a resource. PUT should be used for making modification or updates on an existing resource. Finally, DELETE should be used when deleting or removing a resource. These are standards that should be followed as per REST. However, technology does not prevent one to deviate from this. For example, in a typical SOA*-style web-service, a client may well use a POST to request just the details about an existing resource. In this case, the client may send parts of its query in the HTTP request content. However, REST dictates that for reading a resource, only GET should be used. Every part of the query should be sent as part of the query-string, since, HTTP GET does not allow the request to have a content part.

The general import of the above guidelines promotes a simple, lightweight, modular and scalable architecture. As observed in practice, HTTP traffic in RESTful web-services tend to be lighter as compared with SOAP enveloped messages of SOA* web-services. Stateless messages wear well with unreliable communication channels like WWW since they tend to avoid harming the system when they fail to serve its functioning. Also, REST and SOA* are not entirely competing architectural choices. It is possible to implement RESTful web-services using SOA* technologies. As of now, certain important aspects like security are not completely worked out for REST implementations. Implementation of public web-services with security issues do not yet prefer REST. Similarly, development of other auxiliary technologies like description languages is still in its infancy in the case of REST.

2.2 Connectedness

There is a requirement of all RESTful web-service that they should be *connected*, or their resources should be *interconnected*. By this, it is informally understood that the web-service has a property such that resources should be accessible from the other resources. To access any resource in the web-service, it should not be required of the client to frame the URI of that resource explicitly by following any rule. Rather, the URI should be directly available in the response payload of some other *appropriate* resource.

The reader may think of web-services as web-sites and of resources as web-pages. The idea of connectedness then applies equally well to traditional web-sites (indeed, the traditional WWW is considered the original implementation of REST architectural style).

The above description of connectedness is informal. Hence, it is difficult to quantify and to verify. However, from practical perspective, connectedness is an important property of

web-applications. Following are some practical ramifications of connectedness:

1. **Usability:** It is not easy, and sometimes even impossible, for a client to frame the URI of a resource it wants to visit. A web-service which requires its clients to frame URIs of the required resources following some rule scores low in usability.
2. **Unlinked URIs and in-memory garbage:** Though, resources are more abstract than in-memory objects, existence of an unused (or unaccounted for) resource in the server will generally map to consumption of some physical resource. In that sense, URIs which have no link available to them, and have to be explicitly framed at the client side, are similar to garbage objects in memory in a traditional program run-time.
3. **Broken links and dangling pointers:** Presence of broken links is equivalent to dangling pointers in a traditional program run-time and may have similar consequences in the execution of an interactive use-case between the client and server.

To the best of our knowledge, there is no rigorous definition of connectedness available in the existing literature. Therefore, to form the basis of the ensuing discussion in this article, we define connectedness as follows:

> A web-service (or web-site) is connectedness if:
>
> 1. All its resources (except the base resource) are accessible by a sequence of HTTP GETs done on the URIs returned in the response payloads of other resources, starting with the base URI. For example, suppose that in a blogging web-service named `eblog.com`, there appears a list (supposedly exhaustive) of blogs owned by a user `abc` in his homepage `eblog.com/abc`. However, suppose that there exists a blog `B1` whose link does not appear in the above list, nor is a part of the response payload of any other resource in the web-service, then this web-service is not connected.
> 2. The response payload of no resource should have a URI of a resource which does not exist. For example, If there appears the list of all blogs owned by a user `abc` in his homepage `eblog.com/abc`, and suppose one URI appearing in the return payload is of a blog `B1` which existed in the past but now has been deleted, then this web-service is not connected.

Please note that we have kept the notion of *appropriateness* of referring resources out of our definition. Though, from a practical standpoint, it is an important property, it is hard to quantify 'appropriateness' of a link. Thus, not including it in the definition of connectedness simplifies the problem of automatic verification and testing of connectedness.

The remainder of this article presents our method of testing connectedness of a RESTful web-service. This method is specification based, i.e. the testing is guided by a formal specification of the web-service under test. We present the specification notation in the next section. The testing algorithm is presented in section 4.

3. WEB-SERVICE SPECIFICATION

In this section, we describe a simple visual notation to describe an aspect of a RESTful web-service relevant to the problem of connectedness. We have named the diagrams formed using this notation as *POST class graphs* and *POST object graphs* owing to their similarity with UML class diagrams and object diagrams respectively.

