Compatibility Modeling and Testing of REST API based on REST Chart

Li Li and Wu Chou
Huawei Shannon IT Lab, Huawei, Bridgewater, New Jersey, U.S.A.

Keywords: REST API, Petri-Net, REST Chart, REST Compatibility, Client Oracle, and Client Agent.

Abstract: Compatibility of REST API has become an acute issue in many large scale distributed software systems where many REST APIs evolve rabidly with new services and service updates. To address this problem in a generic fashion independent of the REST API implementations, this paper presents an approach based on REST Chart, a Petri-Net based XML language and modelling framework to describe and track down the variations among REST APIs. In addition, an efficient algorithm is developed that can perform the fast model checking to determine the compatibility between two REST APIs from their REST Chart representations. Unlike conventional monolithic client architecture based backward compatibility testing approaches, REST Chart compatibility modelling is defined formally in terms of a client operational model that decomposes the client side service infrastructure 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. A prototype system has been implemented and the preliminary experimental results show that the approach is feasible and promising.

1 INTRODUCTION

In recent years, the REST architectural style (Fielding 2000) has been widely applied in API design for multiple areas, including Real-Time Communications (GSMA OneAPI 2013), Cloud Computing (OpenStack REST API v2.0), and Software-Defined Networking (SDN) (Floodlight REST API 2014). It is an efficient and flexible way to access and integrate large-scale complex systems which may have many interacting REST APIs to provide their resources as service for applications. However, in large scale distributed systems, these interacting REST APIs are evolving rapidly and under frequent updates. An acute problem in REST based system is how to efficiently migrate REST clients to keep up with the rapid updates and service variations that are frequently made to numerous REST APIs - a situation may cause the backward compatibilities to break.

For example, OpenStack is an open source IaaS platform that currently supports 14 REST APIs (OpenStack API Complete Reference 2014), implemented by over 30 components - managing compute, storage, network, VM image, and identity services. To maintain backward compatibility, OpenStack simultaneously support different versions of the same API - for example, there are 3 versions of Compute API, 2 versions of Block Storage API, 2 versions of Identity API, and 2 versions of Image API. But the actual number of REST APIs in an OpenStack installation can be much more if we count the third party REST APIs.

On the other hand, OpenStack development follows a very rapid release cycle, and the development cycle of different versions of OpenStack often overlap in time (OpenStack Releases 2014). For example, the Grizzly and Havana versions of OpenStack overlap each other by 6 months, and each has 6 releases within 11 months respectively. The Havana and IceHouse versions of OpenStack overlap each other by 6 months, and the IceHouse version of OpenStack has 4 releases within 6 months. Each new release may introduce the following changes to REST APIs that can break the REST clients programmed for the previous release versions:

- schemas: add, delete, modify, or relocate schema elements in the media types;
- hyperlinks: add, delete, modify, or relocate hyperlinks in the schemas.

For example, Floodlight REST API, which is

Li L. and Chou W..
Compatibility Modeling and Testing of REST API based on REST Chart.
DOI: 10.5220/0005441301940202
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 194-202
ISBN: 978-989-758-106-9

adopted by OpenStack for network control, has these changes between its versions v1.0 and v2.0. Both versions allow a client to navigate to a port resource by the path: `initial→networks→ports →port.` In addition, version v2.0 adds an equivalent but shorter path: `initial→ports →port`. In both cases, a v1.0 client looking for a port resource can still find it, although it may take a different path in version v2.0. However, version v2.0 also introduces changes that break the clients, such as renaming the attachment resource in v1.0 to device resource. A v1.0 client looking for an attachment resource will not find it in v2.0, unless it is told that attachment is equivalent to device.

To find incompatibilities within REST APIs is difficult for two reasons. First, many REST APIs are not described by a machine-readable language which is needed to lend themselves to an automated analysis. Second, the available service description languages do not come with automated methods for compatibility analysis.

To address the compatibility problems in such rapid development cycles as in the large scale open source development, this paper adopts REST Chart as the service description language and modelling framework for REST API – it transforms the problem of compatibility between REST APIs to compatibility between two REST Charts. Since REST Chart is based on Petri-Net, the compatibility of two REST Charts can be formally defined and resolved based on a Petri-Net behaviour model, making it feasible for a systematic and automated compatibility checking of REST APIs.

