Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 309 (2014) 75–93

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formalization and Verification of REST on HTTP Using CSP

Ting Yuan, Yiting Tang, Xi Wu, Yue Zhang
Huibiao Zhu, Jian Guo

Shanghai Key Laboratory of Trustworthy Computing
Software Engineering Institute, East China Normal University
3663 Zhongshan Road(North), Shanghai, China, 200062

Weijun Qin

State Key Laboratory of Information Security
Institute of Information Engineering, Chinese Academy of Sciences
No.89 Min Zhuang Road, Haidian District, Beijing, China, 100093

Abstract

Representational State Transfer (REST), as a promising software architecture style, has been used in large scale since proposed. But considerable confusions about REST exist and many examples of supposedly RESTful applications violate key REST constraints. In this paper, we focus on the most important constraints of REST, stateless property and hypertext-driven property. First we establish a formal model for REST on HTTP in CSP. In the model, components in RESTful systems communicate with each other using standard HTTP methods and are modeled as CSP processes. From the model we can find the effects of HTTP methods on resources. Then we give formal descriptions for failure cases of stateless, hypertext-driven constraints of REST, and safe, idempotent properties of HTTP methods, within which whether a system breaks REST constraints or basic HTTP requirements can be checked. Furthermore, we use model checker PAT to prove all the constraints hold in our model. In the end, a case study about the process of buying food is mapped to our model to better illustrate the REST concepts and our approach.

Keywords: REST, HTTP, CSP, Stateless, Hypertext-driven

1 Introduction

Protocols and standards like SOAP (Simple Object Access Protocol) [1] and HTTP (Hypertext Transport Protocol) [2] are used to construct web services. REST is a description of potential design principles of current Web architectures. It focuses on the scalability of component interactions, generality of interfaces and independent deployment of components.

The notion of REST was firstly proposed by Fielding in 2000 [3], who is also the co-author of the HTTP RFC [2]. Since then, the research and practice based

http://dx.doi.org/10.1016/j.entcs.2014.12.007

on REST have kept developing. In practice, we can find more and more web APIs which claim to be RESTful, such as the photo sharing application Flickr and the online shopping web service Amazon. Books such as [4] and [5] are published to guide coders to develop RESTful applications. Researchers have done a lot of work related to clarifying REST. In [6], Tilkov et al. introduce REST's key principles in the plainest languages with some easy examples to get readers to know more about them. In [7], Tomayko explains REST in the form of a daily dialog with his wife. The key ideas of the complicated architecture in this article turns out to be easily understood even by a housewife who has no background of computer. Webber et al. use the example of Starbucks to clarify each property of REST [8]. All the articles mentioned above make contributions to understanding better of the REST architecture in the informal way.

As to formal fields, Klein et al. apply temporal logic in describing REST's two key principles [9]. In [10], Zuzak et al. use finite-state machines to model REST and propose a case study about weather forecast to show the feasibility of the model. The model maps the transactions between states to messages requested and replied. What the model ignores is the transfer process of the request, making the communication properties of REST hard to check. In [11], the static resources are described in triple spaces while the dynamic communications are modeled in CCS (Calculus of Communicating System) [12]. The model focuses on the feature of resources and ignores the details of interactions between them. Wu et al. give a basic model of REST architecture in CSP (Communicating Sequential Processes) [13] and check several REST constraints [14]. But considerable confusions about REST exist and the REST architecture is often misunderstood and misapplied. In Fielding's blog [15], he has criticized the design of some RESTful applications. Some systems that claim to be RESTful violate the constraint of hypertext-driven behavior, Flickr is such an example.

Inspired by [14] and [9], we formalize REST on HTTP in CSP. Compared to [14], we focus on the most two important constraints of REST and we give a more detailed representation of the REST behavior, where the method used in a request is specified. The constraints described in our paper are the most confusing concepts of REST. We aim to present a better understanding of REST. And in our paper, we also take resources into consideration, thus effects on resources through request/response between clients and server can be depicted. One of the REST constraint is uniform interface. The constraint stresses the uniform methods to manipulate the resources. And till now, HTTP's four methods, GET, PUT, POST and DELETE, have been put into practice as the uniform methods. Regarding to this, we formalize REST based on HTTP in this paper, with the motivation of clarifying the architecture. We first introduce the basic knowledge of REST on HTTP. Then we use CSP to model a RESTful system which uses HTTP to communicate. Components of Client, Server and Resource are modeled as processes, thus communications between each entity are converted to the communications between CSP processes. We give formal descriptions of failure cases of REST's key constraints, stateless constraint and hypertext-driven constraint, and HTTP's safe property and idempotent property. Besides, model checker PAT (Process Analysis Toolkit) [16] is used to verify the constraints and properties in our model. Finally, giving a case study about the process of buying food, we use our achieved architecture to model the whole process and show the constraints and properties fulfilled in the case.

The remainder of the paper is structured as follows. The next section introduces the preliminaries about CSP and PAT. The overview of REST is shown in section 3. Section 4 presents the formal model of a RESTful system on HTTP in CSP. We give the formal description of REST's key constraints and check them in PAT in section 5. Section 6 shows the case study of buying food. The last section concludes the paper and presents possible future work.

2 Preliminaries

2.1 CSP method

CSP (Communicating Sequential Processes) was first described by Hoare [13]. As developed and evolved constantly, it has already become one of mainstream process algebras. It specializes in describing patterns of interaction in concurrent systems. Due to the powerful expressive ability, CSP has been practically applied as a method for specifying and verifying the concurrent aspects of a variety of different systems, such as real time systems and web services. In CSP, processes are composed of basic processes and actions, and they are connected by operators.

