Formalization and Automated Verification of RESTful Behavior

Uri Klein[1] and Kedar S. Namjoshi[2]

[1] Courant Institute of Mathematical Sciences, New York University
uriklein@courant.nyu.edu
[2] Bell Labs, Alcatel-Lucent kedar@research.bell-labs.com

Abstract. REST is a software architectural style used for the design of highly scalable web applications. Interest in REST has grown rapidly over the past decade, spurred by the growth of open web APIs. On the other hand, there is also considerable confusion surrounding REST: many examples of supposedly RESTful APIs violate key REST constraints. We show that the constraints of REST and of RESTful HTTP can be precisely formulated within temporal logic. This leads to methods for model checking and run-time verification of RESTful behavior. We formulate several relevant verification questions and analyze their complexity.

1 Introduction

REST – an acronym for Representational State Transfer – is a software architectural style that is used for the creation of highly scalable web applications. It was formulated by Roy Fielding in [8]. The REST style provides a uniform mechanism for access to resources, thereby simplifying the development of web applications. Its structure ensures effective use of the Internet, in particular of intermediaries such as caches and proxies, resulting in fast access to applications. Over the past decade, interest in REST has increased rapidly, and it has become the desired standard for the development of large-scale web applications. The flip side to this is a considerable confusion over the principles of RESTful design, which are often misunderstood and mis-applied. This results in applications that are functionally correct, but which do not achieve the full benefits of flexibility and scalability that are possible with REST. Fielding has criticized the design of several applications which claim to be RESTful, among those are the photo-sharing application Flickr [9] and the social networking API SocialSite [10].

The criticisms show that some of the confusion is between REST and the Hypertext Transfer Protocol (HTTP) [7]. (Aside: Fielding is also a co-author of the HTTP RFC.) While RESTful applications are implemented using HTTP, not every HTTP-based application is RESTful, and not every RESTful application must use HTTP: REST is an architectural style, while HTTP is a networking protocol. Another common mistake is to call a application RESTful if it uses simpler encodings than those in the Remote Procedure Call (RPC) based SOAP/WSDL [26] mechanism. The distinction goes far beyond this superficial difference. These and other, more subtle, confusions motivate our work.

A question which arises naturally is whether it is possible to automatically check an application for conformance to REST. Doing so requires a precise specification of REST. In this paper, we address both questions. A formal characterization of REST has benefit beyond its use in automated analysis. It should also result in clear and effective communication about REST, and can enable deeper analysis of this elegant and effective architectural style.

We begin by formulating RESTful behavior in a general setting. A key contribution is to show that REST can be formalized within temporal logic. Two constraints define RESTful behavior. One, statelessness, is a branching-time property. The other, hypertext-driven behavior, is expressible in linear temporal logic. Both are safety properties. We then consider the common case of RESTful HTTP, and discuss how HTTP induces variants of the temporal properties.

The temporal specifications may be applied in several ways for verifying that a client-server application is RESTful. One is to model-check a fixed instance of the application [4, 23]. The parameterized model checking question is also of much interest, as web applications typically handle a large number of clients. These questions presume a 'white-box' situation, where implementation code is available for analysis. A second group of questions concern run-time checking of RESTful behavior, a 'black-box' approach, where the only observable is the client-server communication. A third group of questions concern the synthesis of servers which meet a specification under RESTful constraints.

We show that, for a fixed instance, model-checking statelessness can be done in time that is linear in the size of the state-space of the instance and polynomial in the number of resources. On the other hand, checking that an instance satisfies a specification assuming hypertext-driven client behavior is PSPACE-complete in the number of resources. This property can be checked at run-time, however, in time that is polynomial in the number of clients and resources. We show decidability for parameterized model-checking under certain assumptions; the general case remains open. The full version of this paper [15] has complete proofs of theorems and further detail on RESTful HTTP.

2 REST and its Formalization

Our goal in the formalization is to stay as close as is possible to its description by Fielding in [8], which should be consulted for the rationale behind REST.

2.1 Building Blocks for REST

REST is built around a client-server model which includes intermediate components, such as proxies and caches. An application is structured as a (conceptually) single *server component* (server, for short) and a number of *client components* (clients). All relevant communication is between a client and the server. Each *request* for a service is sent by a client to the server, which may either reject the request or perform it, returning a *response* in either case to the client. A server manages access to *resources*. A resource is an abstract unit of information with an intended meaning. Examples are a data file, a temporal

service (e.g., 'current time in France'), or a collection of other resources (e.g., 'all files in a directory').

An *entity* describes the value of a resource at a given time; it can be viewed as the *state* of a resource. A resource state may be constant (e.g., 'Uri's birth date') or changing (e.g., 'current time in France'), but it must take on values which correspond to the intended meaning of the resource. A state may contain both uninterpreted data and links to other resources. This creates a 'Linked Data' view [3] of all the information under the control of an application. A *resource identifier* (resource id, for short) is a name by which a resource is identified. The mapping of names to resources is fixed and unique. In HTTP-based applications, Uniform Resource Identifiers (URIs) [27] are the resource identifiers. A *resource representation* is a description of the state of the resource at a given time. A state may have multiple representations (e.g., a web page may be represented as HTML, or by an image of its content).