The rest of this paper is organized as follows. Section 2 surveys the related work. Section 3 introduces the framework of REST Chart modelling for REST APIs. Section 4 presents the proposed REST client operational model which provides a theoretical basis for the REST Chart based compatibility testing and verification. Section 5 derives the compatibility conditions and the REST Chart comparison algorithm based on the proposed operational model. Section 6 discusses the implementation and experimental results, and we conclude the findings of this paper in Section 7.

2 RELATED WORK

Several REST service description languages have been developed since 2009. WADL (Hadley 2009) is an early effort to describe REST services, followed by RAML (RAML Version 0.8), Swagger (Swagger 2.0), RSDL (Robie 2013), API-Blueprint (API Blueprint Format 1A revision 7), SA-REST (Gomadam 2010), ReLL (Alarcon 2010), REST Chart (Li 2011), RADL (Robie 2010), and RDF-REST (Champin 2013). All these description languages are encoded in some machine-readable languages, such as XML, and most of them are standalone documents, but a few of them, such as SA-REST, are intended to be embedded within a host language, such as HTML. We are not aware of any methods to compare different versions of a REST API based on these description languages.

There are several open source Java packages and Web tools (SOA Membrane WSDL tool 2014, WSDL Auditor 2014, WSDL Comparator 2014) that compare two WSDL files for WS-* based web services – they include XML Schema (Thompson 2004) files to identify changes (addition, deletion, modification, and reorder) to the WSDL elements, such as port types and operations, and the XML elements and attributes. Some tools distinguish changes that will break interface compatibility from those that will not. For example, adding an optional XML element to a XML Schema of an input element of an operation will not invalidate the interface, but adding a required XML element, or changing the type, name, or position of an existing required XML element will.

However, these methods of WSDL comparison for WS-* based web services cannot be applied to compare REST Charts for REST APIs, because a REST Chart is not structured as WSDL. Despite that REST Chart is represented as a XML dialect, we cannot use generic XML diff tools to compare REST Charts, because REST Chart has special semantics not understood by those tools.

There are a few open source tools (SOA Membrane XSD tool 2014) that compare two XML Schema definitions and identify their differences as changes (addition, deletion, modification, and reorder). These tools can assist the comparison of REST Chart, as well as other service description languages that use XML Schema to define the input and output messages of a service. However, they fall short to provide a generic framework for comparing the compatibility of REST service APIs, which is critically needed but serious lacking for large scale distributed software systems.

3 REST CHART MODEL

REST Chart (RC) is proposed in 2011 (Li 2011) to design and describe REST APIs without violating the REST principles (Fielding 2000), especially the

7 rules that codify the REST services (Li 2011). Following these guidelines, REST Chart models a REST API as a High-Level Petri-Net. REST Chart uses a XML dialect to encode a Coloured Petri-Net such that it is machine readable and extensible, and it can be validated by XML Schemas. Each REST Chart always has a designated entry point, from which a REST client can reach all the connected places. We refer readers to the original REST Chart paper (Li 2011) for REST Chart XML syntax and examples.

The structure and behaviour of REST Chart can be explained informally by a basic REST Chart illustrated in Figure 1 with one transition, two input places, and one output place. It models a login request-response interaction with a resource. The REST Chart indicates that a REST client can follow a hyperlink to transition from the login place to the account place if it presents the correct credential.

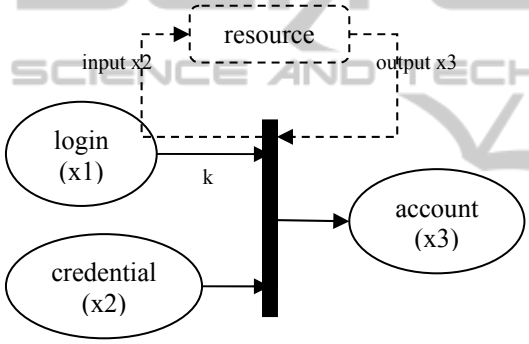

Figure 1: Example of a basic REST Chart.

The client state for this transition can be modelled by a sequence of token markings of the Petri-Net that underlines the REST Chart, where each token is a resource representation defined in (Fielding 2000). In particular, token x_1 denotes a login representation from the server, token x_2 the credential representation from the client (request), token x_3 the account representation from the server (response), and token *0* denotes nothing. Each token marking is a 3-vector for the three places:

$$(x_1, 0, 0) \rightarrow (x_1, x_2, 0) \rightarrow (0, 0, x_3).$$

To distinguish these tokens, REST Chart adopts Coloured Petri-Net (Murata 1989) to associate each place with a media type schema that can be used to process the tokens in that place.