The syntax of CSP is shown below:

$$P,\ Q ::= a \rightarrow P \mid c?x \rightarrow P \mid c!x \rightarrow P \mid P \| Q \mid P[|X|]Q$$

Here P and Q represent processes which have alphabets $\alpha(P)$ and $\alpha(Q)$ denoting the actions that the processes can perform respectively. Meanwhile a and b stand for the atomic actions and c is the name of the channel.

- $a \rightarrow P$ represents that the process first performs action a, then behaves the same as process P.
- $c?x \rightarrow P$ gets a message through channel c and assigns it to a variable x, then behaves like P.
- $c!x \rightarrow P$ sends a message x using channel c, then behaves like P.
- $P \parallel Q$ describes the concurrency of P and Q.
- $P[|X|]Q$ represents that P and Q perform the concurrent events on set X of channels.

More details about syntax of CSP can be found in [13].

2.2 PAT

PAT (Process Analysis Toolkit) [17,16] is a self-contained framework for composing, simulating and reasoning of concurrent, real-time systems and other possible domains. It is designed as an extensible and modularized framework based on CSP and it implements various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. We list some notations in PAT as follows:

- $\#define\ N\ 0$ defines a global constant N which has the initial value 0.
- $var\ cache\_user\_list[N]$ defines an array named $cache\_user\_list$ and the size of it is N.
- $Channel\ c\ 5$ defines a communication channel named c and its capacity is 5.
- $x = x + 1 \longrightarrow Skip$ defines an event that can be attached with an assignment, using which we can update the value of a global variable x.
- $c!a.b \longrightarrow P$ and $c?x.y \longrightarrow P$ refer to sending message $a.b$ and receiving message from channel c respectively.
- $ifa(statement)\{\ P\ \}\ else\{\ Q\ \}$ means if the statement is true, the behavior is like P, otherwise, like Q.
- $a\{\ x = x + 1;\ \} \longrightarrow P$ means updating the global variable x in the block a and then do the process P.

More details about PAT can be found in [17,16,18].

3 Overview of REST on HTTP

REST is a set of principles that define how Web standards, such as HTTP and URIs, are supposed to be used (which often differs quite a bit from what many people actually do) [6]. Here we conclude all REST constriants into several key principles to be easily understood.

(1) Each resource has an id. Resource is a concept newly shows up in [3]. It is an abstract unit of information with an intended meaning [9]. A book is a resource, and at the same time, a collection of books is also a resource. A service, e.g., "current temperature of Shanghai", is a resource too. In a word, everything merits being identifiable is a resource. "Each resource has an id" means we have a unified naming function mapping identifiers to resources. In HTTP_based applications, Uniform Resource Identifiers (URIs) [19] are the resource identifiers.

(2) Resources are linked together. We can always see a confusing concept named "Hyper-media as the engine of application state" in REST related articles. At its core is the concept of hyper-media, or in other words, the idea of links. A resource is never a link, but a link exists between resources. When we request a resource, the server returns what we want, together with some more links that can take us to get more information. It is the link that makes the state of system change [20]. Resources are linked together so that we can get every resource we want just by following links. We give related representations in our model about this constraint.

(3) Using standard methods. With the URIs, the system aims to do something meaningful, so browser should know what to do with the URI. The browser knows it because every resource supports the same interface. The uniform interface between components makes REST distinguished from other network-based architectures. It is the most important feature of REST. The application of the generality to the component interface improves the visibility of the interaction and makes it possible for information to communicate between independent components.

HTTP calls these uniform interfaces (or standard methods) *verbs*, and the set of standard methods includes GET, POST, PUT, DELETE, HEAD and OPTIONS. In this paper, we only take the first four verbs into consideration because only these four methods are related to resources. Almost all applications which claim to be RESTful use HTTP methods to communicate, but many of them misuse these four basic methods. So we introduce the whole REST architecture together with HTTP methods in our following model to approach the practical use of REST, with a hope of reducing the misuse of the great and promising architecture. The meanings of these methods are defined in the HTTP specification [2] and are introduced briefly in the following part, along with some guarantees about their behaviors. A method is called safe if it has no effect on the resource. And if a method is idempotent, it means duplicate actions cause no effect.

- GET: The GET method means to retrieve whatever information is identified by requesting URI. It should be safe and idempotent.
- PUT: The PUT method is used to create (if it does not exist) or modify the requested resource identified by URI. It should be idempotent.
- POST: The POST method is used to create a new resource under the requesting URI and make the new resource as its subordinate. It is neither safe nor idempotent.
- DELETE: The DELETE method is used to delete the requested resource in the server identified by requesting URI. It should be idempotent.

More details about impacts on the entire system and the specifications of methods will be given in the modeling section.

(4) Representations of resources. A resource representation is a description of the state of the resource at a given time [21]. A state may have different multiple representations. For example, a web page can be presented as HTML, or XML, or just an image of its content. The format of representation is negotiated by the client and server and should be understood by the application. The multiple representations of a resource make it available to anyone who knows how to use the web.

(5) Stateless behavior of the communications. The communication between client and server is stateless in REST. Each request contains all the information needed to understand the message. And all the session states are stored in clients. Any context remains in the server cannot be used.

