Journal of Web Engineering, Vol. 10, No. 4 (2011) 353–390
© Rinton Press

A FINITE-STATE MACHINE APPROACH FOR MODELING AND ANALYZING RESTFUL SYSTEMS

IVAN ZUZAK IVAN BUDISELIC GORAN DELAC
School of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
izuzak@gmail.com ibudiselic@gmail.com gdelach@gmail.com

Received November 3, 2011
Revised December 20, 2011

Representational State Transfer (REST), as an architectural style for distributed hypermedia systems, enables scalable operation of the World Wide Web and is the foundation for its future evolution. However, although described over 10 years ago, no formal model for representing RESTful systems exists that is comprehensive in following REST principles, intuitive to Web engineers and researchers alike, and offers practical development guidelines. The lack of such formal models has hindered understanding of both the REST architectural style and the Web architecture, consequently limiting Web engineering advancement. In this paper we present a generic model of RESTful systems based on a finite-state machine formalism. We show that the model enables intuitive formalization of REST design principles, including uniform interface, stateless client-server operation, and code-on-demand execution. Furthermore, we describe the model's mapping to a system-level view of operation and apply the model to an example Web application and several real-word Web applications. Finally, we explore the practical challenges and benefits of using the model in the field of Web engineering, ranging from better understanding of REST to designing frameworks for RESTful system development.

Keywords: representational state transfer, World Wide Web, software architectural styles, formal model, finite-state machines, hypermedia
Communicated by: O. Diaz & S. Auer

1 Introduction

One of the main reasons for the wide adoption of the World Wide Web (Web) as a global information system has been its ability to scale and remain reliable with the rapid growth in the number of its users and applications. Enabling this growth is an architecture [1] designed just for the purpose of developing large-scale distributed hypermedia systems such as the Web. The foundation of this architecture is a set of software design principles called the *Representational State Transfer* architectural style (REST) [2]. In essence, REST describes how a Web application should behave in order to maximize beneficial properties, such as simplicity, evolvability, and performance.

From its introduction in year 2000, REST has not only guided many incremental changes in the Web's continuous evolution, such as the recently standardized HTTP PATCH method [3], but has also been guiding the development of its new dimensions in order to preserve its desirable properties. These efforts include the expansion of the Web with higher-level applications, interlinked data, physical devices and real-time access, through mashups [4], the Semantic Web [5], Web of Things [6] and the Real-Time Web [7]. However, as the Web grows in functionality, it also grows in complexity and is consequently becoming harder to understand, analyze and develop at the architectural level. Web engineers no longer work only with the HTTP protocol [8] and HTML media type [9] as during the "Web 1.0" era, rather with many HTTP extensions [10], new application-level protocols such as ATOM [11], SPDY [12], Constrained Application Protocol (CoAP) [13] and PubSubHubBub (PSHB) [7], many different media types such as ATOM [14], JSON [15], VoiceXML [16], and different RDF serializations [17], and thousands of APIs [18]. In the future, this space of technologies will become even more complex, for example with the introduction of WAKA [19], the long-awaited successor of HTTP. At the same time, in order to keep its usability with this growth, the Web must retain the benets of RESTful design. Therefore, it is increasingly important that Web based systems are developed according to the REST style and understanding REST has become essential for engineering the Web and its future.

However, although defined over 10 years ago, architectural principles of the REST style have only been semi-formally described using diagrams, tabular techniques and natural language descriptions. Furthermore, although formal models of hypermedia systems in general do exist [20], no such model covers fundamental principles of REST and most techniques are used to model the Web which includes many unRESTful properties. This lack of formal explanation has increasingly been causing negative effects, such as confusion in understanding REST concepts [21], misuse of terminology [22] and ignorance of benefits of the REST style. For example, common misunderstandings include the overload in meaning of the word state, such as state, application state, resource state and session state [23] [24], and identifying functionality of REST user agents with Web browsers [25].

In result, Web researchers and engineers experience difficulty in concisely explaining both small-scale and large-scale Web patterns or requirements, such as defining Web application interaction [26] [27], formalizing Web browser functionality [28] and defining future Web architectural goals [29]. Furthermore, development of systems which adhere to the REST style is difficult due to a lack of software frameworks which guide their implementation [30]. This is especially true for developing machine-driven RESTful client components and their application in machine-to-machine RESTful interaction and service composition [31]. For example, the Web application description language (WADL) [32] is infamous for not being well aligned with REST principles. We believe this to be a direct consequence of the absence of formal models which are used as the practical encoding of general architectural principles and serve as the foundation for the software development process in such frameworks and description languages.

In this paper we present a finite-state machine (FSM) [33] formalism for modeling RESTful systems, with the primary motivation of contributing to the understanding of the REST style. Our choice of using a FSM formalism was inspired by *The Rule of Least Power* [34] which originally suggests the use of the least powerful language suitable for expressing constraints or solving a problem. Consequently, one of our goals was to explore the possible limits of the FSM formalism for this specific purpose in order to suggest the use of more a powerful model. Furthermore, one of the core principles of the REST style, transferring resource representations for transitioning agents from one state to another, suggests the usage of a state transition system formalism.

Our generic model is based on the nondeterministic finite-state machine formalism with epsilon transitions (ε-NFA). We first explain the mapping of the model's abstract elements to those of a RESTful system. In order to illustrate model usage, we introduce an example Web application and present its ε-NFA model. Next, we explain how each of REST principles map to the model, including uniform interface, stateless client-server operation, and code-on-demand execution. We show that the transition function of the ε-NFA enables formalization of the transformation of the system's application state, following the *hypermedia as the engine of application state* principle. Furthermore, we show that nondeterministic transitions of the model enable formalization of the temporally varying mapping of resources to representations and that ε-transitions enable formalization of code-on-demand execution. The presented model naturally translates to the client-centric view of RESTful system operation with client components storing application state, issuing resource manipulation requests and integrating responses into application state, while server components perform request processing.

This paper extends and refines our previous research in this field [35]. The remainder of the paper is organized as follows. In Section 2, we give an overview of related work, including a description of principles of the REST architectural style, existing approaches for formalization of RESTful and Web-based systems, and the finite-state machine formalism. Section 3 defines our approach for modeling RESTful systems with finite-state machines and presents the model of an example Web application. In Section 4 we discuss practical applications of the presented approach, as well as its limitations. Specifically, we create models of two real-world Web-based systems, explore issues concerning model creation and state explosion for models of complex system, and present advice for designing software frameworks for building RESTful systems. Section 5 concludes the paper with directions for future work.

2 Background and Related Work

This section presents the background and related work for our research. First, in the following subsection, we describe the principles of the REST architectural style. Next, we give a broad overview of existing approaches for modeling RESTful systems. Finally, we recapitulate the finite-state machine formalism which we use as the basis of our formalism for modeling RESTful systems.

2.1 Representational State Transfer Architectural Style

A software architectural style [36] is a set of abstract design principles which, when applied to a specific system, incur beneficial properties. Examples of well known architectural styles include the pipe-and-filter style, layered style and client-server style. The REST architectural style [2] is intended to be used for distributed hypermedia systems and when applied to such systems, REST enables their scalability, simplicity, modifiability, visibility, portability, performance and reliability.

REST may be seen as a composition of six primitive architectural styles, also sometimes called principles or constraints: client-server, layered, cacheable, stateless, code-on-demand and uniform interface principle. A simplified view of the principles of the REST style is presented in Figure 1.

The *client-server principle* promotes separation of concerns by dividing the system into

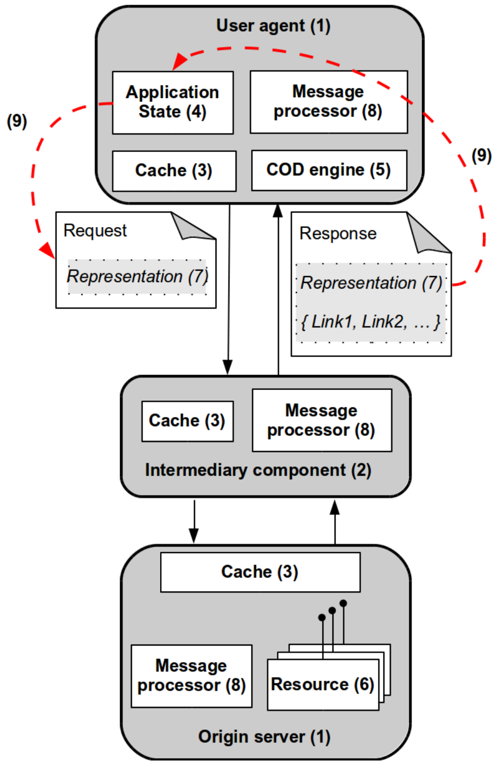

Fig. 1. Simplified visualization of REST principles

client-based and server-based components, such as user agents and origin servers (1). Client-based components are responsible for sending request messages while server-based components respond with response messages. The *layered principle* defines that client-server communication may be proxied through intermediary components, such as caching middleware and gateways (2). The *cacheable principle* defines that request and response messages may be cached by any of the components in the system (3). The *stateless principle* defines that requests from client components should contain all the information necessary to process the request, and that session state is stored by the client component (4). Session state is often called *application state* in the context of RESTful systems. The *code-on-demand principle* defines that client functionality may be extended by downloading and executing code in the form of applets or scripts (5).

Finally, the *uniform interface principle* guides the behavior of components through four sub-principles: identification of resources, manipulation of resources through representations, self-descriptive messages and hypermedia as the engine of application state. The *identification of resources principle* defines that functionality on server-based components is exposed through uniquely identifiable resources (6). The *manipulation of resources through representations principle* defines that client-server communication is based on transferring representations of resources, which are a serialization of the resources' state (7). The *self-descriptive messages principle* defines that each request and response message must include enough information to describe how to process the message, e.g. to indicate cacheability (8). On the Web, This principle is implemented with a standard well-known set of protocols, protocol control data, representation media types and link relations, such as the HTTP protocol, and HTML and ATOM media types. The *hypermedia as the engine of application state principle* restricts requests of client-based components only to resources dynamically identified with links in hypermedia representations of previously received responses (9).

Overall, the operation of a RESTful system may be seen as the temporal transformation of the user agent's application state in order to achieve some application-specific goals. The user agent uses the media types of representations in the current application state to determine the set of hypermedia links. A single link is then chosen based on application goals to form a resource manipulation request. The response representation is then integrated into the application state, where the exact integration method is determined by the link type of the chosen link, after which the cycle is repeated.

2.2 Existing Approaches for Modeling RESTful Systems

This section gives an overview of existing approaches for modeling RESTful systems with the goal of examining the degree of completeness in which REST principles are covered by each approach. We give an overview of related work focused both on REST and similar styles and on modeling Web applications and hypermedia systems in general. While we acknowledge that some modeling approaches have been created with different motivation than ours, such as documentation of systems, we still strongly believe that a comprehensible, intuitive and practical formalism should be the foundation upon which other aspects are defined upon.

In summary, our analysis shows several issues with existing formal and semi-formal models that motivate our research. First, most models are not focused on REST, rather on hypermedia applications in general or Web applications, and thus do not include many of REST principles. Second, most models do not offer an explicit mapping from REST principles to the chosen formalism, and in general do not use the terminology and concepts originally proposed for REST. Third, most models address only REST's static properties or do not offer a mapping of REST principles to a system-level view of operation dynamics. Fourth, some principles of REST are rarely included in models, such as the temporally varying mapping of resources to representations, code-on-demand execution and steady application states.

In [22], a model based on extended influence diagrams and visual modeling tools is presented with the goal of visualizing the structure and design rationale of architectural styles. However, although simple to understand, such a visualization of the hierarchy of principles used to describe architectural styles does not formalize how those principles translate to a system-level architecture. For example, the presented model of the REST style does not explain the roles of different types of components and data elements, or their interaction.

In [37], the authors present Alfa, a framework for characterization of architectural styles, based on composing a small set of architectural primitives. The authors use Alfa to describe many architectural styles, including a significant subset of REST, the layered-client-code-on-demand-cache-stateless-server (LCCOD$SS) style. However, this style does not include the uniform interface principle, one of the most important and distinct principles of REST, while its model does not explain key REST concepts, such as resources, representations and media types.

In [38] the authors present a definition of RESTful semantic Web Services using a process calculus formalism. In this model, a RESTful system is described as a set of processes, representing origin servers and user agents, which exchange request and response messages over uniquely identified channels. This approach is very promising as it allows that channel names exchanged between processes be used to model the exchange of messages containing resource identifiers. This property of the model enables formalization of the REST principle of using hypermedia links as the engine of application state. We encourage further work on using process algebras for modeling RESTful systems which would include a mapping of resources, representations, media types, steady and transition states to such a model together with a generalization of the model which would not be bound to standard HTTP methods as it currently is.

The following two approaches are based on Petri net-based models. In [39] the authors present a promising formal model for specifying RESTful execution of processes specified by Service nets, a specific class of Petri nets that include value passing. The main advantage of the model is its integration of hypermedia-driven application flow while its main use is in modeling composition of RESTful processes. However, the model is not explicit on where application state is stored and does not explain its transformation in response to initial fetches of resource representations which occur at the beginning of an application flow. Furthermore, the model introduces a notion of static and dynamic ports, metaphors for static and temporary resources, which is not RESTful since client components never know and do not need to know if a resource is static or temporary.