In the following discussion, we refer to two related terms, namely, *resource class* and *resource object*. Their relation to each other is exactly the same as that between classes and objects in the object- oriented world. A resource class is a type of resource, e.g. $User$, $Inbox$ etc. A resource object is an instance of a resource class. For example, multiple instances of $User$ resource class may exist in a web-service. Each of these are examples of resource objects, as the $Inbox$es belonging to each of them.

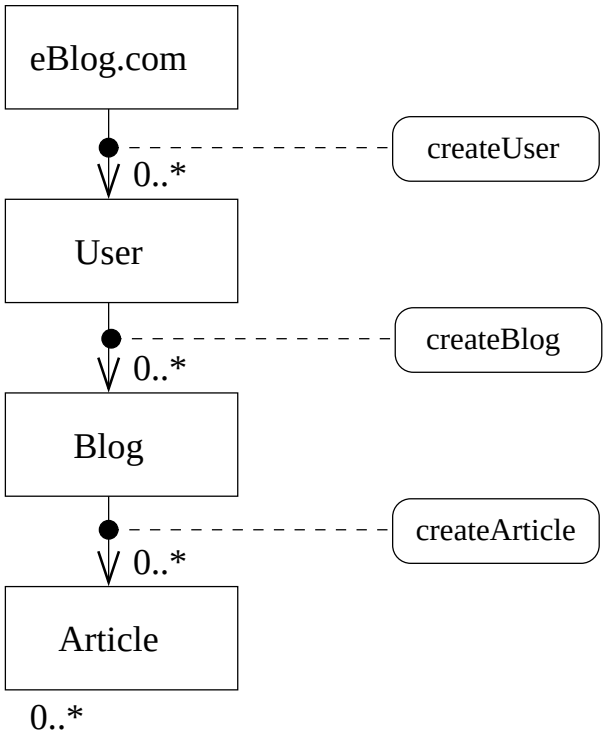
3.1 POST Class Graph (PCG)

A POST class graph is a directed graph whose nodes represent the *resources* (more specifically *resource classes*) and the edges represent HTTP POST operations. If there is an edge from a resource class A to a resource class B, it means that when there is an POST request done on the URI of an instance *resource object* of A, it will result in the creation of an instance of B. We say that A is the *source* of the given POST request and B is the $target$. Consider a graph shown in figure 1(a). This shows the POST class graph of a RESTful web-service by the name eBlog.com. The base resource class *eBlog* is the POST source of resource class $User$. The cardinality on target end means that a single *eBlog* resource object may be a POST source to zero or more $User$ resource objects. Similarly, the graph also says that one $User$ may be a POST source to zero or more *Blog* resource objects. Note that cardinality at the source of a POST edge is 1 by default since any resource can be created only once by only one other resource object. Hence, there is no need to mention the source end cardinality in a PCG.

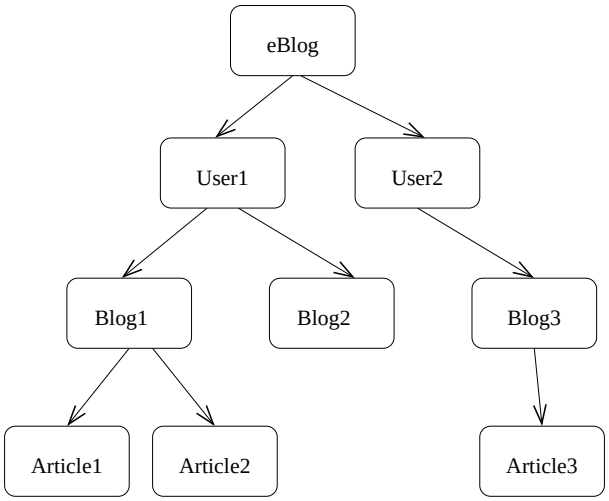
Figure 1(b) shows a POST object graph which is an instance of the POST class graph in figure 1(a). It shows that links of two $User$s are present in the root resource representation namely `User1` and `User2`. There are three *Blog*s: `Blog1` and `Blog2` being linked to by `User1`, `Blog3` being linked to by `User2`. `Blog1` provides links to two *Article*s: `Article1` and `Article2`; and `Blog3` provides links to `Article3`. `Blog2` is an empty blog. Note that while a PCG can be a directed graph of any shape, the POG is always a tree. This is because a resource object is created only once by a unique execution of a POST method. In the POG this translates to a node having a unique parent – the condition for a directed graph to be a tree.