One benefit of statelessness is the scalability of the system. If the server needs to save clients' states, its footprint will be impacted. And stateless behavior brings another advantage. If a server crashes while running, it can be replaced without clients' notice. We give a formal description of this constraint in the verification part.

The formal descriptions of the principles mentioned above can be found in the

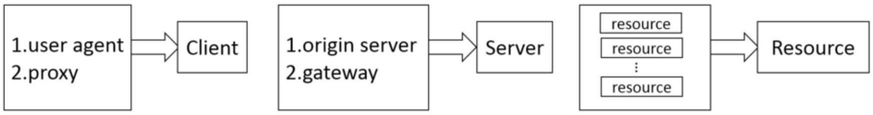

Fig. 1. Combination strategy in our model

next two sections.

4 Modeling

We use CSP to model the REST architecture on HTTP in this section. The REST architecture has the components like user agent, the cache, origin server and intermediary components (proxies and gateways). Since the proxy and gateway are used to forward requests and responses with possible translations, we here combine user agent and proxy into the concept of **Client**, and origin server and gateway into the concept of **Server**. Cache exists in user agents and intermediaries to improve network efficiency. Since our focus has no relationship with cache, we do not take cache into consideration in our model. Resource is the key idea of REST and most communications between clients and server have effects on resources. We also model a process called **Resource**. Server can access to Resource. Client includes a number of clients identified by client id. And Resource includes a number of resources identified by resource id. Figure 1 shows the combination strategy in our model.

First we define the following sets and channels. We use the set **C** to represent client identifiers. The set **Oper** stands for four HTTP methods related to resources. Hence, $Oper = \{get, post, put, delete\}$. **ID** is a set of resource identifiers; **Root** $\subseteq$ **ID** is a finite set of root identifiers; **Data** is a set of data involved in requests and responses and it can be empty (null); **Comm** is a set of communications; and **Repre** is a set of representations of resources, a representation is a specific concept of data, it includes all links and other data in a resource. So $representation \in Repre$, $Repre \subseteq Data$. And $repre(i)$ means the resource i's representation. The set of root resource identifiers, Root, is considered as "common knowledge" for each specific web application, e.g., "www.example.com". So usually $|Root| = 1$, the meaning of $|s|$ is element number of the set s. We define **assoc(c)** as the set of resource identifiers that are 'known' by c, $assoc(c) \subseteq ID, c \in C$. And for each $c \in C$, $assoc(c) = Root$ at the beginning. The set **S(i)** stands for the subordinate resources of the resource i, formally, $S : ID \mapsto 2^{ID}$, mapping each resource identifier to the set of resource identifiers for its subordinate resources. Messages in our model are defined as follows:

$$\begin{aligned}
MSG_{cs} &=_{df} \{\, oper.c.i.d \mid oper \in Oper, c \in C, i \in ID, d \in Data\} \\
MSG_{sc} &=_{df} \{\, d\_1.d\_2 \mid d\_1, d\_2 \in Data\} \\
MSG_{sr} &=_{df} \{\, oper.i.d \mid oper \in Oper, i \in ID, d \in Data\} \\
MSG_{rs} &=_{df} \{\, d\_1.d\_2 \mid d\_1, d\_2 \in Data\} \\
MSG &=_{df} MSG_{cs} \cup MSG_{sc} \cup MSG_{sr} \cup MSG_{rs}
\end{aligned}$$

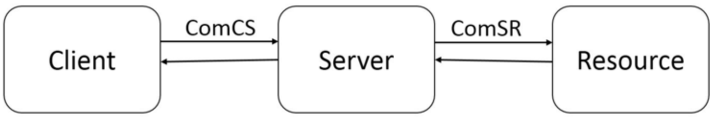

Fig. 2. Model architecture via Channels

MSG_{cs} represents the message sent by Client to Server; MSG_{cs} represents the message sent by Server to Client. And MSG_{sr} and MSG_{rs} stands for the message between Server and Resource. We define a **communication** as "$message\_cs + message\_sr/message\_sc + message\_rs$", that is, the whole process from client sending a request to client receiving a response. Here $message\_cs \in MSG\_cs$, $message\_sr \in MSG\_sr$, $message\_sc \in MSG\_sc$, and $message\_rs \in MSG\_rs$. The set **L(m)** describes resources that are made known to the requesting client, and includes resource identifiers that are returned as results in the communication m, or those are created by m. The set **UL(m)** represents the resource identifiers that are revoked at the client. Here, $m \in Comm$, $L : Comm \mapsto 2^{ID}$, also $UL : Comm \mapsto 2^{ID}$.

We use two channels to model the communications between the three components: ComCS and ComSR.

- **ComCS**: it is used to send and receive messages between Client and Server.
- **ComSR**: it is used to send and receive messages between Server and Resource.

Figure 2 shows the architecture of our model.