A RESTful architecture has a fixed set of uniform *methods*. Hence, every application following that architecture must be based on these methods, which effectively decouples interface from implementation. In contrast, for an abstract data type or RPC model, the method set is unconstrained. Properties of a method, such as *safety* (no invocation changes server state) and *idempotence* (repeated invocation does not change server state) are required to hold uniformly, i.e., for all instantiations of the method.

2.2 Formalizing Resource-Based Applications

A resource-based application is one that is organized in terms of the previously described building blocks, which are formally defined by a *resource structure*: a tuple $RS = \langle R, I, B, \eta, C, D, \sim, OPS, RETS \rangle$, where R is a set of resources; I is a set of resource identifiers; $B \subseteq I$, is a **finite** set of *root identifiers*; $\eta : I \mapsto R$ is a *naming function*, mapping identifiers to resources, a partial function that is injective on its domain; C is a set of *client identifiers*; D is a set of *data values*, with an *equivalence* relation $\sim \subseteq (D \times D)$; OPS is a finite set of methods; and $RETS$ is a finite set of *return codes*.

For simplicity, we use a specific form of resource representation, a pair $\langle ids; d \rangle$ in $2^I \times D$. Here, ids is a set of resource identifiers, and d a piece of data. This abstracts from HTML or XML syntax and formatting, and clearly separates resource identifiers from data values. The relation '$\sim$' may be used to ignore irrelevant portions of data, such as counters or timestamps. We extend it to resource representations as $\langle ids_1; d_1 \rangle \sim \langle ids_2; d_2 \rangle$ **iff** $ids_1 = ids_2$ and $d_1 \sim d_2$.

A *client-server communication* (a communication, for short) is represented by a 'request/response' pair, with the syntax $c :: op(i, args)/rc(rvals)$, where: $c \in C$ is a client identifier; $op \in OPS$ is a method; $i \in I$, is a *target resource identifier*; $args$ is a finite list of *arguments*; $rc \in RETS$ is a return code; and $rvals$ is a finite list of *return values*. The arguments and return values are specific to the method. Both may include resource identifiers, data values, and resource representations. (We omit more complex data types for simplicity.)

With each communication m are associated two *disjoint* sets of resource identifiers, denoted $L(m)$ (linked) and $UL(m)$ (unlinked). The set $L(m)$ describes resources that are made known to the requesting client, and includes resource identifiers which are returned as results in the communication, or that are created by it. The set $UL(m)$ are identifiers which are revoked at the client.

Given a resource structure RS, a *RS-family* is a collection of client and server processes, defined over elements of RS. A *RS-instance* is a specific choice of clients and a single server from a RS-family, with the processes interacting using CCS-style synchronization [17] on communications. A *global state* of a RS-instance is given by a tuple with a local state for the server process and a local state for each client process. A *computation* is an alternating sequence of global states and *actions*, where an action is either a (synchronized) communication between a client and the server, or an internal process transition.

Caveats: In reality, requests and responses are independent events, which allows the processing of concurrent requests to overlap in time. The issue is discussed further in Section 4, as treating it directly considerably complicates the model. There is also an implicit assumption that methods have immediate effects. In practice, (e.g., HTTP DELETE) a server may return a response but postpone the effect of a request. This issue is discussed in Subsection 3.2.

A *communication sequence* σ is a (possibly infinite) sequence of communications carried out between a set of clients and the server. The *projection* of a communication sequence σ on a client c, written $\sigma|_c$, is the sub-sequence of σ which contains only those communications initiated by client c. A computation of a RS-instance *induces* a communication sequence given by the sequence of actions along that computation.

It is important to distinguish between the case where a method is successfully processed by the server, and where it is rejected without any server state change. This is done by mapping return codes to the abstract values $\{\mathsf{OK}, \mathsf{ERROR}\}$, where OK represents the first case and ERROR the second.

For a finite communication sequence σ, the set $assoc(\sigma)$ of resource identifiers defines those resources 'known' at the end of σ. For the empty communication sequence, $assoc(\lambda) = B$. Inductively, $assoc(\sigma; m)$ is $(assoc(\sigma) \cup L(m)) \setminus UL(m)$, if m has return code OK, and it is $assoc(\sigma)$, if the return code is ERROR.

For a finite computation with induced communication sequence σ, $assoc(\sigma)$ and $I \setminus assoc(\sigma)$ define the *associated* and *dissociated* resource identifiers, respectively. We associate a partial function $deref : I \mapsto 2^I \times D$ with the state of the server; $deref(i)$, if defined, is the current representation of the resource $\eta(i)$ (which must be defined if $deref(i)$ is defined).

2.3 Formalization of RESTful Behavior

For this section, fix a structure $RS = \langle R, I, B, \eta, C, D, \sim, OPS, RETS \rangle$, and consider RS-instances. The two temporal properties discussed below define whether the behavior of an RS-instance is RESTful. It is usually more convenient to describe the failure cases, and also more helpful for the purpose of automatic verification. In the temporal formulas, we use a modified next-time operator,

$\mathsf{X}_{\langle a \rangle}$, where a is an action. Its semantics is defined on a sequence with atomic propositions on each state and an action label on each transition. For a sequence σ and position i, define $\sigma, i \models \mathsf{X}_a(\phi)$ to hold if $\sigma, i+1 \models \phi$ and the transition from step i to step $i+1$ is labeled with a.