3.2 Multi-POST

A single POST method call may result in the creation of one or more than one resource classes. In other words, a POST method may have one or more than one $target$s. If a POST has more than one targets, we call it a $multi-post$ and represent it as shown in figure 2. This example means that if the POST method shown in the diagram is called on base resource *eBlog* resource it will result in the creation of a $User$. Two other resource objects, $UserName$ correspond-



(a)



(b)

Figure 1: POST graph: (a) POST class graph; (b) POST object graph

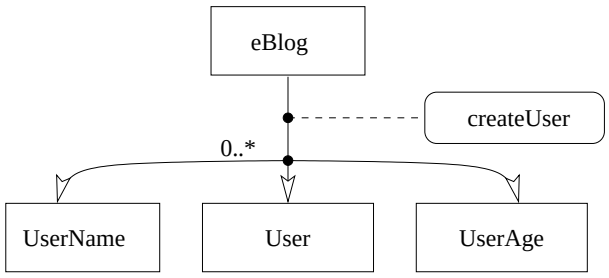


Figure 2: Multi-POST method for creation of $User$ resource

ing to the name of the $User$ just created and a $UserAge$ corresponding to its age, are also created. However, the relation between the simultaneously created resources is not a part of the information provided by this diagram.

3.3 Textual Annotations

POST class graphs are a partial specification of RESTful web-services describing which resource classes create which resource classes. To carry out any analysis of a PCG, either for verification or for test generation, we need some additional information. This additional information is presented as textual annotations which we present here.

Id: The identity of each resource class is denoted by its *id* attribute. This *id* is unique to each resource class.

URI: Each resource class contains a unique URI attribute which denotes its universal resource identifier.

Each HTTP method call is actually a pair of a *request* made by the client to the server, and a *response* from the server as a reply to the client's request. Our specification language allows us to specify what should go into the request and the response as defined below.

Request: Each edge in a PCG corresponds to a POST method. Each POST method (i.e. an edge in PCG) has a *request* attribute which corresponds to the HTTP request for the POST method call. Each *request* in turn contains the details about the *header* with each of its *field*s and their values, and the *body* of the *request*. The third part of the *request* is the list of *query_variable*s which represent the query parameter *name*s and their *value*s. For example,

```
request {
  query_variable {
    name = uname
    value = {username}
  query_variable {
    name = uage
    value = {userage}
  }
}
```

specifies a request that should contain a query-string constituted of two query-variables named `uname` and `uage`. The values that these query-variables get are from the named-values (explained below) `username` and `userage`.

Response: The second part of the annotations on a PCG edge is its *response* attribute. This in turn contains a *header* with its *field*s each with a *name* and a *value*. The response also contains a *representation* part which corresponds to the body of the HTTP response. For example,

```
response {
  header {
    field {
      name = Location
      value = http://eblog.com/{userid}
    }
```

```
postmethod createUser {
  cardinality = (0, *)
  request {
    query_variable {
      name = uname
      value = {username}
    query_variable {
      name = uage
      value = {userage}
    }
  }
  response {
    header {
      field {
        name = Location
        value = http://eblog.com/{userid}
      }
    }
  }

  target {
    id = User
    uri = http://eblog.com/{userid}
  }
  target {
    id = UserName
    uri = http://eblog.com/{userid}/name
  }
  target {
    id = UserAge
    uri = http://eblog.com/{userid}/age
  }
}
%  ...
%  ...
```

Figure 3: Textual annotation on `createUser` method

```
    }
```

specifies that the HTTP response to the corresponding method should contain a header field named `Location`, whose value is a parametrised string `"http://eblog.com/{userid}"`.

Target: A target in a POST method specifies the URI of the resource that gets created on its execution. The URI is represented as a parametrised string in terms of named-values in scope. *Target* can be used by the client to know the URI of the resource created. In testing, it allows us to create the corresponding resource object.

3.4 Parametrised Strings and Named Values

Many of the textual annotations on a PCG can be written in the form of parametrised strings. For example, the $Blog.URI$ = `"eblog/{userid}/{blogid}"`. Here, "`eblog/{userid}/{blogid}`" is a parametrised string because it has two *named values* as its parts, namely `{userid}` and `{blogid}`. A named parameter is similar to a variable in that it can be instantiated to a string literal value. For instance, if `{userid} = "abcd"` and `{blogid} = "xyz"`, the value of $Blog.URI$ becomes `"eblog/abcd/xyz"`.

3.5 Semantics

Consider the specification of resource classes as class definitions, and contained method as function prototypes to be used by a client. Here, a client could be a testing algorithm as the one presented in this paper, or a web-service client. In this subsection, we discuss, in the context of a client execution, how various parts of the specification are interpreted, what (if any) constraints govern the execution of its parts, and the semantics governing the use of the named values. In this, *scope* of a named value corresponds to the part of the specification where references to this value may be made. *Lifetime* of an instance of a named value refers to those contiguous stretches of time when this instance exists and may be used, during the execution of a client which uses this specification to define its interactions with the server.