4.1 Client

As we mentioned above, the process Client includes the part of user agent and proxy. In the whole communication, Client sends message to the next component of the whole system. HTTP defines 8 methods [2], they are GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE and CONNECT. But only the first four methods have effects on resources. So in our model, we only take these four methods into consideration. The model of Client is given in the following part:

$$\begin{aligned}
Client =_{df} & ComCS!get.c.i.null \\
& \quad \rightarrow ComCS?representation.other\_data \rightarrow Client \\
\Box\ & ComCS!post.c.i.resource\_data \\
& \quad \rightarrow ComCS?new\_id.other\_data \rightarrow add\_assoc(c, new\_id) \rightarrow Client \\
\Box\ & ComCS!put.c.i.resource\_data \\
& \quad \rightarrow ComCS?signal.other\_data \rightarrow add\_assoc(c, i) \rightarrow Client \\
\Box\ & ComCS!delete.c.i.null \\
& \quad \rightarrow ComCS?signal.other\_data \rightarrow unlink\_assoc(c, i) \rightarrow Client
\end{aligned}$$

The client sends message to the server and waits for the response. Here, $get, post, put, delete \in Oper$ are the names of methods; $c \in C$ is the id of the client; $i \in ID$ is the requested resource identifier; $reource\_data \in Data$ includes all links and data of a resource; $null \in Data$ represents empty information; $other\_data \in Data$ are other data included in the returning message; $representation \in Repre$ is the representation of a resource; $add\_assoc(c, i)$ is a function, adding i to the client $c's$ association set, that is, after $add\_assoc(c, i)$, $assoc(c)$ turns to be $assoc(c) \cup \{i\}$; $unlink\_assoc(c, i)$ is a function, unlinking the resource i from the client c, which means after $unlink\_assoc(c, i)$, $assoc(c)$ turns to be $assoc(c) - \{i\}$. From the model we can see the client can do any of the four methods every time:

- get: $get.c.i.null$ means the client c sends a request to get the representation of i. The client then waits for the resource representation of i. Since the communication does not link any new resource id to the client c, the set $assoc(c)$ keeps the same. Both L and UL are empty.
- post: $post.c.i.resource\_data$ means the client c creates a new resource under i. The method uses $resource\_data$ to create a fresh resource identified by new_id and associates it to the client c. So $repre(new\_id) = resource\_data$. After the communication, the client *c's* linking set adds the new identifier new_id. And the new_id becomes a subordinate of i and is added to $S(i)$ since it is created under i. Here $L = \{new\_id\}$ and UL is empty.
- put: $put.c.i.resource\_data$ means the client c requests to update the resource i. This method waits for the signal whether the communication is processed successfully. If i exists, the request will update i's information using $resource\_data$. If not, it will create a resource identified by i using $resource\_data$. After the communication, $repre(i)$ turns to be $resource\_data$. If i is updated, then $i \in assoc(c)$, both L and UL are empty. If i is newly created, then i is added to $assoc(c)$, and $L = \{i\}$ while UL is empty, $S(i) = \{\}$. In either case, $assoc(c)$ becomes $assoc(c) \cup \{i\}$ (if $i \in assoc(c)$, $assoc(c)$ equals $assoc(c) \cup \{i\}$).
- delete: $delete.c.i.null$ means the client c requests to delete the resource i. The response is the signal whether the resource is deleted. The communication disassociates i from client c, so $assoc(c)$ becomes $assoc(c) - \{i\}$. Here L is empty while $UL = \{i\}$.

We should clarify the difference between the *post* method and *put* method when it comes to create a new resource. As we can see above, when the client uses *post* to create a new resource with the parameter i, it creates a new id new_id under i. The server determines what new_id is. For example, we suppose i is "www.example.com/books", then new_id may be "www.example.com/books/1" or "www.example.com/books/27". But the *put* method creates a new resource that is exactly identified as i. Hence, we usually use *post* to create a subordinate of a collection. In our example, that is a book of books.

4.2 Server

In our model, the process of $Server$ stands for the combination of gateway and the origin server. It receives requests from the client and sends responses accordingly. It also accesses to resources. The model of $Server$ is shown below:

```
$Server =_{df} ComCS?method.client\_id.resource\_id.data \rightarrow$
        $if(method == get)$
          $\{ComSR!get.resource\_id.data$
          $\rightarrow ComSR?representation.other\_data$
          $\rightarrow ComCS!representation.other\_data\}$
        $else\ if(method == post)$
          $\{ComSR!post.resource\_id.data$
          $\rightarrow ComSR?new\_id.other\_data \rightarrow ComCS!new\_id.other\_data\}$
        $else\ if(method == put)$
          $\{ComSR!put.resource\_id.data$
          $\rightarrow ComSR?signal.other\_data \rightarrow ComCS!signal.other\_data\}$
        $else\ if(method == delete)$
          $\{ComSR!delete.resource\_id.data$
          $\rightarrow ComSR?signal.other\_data \rightarrow ComCS!signal.other\_data\}$
        $\rightarrow Server$
```

The server checks the type of method. If it requests to *get* the resource identified by *resource_id*, the server sends the request forward to Resource with *data* (when *method* == *get*, *data* is empty since the client sends *null* with the *get* method). After that, the server waits the representation of *resource_id* from Resource and then responses to Client. The server handles the other three cases in a familiar way, that is, sends forward the request to resources and sends back the returning data to clients.