Before diving into the specifics, it is worthwhile to point out a couple of important considerations. First, as in any formalization of a hitherto informal concept, there may be subtle differences between an informal idea and its formalization; we point out those that we are aware of. Second, a large part of the usefulness of a formalization lies in the testability of these properties. It is helpful to make a distinction between formal properties which can be tested given complete information of the implementation of clients and the server (a 'white-box' view), and those which can be tested only on the observable sequences of interaction between clients and the server (a 'black-box' view). The first viewpoint is interesting for model-checking; the second for run-time verification. Since we are targeting both approaches, we present the properties from both points of view, making it clear if one leads to a weaker test than the other. This distinction is important only for the safety and idempotence properties.

1. Stateless Behavior. In ([8], Chapter 5), this property is described as follows: *"... each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server."* We formalize it by requiring that the server response to a request be functional; i.e., independent of client history or identity. (A 'client' should be understood to be a machine, rather than a user.) Failure of statelessness is shown by a finite computation followed by a two-way fork, where for some **distinct** client identifiers c, d, one branch of the fork contains the communication $c{::}op(i,a)/r_1(v_1)$, and the other branch contains the communication $d{::}op(i,a)/r_2(v_2)$, and either $r_1 \neq r_2$, or $v_1 \neq v_2$. This failure specification captures the situation where, given an identical history, the same method carried out by different clients has distinct results.

This is a branching-time property. The failure case is expressed as follows in a slight modification of Computation Tree Logic (CTL) [5], which allows the operator $\mathsf{EX}_{\langle a \rangle}$, for an action a.

$$(\exists c, d\ \exists i, op, a, r_1, v_1, r_2, v_2 : c \neq d\ \wedge\ (r_1 \neq r_2\ \vee\ v_1 \neq v_2) \wedge \\ \mathsf{EF}(\mathsf{EX}_{\langle c::op(i,a)/r_1(v_1)\rangle}(\mathit{true})\ \wedge\ \mathsf{EX}_{\langle d::op(i,a)/r_2(v_2)\rangle}(\mathit{true})))$$

The property suffices to detect the common cases of hidden per-client state. One subtlety is that the the property is based on *observable*, semantic effects of a hidden state, not its syntactic presence. Hence, it holds of a server which retains auxiliary per-client information – such as a request counter – but does not use that information to influence the response to a request.

The formalization is also slightly stronger than the intended informal notion of statelessness, in the following sense. Consider a server which implements a method as "`if (client=c) then return 3 else return 4`". This has no

hidden state, yet the method has different results for distinct clients c and d, and fails the property.

2. Hypertext-driven behavior. Informally, this property requires a client to access a resource only by 'navigating' to it from a root identifier. It is also referred to by the acronym HATEOAS, which stands for "Hypertext/Hypermedia As The Engine Of Application State". The failure specification is a finite computation with induced communication sequence of the form $\sigma; c{:}{:}op(\ldots, i, \ldots)/rc(\ldots)$, for some σ, return code rc, method op, and resource identifier i among the arguments of op, such that all of the following hold: $i \notin assoc(\sigma|_c)$, and if L is the linked set of the last communication, then $i \notin L$. The return code and values are not important. It suffices that the identifier i is currently not associated from the perspective of the client c.

This condition can be expressed in Linear Temporal Logic (LTL) [20], most conveniently by using past temporal operators [16] to express the condition $i \notin assoc(\sigma|_c)$. The past-LTL formula for failure, denoted φ_{HT}, can be built up as shown below.

In the following, the predicate $by(m, c)$ is true if communication m is by client c; $\mathsf{OK}(m)$ is true if m has return code OK; $arg(m, i)$ is true if resource id i is an argument to the request in m; Y is the 1-step predecessor operator with variant $\mathsf{Y}_{\langle a \rangle}$ (formally, $\sigma, i \models \mathsf{Y}_{\langle a \rangle}(\phi)$ if $(i \geq 1)$ and $\sigma, (i-1) \models \phi$, and the transition from step i to step $i+1$ is labeled by a); and $p\ \mathsf{S}\ q$ is the 'since' operator which holds if q holds in the past, and p holds since then. Precisely, $\sigma, i \models p\ \mathsf{S}\ q$ **iff** $(\exists k : 0 \leq k \leq i : \sigma, k \models q\ \wedge\ (\forall j : k < j\ \wedge\ j \leq i : \sigma, j \models p))$. Note that $\neg Y(true)$ is true only at the initial state of a sequence.