Token x_1 may have many hyperlinks besides the one for login. To select the login hyperlink *h* with token in the login place, REST Chart adopts Predicate/Transition Petri-Net (Murata 1989) and attaches a *hyperlink predicate k(h)* to the arc from the login place to the transition arc to the account. Hyperlink predicate *k(h)* qualifies a hyperlink *h* with two information items (Li 2008, 2009):

1) *[service]*: a URI (Berners-Lee 2005) that represents the service provided by the hyperlink;
2) *[reference]*: a URI Template (Gregorio 2013) that identifies the locations of the resource.

Two hyperlink predicates k_1 and k_2 are deemed equal if $k_1.[service]=k_2.[service]$ and $k_1.[reference]=k_2.[reference]$.

Predicate *k(h)* is true if and only if the following conditions hold:

1) *k.[service] = h.[service]*;
2) *match(k.[reference], h.[reference])* is complete.

Function *match(x, y)* returns a set of *v=s* pairs, for each variable *v* of URI template *x* that is instantiated by string *s* from URI string *y*. Function *match(x,y)* is *complete* if and only if all the variables of *x* are instantiated by *y*. The following XML represents a hyperlink predicate *k*, where *k.[service]=link/rel/@value*, and *k.[reference]=link/href/@value*:

```
<link id="k">
 <rel value="http://a.b.com/login" />
 <href
value="http://{d}/users/{u}/account" />
</link>
```

The following XML represents an ordinary hyperlink *h*, where *h.[service]=link/@rel* and *h.[reference]=link/@href*:

```
<link id="h" rel="http://a.b.com/login"
href="http://a.b.com/users/john/account"
/>
```

Clearly *k(h)* is true because the two conditions hold:

1) *k.[service] = h.[service] =* `http://a.b.com/login`;
2) *match(k.[reference], h.[reference]) = {d=http://a.b.com, u=john}*.

With hyperlink predicate, we can represent the client state in Figure 1 as a sequence of *p-k* pairs, where *p* denotes a place, *k* denotes the hyperlink predicate selected at *p*, and *0* means no hyperlink is selected:

login-k account-0

This *p-k* representation is equivalent to the token marking vectors, but it highlights the two main operations a client must perform at each step: 1) it must selects a hyperlink at each place to move towards the goal place, in this case the account

place; 2) it must move tokens, e.g. x2 and x3, between the places by interacting with the selected hyperlink. To reduce the cost of client migration, we delegate these two distinct responsibilities to two components: 1) *client oracle* responsible to select hyperlinks; and 2) *client agent* responsible to interact with the selected hyperlink.

To understand the distinction between these two components, we can regard a REST Chart as a maze where the transitions are the locked "doors" that protect the places. The oracle knows which door (hyperlink) to open at each place, but it does not have the keys (interactions) to unlock the doors. The agent has the keys, but does not know which doors to open. To move through a maze towards a goal, a REST client needs the right kind of client oracle and client agent for that maze. Applying this analogy to the REST Chart in Figure 1, where the client is at the login place, the oracle selects the door (hyperlink predicate k) to the next place, account, and the agent produces the key (token x2) to open the door for the client to enter the account place.

When a REST API is updated, the new and old versions usually overlap in places (keys) and transitions (doors), making it possible for us to reuse the client oracle and client agent. The purpose of REST Chart compatibility analysis is to determine which part of a client can be reused when migrating it to a new REST API. Ideally, we want to keep both components (e.g. keys and doors) unchanged. But if this is not possible, we hope to update only the client oracle, since the client agent can be shared by different clients that speak the same media types and network protocols. Moreover, modification to a client agent is often expensive as it consists of several layers of media type and network protocol stacks defined by complex rules.

To understand which component of client *C* needs to be updated, we examine all 4 possible compatible changes to the following *p-k* path:

$$path = p_0\text{-}k_0\ p_1\text{-}k_1\ p_2\text{-}k_2\ p_3\text{-}0$$

Without losing generality, we assume this path exists in version v1.0 of the Floodlight REST API such that p_0=*initial*, p_1=*networks*, p_2=*ports and* p_3=*port* and we try to find compatible paths in v2.0.

Case 1: the new path is identical to the original *path*. Obviously, *C* can reuse its client oracle and client agent to traverse the new path.

