Finding Optimal REST Service Oracle Based on Hierarchical REST Chart

Li Li, Wu Chou
Shannon IT Lab
Huawei
Bridgewater, New Jersey, USA
{li.nj.li, wu.chou}@huawei.com

***Abstract*—Based on the hypertext-driven nature of REST API, this paper presents a structural approach for REST client design and implementation, in which a REST client is decomposed into two reusable functional modules: a client oracle that selects hyperlinks to follow for a given goal, and a client agent that carries out the interaction as instructed by the oracle. This decomposition has several advantages over a monolithic REST client where the two functions are intertwined and inseparable. To automatically find an optimal client oracle from a machine-readable description of a REST API, we introduce the path selection framework and apply Dijkstra's Shortest Path algorithm to Hierarchical REST Chart, which is an enhancement and extension to the original REST Chart that describes REST API based on Colored Petri-Net. The proposed method has been implemented in Java and tested on two sets of Hierarchical REST Charts. Experimental results indicate that the proposed approach is effective and promising.**

Keywords—REST API; Hierarchical REST Chart; REST Oracle; Petri-Net; Shortest Path

I. INTRODUCTION

In recent years, the REST architectural style [1] has become increasingly popular, and it has been applied widely to API designs in various areas, including Real-Time Communications [2][3], Cloud Computing [4], and Software-Defined Networking [5]. It provides an efficient and flexible way to access and integrate large-scale complex systems and distributed applications. REST is based on the principle that *any client of a REST API should be driven by nothing but hypertext*. This principle seems abstract but it is easy to understand if we treat a REST API as a distributed finite state machine, where the states are resource representations and the transitions are the links between the representations. In this model, hypertext-driven means that a client should enter a REST API from an entry point, and then be guided by the hypertext from the resources to reach a final representation.

This principle makes it possible for a user without any technical background to use the Web by following the hyperlinks on the pages until the desired page is retrieved. In this process, the user decides which links to follow based on the information on the page, and the user agent carries out these decisions by interacting with the resources identified by the links. This separation of users from the user agents make both of them "reusable" in the sense that a user can use any user agent, and a user agent can use any user, to navigate the Web.

To mirror this process for a REST client that navigates a REST API without user involvement, it would be beneficial to decompose the REST client into two functional components: a client oracle responsible to select links to follow from the resource representations, and a client agent responsible to interact with the resources based on the selected links. This decomposition of a REST client has several advantages over a monolithic REST client where these two functions are intertwined and inseparable:

- a client oracle can be reused with different versions of a REST API, especially if a REST API version update changes some resource representations and identifications, but does not change the link relations in the representations;
- a client oracle can be reused across different service description languages of the same REST API;
- a client oracle can drive client agents in different programming languages to achieve consistent behavior;
- a client agent can be driven by different client oracle to accomplish different tasks;
- a client oracle can significantly reduce the size of a client agent if only a small portion of the resource representations are selected by the client oracle.

To realize these benefits for a given REST API, a client oracle and client agent can be written by developers manually. But this can be difficult, time consuming, and error prone. A better approach is to generate a REST oracle and agent automatically from a machine-readable service description of a REST API, such as REST Chart [6]. If the manual programming process becomes unnecessary or greatly reduced, we can significantly speed up the REST client development process. To tackle REST client generation in two phases, this paper describes a method to find optimal client oracles based on the Hierarchical REST Chart, an extension to REST Chart [6], and client agent generation will be our future work.

The rest of this paper is organized as follows. Section II reviews related work in REST service description languages. Section III describes the Hierarchical REST Chart. Section IV introduces the optimal REST oracle framework. Section V discusses the implementation and experimental results, and our findings are concluded with Section VI.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-387-2

II. RELATED WORK

Since 2009, several new service description languages, including WADL [7], RAML [8], Swagger [9], RSDL [10], API-Blueprint [11], SA-REST [12], ReLL [13], REST Chart [6], RADL [14], and RDF-REST [15] have been developed independently for REST API, but none of them is yet standardized. All these description languages are encoded in some machine-readable languages, such as XML, and most of them are standalone documents, except a few of them, such as SA-REST, are intended to be embedded within a host language, such as HTML.