$$
\begin{aligned}
\varphi_{HT} =& (\exists c, i : \mathsf{F}(access(c, i)\ \wedge\ \neg\ inassoc(c, i))), \text{ where} \\
access(c, i) =& (\exists m : \mathsf{X}_{\langle m \rangle}(true)\ \wedge\ by(m, c)\ \wedge\ arg(m, i)\ \wedge\ i \notin L(m)), \text{ and} \\
inassoc(c, i) =& (\neg\ revoked(c, i))\ \mathsf{S}\ granted(c, i), \text{ where} \\
revoked(c, i) =& (\exists m : \mathsf{Y}_{\langle m \rangle}(true)\ \wedge\ \mathsf{OK}(m)\ \wedge\ by(m, c)\ \wedge\ i \in UL(m)), \text{ and} \\
granted(c, i) =& (\exists m : \mathsf{Y}_{\langle m \rangle}(true)\ \wedge\ \mathsf{OK}(m)\ \wedge\ by(m, c)\ \wedge\ i \in L(m)) \vee \\
& (\neg \mathsf{Y}(true)\ \wedge\ i \in B)
\end{aligned}
$$

3. Safety and idempotence. REST explicitly includes intermediaries in the model, such as caches and proxies. It is encouraged to have methods which are uniformly idempotent or safe, as intermediaries can more effectively use these methods to reduce latency or mask temporary server failures. While these properties are not required of REST methods, they can be formalized in LTL and model-checked. Unlike the two main properties, the formalization of safety and idempotence is different in the white-box and black-box views.

For the black-box setting, we require the following additional constructs. We suppose that there is a distinguished method, $\mathsf{READ}(i)$, where i is the target resource identifier. It returns either ERROR or $\mathsf{OK}(deref(i))$, the representation

of the resource identified by i. The linked and unlinked sets are empty. We extend the equivalence relation '$\sim$' is to a list of return values: for lists a and b, $a \sim b$ holds if the lists have the same length and corresponding elements have the same types and are related by '$\sim$'. In the following, we also assume that one can identify whether a communication affects a resource; this information is typically available for specific instances of REST, such as RESTful HTTP.

- **Safety of a method.** A method is considered safe if it does not modify resources. In the black-box view, changes to resources can be detected by means of READ methods. A failure for the safety of method *op* is a finite computation with communications c_1::READ(i)/OK(r_1) and c_2::READ(i)/OK(r_2) occurring in that order, with $r_1 \not\sim r_2$, where no intervening communication modifies or dissociates the resource $\eta(i)$ but includes at least one communication using *op*. Informally, failure of safety is signaled by a difference in the representation of the resource identified by i before and after method *op*.
- **Idempotence of a method.** For a method to be idempotent, repeated invocation should have no additional effect on resources. In the black-box view, such changes can be detected by means of READ methods. A failure for the idempotence of method *op* is a finite computation where the communications c_1::*op*/*rc*(rv_1), c_2::READ(i)/OK(r_1), c_3::*op*/*rc*(rv_2), and c_4::READ(i)/OK(r_2) occur in that order, $r_1 \not\sim r_2$ and the communications occurring between these distinguished ones do not dissociate i or modify the resource $\eta(i)$. Informally, the property detects failure by detecting a difference in the representation of a resource identified by i before and after the second instance of a communication with method *op*.

Both black-box properties are weaker than their white-box counterparts. For instance, it is possible that method *op* changes the server state of a resource – perhaps by incrementing an auxiliary counter – but this change is not propagated to the representation, and is hence unobservable by a READ. This violates safety in the white box view, but not in the black-box view.

Naming Independence We present an interesting consequence of the RESTful properties, which shows that the specific choice of naming function does not matter, if client-server behaviors are hypertext-driven. To make this precise, consider structures RS and RS' which are identical except for the naming functions. The naming functions, η and η', are required to map each base name to the same resource. The functions induce a name correspondence: a name i in an RS-instance corresponds to a name j in an RS'-instance if both map to the same resource, i.e., if $\eta(i) = \eta'(j)$.

If clients C_i and C_i' in the hypothesis of the theorem are based on the same program text, a sufficient condition for bisimularity up to naming is that names are used *opaquely*: i.e., no constant names are present, names can only be stored to and copied from variables, and the only relational test allowed for names is equality of name variables. The proof of the theorem is given in the full version.

Theorem 1. *Consider an RS-instance M with clients $C_1, \ldots, C_k$ and server S, and an RS′-instance M' with clients $C'_1, \ldots, C'_k$ and server S'. Suppose that, for each i, clients C_i and C'_i are bisimular up to the naming correspondence, as are S and S'. Then, for each hypertext-driven computation σ of M, there is a hypertext-driven computation σ' of M' such that global states $\sigma(i)$ and $\sigma'(i)$ are bisimular, for each i, and the induced communication sequences match up to the naming correspondence.*

3 REST on HTTP, and Variations

In this section, we show how the property templates from Section 2 can be instantiated for a concrete protocol, HTTP, which is the primary protocol used for constructing RESTful applications. The result is a formal definition of RESTful HTTP behavior.

3.1 Formal RESTful HTTP

HTTP is a networking protocol for distributed, collaborative, hypermedia information systems [7]. The bulk of the interest in REST among developers is in the context of HTTP-based applications. We start by demonstrating how HTTP satisfies the framework requirements described in Subsection 2.1.