Another approach based on Petri nets is presented in [40]. The authors propose REST Chart, a model and language for designing and describing REST APIs, based on Colored Petri nets. Using REST Chart, a REST API is modeled as a topology of representational state spaces while the current state of a user agent is described as the active subset of that topology. However, the model does not address the code-on-demand principle of REST and does not explain the nature of change of the application state with regard to different link types and link relations.

The following five approaches are based on various UML diagrams. In [41] the authors present the Resource Linking Language, an XML-based language for describing interlinked REST resources and consequently the service that can be accessed by interacting with those resources. The language is based on a RESTful service description metamodel, formalized as a UML class diagram, and which incorporates many REST concepts, such as resources, representations, media types and links. However, the static metamodel does not explicitly express the important dynamic properties of REST, such as the application state contained on the client side and the effects of the code-on-demand principle, and does not map REST concepts to a client-server architecture.

In [42] the authors present a navigation-oriented hypertext model based on statecharts, as an upgrade over using finite-state machines. The model uses the structure and execution semantics of statecharts to specify both the hierarchical organization and the browsing semantics of a hyperdocument. However, the model is not focused on RESTful system and does not formalize many RESTful principles, such as the client-server style and the uniform interface principle.

In [43] the authors present an approach to model the structural and behavioral interface of RESTful Web services using UML class and UML protocol diagrams. Furthermore, these models describe the behavior of operations in terms of preconditions and postconditions, and may be serialized as extensions of a WADL interface specification. Although the authors claim that services constructed according to such models are RESTful by construction, the models ignore several principles and concepts of REST. First, the models assumes that all resource identifiers are known in advance in order to define preconditions and postconditions, which does not follow the hypermedia principle. Furthermore, the model ignores the self-descriptive messages principle of the uniform interface principle, does not account for temporally varying resources and confuses resource state with application state.

In [44] the authors describe the process of developing RESTful Web service interfaces from high-level functional specifications. For each phase of the process, the authors use appropriate UML diagrams for modeling service interfaces, and describe the transformation of models for the next phase. While the described approach does attempt to solve the important problem of deriving the set of service resources and links from functional descriptions, it does not account for the overall operation of distributed systems using those resources and links, which is the focus of our research. Specifically, the described approach does not address several key concepts of REST, such as application state and its change as guided by hypermedia controls, and code-on-demand.

In [45] the authors extend the OOH4RIA model-driven development process for Rich Internet Applications (RIAs) in order to support RESTful systems. Similar to the process described in the previous passage, OOH4RIA is based on models and transformations of those models in order to obtain an architecture of a RIA system. The authors extend OOH4RIA with a new model based on UML class diagrams which maps the underlying server-side services to a RESTful interface consisting of resources, protocol actions and links. However, again similar to the process described in the previous passage, this approach focuses only on the static description of server resources, and does not take into account the concept of application state or its transformation.

In [46] the authors present a metamodel for modeling RESTful applications using diagrams similar to UML class and state diagrams. The presented metamodel enables both structural modeling, as a set of resource types, their attributes, relations and representation, and behavioral modeling, as resource state machines. However, the metamodel is strongly focused on modeling server resources, which, although practical, is not a concern of RESTful design. At the same time, the proposed metamodel ignores the concerns of client-based components and interactions, such as defining the application state and code-on-demand execution.

In [47], an agent-based model of RESTful applications is presented. In the model, an agent represents the user agent side of the application while the environment represents the origin server side. The agent has several pools of predefined logic, including application, action and protocol logic, formalized as a hierarchical state machine which drives the agent's action selection. Although explaining high-level dynamics of a RESTful system, the model is more descriptive than formal, not providing an explicit mapping of many REST principles to the model, including code-on-demand execution, resource representations and temporally varying mapping of resources to representations.

In [48], the authors present a broad overview of modeling methods for Web application verification and testing, using a categorization of criteria for classifying models of Web application. Although some criteria may be regarded as reflections of REST principles, such as the *dynamic navigation* criterion which asserts the possibility of modeling servers that may nondeterministically return responses for the same requests, this work is focused on Web applications only and most principles of REST were not considered. For example, in [49], the authors introduce a finite-state machine (FSM) behavioral modeling approach for hypermedia Web applications. The model is based on presenting Web pages as states and links as transitions in a FSM. Furthermore, the authors define multiple types of pages and transitions in order to model activity-initiated transitions and automatic transitions. However, the model is based on a deterministic FSM and does not explain the temporally varying mapping of resources to representations in RESTful systems. Furthermore, the model is based on using only "clickable links" for transitions, i.e. only navigation is used for changing application state, which is an incomplete definition of transitions in RESTful systems.

In [20], the authors give a broad systematization of formal and semi-formal reference models for hypermedia systems and a comparison of hypermedia engineering methodologies. Hypermedia reference models capture important abstractions found in hypermedia applications and describe the basic concepts of these systems, such as the node/link structure. Semi-formal models include the Amsterdam Hypermedia model while formal models include the Trellis and Dexter reference models. However, although these models describe the mechanisms by which the links and nodes in the hypermedia network are related, these do not include many principles, concepts and terminology of RESTful systems. For example, the Dexter reference model uses components and instances, while REST uses resources, representations and application state. Furthermore, these models do not offer a dynamic operational system-level view which maps system components to clients, servers and intermediaries.

In [50] the authors present an automata-based model of hyperdocuments with the goal of verifying trace-based properties by model checking. The model is focused on simple hyperlink-based connectedness of hypertext documents for the Trellis hypermedia system, which does not include important properties of RESTful systems, such as the uniform interface principle. However, two interesting ideas are presented. First, the underlying model upon which a link automaton is constructed is based on place/transition nets in order to allow modeling of parallel execution of hyperdocuments. Second, the authors present a temporal logic for model checking link automatons.

In [51] the authors present a formal definition of RESTful Web services and propose a method for RESTful Web service composition based on Linear Logic. As described, the proposed approach improves the searching efficiency and guarantees the correctness and completeness of the service composition process. However, the proposed approach ignores the concept of hypermedia, the core concept of RESTful systems, and does not explain how models based on Linear Logic satisfy system evolvability, another core benefit of RESTful design. The authors also introduce many concepts, such as resource copies, which are not a part of REST principles. Furthermore, linear logic is not a widely known formal model and will be non-intuitive to most Web engineers and REST researchers.

While it is not focused on modeling, in [52] the authors describe a technique for crawling Rich Internet Applications (RIAs) which is based on a hypercube model of crawled applications. In the hypercube model, the state of an application is represented as a set of events, such as user interface events and network events, that are enabled in that state. Each edge in the model represents a transition between states that is triggered by the execution of exactly one event available in the current state. While this description of application state is well aligned with REST's definition, the model does not account for nondeterministic transitions which are the basis of resources that have temporally varying representations.

2.3 Finite-State Machines Formalism

Finite-state machines (FSMs) [33] are a mathematical formalism for describing processes with a finite number of possible states and sequential state transitions. For formalizing RESTful systems, we use a nondeterministic finite-state machine with ε-transitions (ε-NFA), an extension of the basic deterministic FSM model.

A ε-NFA is formally defined as a tuple $(S, \Sigma, s_0, \delta, F)$ where S is a finite, non-empty set of states, Σ is a finite, non-empty set of symbols representing the input alphabet, $s_0 \in S$ is the initial state of the ε-NFA, δ is the state transition function $\delta : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(S)$, where $\mathcal{P}(S)$ is the power set of S, and $F \subseteq S$ is the set of accepting states. A system-level perspective of ε-NFA operation is shown in Figure 2.

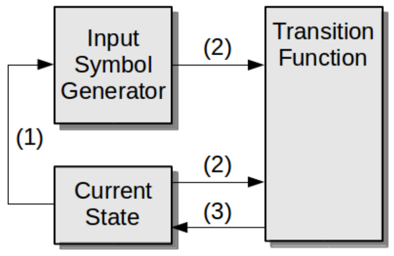

Fig. 2. System-level view of ε-NFA operation

The initial state of the system is stored in the *Current State* module at system startup. The *Input Symbol Generator* module generates an input symbol based on internal rules or environment state (1). For example, the *Input Symbol Generator* module may be a human user of the system or a machine generating input events. Since the generator is not a part of the ε-NFA's formal model but is required to properly model its operation, we define that the generator has access to the system's state stored in the *Current State* module. The *Transition Function* accepts the generated input symbol and the current state (2), determines the next state and stores it in the *Current State* module (3). The described cycle is then repeated until there are no more input symbols generated by the *Input Symbol Generator*. Since the ε-NFA is nondeterministic, the *Transition Function* module returns a set of states, for which the practical meaning is that the system may be in any single state from that set. If at some point at least one state stored in the *Current State* module is marked as accepting, it is said that the ε-NFA accepts the sequence of input symbols read up to that point. Finally, the ε-NFA supports ε-transitions for which the *Transition Function* does not need to read an input symbol in order to modify the system's state.

3 A Finite-State Machine Formalism for Modeling RESTful Systems

Although components of a RESTful system may be viewed as separate agents, each driven by a self-contained FSM, we model the operation of the complete system, often called an *application*, as a single FSM. Such an approach enables the formalization of inter-component interactions required by REST principles, and an higher-level practical view of the system as a whole. While our formalism is focused more on explaining the operation of RESTful systems, the formalism also explains the key elements and their static relations. The central part of this view is the application state of a RESTful system, its definition, transformation during system operation and relation to other concepts, as explained in section 2.1.

In the following subsections we first give an overview of the formalism by defining elements of RESTful systems and mapping them to the elements of the ε-NFA. Second, in order to illustrate the usage of the formalism, we introduce an example Web application and present its ε-NFA model. Next, we describe the formalism in more detail by mapping REST principles to the presented model, including client-server, stateless, code-on-demand and uniform interface styles.

The presented formalism does not explicitly explain the layered and cacheable principles of REST since these are not essential for describing the operation of a system from a functional perspective. The cacheable principle enables that components cache previous responses in order to reduce network requests, therefore only improving performance and not increasing the functional properties of the system. Similarly, the layered principle reduces system complexity and increases scalability by enabling hierarchical organization of components.

3.1 Formalism Overview

In order to explain our formalization of RESTful systems, we first define and give examples of the key elements of RESTful systems. For our examples, we will use two typical Web-based systems; a Web browser navigating to the Google search Web application homepage, and a Web agent using the Twitter API [53].

First, let $ResourceIdentifiers$ be a finite set of **identifiers of system resources** on server components. For example, in Web-based systems the URI namespace [54] is used for resource identification, and examples are `http://www.google.com` for the Google search homepage resource and `http://api.twitter.com/1/statuses/home_timeline` for the Twitter API resource for fetching the user's statuses.

Next, let $Representations$ be the finite set of all **resource representations**, each consisting of resource data, metadata and representation media type identifier:

$$Representations \subseteq data \times \mathcal{P}(Metadata) \times MediaTypeIdentifiers$$

Furthermore, let $Metadata$ be a finite set of key-value pairs $Metadata \subseteq key \times value$, and let $MediaTypeIdentifiers$ be a finite set of **identifiers of possible representation media types**. Example representations include the document representing the Google search homepage and the document containing a user's Twitter timeline statuses. The Google search homepage is a textual document of the HTML media type, i.e. `text/html`, and defines data which is rendered into a visual interface by a Web browser, i.e. the Google search-box control and the Google doodle image. The Twitter timeline document received via the Twitter API is a textual document of the ATOM media type, i.e. `atom+xml`, and contains data which is usually interpreted by machine-driven user agents to perform some task e.g. periodically extract data. For the Web, the IANA organization maintains the set of registered media type identifiers [55].

Furthermore, let *Resources* be a finite set of **server resources** and let *Operations* be a finite set of **resource manipulation operations**. By definition [2], a resource is a temporally varying mapping to a set of equivalent resource representations:

$$\begin{aligned}&Resources : ResourceIdentifiers \rightarrow \\ &\quad (\mathit{time} \times \mathit{Operations} \times \mathit{Representations} \rightarrow \mathit{EquivalentRepresentations})\end{aligned}$$

where $\mathit{EquivalentRepresentations} \subseteq \mathcal{P}(\mathit{Representations})$. Therefore, a resource may both produce a representation of its state and take a representation as input in order to change its state, as signified by the given resource manipulation operation. The set of representations which a resource produces may change over time, but it is required that the semantics of the resource do not change. Furthermore, although a resource may produce a representation of its state in more than one media type it is required that these representations be semantically equivalent.