RAML is a YAML language that organizes a REST API as a tree rooted at a base URI (template or reference) that denotes a REST API entry point. Underneath the root are a set of URI (templates or references) that identify available resources. Each URI may be associated with the access methods that define the input and output representations. While RAML offers a minimalist structure and several interesting mechanisms, such as inline documentation, resource traits and types, it could lead to inadvertent violation of the REST constraints [6] by exposing a list of fixed resource locations. Also, RAML does not seem to have a way to tie the hyperlinks in hypertext representations with the URI templates in the REST API description tree. Without these ties, it would be difficult for a REST client to know the method to access a hyperlink and the response representation.

Swagger has bindings to both YAML and JSON, and its REST API descriptive structure is very similar to RAML, except using a different set of vocabularies. For this reason, it has the same problems as RAML.

RSDL is a XML language that organizes a REST API around a list of <resource> elements, each of which may contain elements <location> that define its URI, <link> that links it to other resources, and <method> that define the access. As a result, RSDL could also inadvertently violate the REST constraints [6] by exposing a fixed set of resource locations to the clients. Moreover, there are no ties between the <link> elements, the <method> elements, and the actual resource representations, such that a hyperlink in a representation can point to its access method and response representation.

RADL is a XML language that organizes a REST API around the <resource> element that defines the resource location in the child element <uri> and resource methods in the child <interface> element. The request and response of a method are defined by <document> elements, which may contain <link> elements pointing to other <document> elements. Like RSDL, this resource centric design could lead to fixed resource locations. Moreover, even if a client knows the interface of a resource, it may not know the method in the interface to access a hyperlink of a document, if two or more methods exist in the interface.

Some open source toolkits are available for some service description languages [8][9][11] to generate client and server skeleton code in Java and node.js, such that the developers can edit the generated source code to complete the implementation. However, when generating the client code, these toolkits do not separate client oracle and client agent, as far as we know.

In addition, none of these service description languages supports nested REST API descriptions, such that a complete REST API description can be incrementally refined or composed seamlessly with the same mechanism. Breaking a large service description file into small parts can be helpful but it is still not sufficient, since the large service description is incomplete without the parts.

III. HIERARCHICAL REST CHART

We adopt the REST Chart model [6] as the basis for finding optimal client oracles. A major feature of REST Chart is the ability to combine the static aspects of a REST API, e.g. media types and link relations, with the dynamics of the REST API, e.g. the hypertext driven client-server interactions, into one coherent model. The REST Chart models a REST API as a Colored Petri Net where the places are “colored” by types. A typed place denotes a media type schema that defines valid resource representations. A transition denotes a valid resource interaction following a hyperlink in a schema. A token in a typed place denotes a valid resource representation defined by the schema. In this model, the connected schemas collectively define the hypertext media types of a REST API without creating any out-of-band dependences to the resource organization of the REST API.

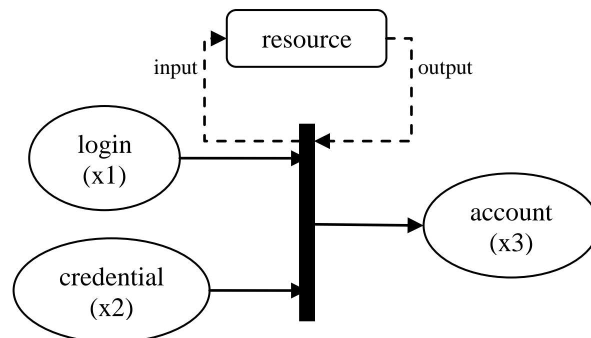

Figure 1. Example of a basic REST Chart

A basic REST Chart defines the contract between the server and client for a single interaction. This contract consists of two server places and a client place connected by a transition, as illustrated in Figure 1. This REST Chart indicates that a client can transfer its representational state from the login place (server place) to the account place (server place) if the client can create a token for the credential place (client place). To make the transition, the server first puts a token x1 in the login place (i.e. returns a valid login page), to provide a login hyperlink to the client. Then the client selects that link and puts a token x2 in the credential place (i.e. enters valid username and password). At this point, the transition (solid bar) fires and the user credential is sent to the login resource identified by the hyperlink. On success, the server deposits a token x3 in the account place (i.e. returns the valid account information).