HTTP is typically used in a client-server model. HTTP resources are uniquely identified using their Uniform Resource Identifiers (URIs) [27]. For HTTP applications, the fields of a resource representation $\langle uris \in 2^I; d \in D \rangle$ are used as follows: *uris* is a set of URIs, *links* that exist in the resource, and *data* is any data, of any type, that is contained in a resource. It may include auxiliary data, such as counters, which is relevant to server-internal processes, but has no relevance to client behavior. Such data can be elided through an appropriate definition of '$\sim$'. The HTTP RFC [7] defines nine methods. We present here the four main methods, the remaining five have no impact on resources. To represent the HTTP concept of *subordinate* resources, we use a partial mapping, $S : I \mapsto 2^I$, which maps each resource identifier to the set of resource identifiers for its subordinate resources, if any. We only describe successfully processed communications, which return the abstract return code OK, all other codes map to ERROR. The main HTTP methods, with their linked and unlinked sets, are as follows.

- GET(i)/OK($deref(i)$): The method returns the current entity (resource representation) of the resource identified by i from the server. Both L and UL are empty.
- DELETE(i)/OK: The method dissociates the resource identifier i on the server, resulting in $deref(i)$ bring undefined. Here, L is empty, and $UL = \{i\}$. The HTTP RFC actually only requires that the server 'intends' to dissociate it [7]. We discuss this more complex scenario in Subsection 3.2.
- PUT($i, \langle uris; d \rangle$)/OK: The method associates a resource identified by i, if it is not already associated, and assigns a value to its corresponding entity so

that $deref(i) = \langle uris; d \rangle$. If this is a new association, then $S(i) = \{\}$. Here, UL is empty, while $L = \{i\}$.
- $\mathsf{POST}(i, \langle uris; d \rangle)/\mathsf{OK}(j)$: The method associates a fresh resource, which is identified by j, and sets $S(j) = \{\}$ and $deref(j) = \langle uris; d \rangle$. The resource identified by j becomes a subordinate of the resource identified by i, and j is added to $S(i)$. Here, UL is empty, while $L = \{j\}$.

Instantiating the REST property templates from Subsection 2.3 with the HTTP methods results in a formal definition of RESTful HTTP. This is a rather technical, mostly straightforward translation, and is given in the full paper.

3.2 Variations on RESTful HTTP Properties

In this section we mention several common or reasonable modifications of the HTTP model from Subsection 3.1, and discuss how they impact the RESTful HTTP properties. Complete descriptions of these modifications and the corresponding changes to the properties are in the full paper.

Cascade of DELETE Methods by Subordination. One side-effect of the POST method is the creation of a subordination relation from the target resource identifier to the newly associated one. A common feature in many HTTP applications is the requirement that when a resource identifier is dissociated through a DELETE call, its subordinates are deleted as well (which, in turn, may trigger more dissociations of resource identifiers with higher degrees of subordination to the originally deleted one). In our model, this would translate into a (recursive) modification to the linked set of DELETE communications. In RESTful HTTP, this change would require modifying all properties whose definition relies on the unlinked sets of communications (the hypertext-driven sequences property and any idempotence properties) to use more complex, yet easily computed, definitions of the unlinked sets.

Subordination Expressed as a Link. Here, subordination is expressed as a link, i.e., for every $i \in I$ such that $deref(i) = \langle uris; d \rangle$, if $S(i)$ is defined, then $S(i) \subseteq uris$. In this case, a side effect of the communication $c{:}\mathsf{POST}(i, r)/\mathsf{OK}(j)$ would be the modification of the resource identified by i to include j in $uris$. The idempotence property should account for this case by considering that successful POST methods modify existing resources.

Background Data Modifications by the Server. In some cases, where the semantics of the domain D are such that it is is (partially or fully) dynamic by nature, HTTP allows the server to modify the data field of resource representations arbitrarily, in accordance with their semantics. An example is a 'current time' resource, whose value is updated by the server. Successive GET's on this resource would result in different values for the time, potentially violating the safety property of GET. This case can be handled by a proper definition of the data equivalence relation to ignore such changes.

Delayed Executions of Completed DELETE Communications. In the HTTP RFC ([7]) it is said that when the server successfully processes a DELETE request it merely means that "at the time the response is given, it intends to delete the resource or move it to an inaccessible location." Our interpretation of this quote is that the single resource identifier that is in the unlinked set of successfully processed DELETE communications is dissociated only after some arbitrary, **finite**, delay, unless it is associated in the meantime by another communication. Although this behavior makes our definition of $assoc(s)$ irrelevant, it does not complicate the HTTP application of the hypertext-driven sequences property, due to the fact that HTTP clients must 'assume' anyway that resources on which they performed successful DELETE methods are no longer accessible for them. All other properties, however, need to be modified to account for the fact that DELETE methods are not as well-behaved as assumed earlier. The HTTP statelessness property, for instance, would have to disallow 'temporal forks' that include DELETE methods performed by different clients which take effect at different times.

3.3 Distinguishing REST from HTTP

Following are some interesting hypothetical applications which clarify the differences between HTTP and REST, and which address some common misunderstandings regarding RESTful HTTP.