Next, let *Requests* be the finite set of valid **resource manipulation requests** $\mathit{Requests} \subseteq \mathit{Operations} \times ResourceIdentifiers \times \mathit{Representations}$ and let *Responses* be a finite set of **resource manipulation responses** $\mathit{Responses} \subseteq \mathit{Representations}$. The processing of user agent requests by an origin server may therefore be defined as a function that maps requests to responses $\mathit{RequestProcessor} : \mathit{Requests} \rightarrow \mathit{Responses}$. The processing of a request may be described as internally mapping the request to a set of semantically equivalent representations and selecting a single representation from the set using user agent preferences:

$$\begin{aligned}&\mathit{RequestProcessor}(req) = \\ &\quad Resources[req.resourceIdentifier](currentTime, req.operation, req.representation) \\ &\quad\; [req.representation.mediaTypeIdentifier]\end{aligned}$$

For example, the `http://www.google.com` resource identifier is mapped to a Google search homepage representation and that mapping varies over time because the Google doodle image changes for special events. Similarly, the `http://api.twitter.com/1/statuses/home_timeline` resource identifier is mapped to the Twitter timeline representation in ATOM format. However, the Twitter timeline resource supports many different media types, so the timeline representation may also be returned in semantically equivalent JSON, XML, and RSS formats, depending on user agent's preferences. Therefore, the presented formalization supports the modeling of **content negotiation** as an important property of RESTful systems. Furthermore, since both the Google search homepage and Twitter API resources are exposed using the HTTP protocol, a resource manipulation request contains an HTTP operation, such as `GET` for fetching content, and metadata as HTTP headers, such as `Accept: text/html`, for specifying HTML as the preferred media type.

Furthermore, let *MediaTypes* be a finite set of **media types** which determine the set of hypermedia links present in a representation:

$$\mathit{MediaTypes} : \mathit{MediaTypeIdentifiers} \rightarrow (\mathit{Representations} \rightarrow \mathcal{P}(\mathit{Links})).$$

Each link in the finite set of **hypermedia links** *Links* is defined by a resource identifier, link type and link relation

$$\mathit{Links} \subseteq ResourceIdentifiers \times LinkTypeIdentifiers \times \mathit{LinkRelations}$$

where $LinkTypeIdentifiers$ is a finite set of link type identifiers and $LinkRelations$ is a finite set of link relations. **Link types** are defined in media type specifications; they define the set of valid request generated for links with that link type, and the method by which a user agent should integrate a response to a request for a link with that link type into the application state. Therefore, a link type defines two functions:

$$\begin{gathered} LinkTypes : LinkTypeIdentifiers \rightarrow (RequestValidator, ResponseIntegrator), \text{ where} \\ RequestValidator : Requests \times Links \rightarrow \{true, false\}, \text{ and} \\ ResponseIntegrator : ApplicationStates \times Responses \times Links \rightarrow ApplicationStates \end{gathered}$$

where *ApplicationStates* is the set of all possible **application states**, and a single application state is the set of representations currently present on the user agent. For example, the HTML media type defines many link types, among which are the `<a>` and `<img>` link types. The `<a>` link type defines a navigation link, meaning that the a user agent should generate an HTTP `GET` request to fetch the linked resource's representation and then use it to replace the currently displayed representations. For example, the Google search homepage contains an `<a>` link that navigates the user agent to the Google corporate Web page. In contrast, the `<img>` link type defines an embedding link, meaning that a user agent should generate a `GET` request to fetch the linked resource's representation and then add it into the current set of representations and display the image inline. For example, the Google search homepage also contains an `<img>` link to the Google doodle image which is displayed inline on the search page. Finally, the HTML `<form>` element is an example of a more complex hypermedia link. The `<form>` element may be used to generate both HTTP `GET` and `POST` requests, and may be used to generate request for resource identifiers which are defined only in a template form. Therefore, the request validation function enables complex hypermedia links and usage of **identifier templates**, such as URI templates [56].

Link relations determine the application-specific semantics of a link, and may be considered the machine equivalent of human-readable link text. For example, since the links on the Google search homepage are meant to be interpreted by humans, they do not contain link relations. However, ATOM representations from the Twitter timeline statuses API do contain link relations since the representations are meant to be consumed by machine-driven agents. For example, the ATOM `alternate` link relation signifies that the linked resource is an alternate version of the resource described by the containing element.

Finally, it is said that a RESTful system is in a **steady state** if the client component has no outstanding requests, as explained in more detail later. The set of steady states is therefore a subset of the set of application states: $SteadyStates \subseteq ApplicationStates$. For example, a Web browser navigating to the Google search homepage will be in a steady state when both the top-level Web page and the embedded Google doodle image are fetched.

Finally, we map the defined elements of a RESTful system to the ε-NFA formal model as follows. The **set of states S of the ε-NFA** represents the application states of the system, $S = ApplicationStates$, where an application state is defined as a non-empty, directed graph of representations, as explained in more detail later. Furthermore, **the initial state s_0 of the ε-NFA** represents the initial application state at system startup. The **set of input symbols Σ of the ε-NFA** represents resource manipulation requests and their corresponding link, $\Sigma \subseteq Requests \times Links$. The **transition function δ of the ε-NFA** represents the validation and translation of input symbols into requests, processing of requests into responses, and integration of responses into the next application state, $\delta : ApplicationStates \times (Requests \times Links) \rightarrow \mathcal{P}(ApplicationStates)$:

$$\begin{aligned}
&\delta(state, inputSymbol) = \delta(applicationState, request, link) = \\
&\quad \text{if } link.linkType.RequestValidator(request, link) \text{ is true, then} \\
&\qquad link.linkType.StateIntegrator(applicationState, RequestProcessor(request), link), \\
&\quad \text{else if } link.linkType.RequestValidator(request, link) \text{ is false, then} \\
&\qquad \{\}
\end{aligned}$$

where the result is a set of application states, rather than a single one, to account for the non-deterministic nature of resources. Furthermore, since the ε-NFA includes ε-transitions, for some application states the transition function may change the application state without reading an input symbol, i.e. without the system generating a resource manipulation request.

Finally, the **set of accepting states** F **of the** ε**-NFA** represents the steady application states, $F = SteadyStates$.

3.2 FSM Model for an Example Web Application

In order to illustrate concepts presented in this paper, we introduce a weather forecast Web application. The resources comprising the Web application are shown in Figure 3.

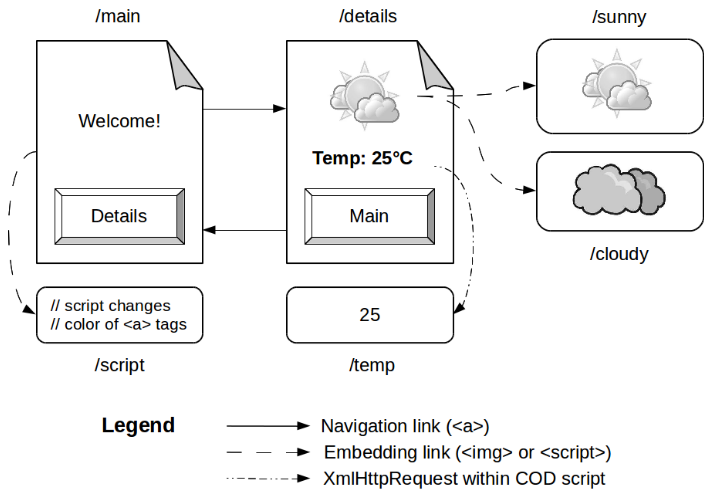

Fig. 3. Example Web application

The base URI of the application is `http://weather.example.com` and we identify its resources using relative addressing. The main Web page, located at `/main`, contains an `<a>` link to the details Web page and a `<script>` element pointing to a JavaScript script located at `/script`. The script periodically highlights the `<a>` link to the details page by changing the color from blue to red. The details Web page, located at `/details`, contains an `<img>` tag pointing either to `/cloudy` or to `/sunny` depending on the current weather. Furthermore, the details page contains a `<script>` element with inline JavaScript code which uses XmlHttpRequest for periodically fetching the current temperature from `/temp` and displaying it in the Web page. Finally, the details page contains an `<a>` link pointing to the main page. The media types of the resources are `text/html` for `/main` and `/details`, `image/png` for `/sunny` and `/cloudy`, and `text/javascript` and `text/plain` for `/script` and `/temp`, respectively. We assume that the user of the application is using a modern Web browser.

The ε-NFA model of the example Web application is shown in Figure 4, with numbers denoting states and letters denoting input symbols. The initial state `0` contains a representation with an `<a>` link to the `/main` page. After an HTTP GET request is issued (`a`), the server returns a response containing the representation of the `/main` page which becomes the current state. State `1` is not steady since the representation contains a `<script>` link to `/script` which must be fetched. An HTTP GET request is issued to fetch the script (`b`) and the application then enters the steady state `2`. Because the script periodically changes the color of the link pointing to the `/details` page, the application's steady state may change between states `2` and `3` using an (ε) transition.

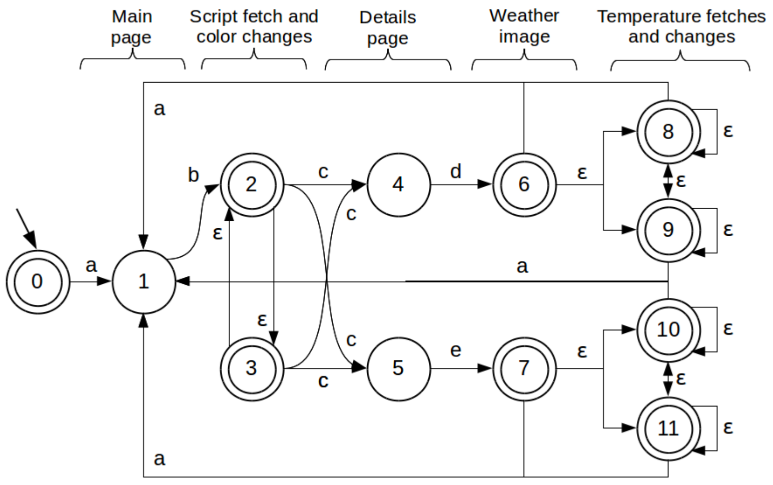
Main page
Script fetch and color changes
Details page
Weather image
Temperature fetches and changes
a
b
c
d
e
ε
0
1
2
3
4
5
6
7
8
9
10
11

Fig. 4. ε-NFA model of example Web application

When the user follows the link to the `/details` page (c), the application makes a nondeterministic transition to transient states `4` and `5` because the representation contains an `<img>` link pointing either to `/sunny` or `/cloudy`. After fetching the linked image using an HTTP GET request (`d` or `e`), the application enters one of the steady states `6` and `7`. Furthermore, the representation of the details page contains an inline script which periodically makes an HTTP GET request for the current temperature. However, since that GET request is a made within a code-on-demand script, it is modeled as an (ε) transition. Furthermore, since the returned temperature may have two possible values (25°C or 30°C), the application nondeterministically enters states `8` and `9` if the weather was cloudy, or states `10` or `11` if the weather was sunny. Because the script executes periodically, the application nondeterministically cycles between states `8` and `9`, or states `10` and `11`, depending on the weather. Finally, because the `/details` page contains a link to the `/main` page, the user may at any time follow the link (`a`) and bring the application back to state `1`.

3.3 Client-Server Style and Stateless Style Principles

Based on the presented formalism, Figure 5 shows the system-level view of a RESTful system as a set of modules, their mapping to the elements of the ε-NFA, and distribution between client and server components. Because client-server interaction in RESTful systems must be stateless, the *Application State* module which stores the current application state is located on the client component. This is also true for modules that are responsible for generating input symbols: the *Media Type Processor*, *Link Storage*, *Application-level Logic*, *Hypermedia-level Logic* and *Protocol-level Logic*. However, the transition function is divided between the client and server components in order to satisfy the client-centric description of RESTful system operation in which the server is responsible only for mapping requests to responses. Therefore, the *Request Preprocessor*, *State Integrator* and *Code-on-demand Engine* reside on the client component, and only the *Request Processor* on the server component.

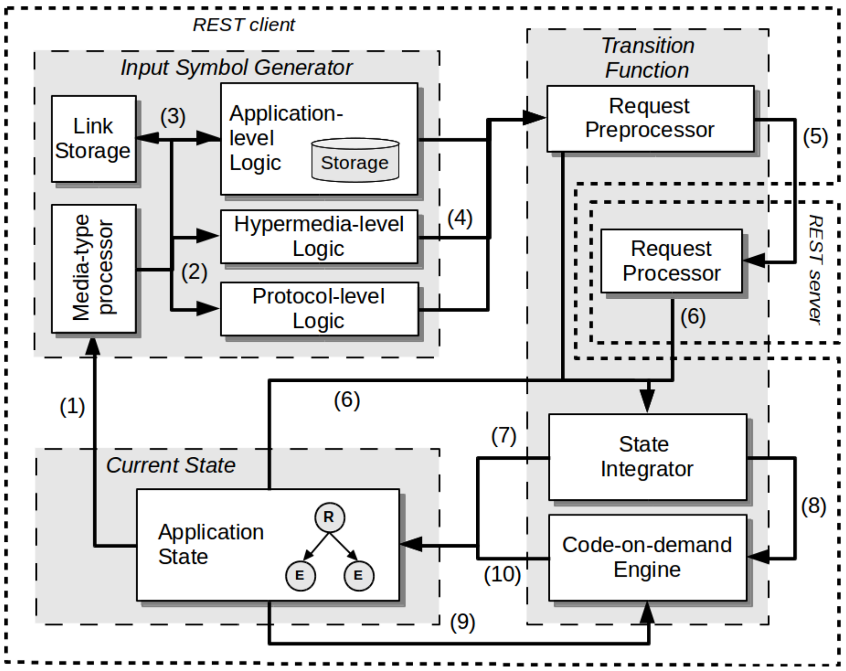

Fig. 5. Mapping of client-server and stateless REST principles to the ε-NFA model

The interaction of the system's modules is defined as follows. The resource representations comprising the current application state are read by the *Media Type Processor* (1) in order to determine the set of available hypermedia links. For example, the `/main` page of the example Web application has the `text/html` media type which enables that the `<a>` link to the `/details` page and `<script>` link to the `/script` script are recognized. The set of links and application state are passed to the *Protocol-level Logic*, *Hypermedia-level Logic* and *Application-level Logic* (2) so that one of the links may be chosen as the basis for the next input symbol. *Protocol-level Logic* and *Hypermedia-level Logic* are responsible for automatically generating input symbols which guide the system to a steady state. *Protocol-level Logic* generates input symbols based on protocol-level links and control data, such as response status codes. For example, in the case of HTTP, if a `301 Moved Permanently` status code and `Location` header are received, the *Protocol-level Logic* automatically generates an input symbol for fetching the resource representation from the new location.