These token markings capture the essential interaction procedure as sanctioned by the REST Chart.

The resource involved in the interaction is identified by a URI template, and there is no fixed resource location, relation, or interface that could lead to violations of the REST constraints R3-R5 [6]. A REST API with more than one interaction can be described by connecting appropriate places to form a large Petri-Net with a single entry place, which is the designated entry point of the REST API.

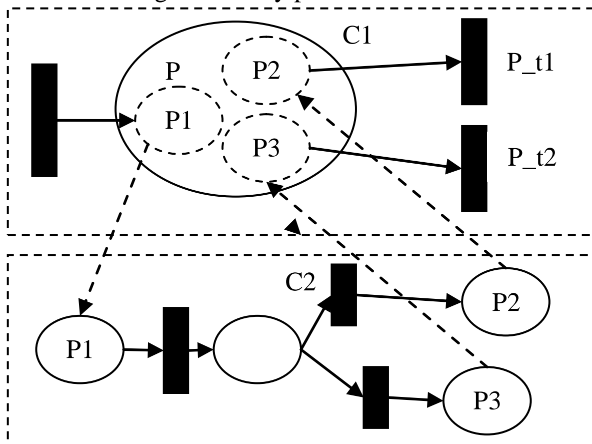

Figure 2. Hierarchical REST Chart C1 that nests REST Chart C2

```
<rest_chart id=”C1”>
   …
   <representation id="P" href="URI_to_C2" />
   <transition id="P_t1">
      <input>
        <representation ref="P/P2" link="P2_k1" />
      </input>
      …
   </transition>
   <transition id="P_t2">
      <input>
        <representation ref="P/P3" link="P3_k1" />
      </input>
      …
   </transition>
   …
</rest_chart>
```

Figure 3. XML of REST Chart C1 that nests C2 by its interface

```
<rest_chart id="C2">
   <representation id="P1" initial="true">…</representation>
   …internal topology…
   <representation id="P2">…</representation>
   <representation id="P3">…</representation>
</rest_chart>
```

Figure 4. XML of REST Chart C2’s interface places

To promote reusability and modularization of REST API, this paper extends the REST Chart to Hierarchical REST Chart based on Hierarchical Petri-Net. In Hierarchical REST Chart, a typed place can contain another REST Chart, as shown in Figure 2 where place P of REST Chart C1 contains REST Chart C2. The REST Charts C1 and C2 communicate as follows: 1) when place P receives a token of type P1, it is moved to place P1 of C2; 2) C2 will fire as usual; 3) when place P2 or P3 has a token, then the token is moved back to place P; 4) C1 will continue to fire as usual. In general, Hierarchical REST Chart can be nested to any number of levels.

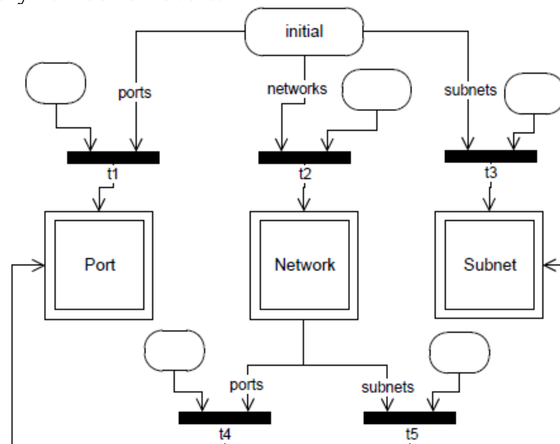

Figure 5. Top-level REST Chart for the network REST API with nested representations in double framed boxes

