Design and Describe REST API without Violating REST: A Petri Net Based Approach

Li Li
Avaya Labs Research
Avaya Inc.
Basking Ridge, NJ, USA
lli5@avaya.com

Wu Chou
Avaya Labs Research
Avaya Inc.
Basking Ridge, NJ, USA
wuchou@avaya.com

***Abstract*—As REST architectural style gains popularity in the web service community, there is a growing concern and debate on how to design RESTful web services (REST API) in a proper way. We attribute this problem to lack of a standard model and language to describe a REST API that respects all the REST constraints. As a result, many web services that claim to be REST API are not hypermedia driven as prescribed by REST. This situation may lead to REST APIs that are not as scalable, extensible, and interoperable as promised by REST. To address this issue, this paper proposes REST Chart as a model and language to design and describe REST API without violating the REST constraints. REST Chart models a REST API as a special type of Colored Petri Net whose topology defines the REST API and whose token markings define the representational state space of user agents using that API. We demonstrate REST Chart with an example REST API. We also show how REST Chart can support efficient content negotiation and reuse hybrid representations to broaden design choices. Furthermore, we argue that the REST constraints, such as hypermedia driven and statelessness, can either be enforced naturally or checked automatically in REST Chart.**

Keywords: RESTful web service; Petri Net; REST Chart;

I. INTRODUCTION

REST stands for Representational State Transfer. It is a term coined by Roy Fielding in his dissertation [1] to refer a software architectural style. According to [1], an architectural style is a coordinated set of architectural constraints that restricts the roles and features of architectural elements, and the allowed relationships among those elements within any architecture that conforms to the style. In other words, we may regard an architectural style as a set of design patterns with which we can create architectures that exhibit the properties induced by the style. Derived from 12 other related styles, REST is a hybrid (union) style that aims to induce certain architectural properties that are important to distributed hypermedia systems. These properties include usability, simplicity, scalability and extensibility. By introducing these properties with carefully considered tradeoffs, REST attempts to minimize latency and network communications, and at the same time, maximizing the independence and scalability of component implementations, including user agent, proxy, gateway, origin server and various connectors, in distributed hypermedia systems.

Because of the close relationship of REST with the Web and HTTP, there are two main confusions about REST. One is to equate REST with the Web. This assumption is wrong because the Web can incorporate architectural styles other than REST and REST can be used outside the Web as well. But the most common use cases of the Web, the distribution of hypermedia content, are guided by REST. The other confusion is to equate REST with HTTP. This confusion often leads to two assumptions: 1) REST can only be realized by HTTP; and 2) any web service defined in terms XML over HTTP is REST. The first assumption is wrong because REST, as an architectural style, is protocol independent, even though HTTP is currently a protocol that supports REST. The second assumption is wrong because REST imposes more constraints than HTTP. Therefore, HTTP is neither a sufficient nor a necessary condition for REST.

One of the important REST constraints not enforced by HTTP is "hypermedia as the engine of application state" [1]. This constraint states that a REST API should be driven by nothing but hypermedia. This is how a web browser interacts with well-behaved web applications, where the browser transitions to different pages based on the selected hyperlinks and actions present in the pages. Since HTTP has been used to transfer information that is not hypertext, this constraint is often ignored by REST API designs. Instead of defining API in terms of hypermedia, some so-called REST APIs are modeled as a set of interfaces implemented by resources. Although this design supports the REST uniform interface constraint, it inevitably creates fixed resource names, types and hierarchies that violate the REST API design rules prescribed by Roy Fielding [2]. This kind of violations leads to an API that depends on the out-of-band information, instead of hypermedia, to drive the interactions between components.

The reason for these confusions to large extent is due to lack of a standard modeling language to define REST API. Without such a definition language that codifies the REST constraints, a REST API designer has to make decisions based on his own interpretation of what REST means. As discussed above, this often leads to an API that is not RESTful. Furthermore, lack of a definition language will lead to a situation where REST APIs are defined in different

978-0-7695-4463-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ICWS.2011.54

formats, creating an unnecessary layer of data coupling between components that is difficult to scale in large distributed systems across organizational boundaries.