Case 2: the new path consists of the same pairs but different inter-pair relations. For example, new path $p_0\text{-}k_0\ p_2\text{-}k_2\ p_3\text{-}0$ removes pair $p_1\text{-}k_1$, whereas new path $p_0\text{-}k_0\ p_2\text{-}k_2\ p_1\text{-}k_1\ p_{3\text{-}0}$ reorders the original pairs. *C* can still keep its client oracle and client agent, because the client oracle can select the same hyperlink at the same place in v2.0.

Case 3: the new path consists of different pairs combined from the same *p* and *k*. For example, new path $p_0\text{-}k_2\ p_1\text{-}k_0\ p_3\text{-}0$ changes the hyperlinks selected at p_0 and p_1. For *C* to traverse the new path, it needs a new client oracle that selects k_2, instead of k_0, at p_0. But *C* can reuse its client agent, since all *p* and *k* in the new path occur in the original path.

Case 4: the new path consists of different pairs combined from different *p* and *k*. For example, new path $p_0\text{-}k_3\ p_4\text{-}k_4\ p_3\text{-}0$ introduces new hyperlink predicates $\{k_3, k_4\}$ and places $\{p_4\}$ to reach the original goal place p_3. New hyperlink predicates mean new services and protocols that *C*'s client agent cannot fire, while new places mean new schemas and tokens that *C*'s client agent cannot process. For this reason, client *C* has to update both client agent and client oracle. However, if the new places and transitions in v2.0 are *covered* by some places and transitions in v1.0, then *C* can update its client oracle but keep its client agent.

4 RC OPERATIONAL MODEL

In order to find generic conditions and algorithms that detect compatible paths between REST Charts, this section introduces a deterministic operational model from REST Chart based on a state-based behaviour model (Murata 1989) of Petri-Net that underlies REST Chart. This model leads to a formal framework of client oracle and client agent, from which the generic REST Chart compatibility model for all 4 cases is derived.

4.1 RC Behaviour Model

A REST Chart is a bipartite graph $RC=(P, T, F, M_0, p_0, L, S, K, type, link, bind)$, where:

- *P* is the finite set of places;
- *T* is the finite set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions and from transitions to places;
- M_0: $P \rightarrow \{0, 1, 2, ...\}$ is the initial marking, a function that maps each place in *P* to 0 or more tokens;
- p_0 is the initial place;
- *L* is a finite set of media type definition language;
- *S* is the finite set of schemas in some type definition language in *L* and *valid(s, x)*

indicates token x is an instance of schema s;
- K is the finite set of hyperlink predicates;
- *type:* $P \times L \rightarrow S$ maps each place and a media type language to a schema;
- *link:* $P \rightarrow 2^K$ maps a place to a set of hyperlink predicates;
- *bind:* $P \times K \rightarrow T$ binds a hyperlink predicate in a place to a transition.

We assume that RC has no isolated places or transitions as typical. For a REST Chat RC with m places and n transitions, let $A=[a_{ij}]$ be the $n \times m$ incident matrix of integers, whose entry is given by:

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-} \quad (1)$$

$$a_{ij}^{+} = w(T_i, P_j), a_{ij}^{-} = w(P_j, T_i) \quad (2)$$

where $w(T_i,P_j)$ is the weight of the arc from transition T_i to its output place P_j and $w(P_j, T_i)$ is the weight of the arc to transition T_i from its input place P_j.

For a given token marking M, let $M(P_j)$ denote the number of tokens in place P_j. Transition T_i can fire if and only if:

$$a_{ij}^{-} \le M(P_j), 1 \le P_j \le m \quad (3)$$

In the deterministic operation model, only one transition fires at each step. To select a transition to fire at the k-th step, we define a $n \times 1$ column *control vector* u_k with exactly one 1 in the i-th position and 0 elsewhere indicating transition i fires. If g is the goal place, then the necessary condition to reach marking $M_d(g)>0$ in d steps from M_0 is:

$$\Delta M_k = M_k - M_{k-1} = A^T u_k \quad (4)$$

$$M_d(g) = M_0 + A^T \sum_{k=1}^{d} u_k \quad (5)$$

Among the three factors of Equations (4) and (5), A^T is fixed by the REST Chart, u_k is controlled by the *client oracle* that selects a transition to fire, and ΔM_k is handled by the *client agent* that moves tokens between the places of the fired transition. The procedures of these two components are defined in section 4.2.