A named value could either be a *parameter* or a *variable*. This distinction is not explicit in a PCG since we have no explicit declaration of named values. However, the usage of a named value determines whether it is a parameter or a variable. A named value which gets bound to a specific value immediately on its creation is a parameter, while one which gets bound subsequently is a variable.

The scope of a parameter spans across a resource and all its outgoing POST edges. Likewise, the lifetime of a parameter coincides with the lifetime of the resource object which contains it. A resource object must have a concrete URI. Consequently, When the first occurrence of a named value is in the URI of a resource, it is a parameter since it must be bound to a concrete value immediately on the creation of the resource object for the resource object to have concrete URI. If the first occurrence of a named value is in the POST method section of the resource class, it is treated as a variable. If the first occurrence of a variable is anywhere in the *request*, it is assumed that the variable is supplied by the client. If the first occurrence of a variable is in the *response*, then it is assumed that the variable gets its value from the server. The scope of variable is limited to the POST method that contains it. The lifetime of a variable instance is the same as that of the instance of the containing POST method.

In the above the term *first occurrence* assumes some kind of ordering among the textual description. This ordering can be understood as follows: When a resource object is created, its URI field is immediately created. Hence, any named value appearing in the URI must be upfront instantiated.

Inside a POST method, the *request* precedes the *response*. In both *request* and *response*, the *header* precedes the *body* (or *representation*). Among the *header fields*, there is no specific ordering. For our purpose (of test generation), we assume a lexical ordering among them as they appear in the representation file.

If there are multiple POST methods belonging to the same resource class, the ordering between their time of occurrence is not specified. For our purpose, we assume a lexical ordering among the POST methods as they appear in the representation file. A different scenario might exercise the POST methods in a different order. However, whatever be the scenario, all named values follow the rule of 'one definition multiple uses'. This means that the first occurrence of a named value defines its value. All other occurrences are its uses which use the same value as defined in the first occurrence.

An example of textual annotations pertaining to the `createUser` method of the PCG in figure 2 is given the figure 3. It shows the textual annotation of the `createUser` POST method. There are two portions named `request` and `response`. `request` contains two `query_variable`s named `{username}` and `{userage}`. `response` contains a header field named `location` whose value is a parametrised string `"http://eblog.com/{userid}"`. There are three named values in the textual annotation of `createUser` method, namely, `userid`, `username` and `userage`. All three are variables since their first occurrence is not in the `uri` attribute of the source

resource `eBlog`. `username` and `userage` appear first in the `query_variable` of the `request`. This means that their values will be provided by the user as query parameters during the creation of a `User` resource object. On the other hand, `userid` appears first in the `location` header field of the `response`. This means that it will be generated by the server and its value will come back as a part of the specified header field.

The named value `userid` appearing in the the `uri` attribute of three resource classes `User`, `UserAge` and `UserName` are all parameters since they appear in the `uri` attribute of the respective resource classes. They are also different from each other since the scope of a named value, whether a parameter or a variable, is limited to its containing resource class. When an instance object of `User` resource class is created, its `uri` attribute and hence the `userid` parameter also get instantiated. Same is the case with `UserAge` and `UserName`.

3.6 WADL++

Web Application Description Language (WADL) is an XML-based description format from Sun Microsystems. We have used the current specification of WADL as our starting point in defining a persistent format for writing PCGs. The reasons were:

1. WADL contains constructs for describing a RESTful web-service.
2. XML parsing is convenient as high quality off-the-shelf XML parsers are easily available.

However, WADL does not have constructs enough to convey all the ideas of PCG. Hence, we have customised (augmented) it to match the semantics of PCGs and have named this notation as WADL++. We will not go into the description of WADL++ here; but a sample WADL++ specification is shown in appendix A.

4. TESTING METHOD

The top-level view of our connectedness testing method is as follows:

1. Analyse the PCG and generate an appropriate server-side resource scenario and a POST object graph (POG).
2. Generate reference URI list from the POG.
3. Generate visited URI list by doing a DFS on the web-service.
4. Check if all the URIs appearing in the URI list formed in step 2 have been visited during step 3. If yes, declare test as PASSED. Else, declare as FAILED.

The above scheme is diagrammatically depicted in figure 4. In the rest of this section we present the steps of this algorithm in detail.