There are other reasons for a REST API definition language. One is for standard development where interested parties work together to design a common REST API before it is implemented. To avoid ambiguity, the REST API needs to be specified in a more precise way than plain English, such that independent implementations can be interoperable. A REST API definition language, which is machine-readable, can also help with the design process and code generation to facilitate the implementation and consumption of the REST API. Moreover, a REST API defined in a standard modeling language can facilitate automated service discovery, service selection, and service composition - all depend on some kind of functional specifications and modeling of the services.

To address this critical issue, we propose REST Chart as a model and a definition language for REST API. The goal of REST Chart is to embody the relevant REST constraints to guide REST API designs. To achieve this goal, REST Chart abandons the notion that an API is a set of interfaces, which is a model for many RPC (Remote Procedure Call) based distributed systems. Instead, REST Chart models a REST API as a set of hypermedia representations and transitions between them, such that a REST API will be driven by hypermedia. This conceptual model is transformed into a special Colored Petri Net whose topology defines a REST API and whose token markings define the representational state space of user agents using that API. Based on this Petri Net model, an XML dialect is developed to encode the Petri Net to guide the design and validation of REST API.

The rest of the paper is organized as follows. Section II reviews the related work, especially the REST API design rules [2] that motivated this work. Section III presents the detail of REST Chart model and language with an example. Section IV discusses the implications of REST Chart on REST constraints and we conclude with Section V.

II. Related Work

Roy Fielding explains how REST API should be driven by hypermedia (hypermedia constraint) with 6 rules as quoted below [2], (the rules are numbered here for ease of reference):

R1. A REST API should not be dependent on any single communication protocol, though its successful mapping to a given protocol may be dependent on the availability of metadata, choice of methods, etc. In general, any protocol element that uses a URI for identification must allow any URI scheme to be used for the sake of that identification. *[Failure here implies that identification is not separated from interaction.]*

R2. A REST API should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols, such as HTTP's PATCH method or Link header field. Workarounds for broken implementations (such as those browsers stupid enough to believe that HTML defines HTTP's method set) should be defined separately or at least in appendices, with an expectation that the workaround will eventually be obsolete. *[Failure here implies that the resource interfaces are object-specific, not generic.]*

R3. A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). *[Failure here implies that out-of-band information is driving interaction instead of hypertext.]*

R4. A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations. *[Failure here implies that clients are assuming a resource structure due to out-of band information, such as a domain-specific standard, which is the data-oriented equivalent to RPC's functional coupling].*

R5. A REST API should never have "typed" resources that are significant to the client. Specification authors may use resource types for describing server implementation behind the interface, but those types must be irrelevant and invisible to the client. The only types that are significant to a client are the current representation's media type and standardized relation names. *[ditto]*

R6. A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations. The transitions may be determined (or limited by) the client's knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand). *[Failure here implies that out-of-band information is driving interaction instead of hypertext.]*

Here "hypertext" is used as a synonym for "hypermedia" which refers to data that combine control information with presentation information.

Content negotiation is another important part of REST API. HTTP 1.1 supports three types of content negotiations [3]: 1) server-driven where the origin server determines the representation for the user agent, based on user agent's preferences; 2) agent-driven where the user agent selects the representation from available ones on the server; 3) transparent where a cache combines the two types of content negotiation.

The disadvantages of server-driven negotiation include: 1) the origin server cannot accurately determine what is best for the user agent; 2) it requires user agent to send preference on every request; 3) it complicates the implementation of origin servers; 4) it may limit a public cache's ability to use the same response for multiple user agents. Agent-driven negotiation avoids these problems but it requires a second request to retrieve the best representation, which is inefficient.