Consider an application which uses only two HTTP methods: PUT and GET. A client encodes methods in the *uri* argument of $\mathsf{PUT}(uri, junk)$ requests, where *junk* - a resource representation - is a meaningless constant. A GET communication is used by a client to examine the state of the server. This application is compliant with the HTTP RFC, as there is no restriction on the PUT communications' return values. However, it is non-RESTful, since it would either have to include an infinite set of root identifiers (each *uri* argument being one), or it would violate the hypertext-driven behavior property. The Flickr API is non-RESTful for a similar reason.

Consider an application which relies entirely on POST communications, and uses a single root identifier, *base*, for all such communications (one may consider $B = \{base\}$). In any $\mathsf{POST}(base, \langle base; data\rangle)/\mathsf{OK}(uri)$ communication, clients encode methods in the *data* field. We consider two variants:

1. The return value of an method is encoded in the newly associated URI *uri*, returned as a result of POST. This is compliant with the HTTP RFC, but it goes against the notion of dividing information into distinct resources, as the base URI must be treated as a single resource. As there is no division into resources (which would be created by – and used to identify – different clients), this application is likely to violate the HTTP statelessness property. Moreover, it is also likely to violate the resource identifier opaqueness assumption from Subsection 2.3, as a program must interpret the URI strings returned by POST. While the opaqueness assumption is not an essential part of REST, it is important to simplify program development and maintenance.

2. The newly associated URI *uri* is used to point to a resource whose representation is the result of the method, and which is later retrieved by a GET on the *uri*. This violates the HTTP RFC, which requires that the result of POST identifies a resource with the supplied data as its representation. As in the previous case, this application is also likely to violate the HTTP statelessness property.

4 Automated Verification of RESTful Behavior

In this section, we formulate and discuss questions relevant to the automated verification of RESTful behavior. We give preliminary results and point to questions that are still open.

4.1 Computation Model

The somewhat informal model used previously can be made precise as follows. Client and server processes are modeled as labeled transition systems. A communication is modeled as a CCS synchronization [17]. Hence, in a communication of the form 'request/response', a client offers this communication at its state, the server offers to accept it, and the two are synchronized to effect the communication. Processes may have internal actions, including internal non-determinism. The CCS model is appealing for its simplicity but assumes atomic communication. We formulate problems and solutions in this model. Subsequently, we discuss how the atomicity requirement may be relaxed, which brings the analysis closer to real implementation practice.

4.2 Fundamental Questions

The two properties of REST, statelessness and hypertext-driven behavior, lead to the following key verification questions.

ST Does a client-server application M satisfy the statelessness property?
HT1 For a client-server application M, does its specification, φ, hold for all computations where client behavior is hypertext-driven?
HT2 For a client-server application M, do all computations which are not hypertext-driven satisfy a 'safe-behavior' property ξ?

These fundamental questions may be asked for a program with a fixed set of clients and resources, or in the parameterized sense. One may also ask if violations of these properties can be detected using run-time monitors. Another interesting question is whether, given an application specification, one can synthesize a server which satisfies it (again, fixed or parameterized).

4.3 Automata Constructions

A nondeterministic automaton which detects a *failure* of the hypertext-driven behavior property works as follows. For a given input word, the automaton guesses the client and resource identifier with which to instantiate the failure

specification, then keeps track of whether the resource id belongs to the current *assoc* for that client. It accepts if, at some point, there is a request by the client using the resource id, but the id is not part of the current *assoc* set. Keeping track of whether a resource id belongs to the *assoc* set for a client does not require computing the *assoc* set. A simple two-state machine suffices, with states $In(c, i)$ and $Out(c, i)$. If the current communication m is by client c and is successful, a transition is made from $In(c, i)$ to $Out(c, i)$ if $i \in UL(m)$, and from $Out(c, i)$ to $In(c, i)$ if $i \in L(m)$. Otherwise, the state is unchanged. The number of automaton states, therefore, is polynomial in $|I|$ and $|C|$.

The deterministic form of this automaton must track all clients and resource ids simultaneously. Thus, the size of a state of the deterministic automaton is $O(|I| \cdot |C|)$, and its state space is exponential: $O(2^{|I| \cdot |C|})$. Non-deterministic failure automata for safety and idempotence can be derived similarly from their failure specifications; these are described in the full paper.

4.4 Model-Checking for Fixed Instances

A fixed instance has a fixed set of resources and clients. The parameters of interest are the sets in the underlying resource structure: the clients, C, the resource identifiers, I, and the data domain, D.

Statelessness is expressed in a slight variant of CTL, as described previously. (The extension does not affect model-checking complexity.) The indexed property expands out to a propositional formula which is polynomial in the sizes of I and D. Hence, using standard CTL model-checking algorithms [4, 23], the ST property can be verified in time linear in the overall application state space and polynomial in the resource structure parameters.

Property HT1 can be verified as follows. A violation of HT1 is witnessed by a computation where all clients are hypertext-driven but φ is false. This can be checked using automata-theoretic model checking [24] by forming the product of the application process with (1) a Büchi automaton for the negation of φ, and (2) an automaton which checks that all clients follow hypertext-driven behavior. The property is verified **iff** the product has an empty language. The second automaton is the deterministic automaton from Section 4.3, with negated acceptance condition.

