Compatibility Checking of REST API Based on Coloured Petri Net

Li Li(✉) and Wu Chou

Huawei Shannon IT Lab, Huawei, Bridgewater, NJ, USA
{li.nj.li,wu.chou}@huawei.com

Abstract. One of the underutilized advantages of REST API is extensibility. Extensibility allows a REST API to make certain changes to its resource representation, identification, interaction, and connection without breaking its clients. A client can cope with these changes in the REST API through hypertext-driven interactions - an iterative process in which the client can determine the resource identification based on its representation, utilize its identification to determine the interactions, and follow the interactions to determine its connections. However, our analysis reveals that there are limitations to the flexibility of the hypertext-driven navigation due to the dependency between these interaction layers in the REST API, and there is a critical need to determine if two REST APIs are compatible for the client. To address this issue, we describe a structured approach to REST client modelling that decomposes a REST client into two functional components: client oracle and client agent. From this client model, we derive a formal definition of compatibility based on the REST Chart representation of the REST API, and an efficient algorithm is developed to verify the compatibility between two REST Charts. A prototype system has been implemented, and the preliminary experimental results show that the approach is feasible and promising.

Keywords: REST API · Coloured petri net · REST chart · Compatibility checking · Client oracle · Client agent

1 Introduction

In recent years, the REST architectural style [1] has been widely applied in API design for multiple areas, including Real-Time Communications [2], Cloud Computing [3], and Software-Defined Networking (SDN) [4]. 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 the 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 [5], implemented by over 30 components - managing compute, storage,

V. Monfort et al. (Eds.): WEBIST 2015, LNBIP 246, pp. 25–43, 2016.
DOI: 10.1007/978-3-319-30996-5_2

network, VM image, and identity services. To maintain backward compatibility, OpenStack simultaneously supports 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. In fact, the actual number of REST APIs in an OpenStack installation can be even much higher if we count the third party REST APIs.

On the other hand, OpenStack development follows a rapid release cycle, and the development cycle of different versions of OpenStack often overlap in time [6]. 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 can introduce changes to its REST APIs that can break the REST clients originally programmed for the previous release.

For example, version 2.0 of Floodlight REST API in OpenStack made significant changes to version 1.0. Version 2.0 allows a client to traverse to a port resource in one of the two paths: (1) *initial* → *networks* → *ports* → *port* or (2) *initial* → *ports* → *port*. The first path is in version 1.0 while the second path is not. A version 1.0 client looking for a port resource in the version 2.0 REST API can follow the first path without change, but it cannot take advantage of the second path, unless it is pre-programmed to take alternative paths. Version 2.0 also introduces changes that a version 1.0 client does not recognize, such as renaming the attachment resource in version 1.0 to the device resource. A version 1.0 client looking for an attachment resource will not find it in version 2.0, unless it is reprogrammed to look for device resource.

To find such incompatibilities between versions of a REST API is therefore important to design and port clients in face of the frequent changes and updates. This task is difficult because a REST API permits 4 types of changes on its resources, and it can happen at the layers of its resource representation, identification, interaction, and connection. These changes can occur in any combination and each of them may require a special method to deal with. Even if a well-designed REST API can navigate its clients through some of the changes with hypertext-driven interactions, e.g. content negotiation and URI redirection, a client is still at risk of not being able to reach its targeted resource if the changes are beyond its programmed flexibility. Although it might be possible to program more flexibility in the client to reduce such risks with a new REST API, it is difficult to know what flexibility is necessary ahead of changes, and adding unnecessary flexibility can impact the performance. For these reasons, we propose a formal and efficient way to check and verify the compatibility for the client between two REST APIs.