4.1 Generation of POG and Server Side Resource Configuration

As shown in figure 4, this step has two outputs:

1. Server-side resource configuration
2. Client-side POST object graph

It is the idea of this step to generate an appropriate server-side resource configuration. The POG on the client side is generated in parallel to keep track of all the resource objects generated and also for the proper execution of this step. As will be explained below, the algorithm makes use of the data in the already created objects while generating new ones.

The POG is created in a recursive manner at the client end. The creation of each POG node happens immediately after the creation of the corresponding resource object at the server end. The creation of a POG node completes with a subsequent creation of nodes corresponding to the POST targets of its resource class in the PCG.

Consider the POST method shown in figure 2 whose textual annotation is shown in figure 3. The algorithm will start by creating a node corresponding to the base resource object of `eBlog` class. It is assumed that the corresponding server-side resource object already exists. In this case, the corresponding resource class `eBlog` has only one POST method: `createUser`. The algorithm creates an appropriate HTTP POST request and sends it to the URI of `http://eblog.com`, i.e. the URI of `eBlog`. Salient points of a typical request composed by the algorithm is shown below:

```
URI: http://eblog.com?uname=132434&uage=4324
```

The values of the query variable `uname` is `132434` which is randomly generated by the algorithm. Same is the case with the second query variable `uage`. Simultaneously the two variables of the current POG node `username` and `userage` get bound to values `132434` and `4324` respectively.

The salient points of the HTTP response that may come back to this request are shown below:

```
Location: http://eblog.com/45342
```

The algorithm matches the string `http://eblog.com/45342` coming back as the value of the `Location` header field with `"http://eblog.com/{userid}"` which is specified in the PCG as the value of the `Location` header field in the response of this POST method. This step results in the node variable `{userid}` getting bound the value `45342`. Consequent to this, the algorithm expects three resources to get created on the server end whose URIs are:

```
http://eblog.com/45342
http://eblog.com/45342/name
http://eblog.com/45342/age
```

As the cardinality of this method is `(0, *)`, it means that the `eBlog` resource object is allowed to create 0 or more `user` resource objects. The algorithm assumes an appropriate value for `*`, e.g. 5, and repeats the above step so many times. This results in the creation of 5 `user` resource objects at the server end (along with 5 instances of `username` and `userage` each), and 15 nodes get added as children to the `eBlog` node(5 of types `user`, `username`, `userage` each) in the client side POG.

Following this, the algorithm descends to the next level of the POG and repeats the above. Since, the resource classes `username` and `userage` do not have any POST methods in the PCG, hence nothing will happen when the algorithm visits nodes corresponding to them in the next level. However, when it visits an instance of `user`, it detects that its resource class node has a POST method defined for it. Therefore, the above step repeats for this node involving sending of POST method, creation of the corresponding POG node, and exploration of the PCG in depth first manner along that path.

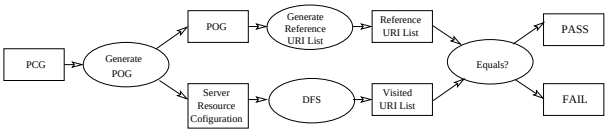


Figure 4: Connectedness Testing Method Schematic

The algorithm traverses each POST method only once (multiplied by its cardinality as defined in the PCG). Since, the POST methods are represented as edges in the PCG, the coverage criterion we employ here is *edge coverage* on the PCG.

4.2 Generation of Reference URI List

By *reference URI list* we mean a list of URIs whose corresponding resource objects have been created in the step 1. For the web-service to be connected, it is necessary that it is possible to visit all the URIs in this list starting at the base URI. The reference URI list is created by simply collecting in a list the URIs of all the resource objects created in the step 1.

4.3 Depth-First Search on the Web-Service

Do a DFS on the web-service in the following manner: Starting by making a GET request on the base URI. Extract all the URIs that appear in the response payload. Repeat the current step on all these URIs in a depth first manner. Continue until no more unvisited URIs are being visited or a maximum limit of search is reached.

4.4 Test Verdict

If all the resource objects created in step 1 are reachable from the base URI by doing the DFS of step 3, the test verdict is PASS. Otherwise, it is FAIL. The decision is made by simply checking the equality of the reference URI list (step 2) and the visited URI list (step 3). Equality means PASS; else FAIL.

5. EXPERIENCE

We developed a prototype tool implementing our connected testing algorithm. We used this tool to test a web-service developed within our team.