4.2 REST Client Components

A *client agent* A of a REST Chart RC consists of the following abstract procedures that operate on tokens in a place:

- *(H, d)=decode(p, l, x)*: decodes a token x in type language l in place p into a set of hyperlinks H and data d, such that:
  - $(\forall h \in H \exists k \in link(p))k(h)$: every decoded hyperlink h matches a hyperlink predicate k at place p;
  - *valid(type(p, l), x)*: is true, indicating that token x is an instance of schema *type(p, l)* at place p with language l;
- *x=encode(p, l, (H, d))*: encodes a set of hyperlink H and data d into a token x in type language l in place p, such that:
  - *valid(type(p, l), x)* is true as described above;
- $(p, x_{out})=fire(t, h, x_{in})$: send token x_{in} to the resource identified by hyperlink h and receives token x_{out} in place p according to protocol defined by transition t.

In this model, each place and language pair defines a schema and each token in a place is processed as an instance of the schema. To decode in place p_j a token encoded in place p_i requires that these places maintain the *coverage* relation denoted by $p_i \subseteq p_j$. More precisely, for any media type definition language l and token x, $p_i \subseteq p_j$ if and only if $type(p_i, l) \subseteq type(p_j, l)$ such that:

$$valid(type(p_i, l), x) \rightarrow valid(type(p_j, l), x).$$

It is evident that if $p_i \subseteq p_j$, then any token encoded in p_i can be decoded in p_j such that:

$$(H, d)=decode(p_j, l, encode(p_i, l, (H, d))).$$

A *client oracle* Q of a REST Chart RC consists of the following abstract procedures that operate on the control vector:

- $(k, t, p_j, H_j, d)=select(p_i)$: selects a hyperlink predicate k, transition t for k, input place p_j for t, data d, and hyperlinks H_j for p_j, based on current place p_i;
- *Bool=goal(d):* return true if data d satisfies the target place.

A client oracle can be derived from a REST Chart to select a shortest path to reach the goal place. It could be implemented as a rule-based system or finite-state machine that is easy to reconfigure when RC or the goal changes.

The client operation model can be represented by a recursive procedure by which the client agent moves towards the goal place guided by the client oracle. More precisely, a REST client $C = (Q, A, reach)$, where Q is a client oracle, A is a client agent, and *reach* is a control procedure that combines Q and A to reach the goal place, starting from the initial place p_0 and token x_0 (Listing 1). Variable V collects the traversed places and transitions with an empty set as the initial value.

```
1.  reach(l, Q, A, V, p0, x0)
2.    (H0, d0) = A.decode(p0, l, x0)
3.    if Q.goal(d0) then return V
4.    (k, t, p1, H1, d1) = Q.select(p0)
5.    if h ∈ H0 ∧ k(h) then
6.      V = V ∪ {(p0, x0, t)}
7.      x1 = A.encode(p1, l, H1, d1)
8.      (p2, x2) = A.fire(t, h, x1)
9.      return reach(l, Q, A, V, p2, x2)
10.   end
11.   return V
12. end
```

Listing 1: REST client operational model

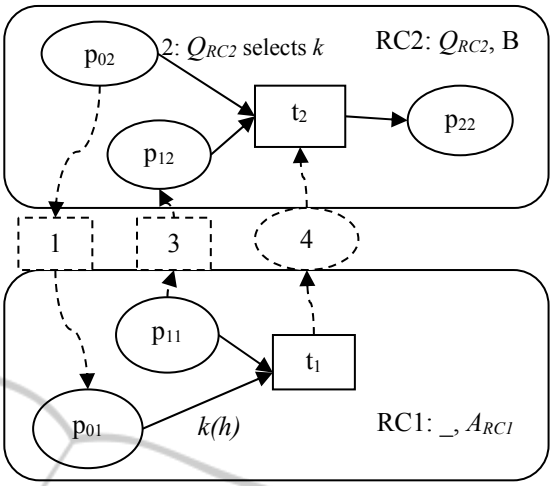

Figure 2: *C* uses *A* to fire a transition of *RC2*.

5 RC COMPATIBILITY

The compatibility between two REST Charts RC1 and RC2 can be defined in terms of the client operation model introduced in Section 4. More formally, a place p in REST Chart RC_2 is compatible with REST Chart RC_1 for client C, if and only if the following conditions hold:

1) $C = (Q, A, reach)$;
2) $(p, x) = reach(Q, A, M_0, p_0, x_0)$;

where:

- Q_{RC2} is a client oracle for RC_2;
- A_{RC1} is a client agent for RC_1;
- M_0 is the initial state of C;
- p_0 is the initial place of RC_2;
- x_0 is the initial token in p_0.