While compatibility checking can be based on either the *implementations (behaviours)* or the *descriptions (structures)* of the REST APIs after they are implemented, this paper takes an approach to allow the compatibility checking and verification for the REST APIs at their design time before they are implemented. This approach requires that the REST APIs be described and represented in a more formal machine-readable way, and it is justified for several reasons: (1) a formal description can be read by both users and machines; (2) a formal description is more accurate than an informal one (such as English); and (3) a formal description can be used to automatically generate

REST API services and clients. The particular formal description adopted by this paper is REST Chart [14], a REST service description language and modelling framework based on Coloured Petri Net. Petri Net is a mathematical model with a graphical representation that is easy to visualize for users. With REST Chart, compatibility checking of two REST APIs is transformed into the compatibility checking of two REST Charts that describe the REST APIs. The main contributions of this paper are summarized below:

1. A layered model is described to analyze how changes to the resource representation, identification, interaction, and connection can impact each other, and it is applied to characterize the capabilities and limitations of hypertext-driven navigation in coping with REST API changes.
2. A REST client model that decomposes a REST client into two structural components: client oracle and client agent, to localize and classify the impact of REST API changes on the client implementation.
3. From this client model, a formal definition of compatibility between REST Charts is derived for compatibility checking and verification.
4. We describe an efficient algorithm to find and identify the compatible paths between two REST Charts in hypertext-driven navigation.

The rest of this paper is organized as follows. Section 2 surveys the related work. Section 3 analyzes the 4 types of changes that can be made to the REST API. Section 4 introduces the framework of REST Chart modelling for REST APIs. Section 5 presents the proposed REST client operational model which provides a theoretical basis for the REST Chart based compatibility testing and verification. Section 6 derives the compatibility conditions and the REST Chart comparison algorithm based on the proposed operational model. Section 7 discusses the implementation and experimental results, and the findings of this paper are summarized in Sect. 8.

2 Related Work

Several REST service description languages have been developed since 2009. WADL [7] is an early effort to describe REST services, followed by RAML [8], Swagger [9], RSDL [10], API-Blueprint [11], SA-REST [12], ReLL [13], REST Chart [14], RADL [15], and RDF-REST [16]. 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. However, they lack a formal method to efficiently compare different versions of a REST API based on these description languages.

There are several open source Java packages and Web tools [17–19] that compare two WSDL files for WS-* based web services – they include using XML Schema [20] files to identify changes (addition, deletion, modification, and reorder) made 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 for 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.

Moreover, 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 a WSDL file. 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 these tools.

In addition, there are a few open source tools [21] 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 for large scale distributed software systems.

Petri Net is a mathematical model that has been used to model and analyze concurrent and distributed communication and computation systems [22–24]. A Petri Net consists of places and transitions connected by arcs. A place can store tokens, which can represent data, messages or conditions, and a transition can move tokens between places to model computation, processing or inference. Coloured Petri Net assigns colours (data types) to places and tokens, such that it can realistically model systems in which tokens can have complex structures. However, it lacks a structured way to check and verify the compatibility of Petri Net in RESTful interactions.

3 A Layered Model of REST API Changes

Hypertext-driven navigation, also called “hypermedia as the engine of application state” in [1], is an iterative process in which a client uses representations (e.g. XML) to select identifications (e.g. URI), utilizes identifications to determine interactions with resources (e.g. HTTP), and follows the interactions to obtain representations and connections. This process is illustrated in Fig. 1, where a client moves from resource *a* to resource *c* through three layers of objects: representations, identifications, and connections, using two layers of operations: selections and interactions. At first, the client selects the entry point URI_a. It then uses the appropriate protocol to interact with resource *a* identified by URI_a. After the interaction returns Hypertext_a, the client selects URI_b from the Hypertext_a and the cycle continues until the client reaches the target resource *c*. Because hypertext-driven navigation encourages a client to make its decisions based on current resource states, a REST API can make certain changes at these layers where a client can cope with these changes at runtime through hypertext navigations without modification.

Compared to direct access from a fixed identification, hypertext-driven navigation seems less efficient. For example, navigating to resource *c* from resource *a* requires 4 navigation messages, whereas the direct access to resource *c* from URI_c requires 0 navigation message. However, direct access to a resource is possible only if the resource already exists. If the resource (e.g. *c*) does not exist and must be created by another resource (e.g. *b*), hypertext-driven navigation is the only option. Direct access binds resource identifications to resource connections and makes them difficult to change at runtime. For example, if a REST API advertises URI$_c$ = http://www.example.com/a/b/c for direct access, it removes the possibility for a client to find the changes to the connection at runtime - the only way is to modify the client to use the new URI.