The web-service we tested was a prototype of a blogging service, named eBlog, for sharing project specific information amongst project team members. The PCG of this web-service is the same as in figure 1(a). In the interest of space, we present only a portion of the textual annotation for this PCG in figure 2 which shows the textual annotation for a POST method named `createUser`.

The POG generated from the execution of step 1 of the algorithm is shown in figure 5. We have used different shapes to denote resource objects belonging to different resource classes (circle for the base resource eblog; rectangle for *User*; ellipse for *Blog* and hexagon for *Article*). Please note that where the specification says that a resource A may create any number of instances of B (e.g. in `createUser` method, `cardinality=(0, *)`), our algorithm creates only 5 instances of B (e.g. 5 users are created in the POG shown denoted by 5 rectangles).

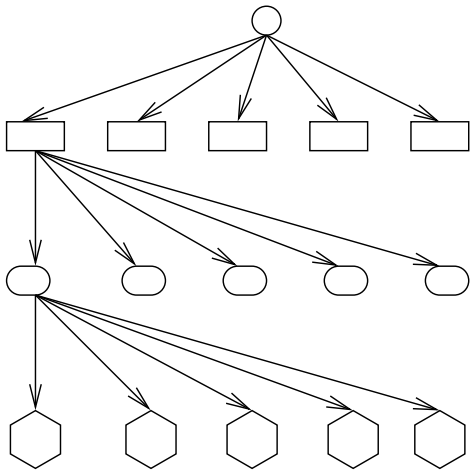


Figure 5: POG generated after the execution of step 1 of the algorithm

```
http://localhost/eblog/
http://localhost/eblog/user/39667
http://localhost/eblog/user/39667/blog/46637
http://localhost/eblog/user/39667/blog/46637/article/5804
http://localhost/eblog/user/39667/blog/46637/article/8711
http://localhost/eblog/user/39667/blog/46637/article/28461
http://localhost/eblog/user/39667/blog/46637/article/592
http://localhost/eblog/user/39667/blog/46637/article/12590
http://localhost/eblog/user/39667/blog/4654
http://localhost/eblog/user/39667/blog/1563
http://localhost/eblog/user/39667/blog/18405
http://localhost/eblog/user/39667/blog/15314
http://localhost/eblog/user/10469
http://localhost/eblog/user/32156
http://localhost/eblog/user/7378
http://localhost/eblog/user/2534
```

Figure 6: Reference list generated after execution of step 2 of the algorithm

The reference list is nothing but the list of all the URIs present in the server after execution of step 1. This reference list is presented in figure 6.

After a number of build-test-fix cycles, we arrived at a fully connected version of eBlog. There were many functional issues with the initial implementation. Here are some examples of the defects that were uncovered and corrected:

- None of the POST methods was returning a `Location` header field in the response.
- The URIs returned in the body were relative to the base URI while the specification said that they would be full URIs.

Our tool raised an exception in all such cases exactly at the point where the fault occurred resulting in a clear fault identification. Finally, once these faults were removed, there were many connectedness bugs which got caught after subsequent testing. Some examples are:

- While the specification mentioned `http://localhost/eblog` as the base URI, the list of all the users was actually provided in a *user list* page with a URI `http://localhost/eblog/users/`. This was fixed by adding a link to user list page in the eblog. The corresponding update was made in the specification.
- The specification did not have any mention of following resources: *blog list* (containing links to the blogs belonging to a user) and *article list* (containing links to the articles belonging to a blog). The problem got resolved by adding a link to the blog list resource in the respective user resource and by adding a link to the article list resource in the respective blog. The POST methods `createUser` and `createBlog` were made multi-POSTs in the specification.

Our overall experience from using our testing method (via our prototype tool) was that following the three-step process of *specify-implement-test* while developing a RESTful web-service yielded early benefits. Since the output of web-service calls are supposed to be machine consumable, it important that the implementation precisely follows a specification. Having a formal specification of the web-service ready in the beginning made it possible to initiate testing early during implementation phase. Early testing helped uncover both implementation and requirement errors early.

Currently, we are engaged in scaling up our prototype implementation to test an industrial strength RESTful web-service that is a part of a large healthcare informatic infrastructure within our organisation. This web-service is in its early stages of development. We are partnering with the development team in writing a formal specification of this web-service, and testing the same with our tool.