An alternative to the above negotiation mechanisms is to express the available media types in the REST API. This approach enables agent-driven negotiation without the need for a second request, as the user agent can select the best representation from the REST API directly. The disadvantage of this approach is that an origin server cannot change media types at runtime. But in most cases, the available media types for a REST API are unlikely to change frequently. For this reason, we introduce a new rule in addition to R1-R6 from Fielding [2]:

R7. A REST API should support multiple representations of a resource *[Failure here implies that identification is not separated from representation]*.

WSDL 2.0 [4] is a W3C specification that was designed to describe REST API, in addition to SOAP based web services. Using WSDL 2.0, a REST API can be defined in two parts: 1) a WSDL file defines the entry point of the REST API (R6); and 2) WSDL interfaces and bindings attributes inserted to XML schemas that define the transition relations between the schemas (R3). An example that demonstrates this approach can be found at [5]. Although this approach conforms to the hypermedia constraint, it lacks the notion of media types as XML is assumed in typical use cases. Without support for media type, it is difficult, if not impossible, to define a REST API that is not based on XML, for example, JSON, or a REST API that supports different media types through content negotiations. Modifications to XML schemas also create two problems: 1) a developer has to look at both schemas and WSDL to figure out the transitions; 2) the information in XML schemas are no longer shared by both SOAP and REST APIs as the modifications are only for REST API. On the other hand, WSDL 2.0 is too complex for REST API as many of its components are designed for custom interfaces instead of uniform interfaces. For example, the interface component in SOAP based web services can contain multiple operations, whereas in REST API there is only one operation attached to each URI element [5]. In WSDL 2.0, interfaces can also be extended, whereas there is no such need in REST API. It is possible to profile WSDL 2.0 for REST, but profiling cannot remove required elements and attributes.

WADL, Web Application Description Language [6], is a W3C member submission that aims to provide machine processable description of HTTP-based web applications. WADL describes a web application as hierarchies of resources, each of which has a set of methods from the uniform interface. Each method has request and response that contain representations grounded on XML elements. WADL also introduces resource types that define behaviors for multiple resources. Even though WADL has been used to describe REST API [7], the model taken by WADL is inconsistent with the hypermedia constraint:

- R3 is violated as what HTTP methods to use in what URIs is defined outside the scope of media types;
- R4 is violated as it introduces resource hierarchies;
- R5 is violated as typed resources are permitted;
- R6 is violated as multiple resources tend to expose multiple entry points;

Due to these violations, web applications described with these WADL features cannot be RESTful.

RIDDLE [8] is another attempt to develop a description language for REST API. However, RIDDL also ignores the hypermedia constraint by organizing a REST API around resource tree and interfaces as WADL does.

There are some research efforts that use Petri Net for process enactment that invokes RESTful web services [9][10]. However, the Petri nets used there are to describe the processes based on the web service interfaces. It does not constitute the interfaces for web services, as a process is just one way to use web services in sequence.

III. REST CHART MODEL

In this section, we present REST Chart as a model to describe REST API in terms of media types. The hypermedia constraint rules R1-R7 in section II are treated as requirements for the REST Chart model.

A. Media Type Representation

A REST API typically uses one or more media types and many data types within one media type. For example, a banking service may choose `application/xml` media type for all its resource representations. Each of the resource representations, such as user login and account, has its own XML Schema (or some other type definition language). We treat these XML Schemas as part of the media type specification of a REST API. This approach has two advantages. First, the schemas are necessary for a user agent to understand the resource representations in the REST API. Second, we can create as many schemas as necessary within one media type, without having to invent and register any new media types. This is actually how `text/html` media type is defined: it contains many DTD rules to specify various components of HTML.

To generalize this approach, we distinguish two types of representations: *type representation* and *resource*

representation. Type representation represents the media type and schemas used by a REST API. The schemas could be defined by any type definition language, such as XML Schema, RELAX-NG, DTD, or even presentation language like HTML tables. To facilitate processing, a schema may have its own media type (e.g. `application/relax-ng-compact-syntax`). Resource representations are instances of the schemas of the type representations. For example, a valid account XML representation should be an instance of the account schema. It should be pointed out that "instance" does not imply that schema validations are required to process the resource representations.