By the maze analogy in Section 3, this definition implies that RC_2 is compatible with RC_1 if client C can reuse the keys for RC_1 to open the doors in RC_2, when guided by oracle Q_{RC2}. Here a key refers to the *decode()*, *encode()* and *fire()* procedures defined in Section 4.2. The situation is illustrated in Figure 2, where client C has a token in place p_{02} and its oracle Q_{RC2} selects door t_2 to enter place p_{22}.

To find a reusable key, we introduce agent B to RC_2 whose job is to search RC_1 for a door (transition) t_1 equivalent to t_2 so C can use the key for t_1 to open t_2. This equivalent relation can be defined with an auxiliary Petri-Net that connects RC_1 and RC_2 with dashed places and transition 1, 3, 4 shown in Figure 2. Transition t_1 and t_2 are equivalent if C can fire transition t_2 in the following 4 steps. At **step 1**, agent B encodes token x_0 in place p_{02} and sends it to place p_{01} where agent A_{RC1} decodes x_0 to extract the hyperlinks H_0. At **step 2**, oracle Q_{RC2} selects from p_{02} hyperlink predicate k that leads to place p_{22}. Client C applies k to H_0 to choose hyperlink h to follow at place p_{01}. At **step 3**, B finds place p_{11} for A_{RC1} to encode token x_1 (request). Agent A_{RC1} sends token x_1 to place p_{12} for B to decode it. At **step 4**, agent A_{RC1} fires transition t_1 which in turn fires transition t_2 to produce token x_2 (response) in p_{22}. The procedures of A_{RC1} and B at each step are correlated in Table 1 (for brevity, the subscripts of Q and A are removed).

Table 1: Procedures called by client agents A and B.

	RC2: Q, B	**RC1: Q, A**
1	x_0 = $B.encode(p_{02},l,(H_0,d_0))$ $valid(type(p_{02}, l), x_0)$	(H_0, d_0) = $A.decode(p_{01}, l, x_0)$ $valid(type(p_{01}, l), x_0)$
2	$(k, t, p_1, H_1, d_1) = Q.select(p_{02})$ $\exists h \in H_0 \wedge k(h)$	
3	(H_1, d_1) = $B.decode(p_{12}, l, x_1)$ $valid(type(p_{12}, l), x_0)$	x_1 = $A.encode(p_{11},l,(H_1,d_1))$ $valid(type(p_{11}, l), x_0)$
4	(p_{22}, x_2)= $B.fire(t_2, h, x_1)$ $t_2 \subseteq t_1$	(p_{22}, x_2)= $A.fire(t_1, h, x_1)$ $t_1 \subseteq t_2$

Table 2: Constraints between RC_2 and RC_1.

1	$p_{02} \subseteq p_{01}$
2	$k \in RC_1.link(p_{01}) \cap RC_2.link(p_{02})$
3	$p_{11} \subseteq p_{12}$
4	$RC_1.bind(p_{01},k) = t_1 = t_2 = RC_2.bind(p_{02}, k)$

For client agent A_{RC1} to fire the transition, all the procedures in Table 1 must succeed in the right order. However, these conditions rule out non-validating client agents that do not use schemas. For

```
1.  traverse(RC2, RC1, V, p0)
2.    if p0 ∈ V then return
3.    V = V ∪ {p0}
4.    K = RC2.link(p0)
5.    for each k ∈ K
6.      t2 = RC2.bind(p0, k)
7.      if t1 = RC1.cover(t2) then
8.        V = V ∪ {(t2, t1)}
9.        p2 = RC2.output(t2)
10.       traverse(RC2, RC1, V, p2)
11.     end
12.   end
13. end
```

Listing 2: REST Chart comparison algorithm.

this reason, Table 1 summarizes the necessary conditions for finding a compatible path. Using the hyperlink predicate equality relation defined in Section 3 and the schema coverage relation defined in Section 4.2, the Table 1 conditions can be reduced to Table 2, where the dependences on the operational procedures are removed and the conditions depend only on the structures of RC_1 and RC_2. These structural conditions allows us to compare RC_1 and RC_2 with a Depth-First search algorithm to traverse RC_2 aided by RC_1 as outlined in Listing 2.