4.3 Resource

The process Resource consists of numbers of resources in a RESTful system. It changes the status of resources and communicates with server. The model of Resource is as follows:

```
$Resource =_{df} ComSR?method.resource\_id.data \rightarrow$
          $if(method == get)$
            $\{representation = get\_repre(resource\_id)$
            $\rightarrow ComSR!representation.other\_data\}$
          $else\ if(method == post)$
            $\{j = create\_resource(resource\_id, data)$
            $\rightarrow ComSR!j.other\_data\}$
          $else\ if(method == put)$
            $\{set\_resource(resource\_id, data)$
            $\rightarrow ComSR!OK.other\_data\}$
          $else\ if(method == delete)$
            $\{set\_resource(resource\_id, null)$
```

$$\rightarrow ComSR!OK.other\_data\}$$
$$\rightarrow Resource$$

Here, $method \in Oper$ is a parameter passed from server, and it can be get or $post$ or put or $delete$; $resource\_id \in ID$ represents the requesting resource identifier; $OK \in Data$ is a signal of whether the method is processed successfully; $get\_repre(resource\_id)$ is a function, returning the representation of $resource\_id$; the function $create\_resource(resource\_id, data)$ creates a new resource under the identifier $resource\_id$, sets the representation of the new resource $data$ and returns the resource identifier newly created; $set\_resource(resource\_id, data)$ changes the representation of the resource $resource\_id$ to $data$. When Resource receives a request of getting a resource, it uses function $get\_repre(resource\_id)$ to get the representation and returns it. A $delete$ method makes $repre(resource\_id)$ undefined, so we use the parameter $null$ in the $set\_resource()$ function when $method == delete$.

4.4 System

The whole application can be modeled as a concurrent composition of Client, Server and Resource.

$$System =_{df} Client[|ComCS|]Server[|ComSR|]Resource$$

5 Verification

In this section, we use first order logic to give formal definitions of failure cases of REST's most important constraints. We also check safe and idempotent properties of HTTP methods in our model. Then the model checker PAT is used to prove the constraints, indicating our achieved model is reasonable. To better describe and understand the constraints, we first give the functions below:

- $set(a)$ represents the set a, e.g., $c \in set(client)$ means c is a client. We define three sets: *communication*, *client*, *method*. A communication is the whole process that a client requests a resource and the server responses it. And the set*communication* represents the set of communications. And $method \in \{get, put, post, delete\}$.
- $resource(comm)$ represents the *resource identifier* requested in the communication *comm*.
- $response(comm)$ represents the *response* of the communication *comm*, that is, the returning message from server to client.
- $method(comm)$ represents the HTTP *method* of the communication *comm*. Here $method(comm) \in set(method)$.
- $link(comm)$ describes resources that are made known to the requesting client, and includes resource identifiers that are returned as results in the communication *comm*, or those are created by *comm*.
- $comm\_sequence(c)$ is a sequence of communication carried out between client c and the server.

- $assoc(s,c)$ defines the set of resource identifiers that are "known" to the client c after the communication sequence s. Here, $c \in set(client)$ and $s \in comm\_sequence(c)$.
- $next(c)$ means the next communication carried out between client c and the server.

5.1 Stateless Behavior

In REST architecture, the communication is stateless. In other word, the client's environment has no influence on the response and each request includes all the message needed. And the server does not store anything related to clients. We here give the failure case of stateless behavior:

Failure Case 1:

$$\begin{aligned}
&\forall c, d \in set(client) \wedge c \neq d, \exists com\_c, com\_d \in set(communication) \bullet \\
&\qquad comm\_sequence(c) = comm\_sequence(d) \\
&\qquad \wedge next(c) = com\_c \wedge next(d) = com\_d \\
&\qquad \wedge resource(com\_c) = resource(com\_d) \\
&\qquad \wedge method(com\_c) = method(com\_d) \\
&\qquad \wedge method(com\_c) = get \\
&\qquad \Longrightarrow response(com\_c) \neq response(com\_d)
\end{aligned}$$

The failure case shows that, if two different clients request for reading the same resource, they get different results, then the system breaks the stateless behavior. This proves server's actions are irrelevant to the clients' environments.

5.2 Hypertext-driven Behavior

Hypertext-driven is the most important principle of REST which makes it distinguished from other architectures. It focuses on the point that all data are linked to each other [20]. And the REST architecture insists on the principle of hypertext being the engine of application state. We give the following failure case of Hypertext_driven behavior:

Failure Case 2:

$$\begin{aligned}
&\forall c \in set(client) \wedge \forall s \in comm\_sequence(c), \exists com \in set(communication) \bullet \\
&\qquad next(c) = com \wedge resource(com) \notin assoc(s,c) \\
&\qquad \Longrightarrow resource(com) \notin link(com)
\end{aligned}$$

We can see from the failure case that a client can only access to the linked resource. In an application, if a client requests for an unlinked resource, and it is not returned or created in the requesting process, the application then breaks hypertext_driven behavior.

5.3 Properties of HTTP Methods

In the constraint of uniform interface [3], REST should have uniform manipulations of resources. HTTP is such a choice. But the misuses of HTTP's four methods make it not as powerful as it supposes to be. So it is of great importance to understand and keep the properties of each method. As the network's instability and insecurity, safety and idempotence are two properties that should be kept.

5.3.1 Safe method.

A safe method have no effect on any resource [2]. The method *get* should be safe. A way to check whether a method is safe or not is as follows:

Constraint 1:

$$\begin{aligned}
&\forall c \in set(client), \exists com\_c\_1, com\_c\_2, com\_c\_3 \in set(communication)\\
&\qquad \bullet method(com\_c\_1) = method(com\_c\_3)\\
&\qquad \wedge method(com\_c\_1) = get\\
&\qquad \wedge method(com\_c\_2) = m\\
&\qquad \wedge resource(com\_c\_1) = resource(com\_c\_2)\\
&\qquad \wedge resource(com\_c\_2) = resource(com\_c\_3)\\
&\qquad \Longrightarrow response(com\_c\_1) = response(com\_c\_3)
\end{aligned}$$

Here, $com\_c\_1, com\_c\_2, com\_c\_3$ are taken sequentially, and m is the method needs to be checked whether it has the safe property. $m \in \{get, delete, put, post\}$. First we let a client c *get* a resource, then c uses method m to request it and in the end we let c get it again. If c gets the same response, then the method m does not modify the resource and it has the safe property. HTTP defines only the method *get* has the property of safety and we will prove it in PAT.