These relations between media type, type representation, and resource representations are illustrated in the UML class diagram in Figure 1.

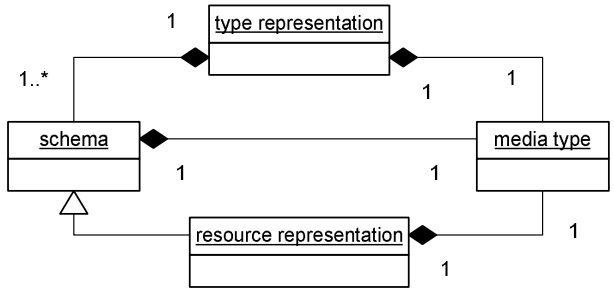

Figure 1: Media Representation Model

B. REST Chart Model

The REST Chart model treats a REST API as a set of type representations connected by transitions. Each transition specifies the possible interactions with the resource referenced by a hyperlink in a type representation. The model uses a special kind of Colored Petri Net [11][1] to define these type representations and transitions. In particular, a type representation becomes a Petri Net place that can have Petri Net tokens denoting resource representations of that type (color). A transition always has two input places and several output places (response and faults) and one of the input places contains the hyperlink to the resource. The hypermedia constraint is enforced by the Petri Net transition rules: 1) a transition can be fired if and only if all its input places have the right kind of tokens; 2) After a transition fires, its output place will have the right kind of tokens.

This idea is elaborated by the following example. Suppose a REST API for a banking application asking a user agent to login with user name and password in order to access the account information. To represent this logic in hypermedia, the login type representation contains a hyperlink to which a user agent can submit the credential data and receives the account information. This REST API is a Petri net with three places (type representations) and one transition connecting the three places (Figure 2). This Petri net indicates that the user agent can transfer its representational state from the login place to the account place aided by the credential place. To make the transfer, the user agent must first put a token x1 in the login place and a token x2 in the credential place – it must obtain a login document with a hyperlink (x1) and created valid credential data (x2). Then the user agent fires the transition by submitting the credential to the resource at the hyperlink. On success, the account resource representation is obtained from the response. As the result, this transition causes the third token x3, the account document, to occur in the account place. The token marking of the Petri net after this firing is shown in Figure 2. The effect of this interaction on Petri Net can be modeled by a time delay for the transition.

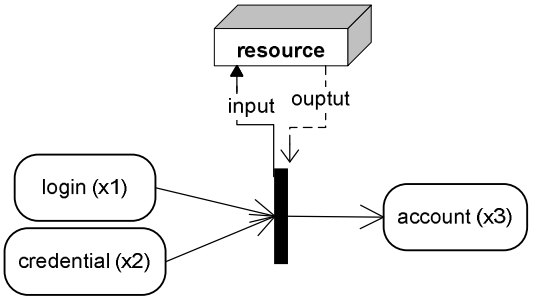

Figure 2: A Petri net with interaction on transition

When firing a transition, a user agent uses a client connector to interact with the origin server. Section 5.2.2 in Chapter 5 of [1] provides a high level description about the uniform interface to be implemented by the client connectors. The interface consists of input and output parameters. In each interaction, the input and output parameters contain: 1) control data (e.g. HTTP verbs and status); 2) optional metadata about the resource representation (e.g. HTTP headers); and 3) an optional resource representation. In addition, a user agent also needs to know the transfer protocol, e.g. HTTP, in order to call the right client connector. These data thus constitute the interaction instructions for firing the transitions.

To force a user agent to start from an initial URI, a Petri net can be marked with one token in the initial place. This marking is also the initial representational state of the user agent. As the user agent fires the enabled transitions based on its selection of hyperlinks, it creates more tokens (resource representations) in different places. These markings represent different representational states of the user agent. In typical situations, the user agent reaches the goal state when a token (e.g. confirmation of a money transfer) occurs in the desired place. The enabled transitions can fire sequentially or concurrently, depending on the search strategy of the user agent.