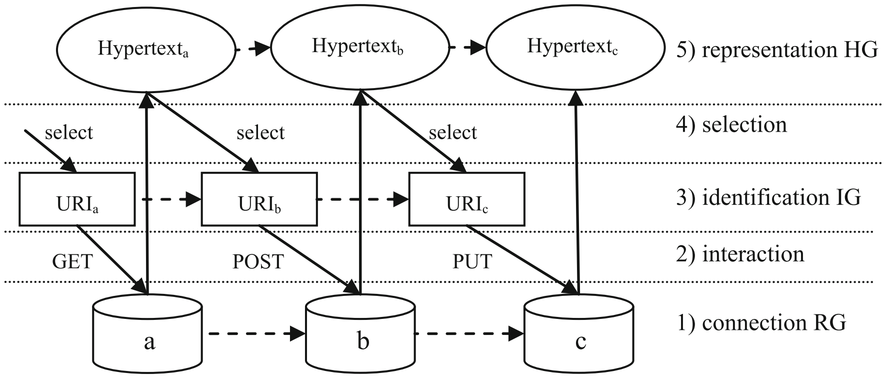

Fig. 1. Layered REST API structure for hypertext-driven navigation.

Although hypertext-driven navigation is a powerful mechanism in the RESTful design framework, it has limitations in dealing with changes in REST APIs. For hypertext-driven navigation to work, a client must recognize the objects and perform the operations in Fig. 1.

If a change introduces unrecognized objects or operations, a client will not be able to navigate through the REST API. In such cases, the client has to be modified in order to work with the new changes. To understand these limitations, we analyze the possible changes at each layer and describe how these changes in one layer can impact the other and affect the clients.

In our analysis, we treat 3 layers of objects as 3 directed graphs so that we can determine how changes in one graph can impact other graphs:

1. Connection Graph *RG*: a directed graph whose nodes are resources and edges are connections between the resources. In particular, we use triple *(x, r, y)* to represent connection *r* from resource *x* to resource y.
2. Identification Graph *IG*: a directed graph whose nodes are identifications and edges are relations between the identifications. In particular, we use *u(x)* to denote identification *u* of resource *x*.
3. Representation Graph *HG*: a directed graph whose nodes are hypertexts and edges are relations between the hypertexts. In particular, we use *h(u(x))* to denote hypertext *h* obtained from the identification *u(x)* and *l(r, u(x))* to denote a hyperlink *l* corresponding to the connection *r* to identification *u(x)*, and *m(h(u(x)))* to denote the media type of the hypertext *h*.

Because the selection operations are performed by a client and not controlled by a REST API, we only analyze interaction operations. In particular, we use *p(u(x))* to denote protocol *p* to interact with identification *u(x).* Within these notations, we consider three types of basic changes at each graph: (1) replacing element *x* by *y* is denoted by $x \rightarrow y$; (2) adding element *x* is denoted by $+x$; and (3) removing element *x* is denoted by $-x$.

3.1 Connection

The basic changes to a resource graph *RG* are summarized in Table 1.

Table 1. Impacts of the basic changes to RG.

(x,r, y)	Interaction	Identification	Representation
$\rightarrow(x, r, z)$	$+p(u(z))$	$+u(z)$	$l(r, u(y)) \rightarrow l(r, u(z))$ *in* $h(u(x))$
$+(x, r, z)$	$+p(u(z))$	$+u(z)$	$+l(r, u(z))$ *in* $h(u(x))$
$\rightarrow(x, s, y)$	*no*	*no*	$l(r, u(y)) \rightarrow l(s, u(y))$ *in* $h(u(x))$
$+(x, s, y)$	*no*	*no*	$+l(s, u(y))$ *in* $h(u(x))$
$-(x, r, y)$	$-p(u(y))$	$-u(y)$	$-l(r, u(y))$ *in* $h(u(x))$

When a REST API replaces resource *y* by *z* or adds resource *z*, the REST API must create new identification *u(z)* in graph *IG*. The new identification leads to new interaction operation *p(u(z))*. However, if resource *z* already exists in the REST API, there will be no changes to the interaction operations or identification graph.

A REST API can replace the connection between the resources. Since the resources remain the same, no changes are necessary to the interaction operations and graph *IG*, except that the new connection must be advertised in the *HG* as hyperlinks.