5.3.2 Idempotent method.

According to HTTP RFC [2], a method is called idempotent if duplicate actions cause no effect. In HTTP standards, *get*, *put* and *delete* are defined idempotent. We can check a method *m's* idempotence through following constraint:

Constraint 2:

$$\begin{aligned}
&\forall c \in set(client), \exists com\_c\_1, com\_c\_2, com\_c\_3, com\_c\_4 \in set(communication)\\
&\qquad \bullet method(com\_c\_2) = method(com\_c\_4)\\
&\qquad \wedge method(com\_c\_2) = get\\
&\qquad \wedge method(com\_c\_1) = method(com\_c\_3)\\
&\qquad \wedge method(com\_c\_1) = m\\
&\qquad \wedge resource(com\_c\_1) = resource(com\_c\_2)\\
&\qquad \wedge resource(com\_c\_2) = resource(com\_c\_3)\\
&\qquad \wedge resource(com\_c\_3) = resource(com\_c\_4)\\
&\qquad \Longrightarrow response(com\_c\_2) = response(com\_c\_4)
\end{aligned}$$

Here, $com\_c\_1, com\_c\_2, com\_c\_3, com\_c\_4$ are taken sequentially. A client c uses method m to manipulate a resource and uses *get* to see the effect. Then it uses the same method to manipulate the resource again and uses *get* to check the result. If the two *get* methods have the same results, then method m is idempotent. We will check whether *get*, *put* and *delete* keep the property in our model via PAT.

5.4 Modeling in PAT

We implement our model using PAT in this subsection. There are three processes in our model, *client*(), *server*() and *resource*(). We give the relevant codes about the client part:

```
$client(client\_id, method, resource\_id, data) =$
  $ifa(method == 1)$
      $\{ComCS!method.client\_id.resource\_id.data$
      $\longrightarrow ComCS?representation.other\_data$
      $\longrightarrow client(client\_id, method, resource\_id, data)\}$
  $else\ ifa(method == 2)$
      $\{ComCS!method.client\_id.resource\_id.data$
      $\longrightarrow ComCS?new\_id.other\_data$
      $\longrightarrow client(client\_id, method, resource\_id, data)\}$
  $else\ ifa(method == 3)$
      $\{ComCS!method.client\_id.resource\_id.data$
      $\longrightarrow ComCS?signal.other\_data$
      $\longrightarrow client(client\_id, method, resource\_id, data)\}$
  $else$
      $\{ComCS!method.client\_id.resource\_id.data$
      $\longrightarrow ComCS?signal.other\_data$
      $\longrightarrow client(client\_id, method, resource\_id, data)\};$
```

Here, *client_id* is the requesting client's id; *method* represents the method, 1,2,3,4 for *get*, *post*, *put*, *delete* respectively, *data* can be null or other data of resource. First we use

$$r1() = client(1, 1, 1, 111)||server()||resource();$$

to create a new resource under resource 1. Then we use two processes

$$r2() = client(1, 0, 2, 0)||server()||resource();$$
$$r3() = client(2, 0, 2, 0)||server()||resource();$$

to represent two different clients identified by 1 and 2 to get the same resource 2. Then we observe the traces of these two systems to check the responses of them. The result shows that these two clients get the same response. In this way, we prove the server's responses are irrelevant to the client's context. So the stateless constraint is proved. The complete codes of this proof will be given in the appendix.

We also check the safety and idempotence properties of HTTP methods in our model in similar ways and prove that method *get* keeps safety, and methods *get*, *put* and *delete* keep idempotence in our model. The result lives up to the HTTP standards.

6 Case Study

In order to illustrate the model and constraints presented in this paper, we introduce an example: ordering a meal in a restaurant.

Here we choose a more accessible domain to illustrate our model rather than the one with a wealth of technical and domain-specific details:

Our restaurant, named "RESTaurant", like most other businesses, is primarily interested in maximizing the throughput of orders. As a result, we use asynchronous processing. In other words, we have waiters taking customers' orders and at the same time we have a cook making food in the kitchen. This allows the waiters to keep taking orders even if the cook is not available for the moment.

6.1 Model Mapped to the Whole Process

6.1.1 Customers–Client.

The customers in the whole process are clients in our model. They can place an order using *post* method, update an order using *put* method, ask for the cost of the food using *get* method and take the food away using *delete* method. We show each of them in the following part and the mapping of the model can also be found.