On the other hand, *Hypermedia-level Logic* generates input symbols based on hypermedia-level links in the representation data in the *Application State*, such as embedding links. Since protocol-level rules have priority over hypermedia-level rules, *Hypermedia-level Logic* is executed only after *Protocol-level Logic* has finished executing. Furthermore, because steady states are determined exclusively from the protocol control data and media types of the representations in the current application state, *Protocol-level Logic* and *Hypermedia-level Logic* execute before *Application-level Logic*. If the system is in a steady state, *Application-level Logic* is responsible for generating input symbols based on application-specific goals, which are derived either from user input or from application-specific rules encoded in the module. For example, after the `/main` page has been fetched, two links are available: `<script>` for embedding the `/script` script and `<a>` for navigating to the `/details` page. The former link would be selected by *Hypermedia-level Logic* for downloading the script, while the latter link would be selected by *Application-level Logic*, but only in response to a user clicking on the link.

Application-level Logic may internally store application-specific data from the application state. For example, a machine-driven agent may keep a history of temperatures in the `/details` page from the example Web application. Furthermore, if the system is in a steady state, *Application-level Logic* may store links used to fetch representations present in the *Application State* into the *Link Storage* module (3), which is a capability for bookmarks. The links present in *Link Storage* may be used by *Application-level Logic* as entry points of application navigation, as if they are present in the *Application State*. Therefore, if a link from *Link Storage* is used, the *Application State* is emptied of existing representations before the request is sent. For example, if the `/details` page in the Web application example didn't contain a link back to the `/main` page, the user agent could add the link to the `/main` page to the *Link Storage* and navigate back from the `/details` page.

The input symbol generated by either of these modules consists of a resource manipulation request and the chosen link (4). The *Request Preprocessor* first validates the request with respect to the chosen link and current application state, i.e. validates the hypermedia principle for the generated request. For example, the `/details` page contains an `<a>` link only to the `/main` page, so if the *Application-level Logic* generates an input symbol with any other resource identifier, that input symbol would be marked as invalid and rejected. Next, the *Request Preprocessor* stores and removes the link of the input symbol and adds a request identifier to the request before forwarding it to the *Request Processor* on the server (5). The server's response therefore contains the request identifier and a representation of the identified resource. Although request identifiers are a conceptual requirement for coupling responses with requests, they are not currently used on the Web since requests and responses are related through the TCP connection by which they are sent and received. The *State Integrator* uses the link type of the chosen link, the corresponding server response and the current application state (6) to synthesize the next state (7).

Representations in the *Application State* are organized in a directed graph structure, where one node is marked as a root node. A node in the graph symbolizes a resource representation, while an edge symbolizes an embedding link from a representation to another representation fetched based on that link, e.g. an `<img>` link. For each representation in the *Application State* there is exactly one node, i.e. if two equivalent representations are fetched for embedding links, then there is only one node in the graph for these representation and two edges leading to that node from the parent nodes. Therefore, the root node resource identifier does not completely define an application state, rather it is defined with all the representations and links in the state graph. This definition of application state correctly models the relationship between representations stored in the *Application State* while not duplicating nodes for equivalent representations. Moreover, a graph-based definition of application state supports the existence of cycles which are not forbidden in the sense that a representation may contain an embedding link for itself. However, practical implementations of application state structures will disable infinite cycles from overloading the system, e.g. Web browser have a fixed limit on the depth of nesting `<iframe>` elements.

Based on the link type of the request related to the received response, the *State Integrator* modifies the application state graph to include the response representation. For example, if the `/main` page was fetched and the `/script` script was fetched afterwards via the `<script>` link, the script representation would be added to the application state. On the other hand, if the `/details` page was fetched afterwards via the `<a>` link, the received representation would be the only node in the new application state. Figure 6 displays the described transformation of the application state graph for the example Web application.

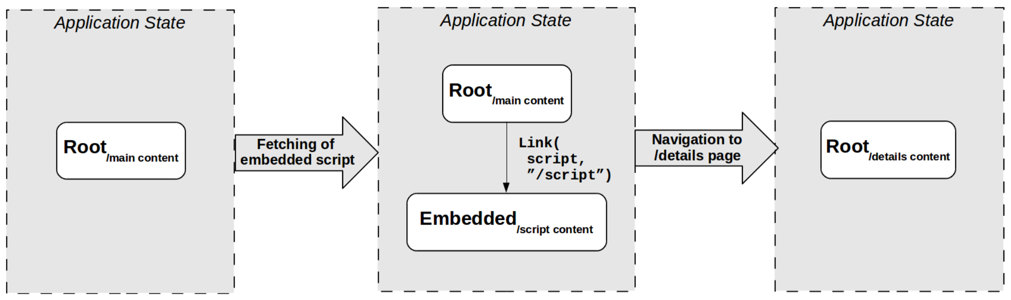

Fig. 6. Transformation of the application state graph for the example Web application

Finally, if the resource representation is an executable script, the *State Integrator* passes the script to the *Code-on-demand engine* for execution (8). The executed script may then examine and change the *Application State* (9) (10) without generating input symbols using hypermedia links. For example, after the `/script` script is fetched and executed using a JavaScript engine, it periodically modifies the application state by changing the color of a link in the representation of the `/main` page.

3.4 Uniform Interface Style Principle

REST is defined by four *uniform interface* principles: identification of resources, manipulation of resources through representations, self-descriptive messages, and hypermedia as the engine of application state. In this section we formalize these principles in the context of our model. *Resource identification* is supported in the model through resource identifiers which are used explicitly in input symbols and application states, where an input symbol consists of a resource manipulation request and link, while the request contains a resource identifier, an operation and a representation. For example, an input symbol $IS_{\text{toDetails}}$ for navigating to `/details` in the Web application example could be represented as:

$$\begin{aligned} IS_{\text{toDetails}} = (\quad & \mathit{Request} : (\mathit{operation} : \text{``}\mathit{GET}\text{''}, \mathit{resourceIdentifier} : \text{``}/\mathit{details}\text{''}, \\ & \quad \mathit{representation} : \text{``...''}), \\ & \mathit{Link} : (\mathit{resourceIdentifier} : \text{``}/\mathit{details}\text{''}, \mathit{linkTypeIdentifier} : \text{``} < a > \text{''}, \\ & \quad \mathit{linkRelation} : \text{``''}) \ , \end{aligned}$$

and the application state AS_{main} of a completely loaded `/main` page as:

$$\begin{aligned} AS_{\text{main}} = \quad & \{\mathit{Nodes} : \\ & \quad [/\mathit{main} : (\mathit{metadata} : \text{``...''},\ \mathit{mediaType} : \text{``...''},\ \mathit{data} : \text{``}/\mathit{main\ contents}\text{''}), \\ & \quad /\mathit{script} : (\mathit{metadata} : \text{``...''},\ \mathit{mediaType} : \text{``...''},\ \mathit{data} : \text{``}/\mathit{script\ contents}\text{''})], \\ & \mathit{Edges} : [/\mathit{main} : [/\mathit{script}]]\} \ . \end{aligned}$$

For readability, we do not include full listings of representation data and metadata. On the Web, metadata in general consists of HTTP headers, while the data is the body of HTTP message.

Manipulation of resources through representations is supported in the model through explicit usage of representations in input symbols and application state. One of REST's foundations is the temporally varying mapping of resources to representations, which is supported through the nondeterminism of the transition function. For example, because the `/details` page contains an `<img>` link to either `/cloudy` or `/sunny`, the navigation from `/main` to `/details` in the example Web application could be represented with the following transition:

$$\delta(AS_{\text{main}}, IS_{\text{toDetails}}) = \{AS_{\text{detailsCloudy}}, AS_{\text{detailsSunny}}\} \ \text{, where}$$

$$\begin{aligned} AS_{\text{detailsCloudy}} = \quad & \{\mathit{Nodes} : [/\mathit{details} : (\mathit{metadata} : [...],\ \mathit{mediaType} : \text{``}\mathit{text}/\mathit{html}\text{''}, \\ & \quad \mathit{data} : \text{``}/\mathit{details\ content\ with\ link\ to\ }/\mathit{cloudy}\text{''})], \\ & \mathit{Edges} : []\} \ , \end{aligned}$$

$$\begin{aligned} AS_{\text{detailsSunny}} = \quad & \{\mathit{Nodes} : [/\mathit{details} : (\mathit{metadata} : [...],\ \mathit{mediaType} : \text{``}\mathit{text}/\mathit{html}\text{''}, \\ & \quad \mathit{data} : \text{``}/\mathit{details\ content\ with\ link\ to\ }/\mathit{sunny}\text{''})], \\ & \mathit{Edges} : []\} \ . \end{aligned}$$

The *self-descriptive messages* style principle is supported in the model through stateless interaction, the limitation of using finite sets for system methods, media types, link types and link relations, and explicitly using these elements in the input symbols and application state. For example, the $IS_{\text{toDetails}}$ input symbol shown above has a link type of `<a>` while the representations in states $AS_{\text{detailsCloudy}}$ and $AS_{\text{detailsSunny}}$ are of the `text/html` media type.

Hypermedia as the engine of application state is supported in the model through the transition function which advances the system from one state to another. Specifically, the output of the transition function is defined only for pairs of states and input symbols for which the input symbol may be derived from the current state or bookmarks. In other words, the current state must contain a hypermedia link used to generate the next input symbol's resource manipulation request. For example, the transition function in the model of the example Web application is undefined for state $AS_{\text{detailsCloudy}}$ and $IS_{\text{toDetails}}$ because the `/details` page does not contain an `<a>` link to itself:

$$\delta(AS_{\text{detailsCloudy}}, IS_{\text{toDetails}}) = \{\} \ .$$

Furthermore, we define that the initial application state is a single representation containing links to resources which are the stable entry points for the system. For example, since the example Web application's entry point is the `/main` resource, the initial state of the model AS_{init} could be represented as:

$$\begin{aligned} AS_{\text{init}} = \quad & \{Nodes : [initial : (metadata : [...],\ mediaType : \text{“}text/html\text{”}, \\ & \quad data : \text{“}HTML\ page\ with\ an\ <a>\ link\ to\ /main\text{”})], \\ & Edges : []\} \ . \end{aligned}$$

Finally, *steady and transient application states* are supported in the model through accepting and unaccepting states. The acceptance of a state is determined by *Hypermedia-level Logic* and *Protocol-level Logic*; if these modules cannot generate any more requests based on hypermedia-level and protocol-level links, the current application state is accepting. For example, in the example Web application, the first representation in an application state is always of the `text/html` media type meaning that all embedded resources, such as resources linked to using `<img>` and `<script>`, should be fetched in order for the system to be in a steady state. Therefore, the state AS_{main} is accepting (steady), while the state $AS_{\text{detailsCloudy}}$ is unaccepting (transient).

3.5 Code-on-Demand Style Principle

The *code-on-demand* principle is defined [2] as client-side execution of downloaded scripts together with the possibility that these scripts extend the functionality of the client. We formalize the code-on-demand principle in the model through ε-transitions i.e. if a script executing on the client component changes the application state from A to B, then this change is modeled with an ε-transition as $\delta(A, \varepsilon) = \{B\}$. In the example Web application, the `/script` script changes the color of the `<a>` link of the `/main` page which can be modeled as $\delta(AS_{\text{main\_link\_blue}}, \varepsilon) = [AS_{\text{main\_link\_red}}]$ and $\delta(AS_{\text{main\_link\_red}}, \varepsilon) = [AS_{\text{main\_link\_blue}}]$, where $AS_{\text{main\_link\_red}}$ and $AS_{\text{main\_link\_blue}}$ are the application states for the `/main` page in which the link is colored red and blue, respectively.

It should be noted that the exact mechanism by which the script obtains data to change the application state is beyond the scope of REST. Therefore, on the Web, it is irrelevant whether a JavaScript script randomly generates data or fetches data from the server using the XmlHttpRequest mechanism.

4 Practical Usage and Limitations

Although our primary motive for researching formal approaches for modeling RESTful systems is to enable simpler and better understanding of REST, another strong motive is to explore the practical applicability of such approaches. In the following sections, we explore the potential practical benefits and limitations of both the presented formalism and models of specific systems. In particular, we are interested in practical aspects of the formalism in the context of two main issues with developing RESTful systems: machine-to-machine (M2M) RESTful interaction and software frameworks for RESTful development.