When a REST API removes a connection including its resource, it needs to remove the corresponding interaction operation, identification, and hyperlink properly. However, if resource *y* is still connected in the REST API, then no change to graph *IG* is needed.

When a REST API changes connections between the same resources, it will break those clients that navigate those connections. To deal with such changes, a client can backtrack to rediscover the connections. For example, if the REST API changes connection $a \rightarrow b \rightarrow c$ to $a \rightarrow d \rightarrow c$, then the client cannot navigate to resource *c* from *b*. The client can backtrack to resource *a* and follow *d* to *c* instead. This technique can eventually find any resources connected to an entry resource.

3.2 Interaction

The basic changes to interaction operations are summarized in Table 2.

The first two rows show that a REST API can change the protocol to interact with identification *u(x)*, by associating different URI resolution procedures with the hypertext.

Table 2. Impact of basic changes to interaction operations.

p(u(x))	Connection	Identification	Representation
$\rightarrow q(u(x))$	*no*	*no*	$\rightarrow h(u(x))$
$+q(u(x))$	*no*	*no*	$\rightarrow h(u(x))$
$-p(u(x))$	*no*	*no*	$-l(r, u(x))$ *in* $h(u(x))$

The last row means a REST API can remove the protocol for a URI without removing the URI itself. A URI without a protocol identifies a resource that cannot be resolved from this hypertext.

When a REST API replaces a protocol, it will break those clients that depend on the protocol. Although a client can prepare some common protocol stacks, it will increase the footprint of the client, and there is still no guarantee that it will cover the protocols used by a new REST API. For the same reason, it is better to install a small set of common protocol stacks in a client, and add the new ones required by a REST API when needed.

3.3 Identification

The changes to *IG* and impacts are summarized in Table 3.

Table 3. Impact of changes to identification graph IG.

u(x)	Connection	Interaction	Representation
$\rightarrow v(x)$	*no*	*maybe*	$l(r, u(x)) \rightarrow l(r, v(x))$
$+v(x)$	*no*	*maybe*	$+l(r, v(x))$
$-u(x)$	*no*	*maybe*	$-l(r, u(x))$

A REST API can replace, add or remove identifications without modifying the underlying connection graph *RG*. When a new identification *v(x)* is introduced, a new interaction operation may be needed, and a new hypertext is always necessary. When identification *u(x)* is removed, the corresponding interaction operation becomes useless, if no other identifications use the operation.

Replacing identification *u(x)* may break those clients that do not know or understand the new identification. A REST API can use HTTP redirection to "teach" a client the new identification using the old identification. A REST API can also use a URI template [28] to prepare its clients for the range of URI changes. However, these techniques require that the new and old identifications use the same interaction operations. If new interaction operations are introduced, the client has to be modified.

3.4 Representation

The changes to HG and impacts are summarized in Table 4.

Table 4. Impact of changes to a representation.

$m(h(u(x)))$	Connection	Interaction	Identification
$\rightarrow n(h(u(x)))$	*no*	*yes*	*no*
$+n(h(u(x)))$	*no*	*yes*	*no*

In these changes, we assume the new media type *n* (e.g. JSON) is equivalent to *m* (e.g. XML). Therefore, there is no change to the resource connections or identifications. However, the interaction protocol messages have to be changed to transmit the new media type.

Although a client can use content negotiation to accept a set of media types, there is no guarantee that these media types will overlap with those supported in a REST API. When a REST API replaces a media type, it breaks those clients that depend on that removed media type. Similarly, adding a new media type requires reprogramming the client in order to use it.

If a REST API modifies the specification of a media type, including its structure and processing rules, a client must be reprogrammed, because hypertext-driven navigation procedures depend on the media types.

4 REST Chart Model

To localize the impact of REST API changes on client, we decompose a client into two functional components: (1) client oracle that copes with changes in connection and identification; and (2) client agent that copes with changes in representation and interaction. In hypertext-driven navigation, a client oracle selects hyperlinks to visit from hypertext, while a client agent interacts with the selected hyperlinks to obtain the new hypertext. In order to define the compatibility between two REST APIs from the perspective of clients, we formalize these two concepts into a client model. Since any REST client model is dependent on the REST API it uses, we develop a formal model of REST API and use it to derive the client model. By defining compatibility based on the abstract models instead of concrete REST API implementations, we are able to derive a generic and efficient algorithm to check the compatibility without REST API implementations.