This Petri Net based approach to REST thus achieves the following goals within one model:

1. Its topology defines the transition relations between hypermedia representations of a REST API.
2. Its token markings define the representational state space of any user agent that uses the REST API.
3. Its transitions define the possible interactions (data formats and protocols) between the user agent and the resources.

Furthermore, this Petri Net model provides a theoretic framework to study and verify the important properties for a REST API, such as deadlock detection and resource consumption [11], which are outside the scope of this paper.

C. XML Representation of REST Chart

To facilitate REST API design, we propose an XML markup language for REST Chart using the following RELAX-NG grammar rules.

```
reference = element representation {
    attribute ref { text },
    attribute link { xs:anyURI }?
}
anyAttribute = attribute * { text }
anyElement = element * {
  (anyAttribute | text | anyElement)*
}
group = (
  element control { anyAttribute+ },
  element metadata { anyAttribute+ }?,
  reference?
)

element rest_chart
{
  element include {
    attribute location { xs:anyURI }
  }*
  element representation {
    attribute id { text },
    attribute media_type { text },
    attribute charset { text }?,
    attribute encodings { text }?,
    attribute languages { text }?,
    attribute initial { xs:boolean }?,
    element schema {
      attribute location { xs:anyURI }?,
      attribute media_type { text },
      anyElement
    }?,
    element alternatives { reference+ }?,
    anyElement
  }+

  element transition {
    element input {
      attribute uri_templates { text },
      group
    }
    element output {
      group
    }+
  }+
}
```

A REST Chart consists of two basic elements: type representation and transitions. A type representation contains attributes for media type name, character set, encodings and languages. It may also contain alternative type representations to facilitate content negotiation (R7). The schema of a type representation may be inline or located somewhere else. We also model the initial URI (R6) to a REST API as a special resource representation, to be uniform and generic.

A transition contains input and output elements. A transition always selects a hyperlink from a type representation in one of the input elements. The hyperlink is defined by the "link" attribute in the reference to the type representation. The URI template of that hyperlink is defined on the input element containing the reference. The URI template can specify the protocols to interact with the resource, without defining any fixed resource names, hierarchies or types (R1, R2, R4 and R5). By referring the hyperlinks from transitions, REST Chart avoids modifications to XML schemas as WSDL 2.0 does. A REST Chart thus defines the processing rules for the media types used by an REST API. As the result, the entire REST API is defined in media types (R3).

D. A REST Chart Example

To illustrate how REST Chart is used to derive the REST API for the service, we take a starbucks coffee order service as a concrete example. This service workflow is motivated by the example in [12]. For brevity, we omit namespaces and fault representations. Also we use short strings instead of long URI for link relations and transition references.

First, we create a resource representation to indicate the initial URI for the REST API (see R6):

```
  <representation                id="initial"
media_type="application/xml" initial="true">
  <link                                id="k1"
rel="create">http://starbucks.example.com/ord
er/v1</link>
  </representation>
```

The URI identifies the resource that creates coffee orders, and the inserted transition `t1` tells the user agent that it can move to the `order_payment` type representation by sending an `order_request` with POST to the initial URI (k1):

```
  <transition id="t1">
    <input>
      <representation ref="initial"
link="k1" />
    </input>
    <input>
      <control method="POST"/>
      <representation ref="order_request" />
    </input>
    <output>
      <control status="201"
reason="Created"/>
      <representation ref="order_payment"/>
    </output>
  </transition>
```