First, we present two models of real-world Web-based systems and explain insights into properties of models of different classes of Web-based systems, such as Web applications and Web APIs. Next, we address the state explosion problem inherent to state machine-based models of complex systems. Furthermore, we address issues concerning who and how develops such models of specific system, and provide insight for advancement of M2M RESTful development. Finally, we explain how the presented formalism can be leveraged to help design software frameworks for developing RESTful systems.

4.1 Models and Analysis of Web-based Systems

In this section, we apply the presented formalism to two real-world Web-based systems and present their models. First, we present the model of a Web browser user agent using the Google search Web application. Second, we present the model of a machine user agent using the Twitter and GitHub APIs. Since the Web may be observed as a single, evolving large-scale distributed hypermedia system, modeling the whole Web obviously is not feasible. Therefore, the presented models describe only a part of the system relevant to the user agent's goals.

Our motivation for modeling real-world Web-based system is not to find deficiencies in the presented formalism. Rather, our motivation is to explore the modeling limits of using such a simple formalism, and to gain insight. Specifically, as we explain in the final part of this section, the presented models point to several problems of creating and using models, and to typical properties of models of different classes of Web-based systems.

4.1.1 Model of a Web Browser Using the Google Search Web Application

In this section, we present the model of a system in which a Web browser user agent is used to search for the JWE journal homepage using the Google search Web application. The scenario consists of the user navigating to the Google search homepage located at `http://www.google.com`, entering "JWE" as the search term, and using either the regular search or the "I'm feeling lucky" search functionality. We assume that the user is using a modern Web browser with JavaScript execution enabled. Furthermore, we assume that the Google "Instant Search" feature is disabled and that the autocomplete feature is enabled. Finally, in order to keep the model reasonably readable, we ignore several links present on the Google search homepage, such as the links pointing to the Google corporate pages and advertising programs. Figure 7 presents the model of the described system, while a summary of states, input symbols, and transitions of the presented model is given in Table 1.

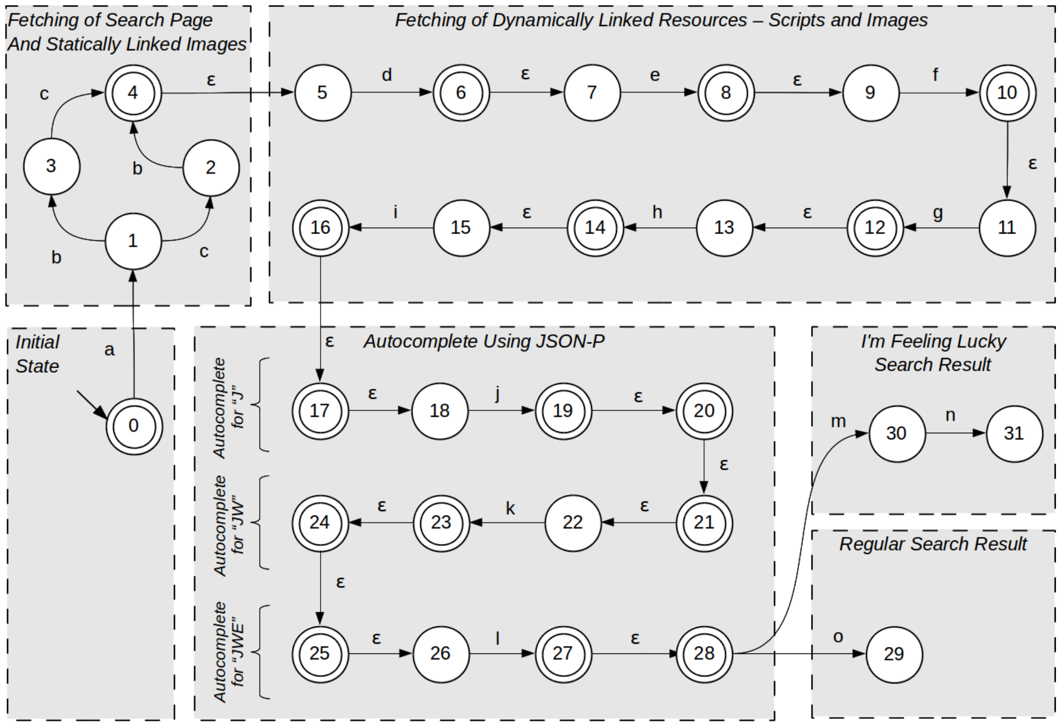
Fetching of Search Page And Statically Linked Images
Fetching of Dynamically Linked Resources – Scripts and Images
Initial State
Autocomplete Using JSON-P
Autocomplete for "J"
Autocomplete for "JW"
Autocomplete for "JWE"
I'm Feeling Lucky Search Result
Regular Search Result

Fig. 7. FSM model of system used to search for the JWE journal homepage, using the Google search Web application

Several interesting parts of the presented model may be noted. First, as indicated by the transition $16\xrightarrow{\varepsilon}17$, the interactions of users with the user agent's interface, such as entering text into the search-box, are modeled as ε-transitions. This approach for modeling user actions is intuitive since user controls may be seen as having background code-on-demand scripts which translate user-generated events into application state changes. Second, as indicated by transitions $1\xrightarrow{c}2\xrightarrow{b}4$ and $1\xrightarrow{b}3\xrightarrow{c}4$, the order by which resources linked to by embedding

Table 1. Summary of states, input symbols and transitions for the FSM model presented in Figure 7

State or transition	**Explanation**
0	Initial state containing the link to the Google search homepage `www.google.com`
$0\xrightarrow{a}1$	Fetching of Google search homepage using an HTTP `GET www.google.com` request, and integration of response as the new root node in the application state.
1	Media type processor parsing HTML links in the application state.
$1\xrightarrow{c}2\xrightarrow{b}4$ or $1\xrightarrow{b}3\xrightarrow{c}4$	Hypermedia-level Logic detects `<img>` links and fetches embedded images using HTTP `GET ssl.gstatic.com/gb/images/b_8d5afc09.png` and `GET www.google.com/images/srpr/logo3w.png` requests containing the Google doodle image and sprites. The State Integrator integrates responses into application state as new child nodes of the homepage representation node.

Table 2. (*Table 1 Continued*) Summary of states, input symbols and transitions for the FSM model presented in Figure 7

State or transition	**Explanation**
$4 \xrightarrow{\varepsilon} 5 \xrightarrow{d} 6$	Upon page load, an inline JavaScript script dynamically adds a `<script>` link (ε) to the homepage representation. The script is then fetched using HTTP `GET www.google.com/extern_js/f/.../uCdHgKP-Nm8.js`, added to the application state as a new child node of the homepage representation node, and executed.
$6 \xrightarrow{\varepsilon} 7 \xrightarrow{e} 8$, $8 \xrightarrow{\varepsilon} 9 \xrightarrow{f} 10$, $10 \xrightarrow{\varepsilon} 11 \xrightarrow{g} 12$, $12 \xrightarrow{\varepsilon} 13 \xrightarrow{h} 14$, $14 \xrightarrow{\varepsilon} 15 \xrightarrow{i} 16$	Code-on-demand JavaScript script dynamically adds several `<script>` and `<img>` links into the Google search homepage representation. Hypermedia-level Logic automatically fetches embedded resource representations using HTTP `GET www.google.com/client_204`, `GET clients1.google.com/generate_204`, `GET www.google.com/images/swxa.png`, `GET www.google.com/images/nav_logo91.png`, and `GET ssl.gstatic.com/.../sem_3498c5d919c7af5cee92648596f33900.js` requests. Fetched representations are added to the application state as new child nodes of the homepage representation node, and JavaScript scripts are executed.
$16 \xrightarrow{\varepsilon} 17$	The user enters the letter “J” into the search-box.
$17 \xrightarrow{\varepsilon} 18 \xrightarrow{j} 19$	JavaScript script detects the letter “J” in the search-box and adds a `<script>` link into the homepage representation, for fetching autocomplete results via JSON-P [57]. The added `<script>` element resource is fetched with an HTTP `GET clients1.google.com/complete/search?client=hp&sugexp=kjrmc&cp=1&gs_id=8&q=J` request, integrated into application state, and executed as a code-on-demand script.
$19 \xrightarrow{\varepsilon} 20$	The script executed in the previous step contains the autocomplete results for the letter “J” and adds them to the homepage representation.
$20 \xrightarrow{\varepsilon} 21$, $21 \xrightarrow{\varepsilon} 22 \xrightarrow{k} 23$, $23 \xrightarrow{\varepsilon} 24$	The user enters the letter “W” into the search-box. JavaScript script detects the change in the search-box text and adds a `<script>` element for fetching the autocomplete results for “JW”. The script is fetched using an HTTP `GET clients1.google.com/complete/search?client=hp&sugexp=kjrmc&cp=1&gs_id=8&q=JW` request, integrated into the application state, and executed. The executed script adds the autocomplete results into the homepage representation.
$24 \xrightarrow{\varepsilon} 25$, $25 \xrightarrow{\varepsilon} 26 \xrightarrow{l} 27$, $27 \xrightarrow{\varepsilon} 28$	The user enters the letter “E” into the search box. JavaScript script detects letters “JWE” in the search-box and adds a `<script>` element for fetching the autocomplete results for “JWE”. The script is fetched using an HTTP `GET clients1.google.com/complete/search?client=hp&sugexp=kjrmc&cp=1&gs_id=8&q=JWE` request, integrated into the application state, and executed. The executed script adds the autocomplete results into the homepage representation.
$28 \xrightarrow{m} 29$	The user presses the “Search” button, initiating the `<form>` link element. The Application-level Logic uses the `<form>` element to make an HTTP `GET www.google.hr/search?hl=en&source=hp&q=JWE&oq=JWE&aq=f&aqi=g6g-s2g2&aql=1` request with the search query passed as a URI query parameter. The returned response contains the Google search result page for the “JWE” string. The response is integrated into the application state by replacing the existing representations as the single new root node. In the following steps, the browser’s Hypermedia-level Logic would continue to fetch the embedded images and scripts for the search results page.
$28 \xrightarrow{n} 30 \xrightarrow{o} 31$	The user presses the “I’m feeling lucky” button, initiating the `<form>` link element as before. However, the HTTP `GET www.google.hr/search?hl=en&source=hp&q=JWE&oq=JWE&aq=f&aql=1&gs_sm=ib&btnI=1` request sent to the server (n) contains a query parameter defining that this is an “I’m feeling lucky” type of search. The server response is an HTTP `302 Found` response containing an HTTP `Location` header with the value of the JWE homepage URI. Therefore, the browser’s Protocol-level Logic generates an HTTP `GET http://www.rintonpress.com/journals/jwe/` request (o) to fetch the JWE journal homepage. In the following steps, the browser’s Hypermedia-level Logic would then continue to fetch the embedded images and scripts for the JWE homepage.

links are fetched does not affect the operation of the application. This shows that the graph data structure chosen for storing the application state correctly models the expected result, i.e. the application state graphs of immediate steady states are equivalent for any order of inserting embedded resources. Third, as indicated by the transition $4 \xrightarrow{\varepsilon} 5$, code-on-demand scripts may transfer the system from a steady state to a non-steady state by dynamically adding new embedding links into the application state. In such situations, the Hypermedia-level Logic automatically fetches the embedded resources representations in order to return the system to a steady state. Fourth, as indicated by transitions $25 \xrightarrow{\varepsilon} 26 \xrightarrow{l} 27$ and $27 \xrightarrow{\varepsilon} 28$, the Google search Web application uses the JSON-P mechanism for dynamically updating the application state using server data, rather than using XmlHttpRequest, and the JSON-P mechanism is correctly formalized by the model.

4.1.2 Model of a Machine Agent Using the GitHub and Twitter APIs

In this section we present a model of a system in which a machine-driven user agent uses the Twitter [53] and GitHub APIs [58]. The goal of the agent is to search for tweets containing the term “JWE” and collect links pointing to those tweets. After collecting the links, the agent stores the links into a GitHub gist by forking an existing gist with the description “JWE” and appending the links to the gist. The presented model does not include the authentication details for using the GitHub API, in order to focus on application-relevant states.

The FSM model for the described system is presented in Figure 8, while a summary of states, input symbols, and transitions of the presented model is given in Table 3.