The REST API model we choose is REST Chart (RC) [14], which is originally proposed to design and describe REST APIs without violating the REST principles [1]. REST Chart models a REST API as a Colored Petri Net [23] that can model concurrent system with complex data structures as colors. The structure and behavior of REST Chart can be explained with a simple example in Fig. 2. This REST Chart with one transition, two input places, and one output place. It models a login REST API with one resource. The login place contains a token x_1 which is the entry point to the REST API. The credential place is initially empty. To navigate to the account place, a client must deposit a valid token x_2 in the credential place. The two tokens x_1 and x_2 cause the transition to fire, emulating the interaction with the resource. If the credential x_2 is

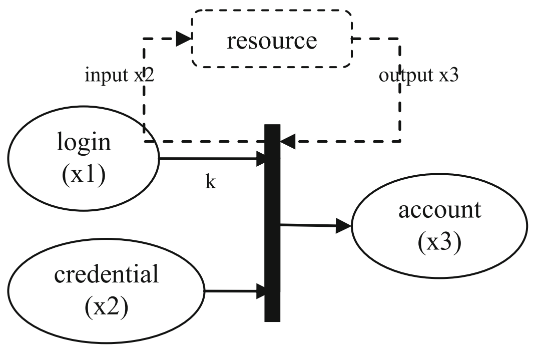

Fig. 2. Example of a basic REST Chart.

accepted by the resource, a token x_3 representing the account information will be deposited in the account place by the REST API.

The above interaction can be modeled by a sequence of token markings of the Petri Net, where each token is a valid resource representation. As this REST Chart has 3 places, each token marking is a 3 dimensional vector and the interaction involves 3 token markings:

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

Token x_1 may have many hyperlinks besides the login. To distinguish the login hyperlink *h* from the rest, REST Chart adopts Predicate/Transition Petri Net [22] 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 [25, 26]:

1. *[service]:* a URI [27] that represents the service provided by the hyperlink.
2. *[reference]:* a URI Template [28] that identifies the locations of the resource.

Two hyperlink predicates *k1* and *k2* are equal if *k1.[service] = k2.[service]* and *k1.[reference] = k2.[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 Fig. 2 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 performed by client oracle and client agent: (1) the oracle selects a hyperlink at each place to move towards the goal place, in this case the account place; (2) the agent moves tokens, e.g. x_2 and x_3, between the places by interacting 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, a REST client needs the right kind of client oracle and client agent for that maze.

Within the REST Chart model, the changes in the layers of a REST API correspond to changes to a *p-k* path:

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

Without losing generality, we assume that this path exists in version 1.0 of a REST API and the version 2.0 changes the path in the following ways.

Case 1: the new path is identical to the original *path*. Obviously, a version 1.0 client 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-k_0 p_2-k_2 p_3-0 removes pair p_1-k_1, whereas new path p_0-k_0 p_2-k_2 p_1-k_1 $p_{3\text{-}0}$ reorders the original pairs. A version 1.0 client can keep its client oracle and client agent, because the client oracle can select the same hyperlink at the same place in version 2.0.

Case 3: the new path consists of different pairs combined from the same *p* and *k*. For example, new path p_0-k_2 p_1-k_0 p_3-0 changes the hyperlinks selected at p_0 and p_1. For a version 1.0 client to traverse the new path, it needs a new client oracle that selects k_2, instead of k_0, at p_0. But the client 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-k_3 p_4-k_4 p_3-0 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 a version 1.0 client agent cannot fire, while new places mean new schemas and tokens that the client agent cannot process. For this reason, the version 1.0 client has to update both its client agent and client oracle. However, if the new places and transitions in version 2.0 are *covered* by some places and transitions in version 1.0, then the client can update its client oracle but keep its client agent.

5 RC Operational Model

In order to identify the compatible paths between REST Charts, this section introduces a deterministic operational model for REST Chart based on a state-based Petri Net behaviour model (Murata 1989). This model leads to a formal model of client oracle and client agent, from which the REST Chart compatibility checking algorithm is derived.