Procedure $RC_1.cover()$ finds a transition t_1 in RC_1 equivalent to transition t_2 in RC_2 based on the Table 2 conditions without actually constructing the auxiliary places and transitions in Figure 2. For each transition in RC_2, this procedure may need to search up to $|P_1|$ places of RC_1 in the worst cases, where P_1 is the set of places of RC_1, since hyperlink predicate k may occur in all places of RC_1. As the algorithm traverses all transitions and places of RC_2, its time complexity is $O(|P_1|(|P_2|+|T_2|))$, where P_2 is the set of places and T_2 is the set of transitions of RC_2.

6 PROTOTYPE AND EXPERIMENTS

We implemented the REST Chart comparison algorithm (Listing 2) in Java and tested it on several REST Charts. The Java tool uses JDOM package to parse two REST Charts, each defined by some XML files, and outputs compatible places and schema relations. An example output of the REST Chart comparison is illustrated in Figure 3, where a compatible place in the new version is marked by arrows pointing to the old places that cover it.

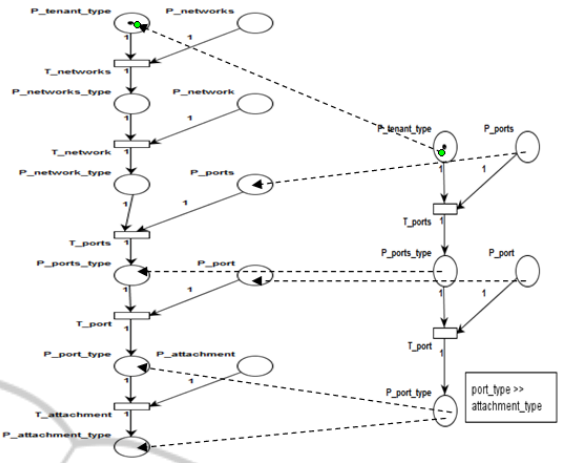

Figure 3: Compatible places between version 1.0 (left) and version 2.0 of Floodlight REST API.

The $RC_1.cover()$ procedure is based on the SOA membrane package (SOA Membrane XSD tool 2014) that compares XML schemas and identifies their differences. In particular, if s_1 and s_2 are two XML schemas, the procedure $compare(s_1, s_2)$ returns a set of differences D, where e denotes the element in s_1 that has been changed in s_2 by operation op:

$$D=\{(e, op)|op=\{add, remove, move, type\}\}.$$

Let B be the set of bad elements in D, where $e.min$ is the minimum occurrence of element e, and $t1 \succ t2$ indicates that $t1$ is a super type of $t2$:

$$B = \{e|(e, remove) \in D,\ (e, move) \in D,\ (e, add) \in D \wedge e.min \neq 0,\ (e, type, t1, t2) \in D \wedge (t1 \succ t2)\}.$$

Let G be the set of good elements in D and s_1:

$$G=\{e|e \in s_1,\ e \notin B\}.$$

Then we have the following decision rules:

1. $D=\{\}$: $s_1 = s_2$;
2. $B=\{\}$, $G\neq\{\}$: $s_i \subseteq s_j$;
3. $B\neq\{\}$, $G\neq\{\}$: s_1 is partially covered by s_2.

To test the correctness of the algorithm, we took a REST Chart and created a dozen versions of it by changing the places, transitions and schemas in various ways, and verified the outputs of the algorithm against the changes.

The performance of the algorithm is summarized in Table 3 for two REST APIs: SDN REST Chart (rows 1 and 2) and flat Coffee REST Chart (rows 3 and 4). The results are averaged over 5 runs of the algorithm on a Windows 7 Professional notebook computer (Intel i5 CPU M560 Dual Core 2.67GHz with 4GB RAM). The results show that the algorithm spent extra (1179.4–1063.8)=115.6ms when the complexity factors increased by (19+21)*17/((10+11)*8)=4 times from the Coffee REST API to the SDN REST API.

Table 3: Performance summary.

Charts	RC1	RC2	Time (ms)/std
place	17	19	1179.4
transition	17	21	30.3
place	8	10	1063.8
transition	10	11	55.9

7 CONCLUSIONS

The contributions of this paper are summarized below:

- We proposed and formalized the REST Chart structural and behaviour models based on Petri-Net semantics.
- We introduced a formal REST client operational model based on REST Chart, which decomposes the client side into two reusable functional modules: a client oracle and a client agent.
- We proposed a formal definition for REST Chart compatibility, and derived the necessary conditions to test REST API compatibility utilizing the proposed client operational model.
- We developed an efficient algorithm to compare REST Chart to determine the compatible paths based on the compatibility conditions.