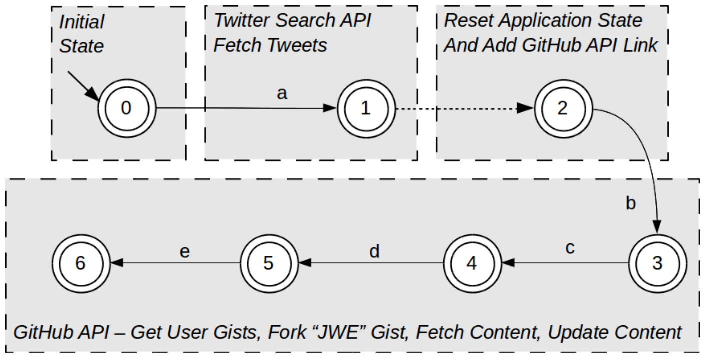
Initial State
Twitter Search API Fetch Tweets
Reset Application State And Add GitHub API Link
0
a
1
2
b
3
c
4
d
5
e
6
GitHub API – Get User Gists, Fork “JWE” Gist, Fetch Content, Update Content

Fig. 8. FSM model of system used to search for and store tweets mentioning “JWE”, using the Twitter and GitHub APIs

Some of the interesting properties of the presented model are highlighted hereafter. First, all state transitions are navigational i.e. resources of both APIs return representations without links for embedded resources. Second, although the GitHub API uses a custom non-standardized JSON-based hypermedia type for representations, it is possible to define a custom Media Type Processor to parse and extract links from such representations. Similarly, it is possible to define custom link types for such custom media types in order to validate requests and integrate responses into the application state, as required by the FSM formalism. Next, API resource representations do not use code-on-demand scripts for changing the application state, rather they rely strictly on hypermedia-driven state change. Finally, as

Table 3. Summary of states, input symbols and transitions for the FSM model presented in Figure 8

State or transition	**Explanation**
0	Initial state containing links to the Twitter API search resources and the GitHub API gists resources.
$0 \xrightarrow{a} 1$	The machine agent's Application-level Logic makes an HTTP `GET search.twitter.com/search.atom?q=JWE` request to fetch the tweets containing the phrase "JWE", and integrates the response ATOM document as the single new root node of the application state.
1	The Media Type Processor parses the search response document with an ATOM parser, while the Application-level Logic stores the found tweet links into internal storage.
$1 \longrightarrow 2$	The Application-level Logic resets the application state to the initial state in order to obtain the entry point link for the GitHub gists API.
$2 \xrightarrow{b} 3$	The Application-level logic makes an HTTP `GET api.github.com/gists` request to fetch the list of the user's gists. The response is JSON document which is integrated into the application state as the single new root node.
3	The Media-Type Processor parses the received JSON document using a custom media type parser, while the Application-level Logic finds the link for the gist described with "JWE".
$3 \xrightarrow{c} 4$	The Application-level Logic makes an HTTP `POST api.github.com/1327010/fork` to create a new fork of the "JWE" gist. The response is an HTTP `201 Created` message with a `Location` header linking to the resource identifier of the created fork. The response is integrated into the application state as the single new root node.
$4 \xrightarrow{d} 5$	The Application-level Logic detects the link for the newly created fork and makes an HTTP `GET api.github.com/gists/1327013` request to fetch the gist content and integrates the response into the application state as the single new root node.
5	The Application-level Logic appends the internally stored tweet links to the downloaded gist content.
$5 \xrightarrow{e} 6$	The Application-level Logic makes an HTTP `PATCH api.github.com/gists/1327013` request to update the gist fork with the new content.

indicated by the transition $1 \longrightarrow 2$, because there are no links in the Twitter API resource representation pointing towards GitHub resources, there is a discontinuation in following hypermedia links for achieving application goals. The machine agent application-level logic is aware of this discontinuation and therefore resets the application state to return to the initial state containing links to well known entry points, including the GitHub gists API link.

4.1.3 Insights from Modeling Real-World Systems

In this section, we present insights from the previously presented models of real-world Web-based systems.

Several questions arise concerning the practical applicability of models. First, models of complex systems will be very large due to a large number of representations, which affects the number of application states, as well as a large number of requests generated for links, which affects the number of input symbols. There are several reasons as to why the number of representations and requests will be very large: resources of interest link to other resources which increases the size of the observed system, the representations of resources may have a high frequency of change over time, and the set of request generated by user-driven user agents is virtually unbounded. For example, the Web is a well connected system in which a user can navigate to almost any page from any other page in several navigation steps. As a partial solution we described in this section, the formalism permits that only a part of the global hypermedia system is modeled, thus focusing on states and input symbols concerning application goals and significantly reducing the model size. Furthermore, in the following section, we explore several possibilities of reducing the size of FSM models by discovering and aggregating equivalent states, with different approaches for defining state equivalence.

Second, the question arises as to who develops such models, how, and when, and what their practical usage would be. We explore these questions in more detail in Section 4.3. Third, while resources of most RESTful systems will be well connected using hyperlinks, there still may be discontinuations in hyperlinks preventing the reaching of application goals. As we demonstrate in Section 4.1.2, the Twitter and GitHub API resources are not mutually connected with hyperlinks, although they do contain hyperlinks for guiding user agents within their API domain. As we describe, a strategy for overcoming this issue is in bookmarking well known entry point links for starting an application flow within one API domain, and using the bookmarks when faced with discontinuations. In the following sections, we explore some of these ideas further in the context of M2M interaction.

Finally, the presented models highlight properties of models of different classes of Web-based systems. For example, simple "Web 1.0" systems will usually only have a single level of embedded resources in the application state graph, typically for embedding images. Furthermore, "Web 1.0" systems will typically not use code-on-demand scripts to change the application state, i.e. models of such systems will not have ε-transitions. In contrast, "Web 2.0" systems typically will use code-on-demand scripts to change the application state, such as using XmlHttpRequest for fetching data from the server and dynamically changing the user interface. Furthermore, "Web 2.0" systems are characterized by scripts dynamically adding new hyperlinks for embedding resources, such as images and JSON-P scripts. By observing the application state graph, "single-page" Web applications may be detected as graphs in which the URI of the root representation never changes. Similarly, "widget-based Web applications" using nested `<iframe>` elements may be detected as having more than a single level of embedded representations in the application state graph. Finally, "Web API"-based systems are characterized by representations not having embedding links, i.e. all links are navigational, and not using code-on-demand scripts, similar to "Web 1.0" systems. Furthermore, models of "Web API"-based systems will have a greater degree of discontinuations as they are almost never connected with resources outside their domain, and often not well connected with other resources within their domain. Therefore, while "Web API" resources are exposed using URIs, they are rarely hypermedia-driven.

4.2 Model Size and Minimization

In this section we address issues which may be raised concerning the finite set limitations of the presented formalism and the size of models of complex systems.

First, because RESTful systems are not restricted to a finite number of states or input symbols, finite-state machines are not a completely suitable model. The ε-NFA model may be relaxed so that the set of states may be infinite or even not countable or, alternately, other kinds of models for describing infinite-state machines may be used. One possible candidate are labeled state transition systems [59], a formalism similar to finite-state machines which permits that the number of states and transitions be infinite. While this approach is directed at supporting infinitely large models, in our opinion, practical use cases will likely look to reduce the number of states of system models, for which there are other alternatives.

In essence, our formalism inherits the state explosion problem from the underlying FSM formalism; the number of states in models of complex systems is extremely large. As explain in the previous section, the reason for a large number of states is the large number of possible representations which constitute application states. In order to reduce the state explosion problem, in the rest of the section we explore tree approaches, which may be used separately or combined:

1. using generic FSM-based algorithms for detecting and removing unreachable states,
2. using generic FSM-based algorithms for detecting and merging equivalent states, and
3. defining custom state equivalence relations based on the contents of the application state representations.

First, the generic formalism of finite-state machines defines that two state machines are equivalent it they accept the same language, where a language is defined as the set of all possible sequences of input symbols over some alphabet [33]. Applying a reachability algorithm to the initial state of the FSM and removing states which are not reachable will therefore not affect the operation of the FSM. This analysis may also be used for error detection; if a state which should be reachable is discovered as unreachable, there is a discontinuation in the expected hypermedia links of available representations.

Next, the generic formalism of finite-state machines defines two states S_1 and S_2 as equivalent if for every sequence of input symbols, the FSM either accepts the sequence starting from both states or rejects the sequence starting from both states. In other words, states S_1 and S_2 are considered equivalent if they accept the same sequences of input symbols, and therefore may therefore be merged into a single state without affecting the operation of the FSM. We apply this technique to the example weather Web application presented in Section 3.2 and present the minimized model in Figure 9.

As show, states 2 and 3 are recognized as equivalent and merged since they are both acceptable and both have the same nondeterministic transition to states 4 and 5 for input symbol (c). However, states 4 and 5 have not been recognized as equivalent because state 4 has a transition only for input symbol (d) for fetching the sunny weather image, while state 5 has a transition only for input symbol (e) for fetching the cloudy weather image. States 6, 7, 8, 9, 10, and 11 have all been merged since they transition either to themselves for ε-transitions used by code-on-demand scripts for fetching temperatures, or back to state 1 for input symbol (a) for navigating to the main page. As can be seen on the image, the number of states in the model is reduced from 12 to 6 states.

This example also illustrates the meaning of the generic FSM state minimization technique in the context of RESTful systems. The technique focuses on the acceptance of states and on the input symbols for supported transitions of states, while at the same time ignores the meaning of each state in terms of representation content. Since the set of possible input

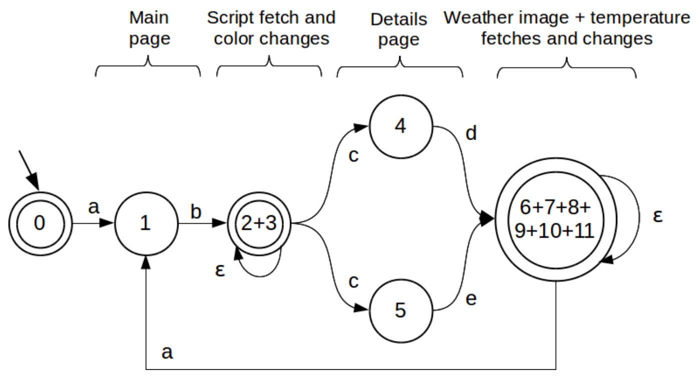
Main page
Script fetch and color changes
Details page
Weather image + temperature fetches and changes
0
a
1
b
2+3
ε
c
4
c
5
d
e
6+7+8+ 9+10+11
ε
a

Fig. 9. Generic FSM minimization of example weather Web application model

symbols for a state depends on the set of links available in the representations, it may be said that this minimization technique merges states S_1 and S_2 if the set of links in their representations is equal. In other words, the technique observes the requests which a user agent may generate at two states in order to determine state equivalence.

While this result intuitively makes sense and significantly reduces the number of states, there are several drawbacks. First, since representation links define the set of input symbols leading out of each state, changes in link resource identifiers will result in different input symbols and consequently nonequivalent states. An example of such two states are states 4 and 5 which contain different identifiers in links for fetching weather images, while the rest of the content is identical. Therefore, generally stated, in situations in which different resource identifiers point to resources which fulfill the same application functionality, differentiating states based only on link identifiers is too fine-grained. Second, in some cases, the technique may be too coarse-grained, for example when differentiating states with the same links, but different content, is needed. For example, states 6, 7, 8, 9, 10, and 11, all have the same links and are merged, but have different weather images and different temperatures, which may be useful criteria for differentiating application states. Similarly, the technique will merge states which have the same content and links, even though the root representations of their respective state graphs were fetched from different resources.

Therefore, in the context of RESTful systems, it may be useful to define custom state equivalence relations [60] based on the contents of the application state. Such equivalence relations may be defined over any part of the application state graphs, such as:

1. aggregate states with equal root representation identifiers. This approach is often found in descriptions of Web applications in the form “one page is one state”, as described in Section 2.2 for some existing research approaches.

2. aggregate states based on link relations of links in representations, rather than by link identifiers. This approach is the basis of descriptions of application-domain protocols based on link relations, as described in [61] [62].

3. aggregate states based on representation content.

In Figure 10 we present a minimized model of the example Web application by aggregating states with the same root representation resource identifier, states which have a temperature higher than 27°C, and states which have a temperature lower than 27°C. As shown, this aggregation scheme reduces the number of states from 12 to 4.

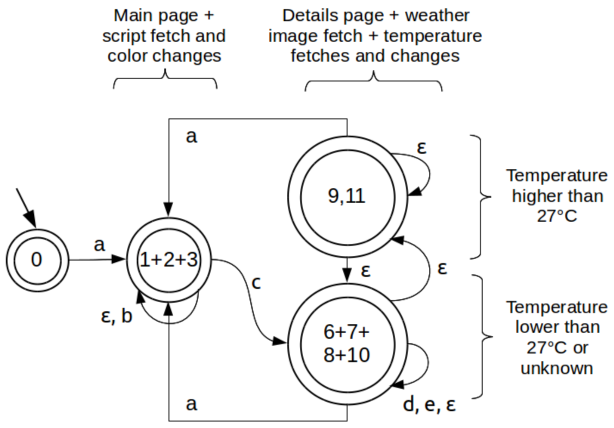
Main page + script fetch and color changes
Details page + weather image fetch + temperature fetches and changes
a
ε
9,11
Temperature higher than 27°C
0
a
1+2+3
ε
ε
c
ε, b
6+7+ 8+10
Temperature lower than 27°C or unknown
a
d, e, ε

Fig. 10. Minimization of example weather Web application model based on custom equivalence relation

Similar aggregation-based approaches may be utilized with the goal of reducing the number of input symbols of a model, which we do not describe in more detail.

4.3 Creating and Using Models of Specific Systems

Some of the core questions of the practical applicability of the presented approach are concerned with benefits of creating models of specific systems, as well as who creates them, when, where and how. In this section we discuss these questions.