6. RELATED WORK

The issue of broken links in websites is well-known to the researchers and practitioners for long. Consequently, there are many tools available that are able to detect and report if there exists a broken link in a website [2, 5, 6]. They are based on crawling through the website, and checking for 404 (resource not found) response. This problem is closely related to the problem of connectedness; in fact it could be cited as a special case of connectedness. However the approach to solving the problem in the context of websites presents difficult issues due to the fact that much of the payload in websites are natural language content. This makes it difficult to employ rigorous specification methods to websites. Moreover, majority of interactive websites maintaining dynamic content do not strictly follow the CRUD directives. For example, most HTML forms in the current day implement all their user commands through HTTP POST method irrespective of whether they create, read, update or delete server resources. Put and delete are seldom used. On the other hand, web-service payloads are meant to be machine consumable by definition. Hence, to describe them in rigorous specification languages with strict semantics is possible. This also opens doors to automating the verification and validation of web-services based on their formal specification. The work presented in this article is an instance of how such an automation can be achieved.

In the direction of formalising specification of web-services, industry standards like WSDL [21] and WADL [20] are XML based and semi-formal. Specification languages coming out of academia tend to be based on formal methods [18, 8]. While the former set lacks formalism and rigour, the latter is hard for most of the software engineering community to use in practice. The specification language presented in this paper aims to use an industry oriented standard (WADL) by incorporating rigour to its semantics. The test generation algorithm presented in this article provides a proof point of the usefulness of this specification language in both specification and automated testing of RESTful web-services.

The interest in generating tests from diagrammatic specifications like UML has been there for long. However, emphasis has been on using behavioural diagrams like statecharts [14] , rather than using graphical structural specifications [10]. Our work uses a specification as input to the test generation algorithm which is structural in nature in the manner of UML class diagrams.

Finally, research in web-service testing has focused on the traditional SOA*-based web-services and technologies centred around that [17, 13, 16, 15]. Though, there are tools available for HTTP testing and debugging [4, 3, 7, 1], these tools are based on sending HTTP requests and checking the response for correctness as per some criterion. In [9], we have presented Test the REST, a framework for functional testing RESTful web-services. These tools provide automation in test execution. But none of them are based on automatic test generation. In particular, none of them has even an implicit support to test connectedness. Popularity of REST as an implementation method for web-services is relatively new. There are some tools already available for testing RESTful web-services. Prior to this, we are not aware of the existence of much research activity in this space. The work presented in this article builds upon [9] (thus inheriting its functional testing capabilities). It extends the same by adding connectedness testing and automatic specification based test generation to the capabilities of TTR.

7. CONCLUSIONS

In this paper, we have presented a specification based method of testing connectedness of a RESTful web-service. Connectedness, though informally understood, and well-accepted as an important property of web-applications, has no formal definition in the literature to the best of our knowledge. We

have provided a formal definition of connectedness in this article. We have presented a simple graphical notation to specify RESTful web-services. This notation is easy to understand and has a well-defined semantics. It also is equivalent to WADL++, an XML based format we have developed. WADL++ is an enhancement of WADL, which is becoming a standard for describing RESTful web-services. The algorithm we have presented in this paper takes as an input a specification written in the notation presented in this article (actually WADL++) and automatically tests the connectedness of the web-service. It is to be noted that the specification language provided has enough constructs to provide a rich specification of the functional features of a RESTful web-service. Our algorithm uses the same to perform automated functional testing of the web-service under test. The algorithm automatically detects it if the implementation deviates from the functional aspects of the input specification.

The algorithm presented in this article tests an aspect of connectedness where resource created are reachable from the base URI. However, for complete testing of connectedness, it is also important to test that deletion of certain resources does not result in creation of orphaned resources or broken links. In our ongoing research, we are covering this aspect of connectedness testing.

Acknowledgement

We cordially thank our colleague Vijaykumar for lending us his implementation of the eBlog service prototype which we tested using our algorithm.

8. REFERENCES

[1] Apache JMeter, http://jakarta.apache.org/jmeter/ [accessed, September 11, 2009].

[2] Broken link checker, http://www.iwebtool.com/broken_link_checker.

[3] eviware soapUI, http://www.soapui.org/ [accessed, September 11, 2009].

[4] Fiddler http debugger, http://www.fiddlertool.com/ [accessed, September 11, 2009].

[5] Link checker pro, `http://www.link-checker-pro.com`.

[6] Linktiger, `http://www.linktiger.com/`.

[7] Webinject - web/http test tool, http://www.webinject.org/ [accessed, September 11, 2009].

[8] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–272, New York, NY, USA, 2008. ACM.

[9] S. Chakrabarti and P. Kumar. Test-the-REST, An Approach to Testing RESTful Web-services. In *To appear in the Proceedings of IARIA Service Computation*, 2009.