Place an order-"post": A customer named Happy comes into RESTaurant and checks the menu to find all the food supplied. The customer makes the decision and says, "Steak, please.". In this way, the customer sends related request and waits for the order number. The trace of Happy's taking order is listed as follows:

$$\begin{aligned}&< ComCS!post.\text{``}Happy\text{''}.\text{``}RESTaurant.org/order\text{''}.\\&\qquad\qquad \{\text{``}steak\text{''}, \text{``}take\_away\text{''}, \text{``}unpaid\text{''}...\},\\&\quad ComCS?\text{``}RESTaurant.org/order/8\text{''}.data1 >\end{aligned}$$

First, customer Happy asks for creating a new order, with the content of "steak" and other related data. Then he waits for the response. The waiter will tell him his order is under the URI of "RESTaurant.org/order". And the *data*1 is in the form like:
order number ("RESTaurant.org/order/8" in our case) and the process of the whole process such as he needs to pay before taking the food. We can see the order created

$$\begin{aligned}&data1 = \{< next\ rel = \text{``}http: //RESTaurant.org/payment\text{''}\\&\qquad\qquad uri = \text{``}http: //RESTaurant.org/payment/order/8\text{''} / > ...\}\end{aligned}$$

Update the order-"put": The customer may forget to order drinks. Before his payment, he can call a waiter to change (update) his order, with a PUT method. The trace of the customer Happy's updating his order is like:

$$\begin{aligned}&< ComCS!put.\text{“}Happy\text{”}.\text{“}RESTaurant.org/order/8\text{”}.\\ &\qquad\quad \{\text{“}steak + orange\ juice\text{”}, \text{“}take\_away\text{”}, \text{“}unpaid\text{”}...\},\\ &\ ComCS?\text{“}OK\text{”}.other\_data >\end{aligned}$$

Pay the order-“get+put”: Before enjoying the meal, the customer should pay for the bill. First, he should *get* his total cost. The trace of getting the total cost is in the following:

$$\begin{aligned}&< ComCS!get.\text{“}Happy\text{”}.\text{“}RESTaurant.org/payment/order/8\text{”}.null,\\ &\ ComCS?\text{“}20dollars\text{”}.other\_data... >\end{aligned}$$

After getting the total cost, the customer can pay his food using *put* method to change the state of “RESTaurant.org/payment/order/8” from “unpaid” to “paid”. The method is similar to updating the order and we no longer list it here. For successful payment, once the customer has received the signal “OK”, Happy can happily take away his steak and juice.

Clean up the order-“delete”: After taking away the food, the customer Happy has no more requirements and says “Goodbye” to the waiter. This is the signal of deleting Happy’s order. The trace is as follows:

$$\begin{aligned}&< ComCS!delete.\text{“}Happy\text{”}.\text{“}RESTaurant.org/order/8\text{”}.null,\\ &\ ComCS?\text{“}OK\text{”}.other\_data >\end{aligned}$$

Seeing the waiter saying goodbye to him, Happy leaves “RESTaurant” with food in his hand.

6.1.2 Cook–Server

In our case, cook is the server in our model. A cook receives orders and returns what customers need. As the resource part, all meaningful things in a system can be a resource. In our restaurant, food are resources, an order of food is a resource too. The mapping of cook and resource are similar to the mapping from customer to client. We do not list the trace of them any more.

6.2 Properties Shown in Our Case

6.2.1 Hypertext-driven Behavior.

When Happy makes his order, he receives *data*1 together with his order number. We see the form of *data*1 again:

$$\begin{aligned}&data1 = \{< next\ rel = \text{“}http://RESTaurant.org/payment\text{”}\\ &\qquad\qquad uri = \text{“}http://RESTaurant.org/payment/order/8\text{”} / > ...\}\end{aligned}$$

With this information, the customer knows the next step is to pay his food using the URI of “http://RESTaurant.org/payment/order/8”. And this URI is added to the client Happy’s association set. Customers can always follow the links to move forward. This is similar to the constraint of hypertext-driven behaviors.

6.2.2 Statelessness.

Think about a situation, a friend of Happy comes into RESTaurant, and he is very generous to pay the bill for Happy. He can also *get* the total cost of the order by “http://RESTaurant.org/payment/order/8”, and pay the bill. The result has no difference, just 20 dollars for a steak and juice. It has the same principle as statelessness in REST, that is, the context of clients has no impact on the communications.

6.2.3 Properties of HTTP Methods.

We give the descriptions of safe and idempotent property constraints in section 5. Here we take the *get* method as an example of safety and take *put* method as an example of idempotence. The method *get* should be safe and idempotent as required. When the customer Happy wants to check the cost of his food, he uses *get* method and knows the food costs him 20 dollars. After checking, the cost is still 20 dollars and the state of the payment is still unpaid. No change happens to the payment order after the *get* method. This means *get* is safe. And the customer asks to update the order, changing the food “steak” to “steak+orange juice”. Maybe waiter A helps to finish it and then he leaves the counter. Then waiter B takes over A’s job and enters the requirement again. In this way, the *put* method is carried out twice and the result is as same as once, that is, “steak+orange juice” for order 8. So *put* is idempotent.

Using our model, we can understand better of the RESTful system in this case study. And also the REST’s most important constraints can be illustrated clearly. Both demonstrate the feasibility of our model.

7 Conclusion

This paper has discussed the concepts of REST on HTTP. The main principles of REST, stateless property and hypertext driven property, are introduced. We model the main components of REST on HTTP in CSP, including three parts, Client, Server and Resource. Then four constraints are come up to check whether the model fulfills the requirements of stateless and hypertext-driven properties of a RESTful system, and the safe and idempotent properties of standard HTTP methods. We use model checker PAT to check all these constraints. Finally, a case study is shown to illustrate the process of a RESTful system on HTTP. All the efforts we have done are to clarify the concepts of REST on HTTP, with an expectation of reducing the misuse of it.

For the future, we will go on the research of REST in formal ways and analyze more properties about it. Besides, we will apply our model to practical applications. Using constraints proposed in section 5, we will check existing RESTful systems to find out whether they fulfill the basic principles of REST and HTTP.