There are several practical benefits of creating specific models. First, creating models for existing systems benefits their documentation and understanding as one of the cornerstones of modern software engineering. While a graphical notation may become unreadable for complex systems, this problem may be alleviated with good user-interface design for navigating such visualizations, or textual notations may be used. Second, creating models as specifications for future systems enables model driven development [63]. It is feasible to create a code generator that inputs a FSM model and application-specific goals, as desired states, and outputs the code of a machine agent that fulfills those goals. Third, creating models of existing systems enables model verification using temporal logic. This aspect of using state machine models is well described in previous research [50] [64] [65] [66], in which the CTL temporal logic is used for model checking, such as reachability and safety analysis. For example, formal model verification may be used for finding broken links or even more complex sequences of specific application states.

Since RESTful systems are in general developed and executed in a decentralized manner, a complete model for a system in practice will not be developed by a single party, such as a client-side developer, or a server-side developer. The reason why a developer of server resources is not capable of modeling the complete system is because she is only aware of resources offered by the server, their representations, and the possible links between those resources. However, server developers cannot anticipate all possible representations sent by user agents, or that user agents may be using multiple server components in order to achieve their application goals. Similarly, the reason why a client developer is not capable of modeling the complete system is because server functionality is exposed through interlinked resources, for which only the entry point resource is known in advance, while the links and resource representations may change at run-time. In other words, a user agent is aware of the part of the application states it has explored in one possible execution flow. Therefore, the system model is divided between client and server components, and the model changes over time due to server evolution.

While this conclusion may be expected, its implications with regard to practical applications of the model may not be as clear and therefore we explore them in the following paragraphs. First, if there is a need for creating a fairly complete model of a system, there are two possible strategies. Since the knowledge for creating the model is divided between client-side and server-side developers, one approach is for the two parties to work together on creating an overall system model. For example, on the Web, many APIs are described in terms of fixed resource identifier templates, resource representations and links between resources. Two examples of such APIs are the GitHub [58] and Twitter APIs [53]. Therefore, a client developer can use such documentation to develop a complete system model. However, the premise of this approach is that the description of resources is stable, which reduces system modifiability, a key benefit of REST.

Another strategy is to apply crawling techniques to discover the model at run-time. Although the concept of crawling is well know and researched, most crawlers do not create models compatible with the presented FSM formalism. However, recent research resulted [52] in a technique for crawling of rich Internet applications (RIAs) which produces a model reasonably compatible to ours. The technique utilizes a hypercube model of possible application states and events as transitions between states. The crawler then explores the state space and amends the model accordingly. The similarity between the models created using the crawling technique and the models created using our formal approach is based on the same view of the application state as the set of representations. In other words, a change in any of the representations in the application state is translated into a transition to a new state. However, there are also differences between the crawled and manually created models, the biggest of which is that the crawled model does not account for nondeterministic transitions.

Finally, we focus the discussions about challenges of creating models from this section, and the previous section on state minimization in order to provide insight for guiding machine agents in machine-to-machine interaction. Machine agents are user agents with specific application goals, typically within an application domain, e.g. fetching the latest 20 tweets from a user's Twitter timeline, and the popular Restbucks example for ordering coffee [61]. Developing such machine agents therefore requires knowledge about the contract for using a server resource: the protocol and media types of representations. Additionally, for implementing the Application-level Logic of such agents, knowledge about the semantics of following each link is required in order to guide the agent through the state space of the system toward the goal state.

Therefore, having a state model of the whole system would help in implementing such machine agents. However, as we explained in this section, client-side developers do not have enough information to create an overall system model. On the other hand, as we explained in Section 4.2, the user agent does not need to know the entire state space, rather just the path leading to the goal state. This path may be seen as a composition of segments, each of which is related to a domain of a particular service. Within each such domain, the service itself contains the information to construct the state-transition segments representing possible service functionality. For example, a service for buying books may expose functionality for ordering books and putting books in a shopping basket, each of which requires a predefined sequence of state transitions. Therefore, a possible solution for constructing machine agents is for servers to provide these models, and as we explained in this section, this may be done by describing the complete service API. On the other hand, the model may change dynamically due to the continuous evolution of resource representations, making such fixed API descriptions less usable.

As we explain in Section 4.2, the user agent does not need to know the exact state space in order to traverse it, rather an aggregated view of it from the perspective of application-relevant information. Therefore, in order for client-side developers to construct machine agents, server-side developers needs to provide such a minimized and aggregated view of state-transition segments for executing the offered services. Figure 11 presents a generic view of this situation; a machine agent's goal is reaching state 3, e.g. buying a book, and therefore needs a guide for traversing the state space which provides several possible links for continuing from the current state 2, e.g. the entry state of a bookstore service. The server exposes two functionalities through its resources, each may be described with a state-transition segment for using it, and one segment ending in the desired state 3. Therefore, the requirement is that the server describes the segments, and that the user agent downloads the descriptions and uses them to implement the Application-level Logic.

The combined issues of describing the server functionality and the transfer of this description to the user agent is one of the open issues of developing RESTful machine-to-machine systems. In essence, the server needs to describe the abstract states representing steps of using a functionality as well as links which should be chosen in each state to reach the goal. One approach, proposed in [61] [62], is based on defining a set of application-specific link relations to describe the semantics of links in representations, and on defining the order of using such typed links in order to reach a goal. These link relations and usage orders would be defined in human-readable documentation for the service, and agent developers would implement the agent's Application-level Logic according to this documentation. However, this approach requires that developers read documentation and additionally hinders the evolvability of the service because its description is delivered statically, rather than on demand.

We believe that the formalism presented in this paper provides insight for improving the previous solution, based on model minimization through state aggregation, and on on-demand model segment transfers. Just as servers should be able to on-demand instruct user agents how to construct resource identifiers based on identifier templates [67], we believe that servers should be able to on-demand instruct user agents how to navigate through their application-level hypermedia processes. In other words, instead of a static description of link relations within service documentation, we propose that the user agent fetches the description when needed at run-time, in a machine-readable format. Machine-agent developers would

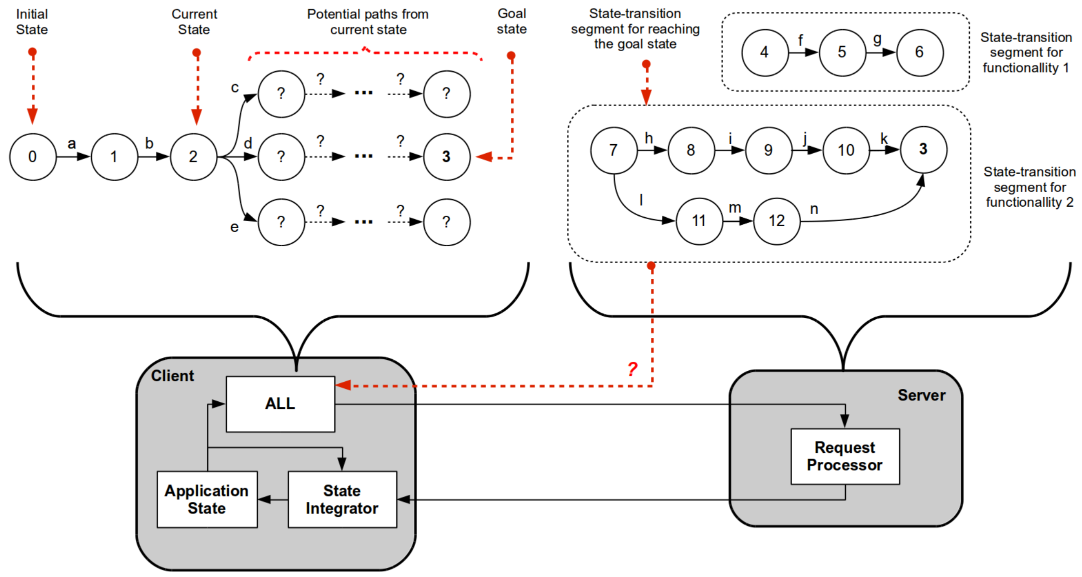
Initial State
Current State
Potential paths from current state
Goal state
State-transition segment for reaching the goal state
State-transition segment for functionallity 1
State-transition segment for functionallity 2
0
a
1
b
2
c
d
e
?
3
4
f
5
g
6
7
h
8
i
9
j
10
k
l
11
m
12
n
Client
ALL
Application State
State Integrator
Server
Request Processor

Fig. 11. View of situation for guiding machine agents using aggregated model segments

then only need to know the entry point for requesting such descriptions and implement the Application-level Logic to receive the description, parse, and integrate it as new rules to guide its execution. Therefore, the service is free to evolve as long the description of the machine-readable description evolves accordingly and the entry point for fetching the description is kept stable. Since we are currently working on an implementation of such an approach, we do not describe work-in-progress in more detail.

4.4 Software Frameworks

Developing systems that conform to the principles defined by REST is difficult, although REST principles have been known for more than a decade. We believe that one of the main issues with RESTful system development is the lack of adequate software development frameworks. Software frameworks are one of the foundations of modern information systems; they significantly reduce the complexity of software development by providing implementations of core technologies and guiding the development process. However, most existing frameworks for "RESTful" Web development are strictly technology-oriented, rather than architecture-oriented. Technology-oriented frameworks provide a flat stack of technology primitives for development but do not incorporate REST principles. As a consequence, developers implement these principles for each application, and often break principles due to a lack of software engineering guidance for their implementation. This is especially true for developing client-side components, such as user agents, on which REST principles are focused on.

In our opinion, software frameworks should provide easy-to-understand development concepts based on formal models of RESTful systems. Without using such models, framework developers have no guidance for developing software engineering abstractions that encapsulate REST principles. Moreover, as noted in [68], separating software engineering practice from the related theory hinders its advancement. Therefore, we believe that the approach for modeling RESTful systems presented in this paper offers many practical guidelines for designing software frameworks for client-side RESTful system development, including Web-based systems. This premise is further supported by the two recently developed software frameworks, Restfulie and RESTAgent, which support several key concepts highlighted in our FSM-based approach. In Restfulie [69], client component development is based on extracting links from representations using application state transitions as the underlying application logic development model. Similarly, RESTAgent [70] defines different link types, such as navigation and embedding, to enable different transformations of application state. Furthermore, several research efforts, such as [71] and [28], are focused on building and formalizing platforms for Web-based systems with the goal of better understanding such systems and their greater modifiability.

The presented FSM-based formalism provides the following insights for designing software frameworks for RESTful development. First, because it is based on finite-state machines, a formalism known to most Web engineers, we believe that our approach provides easy-to-understand and practical guidelines for encapsulating REST principles. Second, we believe that the presented approach, especially the formal definition of key system elements, their interfaces, and their interaction, offers insight for greater decoupling of system development activities. Specifically, we propose that RESTful system development should have three levels of concerns: the framework layer, the architecture layer and the application layer. Figure 12 shows which modules of our FSM-based formalism should be developed in which layer and by which group of developers.

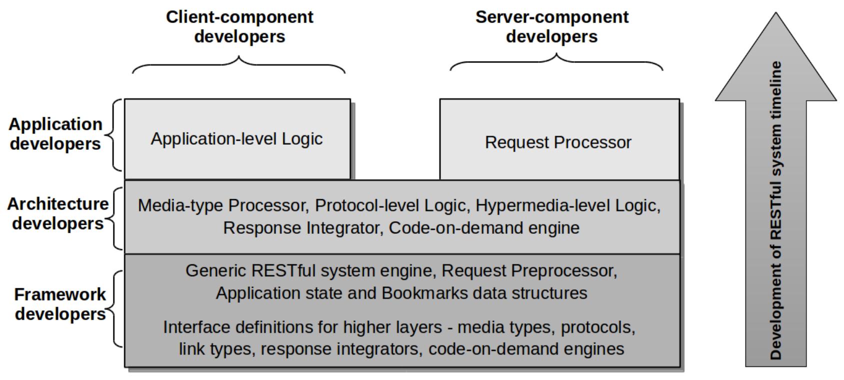
Client-component developers
Server-component developers
Application developers
Application-level Logic
Request Processor
Architecture developers
Media-type Processor, Protocol-level Logic, Hypermedia-level Logic, Response Integrator, Code-on-demand engine
Framework developers
Generic RESTful system engine, Request Preprocessor, Application state and Bookmarks data structures
Interface definitions for higher layers - media types, protocols, link types, response integrators, code-on-demand engines
Development of RESTful system timeline

Fig. 12. Layered separation of concerns in a RESTful framework based on FSM model guidelines

The *framework layer* contains general REST components which are present in every RESTful system and always have the same functionality. Therefore, these components are completely implemented in the framework and are reusable for all architectures and applications. Specifically, framework developers implement the generic RESTful system engine responsible for executing the overall FSM cycle, as described in Section 3.1. Furthermore, this layer provides the Application State and Link Storage modules, for which the required data structure guidelines are also explained by the FSM model, as well as the Request Preprocessor module. Finally, the framework layer defines the necessary interfaces for implementing protocols, media types, link types, state integrators, and code-on-demand engines, in architecture and application layers. Again, the elements of the FSM model, as defined in Section 3.1, provide guidelines for defining these interfaces as they specify the necessary inputs and outputs. For example, a link type definition should be defined with a link type identifier, a request validator function and state integrator function, each of which has a well defined interface.