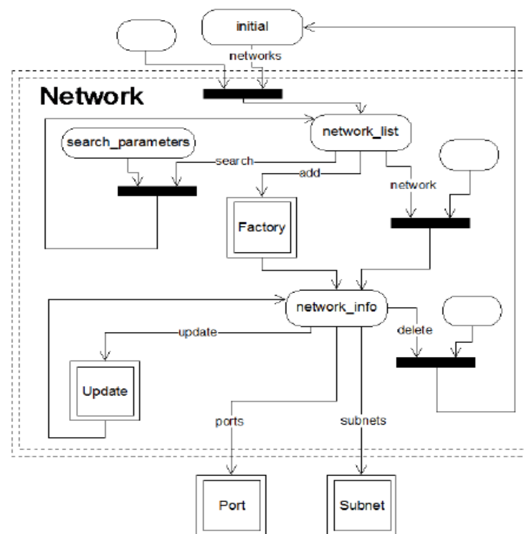

Figure 6. REST Chart nested inside at the Network place of the top-level REST Chart

To represent Hierarchical REST Chart, the XML of C1 is modified but there is no modification to the XML of C2. In C1, we modify place P and its outgoing transitions in REST Chart C1 to point at C2. The relevant modifications to C1 XML are illustrated in Figure 3 in bold font, where the <representation> element has an attribute “href” that points to the location of REST Chart C2, and the two <transition> elements “P_t1” and “P_t2” use notations “P/P2” and “P/P3” to reference the places P2 and P3 respectively in C2 that is nested in place P. The places P1, P2, and P3 of Chart REST C2 act as its interface to hide the topology of C2 as shown in Figure 4, such that the internal changes in C2 will not impact C1.

We have successfully applied Hierarchical REST Chart to describe several practical REST APIs, including the Network Management REST API of OpenStack [16]. Figure 5 depicts a top-level REST Chart with three nesting representations: Port, Network and Subnet, and the nested REST Chart for the Network is depicted in Figure 6, whose interface place Port and Subnet are used by C1 in transition t4 and t5 respectively. In both figures, the empty places indicate the client requests to dereference the hyperlinks.

IV. OPTIMAL REST ORACLE

REST Oracle is based on the idea that a REST client can be divided into two reusable functional modules: a client oracle that decides the resources to interact with, and a client agent that carries out the resource interactions instructed by the client oracle. The decisions made by a client oracle obviously depend on what goal the client is trying to achieve with the REST API. For example, a client oracle that tries to check bank account balance probably should select different resources than a client oracle that tries to deposit a paper check to an account.

If a REST API is described by Hierarchical REST Chart, the goal of a client can be defined in terms of the places it needs to visit. For instance, when a client wants to deposit a check to an account, it must reach these places in the correct order: login place to authenticate itself, the place to deposit a check, the place to scan the check, the place to upload the check, the place to verify the information, and the place for the positive acknowledgement of the entire process.

If there is more than one sequence of places to reach a goal, as most REST API does, a REST client needs to consider which path is optimal. For this purpose, we map each transition in a Hierarchical REST Chart to a positive real number that represents the cost for a REST client to take that transition. The cost can be related to network latency, the message size, the processing time, or a combination of such factors that can be measured based on the environment in which the REST API operates. With the cost factor, an optimal path can be defined as the shortest path from the initial place to a goal place in a Hierarchical REST Chart. A uniform cost of 1.0 at every transition would produce an optimal oracle that takes the smallest number of messages to reach the goal place.

It is possible to find such shortest paths in REST Chart based on Petri-Net reachability algorithms [18] or coverability algorithms [19]. However, these algorithms are usually complex or may take exponential space as they compute all possible token markings in arbitrary Petri-Net. For this reason, we decide to use graph search algorithm, whose time complexity is not dependent on token markings and is polynomial to the number of places and transitions of a Petri-Net.

To apply this approach, we convert the Hierarchical REST Chart to a nested directed graph. The server places become the vertices, and each edge is labeled with corresponding transition including the client place and a cost as illustrated in Figure 7. This process is recursively applied to the nested REST Charts.