Main advantages of our approach are: 1) it is a generic method independent of the REST API structure or messages; and 2) it promotes REST client reusability by migrating only client oracle or client agent. For future work, we plan to extend the REST Chart comparison algorithm to more complex cases and implement an automated client migration process based on the comparison.

ACKNOWLEDGEMENTS

We would like to thank Mr. Tony Tang for implementing the prototype system.

REFERENCES

Alarcon, R., Wilde, E. 2010. Linking Data from RESTful Services, LDOW 2010, April 27, 2010, Raleigh, North Carolina.

API Blueprint Format 1A revision 7. https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md, last access: 12/16/2014.

Berners-Lee, T., Fielding, R., Masinter, L. 2005. Uniform Resource Identifier (URI): Generic Syntax, Request for Comments: 3986, January 2005, https://tools.ietf.org/html/rfc3986, last access: 12/16/2014.

Champin. P-A. 2013. 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, Montpellier (FR), France. pp.10-19.

Floodlight REST API. http://www.openflowhub.org/display/floodlightcontroller/Floodlight+REST+API, last access: 12/16/2014.

Fielding, R. T. 2000: Architectural Styles and the Design of Network-based Software Architectures, DISSERTATION, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

Gomadam, K. et al 2010. SA-REST: Semantic Annotation of Web Resources, W3C Member Submission 05 April 2010, http://www.w3.org/Submission/SA-REST/, last access: 12/16/2014.

Gregorio, J. et al 2012. URI Template, Request for Comments: 6570, March 2012, https://tools.ietf.org/html/rfc6570, last access: 12/16/2014.

GSMA OneAPI 2013. http://www.gsma.com/oneapi/voice-call-control-restful-api/, last access: 12/16/2014.

Hadley, M. 2009: Web Application Description Language, W3C member Submission, 31, August 2009, http://www.w3.org/Submission/wadl/, last access: 12/16/2014.

Li, L., Chou, W. 2008. InfoParser: Infoset Driven XML Processing for Web Services, ICWS 2008, pages 513-520, Beijing China, September 2008.

Li, L., Chou, W. 2009. Infoset for Service Abstraction and Lightweight Message Processing, ICWS 2009, pages 703-710, Los Angeles, July 2009.

Li, L., Chou, W. 2011. Design and Describe REST API without Violating REST: a Petri Net Based Approach, ICWS 2011, pages 508-515, Washington DC, USA, July 4-9, 2011.

Murata, Tadao: Petri Nets: Properties, Analysis and Applications (invited paper), Proceedings of the IEEE, Vol. 77, No. 4, April 1989.

OpenStack API Complete Reference: http://developer.openstack.org/api-ref.html, last access: 12/16/2014.

OpenStack Releases: https://wiki.openstack.org/wiki/Releases, last access: 12/16/2014.

OpenStack REST API v2.0 references: http://developer.openstack.org/api-ref.html, last access: 12/16/2014.

RAML Version 0.8. http://raml.org/spec.html, last access: 12/16/2014.

Robie, J. 2010. RESTful API Description Language (RADL), https://github.com/restful-api-description-language/RADL, 2014, last access: 12/16/2014.

Robie, J. et al 2013. RESTful Service Description Language (RSDL), Describing RESTful Services Without Tight Coupling, Balisage: The Markup Conference 2013, http://www.balisage.net/Proceedings/vol10/html/Robi

e01/BalisageVol10-Robie01.html, last access: 12/16/2014.
SOA Membrane WSDL tool: http://www.membrane-soa.org/soa-model-doc/1.4/cmd-tool/wsdldiff-tool.htm, last access: 12/16/2014.
SOA Membrane XSD tool: http://www.membrane-soa.org/soa-model-doc/1.4/cmd-tool/schemadiff-tool.htm, last access: 12/16/2014.
Swagger 2.0. https://github.com/swagger-api/swagger-spec, last access: 12/16/2014.
Thompson, H. S. et al 2004. XML Schema Part 1: Structures Second Edition, W3C Recommendation, 28 October 2004, http://www.w3.org/TR/xmlschema-1/, last access: 12/16/2014.
WSDL Auditor: http://wsdlauditor.sourceforge.net/, last access: 12/16/2014.
WSDL Comparator: http://www.service-repository.com/comparator/compareWSDL, last access: 12/16/2014.