5.1 RC Behaviour Model

A REST Chart is a bipartite graph $RC = (P, T, F, M0, p0, L, S, K, type, link, bind)\}$, 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: $P \rightarrow \{0, 1, 2, \ldots\}$ 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 $A = [aij]$ be the $n \times m$ incident matrix of integers, whose entry is given by:

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-} \tag{1}$$

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

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:

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

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 to indicate that 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:

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

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

Among the three factors of Eqs. (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 Sect. 5.2.

5.2 REST Client Model

Our REST client model represents the hypertext-driven navigation shown in Fig. 1 in terms of Petri-Net. In particular, a *client agent* A of a REST Chart RC consists of the following abstract procedures that operate on tokens in a place:

- $(H, d) = decode(p, l, x)$: decode 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.
  - $valid(type(p, l), x)$: if it is true, then it indicates that token x is an instance of schema $type(p, l)$ at place p with language l.
- $x = encode(p, l, (H, d))$: encode 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:

- *$(k, t, p_j, H_j, d) = select(p_i)$:* select 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 the 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.

```
reach(l, Q, A, V, $p_0$, $x_0$)
    ($H_0$, $d_0$) = A.decode($p_0$, l, $x_0$)
    if Q.goal($d_0$) then return V
    (k, t, $p_1$, $H_1$, $d_1$) = Q.select($p_0$)
    if h $\in H_0 \wedge$ k(h) then
        V = V $\cup$ {($p_0$, $x_0$, t)}
        $x_1$ = A.encode($p_1$, l, $H_1$, $d_1$)
        ($p_2$, $x_2$) = A.fire(t, h, $x_1$)
        return reach(l, Q, A, V, $p_2$, $x_2$)
    end
    return V
end
```

Listing 1. REST client operational model.

6 REST Chart Compatibility

The compatibility between two REST Charts RC1 and RC2 can be defined in terms of the client operation model introduced in Sect. 5. 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. *$C = (Q_{RC2}, A_{RC1}, reach)$;*
2. *$(p, x) = reach(Q_{RC2}, A_{RC1}, M_0, p_0, x_0)$*;

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 Sect. 4, 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 Sect. 4.2. The situation is illustrated in Fig. 3, where client C has a token in place p_{02} and its oracle Q_{RC2} selects door t_2 to enter place p_{22}.

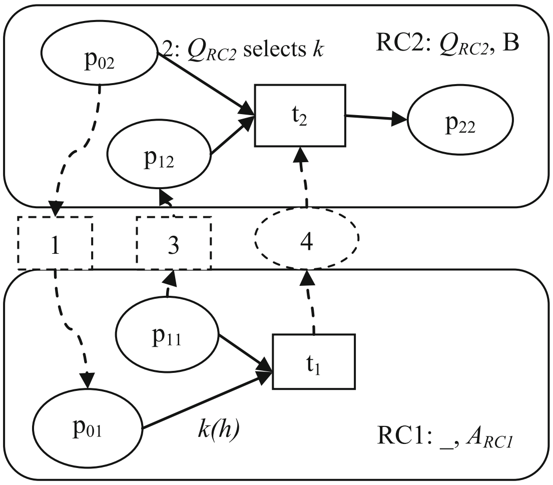

Fig. 3. Client C uses agent A of RC_1 to fire a transition of RC_2.

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 that 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 Fig. 3. 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 5 (for brevity, the subscripts of Q and A are omitted).

Table 5. 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 6. 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 5 must succeed in the right order. However, these conditions rule out non-validating client agents that do not use schemas. For this reason, Table 5 summarizes the necessary conditions for finding a compatible path. Using the hyperlink predicate equality relation defined in Sect. 4 and the schema coverage relation defined in Sect. 5.2, the Table 5 conditions can be reduced to Table 6, 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 RC2 aided by RC1 as outlined in Listing 2.

```
traverse(RC2, RC1, V, $p_0$)
   if $p_0 \in$ V then return
   V = V $\cup$ {$p_0$}
   K = RC2.link($p_0$)
   for each k $\in$ K
      $t_2$ = RC2.bind($p_0$, k)
      if $t_1$ = RC1.cover($t_2$) then
         V = V $\cup$ {($t_2$, $t_1$)}
         $p_2$ = RC2.output($t_2$)
         traverse(RC2, RC1, V, $p_2$)
      end
   end
end
```

Listing 2. REST Chart comparison algorithm.