Property HT2 can be verified by forming the product of the application process with (1) a Büchi automaton for the negation of ξ, and (2) an automaton which checks for failure of hypertext-driven behavior by some client. The property is verified **iff** the product has an empty language. The second automaton is the non-deterministic failure automaton from Section 4.3. The verification takes polynomial time if the size of the application state space is polynomial in the parameter sizes. The verification of HT1 is significantly more difficult.

Theorem 2. *Verification of HT1 for a fixed instance is* PSPACE*-hard in the number of resources. It is in* PSPACE *if a state of the application and of the negated specification automaton can be described in space polynomial in the parameter sizes.*

Proof Sketch. Membership in PSPACE is straightforward, by observing that the automaton used to describe the hypertext-driven property for HT1 has a state size which is polynomial in the the parameter sizes.

PSPACE-hardness for HT1 holds under severe restrictions: a single client, where client, server, and negated specification automaton have a state-space with size polynomial in the parameters' sizes. The reduction is from the question of deciding, given a Turing Machine (TM) M and input x, whether M accepts x within the first $|x| + 1$ tape cells, which is a PSPACE-complete problem (IN-PLACE ACCEPTANCE in [19]). The reduction uses the server state to store the TM head position, while a TM configuration is encoded in the implicitly defined *assoc* set for the client, using resources to represent tape cell contents. □

4.5 Parameterized verification

The parameterized verification question has particular importance, as web applications usually handle a large number of clients and resources. Since statelessness is not a given, it is necessary to assume a server which stores information about each client, which implies that the state space of the server is also unbounded. Nonetheless, the problem can be solved under certain assumptions.

Suppose that clients have a finite state space, X, and that the state space of the server can be written as $Y \times [C \to Z]$, where Y and Z are finite sets. Thus, a global state of an instance with N clients is a triplet (c, a, b), where c is an array of client states, of size N, a is the finite part of the server state, and b is an array of N server-side entries. Assume further that on receiving a request from client i, the server update depends only on, and may only modify, the components a and $b(i)$; i.e., the new entry for client i does not depend on the entries of the other clients. Then, by a change of viewpoint, one may combine the entry $b(i)$ on the server with the state $c(i)$ of client i, obtaining an equivalent application where the new client space is $X \times Z$, and the server space is Y. Both spaces are now finite, although there is still an unbounded number of clients. This situation fits the model in [11], where an algorithm is given for checking linear-temporal properties. The algorithm has very high worst-case complexity, however, so it may be more fruitful to try alternative methods, such as the method of invisible invariants [21, 18], or methods based on upward-closed sets [1].

Several questions remain open. The modeling above implicitly assumes a bounded set of resources and data values. Moreover, the suggested algorithm applies only to linear-time properties and cannot, therefore, be used to check statelessness.

4.6 Run-Time Monitoring

Perhaps the most promising immediate application of the formalization is run-time monitoring. In this setting, the client-server communications are captured by an intermediate proxy, which passes them through analysis automata. This method can be applied to the properties HT1 and HT2; statelessness, being a branching-time property, cannot be checked at run-time, unless some form of

backtracking is implemented. The automata described in Section 4.4 for model-checking HT1 and HT2 can be used for run-time verification of safety specifications. The non-deterministic automata used for checking hypertext-driven behavior must be determinized for run-time analysis. This can be done on the fly, as is the case for implementations of the Unix `grep` command (cf. [2]). The size of the deterministic automaton state is $O(|I| \cdot |C|)$, so the required storage is $O(|I| \cdot |C| \cdot K)$, where K is the state-size of the negated specification automaton. For each communication, the update of the automaton state requires time proportional to the size of the state, and is hence polynomial in the resource parameters. An alternative to run-time verification is off-line testing of a logged communication sequence.

4.7 Synthesizing Servers

A particularly intriguing question is the possibility of synthesizing RESTful servers. A specific question is the following: given a resource structure and a specification φ, synthesize a **stateless** server which satisfies φ. We show below that, under certain assumptions, the statelessness constraint can be dropped. We define a server specification φ to be universally synthesizable if there exists a server implementation which satisfies φ given any set of clients. A sufficient condition for φ to be universally synthesizable is if it is insensitive to client ids and is synthesizable for a single, arbitrary client. Insensitivity means that for sequences σ, δ which agree up to client ids in communications, $\sigma \models \varphi$ **iff** $\delta \models \varphi$.

Theorem 3. *Consider a server temporal logic specification φ. The specification φ is deterministically and universally synthesizable* **iff** *$ST \wedge \varphi$ is deterministically and universally synthesizable.*

Proof Sketch. For the left-to-right direction, given a deterministic server M implementing φ, one can direct all communications to it through an intermediary which replaces all client ids with a single, dummy, client id. By the universality of M, this combination satisfies φ; as M 'sees' only a single client id and is deterministic, the combination is stateless. □

The synthesis problem for LTL specifications, assuming a bounded state-space, was solved in [22]. Implementing the intermediary adds constant complexity.