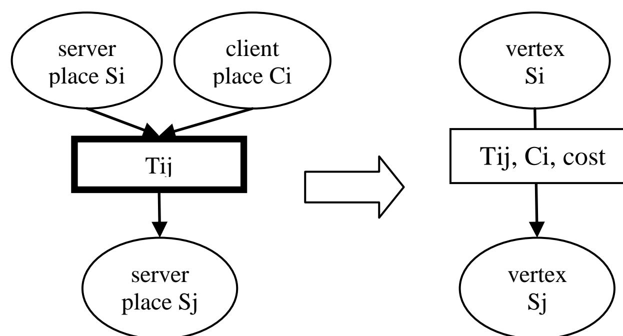

Figure 7. Transforming a REST Chart to a directed graph with cost

It is often useful to direct a REST client to not just one, but a series of goal places when interacting with a workflow. To realize this requirement, we adapt Dijkstra's Shortest Path algorithm [17] to the nested directed graph to find the shortest path from the initial place to the first goal place, and then repeat the algorithm from current goal place to the next goal place in the series, until the last goal place is reached. If a goal place is not reachable, then the process will stop. The shortest path found by this process is an oracle that reaches these goals in the given order.

```
1.  Client_Oracle(C, A, Pi, Pj): Oracle
2.    C: REST Chart
3.    A: adjacency matrix for C
4.    Pi: source place
5.    Pj: target place
6.    IN = {Pi}
7.    For each Pk in P of C do d[Pk] = A[Pi,Pk]
8.    While Pj not in IN do
9.      Pk = a place X in P–IN with minimum d[X]
10.     s[Pk] = C.transition(Pi)
11.     IN += Pk
12.     For each place X in P–IN do
13.       dist = d[X]
14.       d[X] = min(d[X], d[Pk] + A[Pk, X])
15.       if (d[X] < dist) then s[X] = C.transition(Pk)
16.     End
17.   End
18.   T = s[Pj]
19.   Oracle = (C.server_place(T), T, C.client_place(Y))
20.   While C.server_place(T) ≠ Pi do
21.     T = s[T]
22.     Oracle += (C.server_place(T), T, C.client_place(T))
23.   End
24.   Return reverse(Oracle)
25. End
```

Figure 8. Client Oracle algorithm adapted from Dijkstra's Shortest Path Algorithm

The Client Oracle algorithm is outlined in Figure 8. The core of the algorithm (lines 2-17) uses Dijkstra's Shortest Path algorithm on the directed graph A, which is converted from the REST Chart C, to find a shortest path from an initial place Pi to a final place Pj and record the transitions on the path (e.g. s[X] = transition(Pk)). The rest of the algorithm (lines 18-24) reconstructs from the recorded

transitions of the oracle as a sequence of triples (Server_Place, Transition, Client_Place).

If a vertex Si contains a nested directed graph, then we will find the shortest paths from the initial vertex of the nested graph to all its final vertices, and add the total cost of each shortest path to the cost of the corresponding edge from Si. For example in Figure 2, the cost of the shortest path from P1 to P2 will be added to edge P_t1 of P, and the cost of the shortest path from P1 to P3 will be added to edge P_t2 of P.

The following diagram (Figure 9) illustrates 3 shortest paths (oracles) superimposed on an automated coffee service REST Chart with uniform cost 1 labeled on the edges, and the corresponding oracles are summarized in Table I.