Procedure *RC_1.cover()* finds a transition t_1 in RC_1 equivalent to transition t_2 in RC_2 based on the Table 6 conditions without actually constructing the auxiliary places and transitions in Fig. 3. 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.

7 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 Fig. 4, where a compatible place in the new version is marked by arrows pointing to the old places that cover it.

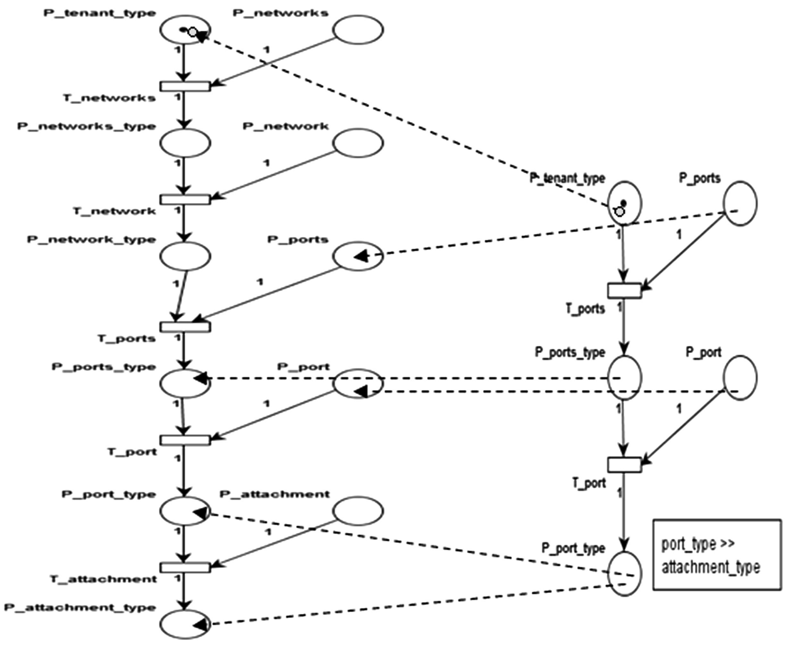

Fig. 4. Compatible places between version 1.0 (left) and version 2.0 of Floodlight REST API.

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. Given two XML schemas s_1 and s_2, the procedure *compare(s_1, s_2)* returns the differences between them as set D of pairs *(e, op)*, where e denotes an element in s_1 or s_2 and op denotes the operation that adds, removes or modifies the position or type of e:

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

Certain pairs in D create incompatible differences such that some XML documents validated by s_1 are not validated by s_2. Let B be the set of such incompatible differences, where *e.min* is the minimum occurrence of element e, and $t_1 \succ t_2$ indicates that type t_1 is a super type of type t_2:

$B = \{\mathrm{e} | (e, remove) \in D \vee (e, move) \in D \vee ((e, add) \in D \wedge e.min \neq 0) \vee (e, type, t_1, t_2) \in D \wedge (\mathrm{t}_1 \succ \mathrm{t}_2))\}$.

Let G be the set of compatible differences:

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

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.*

Table 7. Performance summary.

Charts	RC1	RC2	Time (ms)
Place	17	19	1179.4
Transition	17	21	30.3
Place	8	10	1063.8
Transition	10	11	55.9

To test the correctness of the algorithm, we took a REST Chart and generated a dozen versions of it by changing the places, transitions, and schemas in various ways. Then the outputs of the algorithm against the changes were verified. The performance of the algorithm is summarized in Table 7 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 2.67 GHz with 4 GB RAM). The results show that the algorithm spent extra (1179.4 − 1063.8) = 115.6 ms when the complexity factors increased by (19 + 21) * 17/((10 + 11) * 8) = 4 times from the Coffee REST API to the SDN REST API.