[10] T. T. Dinh-Trong, S. Ghosh, and R. B. France. A Systematic Approach to Generate Inputs to Test UML Design Models. *Software Reliability Engineering, International Symposium on*, 0:95–104, 2006.

[11] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Technical report, 1998.

[12] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. Chair-Taylor, Richard N.

[13] R. Heckel and M. Lohmann. Towards contract-based testing of web services. *Electronic Notes in Theoretical Computer Science*, 82:2003, 2004.

[14] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic test generation from statecharts using model checking, 2001.

[15] H. Huang, W.-T. Tsai, R. Paul, and Y. Chen. Automated model checking and testing for composite web services. In *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 300–307, Washington, DC, USA, 2005. IEEE Computer Society.

[16] E. Martin, S. Basu, and T. Xie. Websob: A tool for robustness testing of web services. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 65–66, Washington, DC, USA, 2007. IEEE Computer Society.

[17] P. Mayer and D. Lübke. Towards a bpel unit testing framework. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 33–42, New York, NY, USA, 2006. ACM.

[18] F. Raimondi, J. Skene, and W. Emmerich. Efficient Online Monitoring of Web-service SLAs. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 170–180, New York, NY, USA, 2008. ACM.

[19] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.

[20] W3C. Web application description language, `https://wadl.dev.java.net/wadl20090202.pdf`.

[21] W3C. Web services description language, `http://www.w3.org/tr/wsdl`.

APPENDIX

A. WADL++ – EXAMPLE

```
<?xml version="1.0"?>
<application xmlns:xsd=
"http://www.w3.org/2001/XMLSchema"
  xmlns:html="http://www.w3.org/1999/xhtml">
  <resources base="http://161.85.98.22:8182/
  eblog">
    <resource path ="http://161.85.98.22:
    8182/eblog/"
    id="#eblog">
      <method href ="#Createuser"> </method>
    </resource>
    <resource path="http://161.85.98.22:
    8182/eblog/
    user/{userid}" id = "#user">
      <method href ="#Createblog">
      </method>
    </resource>
    <resource
    path="http://161.85.98.22:8182/eblog/
    user/{userid}/blog/{blogid}" id =
    "#blog">
      <method href ="#Createarticle">
      </method>
    </resource>
    <resource path="http://161.85.98.22:\
    8182/eblog/\
```

```
    user/{userid}/blog/{blogid}/article/\
    {articleid}" id = "#article">
    </resource>
  </resources>
 <!-- Methods-->
  <method name="POST" id="#Createuser"
  mincardinality ="0" maxcardinality ="*">
    <target  id="#user"
    uri="http://161.85.98.22:8182/eblog/
    user/{userid}"/>
    <request>
      <query_variable name="name" type=
      "xsd:string" value ="{name}"
      required="true"/>
      <query_variable name="age" type=
      "xsd:string" value ="{age}"
      required="true"/>
    </request>
    <response>
      <header>
        <field name = "Location"
        value = "/user/{userid}"/>
      </header>
    </response>
  </method>
  <method name="POST" id="#Createblog"
  mincardinality ="0" maxcardinality ="*">
    <target id="#blog"
    uri="http://161.85.98.22:8182/
    eblog/user/{userid}/blog/{blogid}"/>
    <request>
      <query_variable name="user" type=
      "xsd:string" value ="{userid}"
      required="true"/>
      <query_variable name="title" type=
      "xsd:string" value ="{blogtitle}"
      required="true"/>
    </request>
    <response>
      <header>
        <field name = "Location"
        value = "/user/{userid}/blog/
        {blogid}"/>
      </header>
    </response>
  </method>

  <method name="POST" id="#Createarticle"
  mincardinality ="0" maxcardinality ="*">
    <target id="#article" uri =
    "http://161.85.98.22:8182/eblog/user/\
    {userid}/blog/{blogid}/article/\
    {articleid}"/>
    <request>
      <query_variable name="user" type=
      "xsd:string" value ="{userid}"
      required="true"/>
      <query_variable name="blog" type=
      "xsd:string" value ="{blogid}"
      required="true"/>
      <query_variable name="title"
      type="xsd:string"
      value ="{articletitle}" required=
      "true"/>
      <query_variable name="content"
      type="xsd:string"
      value ="{articlecontent}"
      required="true"/>
    </request>
    <response>
      <header>
        <field name = "Location"
        value = "/user/{userid}/blog/\
        {blogid}/article/
        {articleid}"/>
      </header>
    </response>
  </method>
</application>
```