The `order_payment` type representation contains a RELAX-NG grammar that defines the inline schema of the type representation. The type representation contains three hyperlink elements to indicate how to update, cancel and pay for the order respectively:

```
   <representation id="order_payment"
media_type="application/xml">
    <schema media_type="application/relax-ng-
compact-syntax">
    element order_payment {
       …
     element link {
       attribute rel {"update"}
       attribute id {"k2"}
       xs:anyURI
      }
     element link {
       attribute rel {"cancel"}
       attribute id {"k3"}
       xs:anyURI
      }
     element link {
       attribute rel {"pay"}
       attribute id {"k4"}
       xs:anyURI
      }
    </schema>
   </representation>
```

Transition `t2` tells the user agent how to update the order with a PUT to link k2 and transition to `order_payment`:

```
   <transition id="t2">
     <input>
       <representation ref="order_payment"
link="k2" />
     </input>
     <input
uri_templates="http://{authority}/*">
       <control method="PUT" />
       <representation ref="order_update" />
     </input>
     <output>
       <control status="200" reason="OK" />
       <representation ref="order_payment" />
     </output>
   </transition>
```

The URI template indicates the communication protocol is HTTP while allowing the origin server to create its own namespaces at runtime (see R4).

Transition `t3` tells the user agent how to delete the order with link k3 and transition to the initial type representation:

```
   <transition id="t3">
     <input>
       <representation ref="order_payment"
link="k3" />
     </input>
     <input
uri_templates="http://{authority}/*">
       <control method="DELETE" />
     </input>
     <output>
       <control status="200" reason="OK" />
       <representation ref="initial" />
     </output>
   </transition>
```

Transition `t4` tells the user agent how to carry out the payment over HTTPS with link k4 and moves to the `order_confirmation` type representation:

```
   <transition id="t4">
     <input>
       <representation ref="order_payment"
link="k4" />
     </input>
     <input
uri_templates="https://{authority}/*">
       <control method="POST" />
       <representation ref="payment_info"/>
     </input>
     <output>
       <control status="201" reason="Created"
/>
       <representation
ref="order_confirmation"/>
     </output>
   </transition>
```

The `order_confirmation` type representation contains three links to allow the user agent to 1) monitor the order status by polling; 2) cancel the order before it is made and get a refund; and 3) subscribe to the order status events:

```
   <representation id="order_conformation"
media_type="application/xml">
   <schema media_type="application/relax-ng-
compact-syntax">
    element order_confirmation {
       …
     element link {
       attribute rel {"self"}
       attribute id {"k5"}
       xs:anyURI
      }?
     element link {
       attribute rel {"cancel"}
       attribute id {"k6"}
       xs:anyURI
      }?
     element link {
       attribute rel {"subscribe"}
       attribute id {"k7"}
       xs:anyURI
      }
    </schema>
   </representation>
```

Again, the hyperlinks can be selected by transitions to tell the user agent how to transfer to various type representations. Due to space limit, the detail specifications of these transitions are omitted.

The Petri net for this coffee order REST API is shown in Figure 3, where each link and transition is labeled.

E. Alternative Representations

If the starbucks service decides to support JSON in addition to XML for its coffee order service, it can add the new type representations as the alternative, without changing the transitions in our approach:

```
   <representation id="order_payment"
media_type="application/xml">
      …
     <alternatives>
        <representation
ref="json_order_payment" />
     </alternatives>
   </representation>
   <representation id="json_order_payment"
media_type="application/json" />
```

In general, the alternative representations are supposed to be semantically equivalent in that they carry the same information in different format. By inspecting the available representations, a user agent can select the best media type without constant content negotiations with the cache or origin server.