8 Conclusions

Compatibility checking of REST API is important, because extensibility is one of the main but underutilized advantages in RESTful service framework. Dealing with compatibility at large scale requires not only a well-designed REST API, but also automated methods to detect changes in a REST API. This paper defines compatibility between REST API from the perspective of its clients, and it develops an efficient algorithm to check the compatibility between two REST API models based on Coloured Petri Net. Our approach is independent of the REST API implementations, and it promotes REST client reusability by decomposing the client into the functional modules of client oracle and 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

1. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Dissertation, University of California, Irvine (2000)
2. GSMA OneAPI (2013). http://www.gsma.com/oneapi/voice-call-control-restful-api/. Accessed 20 August 2015
3. OpenStack REST API v2.0 references. http://developer.openstack.org/api-ref.html. Accessed 20 August 2015
4. Floodlight REST API. http://www.openflowhub.org/display/floodlightcontroller/Floodlight+REST+API. Accessed 20 August 2015
5. OpenStack API Complete Reference. http://developer.openstack.org/api-ref.html. Accessed 20 August 2015
6. OpenStack Releases. https://wiki.openstack.org/wiki/Releases. Accessed 20 August 2015
7. Hadley, M.: Web Application Description Language, W3C member Submission, 31 August 2009. http://www.w3.org/Submission/wadl/. Accessed 20 August 2015
8. RAML Version 0.8. http://raml.org/spec.html. Accessed 20 August 2015
9. Swagger 2.0. https://github.com/swagger-api/swagger-spec. Accessed 20 August 2015
10. Robie, J., et al.: RESTful Service Description Language (RSDL), Describing RESTful services without tight coupling. In: Balisage: The Markup Conference 2013 (2013). http://www.balisage.net/Proceedings/vol10/html/Robie01/BalisageVol10-Robie01.html. Accessed 20 August 2015
11. API Blueprint Format 1A revision 7. https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md. Accessed 20 August 2015
12. Gomadam, K., et al.: SA-REST: Semantic Annotation of Web Resources, W3C Member Submission, 05 April 2010. http://www.w3.org/Submission/SA-REST/. Accessed 20 August 2015
13. Alarcon, R., Wilde, E.: Linking data from RESTful services. In: LDOW 2010, Raleigh, North Carolina, 27 April 2010
14. Li, L., Chou, W.: Design and describe REST API without violating REST: a petri net based approach. In: ICWS 2011, Washington DC, USA, pp. 508–515, 4–9 July 2011
15. Robie, J.: RESTful API Description Language (RADL) (2010). https://github.com/restful-api-description-language/RADL, 2014. Accessed 20 August 2015
16. Champin. P-A.: RDF-REST: a unifying framework for web APIs and linked data. In: Services and Applications over Linked APIs and Data (SALAD), Workshop at ESWC, May 2013, Montpellier (FR), France, pp. 10–19 (2013)
17. SOA Membrane WSDL tool. http://www.membrane-soa.org/soa-model-doc/1.4/cmd-tool/wsdldiff-tool.htm. Accessed 20 August 2015

18. WSDL Auditor. http://wsdlauditor.sourceforge.net/. Accessed 20 August 2015
19. WSDL Comparator. http://www.service-repository.com/comparator/compareWSDL. Accessed 20 August 2015
20. Thompson, H.S., et al.: XML Schema Part 1: Structures Second Edition, W3C Recommendation, 28 October 2004. http://www.w3.org/TR/xmlschema-1/. Accessed 20 August 2015
21. SOA Membrane XSD tool. http://www.membrane-soa.org/soa-model-doc/1.4/cmd-tool/schemadiff-tool.htm. Accessed 20 August 2015
22. Murata, T.: Petri Nets: properties, analysis and applications (invited paper). Proc. IEEE **77**(4), 541–561 (1989)
23. Jensen, K., Kristensen, L.M.: Colored petri nets: a graphical language for formal modeling and validation of concurrent systems. Commun. ACM **58**(6), 61–70 (2015)
24. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 2nd edn. Springer, New York (2008)
25. Li, L., Chou, W.: InfoParser: infoset driven XML processing for web services. In: ICWS 2008, Beijing, China, pp. 513–520, September 2008
26. Li, L., Chou, W.: Infoset for service abstraction and lightweight message processing. In: ICWS 2009, Los Angeles, pp. 703–710, July 2009
27. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax, Request for Comments: 3986, January 2005. https://tools.ietf.org/html/rfc3986. Accessed 20 August 2015
28. Gregorio, J., et al.: URI Template, Request for Comments: 6570, March 2012. https://tools.ietf.org/html/rfc6570. Accessed 20 August 2015