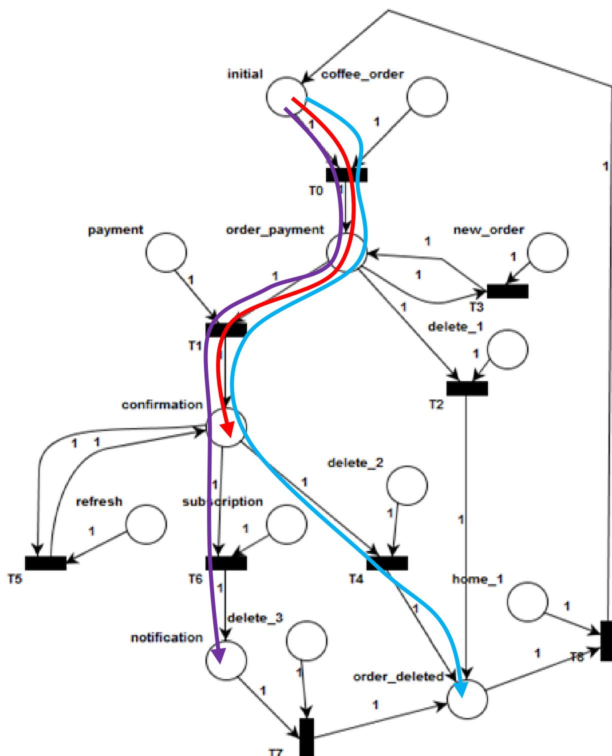

Figure 9. Three client oracles (shortest paths) for three goal series

TABLE I. ORACLES FOUND FOR DIFFERENT GOALS

	Goal Series	*Oracle*
1	**{confirmation}**	**(initial, T0, coffee_order) (order_payment, T1, payment)**
2	**{confirmation, order_deleted}**	**(initial, T0, coffee_order) (order_payment, T1, payment) (confirmation, T4, delete_2)**
3	**{notification}**	**(initial, T0, coffee_order) (order_payment, T1, payment) (confirmation, T6, subscription)**

The oracle triples in Table I contain the crucial information to implement a fully functional oracle program. The server place in a triple specifies the representations that a REST client must understand to select a hyperlink. The transition in a triple specifies the interaction with the selected hyperlink. The client place specifies the representations, e.g. a form definition, that the REST client must supply for that interaction. The only missing information is the actual representation, e.g. the form data, for the client place, which can be saved statically with the client oracle, or input to the client oracle dynamically when it is needed. Alternative optimal paths to a goal can also be included in a client oracle for fail over or load balancing purposes.

V. IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have developed a Java implementation of the Client Oracle algorithm in Figure 8. The overall flow of the implementation is illustrated in Figure 10. Only the top-level REST Chart is need by the tool, which will automatically load any nested REST Chart. Our Java tool implementation accepts a single or a series of goals. In addition to finding a client oracle for a given set of goals, it can also find all the client oracles from the entry place to all other places in a Hierarchical REST Chart. The output of the tool includes the oracles found for the top-level and the nested REST Charts.

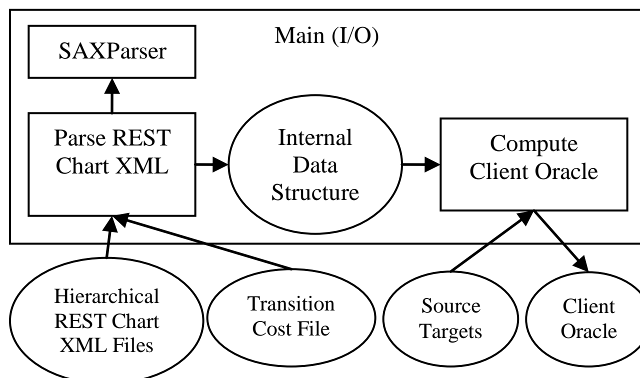

Figure 10. Java implementation of Client Oracle

We ran our Java tool on two sets of Hierarchical REST Charts with randomly generated costs to find all possible client oracles from the initial place. The first set contains 4 Hierarchical REST Charts nested in three levels, where the numbers in the parentheses indicate the number of places and transitions in the REST Chart:

```
Chart1 (8,7)
    Chart3 (4,3)
    Chart2 (6,5)
           Chart4 (4,3)
```

The second set contains 4 REST Charts nested in 3 levels as follows:

```
ABC1 (8,14)
    ABC2 (6,8)
           ABC3 (5,7)
           ABC4 (4,5)
```

The execution time (in millisecond) includes parsing the multiple XML files into the internal data structure, finding all client oracles based on the data structure, and saving them to a log file. The times measured by Java function `System.currentTimeMillis()` averaged over 5 runs on

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-387-2

a 32-bit Windows 7 machine (Intel i5 M560 dual core at 2.67 GHz and 4.00 GB memory) are summarized in Table II, where the numbers in the parentheses indicate the total number of places (V) and transitions (E) in each REST Chart.

TABLE II. PERFORMANCE SUMMARY

	average (ms)	*std*
Set 1 (22, 18)	**59**	**6.8**
Set 2 (23, 34)	**66**	**6.7**

Since the Dijkstra's Shortest Path algorithm has O(E+V*log(V)) time complexity for a graph with E edges and V vertices, the time ratio of these two sets are very close to the time complexity ratio: 66/59=1.1 while (34+23*log(23))/(18+22*log(22))=1.3. For this reason, the the performance is satisfactory and consistent. More important than the performance measurements that can be improved in many ways, the results demonstrate that the approach is feasible in finding optimal client oracles with any Hierarchical REST Chart in polynomial time.