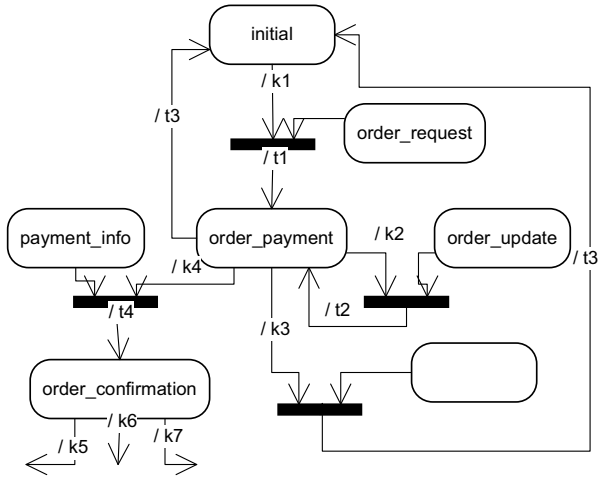

Figure 3: A Petri net for the coffee order service

F. *Hybrid Representations*

In some cases, the media type has no corresponding schema language, such as JSON. In practice, some HTML tables are often used to describe JSON schemas for human readers. The HTML tables are very useful for developers who write the user agent. The tables can also be used by tools for code generation and service discovery. To reuse the HTML content, we simply link the type representations and transitions to the tables.

For example, suppose an HTML table describing the JSON `order_payment` is located at `http://starbucks.example.com/tables#order_payment`, where each row is addressable (Table 1):

property	type
update	xs:anyURI
cancel	xs:anyURI
pay	xs:anyURI

Table 1: A table describing JSON schema

In the REST Chart, we create a JSON type representation whose schema is described by the above HTML table:

```
   <representation     id="json_order_payment"
media_type="application/json">
     <schema
location="http://starbucks.example.com/tables
#order_payment" media_type="text/html"/>
   </representation>
```

This type representation can then be used as input to some transition that selects the "update" hyperlink:

```
   <representation    ref="json_order_payment"
link="#update">
```

Therefore, this approach significantly extends REST API design choices from media types designed only for machines (e.g. XML) to hybrid media types designed for human and machines.

IV. REST CHART AND REST CONSTRAINTS

As our goal is to develop a REST API definition language and modeling framework that codifies the REST constraints, the following sections discuss in more detail the implications of REST Chart with respect to the general REST constraints listed in [7].

A. *Hypermedia Driven*

This constraint states that user agent state change should be driven by hypermedia. This is exactly what happened in a Petri net. The user agent states are token markings, and what marking occurs next depends on which hyperlinks are selected and which transitions will fire. As transitions depend on hypermedia, the state change is driven entirely by hypermedia.

B. *Connectedness*

This constraint states that the resources should be connected. This means that all places in a Petri net should be connected and reachable from the initial place.

C. *Addressability*

The constraint states that each resource should be identified by a URI. We can check this constraint from two directions: 1) all hyperlinks identifying resources should be referenced by transitions; and 2) all transitions should reference a hyperlinks. Test 1) ensures that all resources can be reached and test 2) ensures that all interactions are to resources.

D. *Uniform Interface*

This constraint states that all resources should implement the uniform interface specified by the REST API. This constraint can be enforced by deriving the interface for a resource at a hyperlink from all transitions that select the hyperlink.

E. *Representation*

This constraint states that a user agent must manipulate resources through representations. This constraint is naturally enforced by REST Chart as the input and output places of a transition define all the valid resource representations any user agent can manipulate.

F. Statelessness

To formalize this property, we introduce two concepts into the REST Chart: *idempotent transition* and *stationary place*.

A transition is idempotent if it uses an idempotent operation, i.e. (GET, PUT and DELETE). A transition sequence is idempotent if all its transitions are idempotent. Idempotent transition, by definition, can be repeated many times with the same effect of executing it once. This notion can be extended to non-idempotent transitions (e.g. POST). If for transition t_1 from the place x, there is a "undo" transition t_2 (e.g. DELETE) that goes back to x, then we can regard any sequence containing $t_1\, t_2$ as idempotent.

A place x is stationary if the tokens (resource representation) in the place x never lose any hyperlinks. A transition sequence is stationary if all its input places are stationary. Stationary places guarantee the available resources will not disappear when the places are revisited. For example, if a place at time t has a token with a hyperlink to transfer money, and at time $t+1$, a token in that place still has that hyperlink, then the place is stationary; otherwise, it is not.