Adding the assumption that client interactions are hypertext-driven may make an otherwise-unsynthesizable specification synthesizable, but it may also add significantly to the specification complexity.

4.8 Relaxing The Atomicity of Communications

So far, we have assumed that communications are atomic. In real implementations, however, a request and its response are distinct actions. This allows requests from different clients to overlap in time. To handle this concurrency, we assume that the server is linearizable [12]. Every computation produces results which are equivalent to one where each method takes effect atomically.

Hypertext-driven behavior is formulated entirely in terms of the request and response parameters. If clients are not allowed to issue concurrent requests, hypertext-driven behavior holds of a computation **iff** it holds of its linearization. Assume that the service specification is also defined on communication sequences, and has the same property. Then, it suffices to check properties over the linearized subset of computations, which corresponds to the atomic communication model. This reasoning does not apply to statelessness, which is a branching property, and thus outside the scope of linearizability. Further work is necessary to formulate a notion like linearizability for branching-time properties.

5 Related Work and Conclusions

There is surprisingly little in the literature on formal definitions and analysis of REST. In [13], the authors describe a pi-calculus model of RESTful HTTP. This model, however, comes across as a mechanism for programming a specific type of RESTful HTTP application. The paper does not consider the general properties of REST: statelessness and hypertext-following, nor does it describe a methodology for checking that arbitrary implementations satisfy these properties. There are also a number of books and expository articles on REST, but those do not include formal specifications, nor do they consider analysis questions.

Our work appears to be – to the best of our knowledge – the first to precisely formulate the key properties of REST, and to demonstrate interesting consequences, such as naming independence and the PSPACE-hardness of verification. This work also opens up a number of interesting questions. One is to use the formalization as a basis to investigate questions about REST itself: for instance, how to combine authentication with REST, and how to extend REST to executable representations [6]. We have argued that the parameterized model-checking and synthesis questions are especially relevant for web applications using REST. Constructing a practically usable verifier for REST properties is itself a non-trivial task. We have experimented with simple examples verified using SPIN [14]. An effort to use JPF [25] to verify applications written in the JAX-RS extension of Java was unsuccessful, however, as JPF currently lacks support for key libraries in JAX-RS. Our current focus is on creating a runtime checker, which has the advantage of being independent of implementation language.

To summarize, the formal modeling of REST clarifies its definition, and also raises several challenging questions, both in modeling and in automated analysis.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS (1996)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools, Second Edition. Addison-Wesley (2007)
3. Bizer, C., Heath, T., Idehen, K., Berners-Lee, T.: Linked data on the web (LDOW2008). In: WWW. pp. 1265–1266 (2008), talk by Tim Berners-Lee at TED 2009: `http://www.w3.org/2009/Talks/0204-ted-tbl/`

4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logics of Programs. LNCS, vol. 131. Springer-Verlag (1981)
5. Emerson, E., Clarke, E.: Proving correctness of parallel programs using fixpoints. In: ICALP. LNCS, vol. 85 (1980)
6. Erenkrantz, J.R., Gorlick, M.M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC/SIGSOFT FSE. pp. 255–264 (2007)
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: W3C RFC 2616 (June 1999), http://www.w3.org/Protocols/rfc2616/rfc2616.html
8. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irving (2000)
9. Fielding, R.T.: `http://roy.gbiv.com/untangled/2008/no-rest-in-cmis#comment-697` (2008)
10. Fielding, R.T.: `http://roy.gbiv.com/untangled/2008/rest-apis-must\-be-hypertext-driven` (2008)
11. German, S., Sistla, A.: Reasoning about systems with many processes. Journal of the ACM (1992)
12. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
13. Hernández, A.G., García, M.N.M.: A formal definition of RESTful semantic web services. In: WS-REST. pp. 39–45 (2010)
14. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley (2003), also see `http://spinroot.com`
15. Klein, U., Namjoshi, K.S.: Formalization and Automated Verification of RESTful Behavior. Tech. rep., Bell Labs (2011)
16. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Proc. of the Conf. on Logics of Programs (1985)
17. Milner, R.: Communication and Concurrency. Prentice-Hall (1990)
18. Namjoshi, K.: Symmetry and completeness in the analysis of parameterized systems. In: VMCAI. LNCS, vol. 4349 (2007)
19. Papadimitriou, C.H.: Computational Complexity. Addison Wesley (1994)
20. Pnueli, A.: The temporal logic of programs. In: FOCS (1977)
21. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: TACAS'01. pp. 82–97. LNCS 2031 (2001)
22. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. pp. 179–190 (1989)
23. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Proc. of the 5th International Symposium on Programming. LNCS, vol. 137 (1982)
24. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification. In: IEEE Symposium on Logic in Computer Science (1986)
25. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. 10(2), 203–232 (2003), JPF web page: `http://babelfish.arc.nasa.gov/trac/jpf`
26. SOAP version 1.2 part 1: Messaging framework (second edition). W3C Recommendation (2007), http://www.w3.org/TR/soap12-part1/
27. Uniform Resource Identifier (URI): Generic Syntax. W3C RFC 3986 (2005)