VI. CONCLUSION

The three main contributions of this paper are: 1) a structural approach to REST client design based on two reusable functional modules, i.e. a client oracle that selects hyperlink to follow for a given goal, and a client agent that carries out the interaction as instructed by the oracle; 2) the new modeling mechanism and XML language to support Hierarchical REST Chart, which is a significant improvement over the original REST Chart for REST service modeling; and 3) a path selection framework for finding the optimal REST oracle and the implementation of the path selection framework based on Dijkstra's Shortest Path algorithm to Hierarchical REST Chart. Our approach has several advantages over a monolithic REST client design approach. Experimental results indicated that this approach is feasible and promising.

For future work, we plan to continue improving the framework of the Hierarchical REST Chart and apply the REST oracle approach in automated goal-driven generation of fully functional REST clients based on the REST Chart description of REST API.

ACKNOWLEDGMENT

The authors would like to thank Anita Kurni for implementing the Java Client Oracle tool while working as a contractor for Huawei.

REFERENCES

[1] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Dissertation, University Of California, Irvine, 2000.

[2] Twilio REST API, http://www.twilio.com/docs/api, retrieved: February, 2015.

[3] GSMA OneAPI, http://www.gsma.com/oneapi/voice-call-control-restful-api/, retrieved: February, 2015.

[4] Amazon Simple Storage Service REST API, http://docs.aws.amazon.com/AmazonS3/latest/API/APIRest.html, retrieved: February, 2015.

[5] Floodlight REST API, http://www.openflowhub.org/display/floodlightcontroller/Floodlight+REST+API, retrieved: February, 2015.

[6] L. Li and W. Chou, "Design and Describe REST API without Violating REST: a Petri Net Based Approach," ICWS 2011, July 4-9, 2011, pp. 508-515.

[7] M. Hadley, Web Application Description Language, W3C member Submission, 31, August 2009, http://www.w3.org/Submission/wadl/, retrieved: February, 2015.

[8] RAML Version 0.8, http://raml.org/spec.html, retrieved: February, 2015.

[9] Swagger 2.0, https://github.com/swagger-api/swagger-spec, retrieved February, 2015.

[10] J. Robie, R. Cavicchio, R. Sinnema, and E. Wilde, "RESTful Service Description Language (RSDL), Describing RESTful Services Without Tight Coupling, Balisage," The Markup Conference 2013, http://www.balisage.net/Proceedings/vol10/html/Robie01/BalisageVol10-Robie01.html, retrieved: February, 2015.

[11] API Blueprint Format 1A revision 7, https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md, retrieved: February, 2015.

[12] K. Gomadam, A. Ranabahu, and A. Sheth, SA-REST: Semantic Annotation of Web Resources, W3C Member Submission 05 April 2010, http://www.w3.org/Submission/SA-REST/, retrieved: February, 2015.

[13] R. Alarcon and E. Wilde, "Linking Data from RESTful Services," LDOW 2010, April 27, 2010, pp 100-107.

[14] J. Robie, RESTful API Description Language (RADL), https://github.com/restful-api-description-language/RADL, 2014, retrieved: February, 2015.

[15] P.-A. Champin, "RDF-REST, A Unifying Framework for Web APIs and Linked Data," Services and Applications over Linked APIs and Data (SALAD) workshop at ESWC, May 2013, pp.10-19.

[16] OpenStack API References, http://developer.openstack.org/api-ref.html, retrieved: February, 2015.

[17] J. L. Gersting: Mathmatical Structures for Computer Science, third edition, 1993, pp. 422-423.

[18] C. G. Cassandras and S. Lafortune, Introduction to Discrete Event Processing, 2nd edition, Springer, 2008, pp. 246-246.

[19] J. Esparza and M. Nielsen: Decidability Issues for Petri Nets, BRICS Report Series, RS-94-8, ISSN 0909-0878, May 1994.