A server is stateless if all requests can be processed in isolation. This means a user agent can fire a transition independent of other transitions in a sequence. Combine this property with the repeatability guaranteed by the idempotent transition and stationary place properties, we can conclude that:

If a server is stateless, then all the idempotent and stationary transition sequences can be repeated by a user agent.

Using this definition, it is easy to see how HTTP Cookie violates the statelessness constraint as it breaks repeatable transition sequences. Suppose a cookie is set during a repeatable state sequence as follows, where t_i and p_j represent transitions and places respectively:

t_1-*GET*$\rightarrow p_1 \rightarrow t_2$-*GET [Set Cookie=x]*$\rightarrow p_2$

Now, when the user agent revisits t_1, it may not reach p_1 as before. Instead it may reach a different place as the cookie changes the original request:

t_1-*GET [Cookie=x]*$\rightarrow p_3$

As statelessness is important to the servers, the repeatability is important to user agents in REST. First, repeatable transitions make it easy for a user agent to recover from partial failures by backtracking. Second, a user agent can become more efficient by using cached representations as the responses are predictable. Third, it enables a user agent to compare the cost of alternative transition sequences to optimize its access to the origin servers.

V. Conclusions

This paper presented REST Chart as a model and markup language to describe and design REST API, and to guard against violations of the REST constraints. Main contributions of this paper are summarized as follows:

- We showed that REST APIs and user agent representational state space can be conveniently modeled by a special type of Colored Petri Net.
- We developed REST Chart, an XML dialect based on our Petri Net model to describe REST API.
- We demonstrated that our REST Chart approach does not change schemas and can facilitate content negotiations and broaden design choices with hybrid representations.
- We showed that the core REST constraints are either enforced naturally or can be checked automatically based on REST Chart.

REST API design has become increasingly important as it impacts both user agents and origin servers. We hope this work can address and clarify some of the confusions or even errors surrounding the REST API design, and therefore, benefits of REST can be fully realized based on a sound REST API description and modeling framework.

References

[1] Roy Thomas Fielding, Architectural Styles and the Design of Network-Based Software Architectures, Ph.D. Dissertation, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[2] Roy Thomas Fielding, "REST API Must Be Hypertext Driven," 28 October, 2008, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[3] Fielding et al, RFC 2616 Hypertext Transfer Protocol - HTTP 1.1, Section 12 Content Negotiation, 1999, http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html.

[4] Roberto Chinnici, et al (ed), "Web Services Description Lanaguage (WSDL) Version 2.0 Part 1: Core Language, W3C Recommendation, 26 June 2007, http://www.w3.org/TR/wsdl20/.

[5] Lawrence Mandel, "Describe REST Web Services with WSDL 2.0", IBM DeveloperWorks, 29 May 2008, http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/.

[6] Marc Hadley, Web Application Description Language, W3C member Submission, 31, August 2009, http://www.w3.org/Submission/wadl/.

[7] Leonard Richardson, Sam Ruby, *Restful Web Services*, O'Reilly, 2007.

[8] J. Mangler, et al, "On the Origin of Services – Using RIDDLE for Description, Evolution and Composition of RESTful Services,", 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp 505-508, 2010.

[9] Xiaodong Wang et al, "An Extended Petri-Net Based Approach for Supply Chain Process Enactment in Resource-Centric Web Service Environment," OTM Conferences (1), 2009, pp. 130-146

[10] Gero Decker et al, "RESTful Petri Net Execution," Web Services and Formal Methods, Springer-Verlag LNCS 2009, vol 5387, pp. 73-87.

[11] C. G. Cassandras, et al, *Introduction to Discrete Event Systems*, second edition, Chpater 4, Springer, 2008.

[12] Jim Webber, et al, "How to Get a Cup of Coffee," InfoQ Explores: REST, Issue #1, March 2010, pp. 44-64.

[13] Kurt Jensen. *Coloured Petri Nets*. Springer Verlag, 1997.