Acknowledgement

This work was partly supported by the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61061130541) for the Danish-Chinese Center for Cyber Physical Systems. It was also supported by National High Technology Research and Development Program of China (Grant No. 2012AA011205), National Basic Research Program of China (Grant No. 2011CB302904), National Natural Science Foundation of China (Grant Nos. 61021004 and 91118008), Shanghai STCSM Project (Grant No. 12511504205), and Key Lab of Information Network Security, Ministry of Public Security (No. C12604).

References

[1] Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. Internet Computing, IEEE **6**(2) (2002) 86–93

[2] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol–HTTP/1.1 (1999)

[3] Fielding, R.T.: Architectural styles and the design of network-based software architectures. PhD thesis, University of California (2000)

[4] Richardson, L., Ruby, S.: RESTful web services. O'Reilly Media (2008)

[5] Allamaraju, S.: RESTful Web Services Cookbook. Yahoo Press (2010)

[6] Tilkov, S.: A brief introduction to REST. InfoQ, Dec **10** (2007)

[7] Tomayko, R.: How I explained REST to my wife. The various writings and linkings of Ryan Tomayko. Np **12** (2004)

[8] Webber, J., Parastatidis, S., Robinson, I.: How to GET a cup of coffee. InfoQ.[Online]. Retrieved on January **5** (2010)

[9] Klein, U., Namjoshi, K.S.: Formalization and automated verification of RESTful behavior. In: Computer Aided Verification, Springer (2011) 541–556

[10] Zuzak, I., Budiselic, I., Delac, G.: Formal modeling of RESTful systems using finite-state machines. In: Web Engineering. Springer (2011) 346–360

[11] Hernández, A.G., García, M.N.M.: A formal definition of RESTful semantic web services. In: Proceedings of the First International Workshop on RESTful Design, ACM (2010) 39–45

[12] Moller, F., Tofts, C.: A temporal calculus of communicating systems. Springer (1990)

[13] Hoare, C.: Communicating sequential processesprentice-hall international. Englewood Cliffs (1985)

[14] Wu, X., Zhang, Y., Zhu, H., Zhao, Y., Sun, Z., Liu, P.: Formal Modeling and Analysis of the REST Architecture Using CSP. Web Services and Formal Methods (2012) 49

[15] Fielding, R.T.: REST APIs must be hypertext-driven. Untangled musings of Roy T. Fielding (2008)

[16] Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards flexible verification under fairness. In: Computer Aided Verification, Springer (2009) 709–714

[17] Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: Introducing a process analysis toolkit. In: Leveraging Applications of Formal Methods, Verification and Validation. Springer (2009) 307–322

[18] Chen, C., Dong, J.S., Sun, J., Martin, A.: A verification system for interval-based specification languages. ACM Transactions on Software Engineering and Methodology (TOSEM) **19**(4) (2010) 13

[19] Masinter, L., Berners-Lee, T., Fielding, R.T.: Uniform resource identifier (URI): Generic syntax. (2005)

[20] Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(3) (2009) 1–22

[21] Klein, U.: Topics in Formal Synthesis and Modeling. PhD thesis, New York University (2011)

A Relevant Codes of PAT

```
//number of resources
#define N 50;
//define the channels of the system
channel ComCS 1;
channel ComSR 1;

var repre[N + 1];
//to represent the representations of resources
var root[N+1];
// the root of each resource
var j;
var ok;


client(client_id ,method,resource_id ,data)=
ifa(method==0)//get
{
 ComCS!method.client_id.resource_id.data->
 ComCS?representation.other_data->
  client(client_id ,method,resource_id ,data)
}
else ifa(method==1)//post
{
ComCS!method.client_id.resource_id.data->
ComCS?new_id->client(client_id ,method,resource_id ,data)
}
else ifa(method==2)//put
{
ComCS!method.client_id.resource_id.data->
ComCS?signal->client(client_id ,method,resource_id ,data)
}
else//delete
{
ComCS!method.client_id.resource_id.data->
ComCS?signal->client(client_id ,method,resource_id ,data)
};

server()=
ComCS?method.client_id.resource_id.data->
ifa(method==1)
{
ComSR!1.resource_id.data->
ComSR?representation.other_data->
ComCS!representation.other_data->server()
}
else ifa(method==2)
{
ComSR!2.resource_id.data->
ComSR?new_id.other_data->ComCS!new_id.other_data->server()
}
else ifa(method==3)
{
ComSR!3.resource_id.data->
ComSR?signal.other_data->ComCS!signal.other_data->server()
}
else ifa(method==4)
{
```

```
ComSR!4.resource_id.data->
ComSR?signal.other_data->ComCS!signal.other_data->server()
};

resource()=
ComSR?method.resource_id.data->
ifa(method==1)
{
ComSR!repre[resource_id].0->resource()
}
else ifa(method==2)
{
set{ j=resource_id*resource_id*resource_id+1 }->
create_resource{ root[j]=resource_id; repre[j]=data }
->ComSR!j.0->resource()
}
else ifa(method==3)
{
set_resource{ repre[resource_id]=data }
->ComSR!ok.0->resource()
}
else ifa(method==4)
{
set_resource{ repre[resource_id]=0 }
->ComSR!ok.0->resource()
};

r1()=client(1,1,1,111)||server()||resource();
//client 1 to create a new resource under 1

r2()=client(1,0,2,0)||server()||resource();
// client 1 to get the newly created resource

r3()=client(2,0,2,0)||server()||resource();
//client 2 to get the newly created resource
```