Second, the *architecture layer* contains implementations of protocols and media types which may be reused for many different applications. Specifically, using interface definitions in the framework layer, architecture developers implement Media Type Processor, Protocol-level Logic, Hypermedia-level Logic, and State Integrator modules, as well as Code-on-demand engines which execute different types of scripts. For example, this layer would contain the implementation of the HTTP and CoAP protocols, HTML and ATOM media types, and the JavaScript code-on-demand engine.

Finally, the *application layer* contains implementations of application-specific logic either for client components or for server components. Specifically, client-side developers implement the Application-level Logic module, while server-side developers implement Request Processors. The presented FSM model suggests the use of rules as an appropriate metaphor for implementing the Application-level Logic, where each rule has access to the application state and may generate the next request.

With the presented layered scheme, framework developers are responsible for understanding and implementing generic principles of REST and providing useful software abstractions for these concepts to the layers above. In other words, this scheme allows that the work of supporting REST principles is divided between application developers, architects and framework developers. Therefore, application developers are not burdened with details of REST or underlying protocols and media types. Furthermore, the implementation of both the framework layer and the architecture layer is combined for both client-based and server-based components, while the implementation of the application layer is divided between the client-side and server-side developers, as expected by developers for each layer.

Finally, we believe that this layered framework scheme promotes decentralized development inherent for distributed hypermedia systems, and their modifiability. Using well defined interfaces for each element, the layered scheme promotes independent development of elements from any layer of the system. For example, it should be easy for an architecture developer to implement support for new protocols, such as CoAP, and publish that implementation on the Web. At the same time it should be equally easy for other architects or application developers to download such an implementation and integrate it into their system.

5 Conclusion and Future Work

The study of architectural styles is an essential part for understanding and improving information systems. As the World Wide Web is the most important global information system, the study of its foundational architectural style, Representational State Transfer (REST), is of equal importance from both a theoretical and a practical perspective. Our analysis of previous research in this field has shown that formal models of RESTful systems are unresearched, consider only some core principles of RESTful systems while ignoring others, and are focused on modeling hypermedia systems in general and not RESTful systems.

In this paper we propose a formalism for modeling RESTful systems based on nondeterministic finite-state machines with epsilon transitions (ε-NFA). We show that ε-NFAs are a natural fit for modeling REST principles which are primarily concerned with exchange of representations using states and transitions. Specifically, the states of the ε-NFA represent the application states which the system may be in at some point of execution, where each application state is represented with the set of resource representations obtained by the user agent. The input symbols of the ε-NFA represent the set of resource manipulation requests while the transition function models the hypermedia links between resources i.e. the request which may be issued at some state. In order to support the temporally varying nature of resource representations, transitions may be nondeterministic, while ε-transitions are used to model client-side execution of code-on-demand scripts. The client-server style of REST is therefore naturally modeled by storing the current ε-NFA state and generating input symbols on client components while the transition function is divided between client and server components. Client components are responsible for transforming input symbols into requests and integrating resource representations into the application state, while server components are responsible for processing requests into responses. Therefore, with respect to the presented model, a system may be called RESTful if it can be represented with an ε-NFA and if sequences of generated input symbols do not lead the ε-NFA into an empty error state.

Furthermore, we show that the presented formalism offers many insights related to the Web engineering practice and open problems in RESTful system development. By presenting models of two real-world Web-based systems, we explore the challenges of developing such models and their practical applications. In particular, we address the state explosion problem inherent to modeling complex systems using state machines, and provide several alternatives for state space minimization through aggregation. Finally, we use the presented formalism to draw guidelines for designing frameworks for RESTful system development.

Our research gives the following insights and areas for future work. First, it would be useful to explore the possibility of extending the presented formalism to explicitly accounts for currently unaddressed principles of RESTful systems: the layered and cacheable principles. Furthermore, we are working on utilizing the FSM formalism for modeling composite and hierarchical RESTful systems. In such system, a client component exposes new resources based on aggregate functionality from locally executing code-on-demand scripts and remote resources. Finally, as explained in the previous section, extending the model to support on-demand machine-to-machine interaction is a promising direction. In essence, server components should be able to on-demand provide client components with an aggregated view of the hypermedia state machine segment that implements the application-domain flow of interest to the user agent, such as buying books on Amazon.

In more practical aspects, we have started with the implementation of a software framework for building RESTful client components, based on the presented formalism and framework design guidelines. Furthermore, as described in the previous section, we are interested in implementing a code generator for model driven development based on FSM specifications, as well as a crawler for creating models of existing systems. We are also working on an implementation of the machine-to-machine interaction model based on on-demand transfer of aggregated state machine segments from server components to client components.

Acknowledgements

The authors acknowledge the support of the Ministry of Science, Education, and Sports of the Republic of Croatia through the *Computing Environments for Ubiquitous Distributed Systems* (036-0362980-1921) research project. Furthermore, the authors thank Sinisa Srbljic, Dejan Skvorc, Miroslav Popovic, Klemo Vladimir, Marin Silic, Jakov Krolo and Zvonimir Pavlic from the School of Electrical Engineering and Computing, University of Zagreb. Finally, the authors thank Silvia Schreier from the Faculty of Mathematics and Computer Science, University of Hagen, for her invaluable comments and support.

References

1. I. Jacobs and N. Walsh (2004), *Architecture of the World Wide Web, Volume One*, W3C Recommendation, WWW consortium (2004) `http://www.w3.org/TR/webarch/`.
2. R.T. Fielding and R.N. Taylor (2002), *Principled design of the modern Web architecture*, ACM Transactions on Internet Technology, Vol.2, No.2, pp. 115-150.
3. L. Dusseault and J. Snell (2010), *PATCH Method for HTTP*, Proposed standard, Internet Engineering Task Force (IETF) `http://tools.ietf.org/html/rfc5789`.
4. F. Rosenberg, F. Curbera, M.J. Duftler and R. Khalaf (2008), *Composing RESTful Services and Collaborative Workflows: A Lightweight Approach*, IEEE Internet computing, Vol.12, No.5, pp. 24-31.
5. R. Alarcon and E. Wilde (2010), *Linking Data from RESTful Services*, 3rd International Workshop on Linked Data on the Web, Raleigh, North Carolina, USA.
6. V. Trifa, D. Guinard, P. Bolliger and S. Wieland (2010), *Design of a Web-based Distributed Location-Aware Infrastructure for Mobile Devices*, 1st IEEE International Workshop on the Web of Things, Mannheim, Germany, pp. 714-719.
7. B. Fitzpatrick, B. Slatkin and M. Atkins (2010), *PubSubHubbub protocol*, `http://pubsubhubbub.googlecode.com/svn/trunk/pubsubhubbub-core-0.3.html`.
8. R.T. Fielding, J. Gettys, J.C. Mogul, H.F. Nielsen, L. Masinter, P.J. Leach and T. Berners-Lee (eds.) (1999), *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, IETF draft standard, `http://www.ietf.org/rfc/rfc2616.txt`.
9. I. Hickson (ed.) (2011), *HTML5 - A vocabulary and associated APIs for HTML and XHTML*, W3C Editor's draft, `http://dev.w3.org/html5/spec/`.
10. L. Dusseault (ed.) (2007), *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*, IETF proposed standard, `http://tools.ietf.org/html/rfc4918`.
11. J. Gregorio and B. de hOra (eds.) (2007), *The Atom Publishing Protocol*, IETF proposed standard, `http://www.ietf.org/rfc/rfc5023.txt`
12. M. Belshe and R. Peon (eds.) (2011), *SPDY Protocol*, `http://mbelshe.github.com/SPDY-Specification/draft-mbelshe-spdy-00.xml`.
13. Z. Shelby, B. Frank and D. Sturek (eds.) (2010), *Constrained Application Protocol (CoAP)*, IETF draft `http://tools.ietf.org/html/draft-shelby-core-coap-01`.
14. R. Sayre (2005), *Atom: The standard in syndication*, IEEE Internet Computing, Vol.9, No.4, pp. 71-78.
15. D. Crockford (ed.) (2006), *The application/json Media Type for JavaScript Object Notation (JSON)*, IETF draft, `http://tools.ietf.org/html/rfc4627`.
16. J.A. Larson (2003), *VoiceXML and the W3C Speech Interface Framework*, IEEE Multimedia, Vol.10, No.4, pp. 91-93.
17. D. Beckett (2004), *RDF/XML Syntax Specification*, W3C Recommendation, `http://www.w3.org/TR/rdf-syntax-grammar/`
18. C. Pautasso and E. Wilde (2010), *RESTful web services: principles, patterns, emerging technologies*, 19th international conference on World wide web (WWW '10), Raleigh, NC, USA, pp.

1359-1360.

19. R.T. Fielding (2002), *waka: A replacement for HTTP*, ApacheCon 2002, `http://gbiv.com/protocols/waka/200211_fielding_apachecon.ppt`
20. N. Koch (2000), *Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process*, Ph.D. dissertation, Ludwig-Maximilians-University of Munich, Germany.
21. http://tech.groups.yahoo.com/group/rest-discuss/message/17571
22. F. Fernandez and J. Navon (2010), *Towards a Practical Model to Facilitate Reasoning about REST Extensions and Reuse*, 1st International Workshop on RESTful Design, Raleigh, North Carolina, pp. 31-38.
23. R.T. Fielding (2010), *ACTION-434: Some notes on organizing discussion on WebApps architecture*, W3C TAG mailing list, `http://lists.w3.org/Archives/Public/www-tag/2010Oct/0100.html`.
24. http://www.w3.org/2001/tag/doc/IdentifyingApplicationState-20110930.html
25. J. Kemp (2010), *AWWW and the Web interaction model*, W3C TAG mailing list, `http://lists.w3.org/Archives/Public/www-tag/2010Jun/0034`.
26. J. Rees (2010), *ACTION-434: Some notes on organizing discussion on WebApps architecture*, W3C TAG mailing list, `http://lists.w3.org/Archives/Public/www-tag/2010Oct/0061.html`.
27. J. Ousterhout and E. Stratmann (2010), *Managing state for Ajax-driven web components*, 2010 USENIX conference on Web application development, Berkeley, CA, USA, pp. 7-7.
28. A. Bohannon and B.C. Pierce (2010), *Featherweight Firefox: formalizing the core of a web browser*, 2010 USENIX conference on Web application development, Berkeley, CA, USA, pp. 11-11.
29. *ISSUE-60: Web Application State Management*, W3C TAG Issues list, `http://www.w3.org/2001/tag/group/track/issues/60`.
30. S. Vinoski (2008), *RESTful Web Services Development Checklist*, IEEE Internet Computing, Vol.12, No.6, pp. 96-95, (2008)
31. C. Pautasso (2009), *RESTful Web service composition with BPEL for REST*, Data & Knowledge Engineering, Vol.68, No.9, pp. 851-866.
32. J. Gregorio (2007), *Do we need WADL?*, `http://bitworking.org/news/193/Do-we-need-WADL`
33. J.E. Hopcroft and J.D. Ullman (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing, Reading Massachusetts.
34. T. Berners-Lee and N. Mendelsohn (2006), *The Rule of Least Power*, W3C TAG Finding, `http://www.w3.org/2001/tag/doc/leastPower.html`.
35. I. Zuzak, I. Budiselic, G. Delac (2011), *Formal Modeling of RESTful Systems Using Finite-State Machines*, 11th International Conference on Web Engineering, Paphos, Cyprus, pp. 346-360.
36. R.N. Taylor, N. Medvidovic and E.M. Dashofy (2009), *Software Architecture: Foundations, Theory, and Practice*, Wiley Publishing.
37. N.R. Mehta (2004), *Composing style-based software architectures from architectural primitives*, Ph.D. dissertation, University of Southern California, California, USA.
38. A.G. Hernandez and M.N. Moreno Garcia (2010), *A Formal Definition of RESTful Semantic Web Services*, 1st International Workshop on RESTful Design, Raleigh, North Carolina, pp. 39-45.
39. G. Decker, A. Luders, H. Overdick, K. Schlichting and M. Weske (2009), *RESTful Petri Net Execution*, Web Services and Formal Methods, Springer-Verlag Berlin, Heidelberg, pp. 73-87.
40. L. Li and W. Chou (2011), *Design and Describe REST API without Violating REST: A Petri Net Based Approach*, International Conference on Web Services, Washington, DC, USA, pp. 508-515.
41. R. Alarcon, E. Wilde, J. Bellido (2010), *Hypermedia-driven RESTful Service Composition*, 6th Workshop on Engineering Service-Oriented Applications, San Francisco, California.
42. M.C. Ferreira de Oliveira, M.A. Santos Turine and P.C. Masiero (2001), *A statechart based model for hypermedia applications*, ACM Transactions on Information Systems, Vol.19, No.1, pp. 28-52.
43. I. Porres and I. Rauf (2011), *Modeling Behavioral RESTful Web Service Interfaces in UML*, 26th Annual ACM Symposium on Applied Computing Track on Service Oriented Architectures and Programming, TaiChung, Taiwan, pp. 1598-1605.

44. M. Laitkorpi, P. Selonen and T. Systa (2009), *Towards a Model-Driven Process for Designing ReSTful Web Services*, International Conference on Web Services, Los Angeles, CA, USA, pp. 173–180.
45. S. Perez, F. Durao, S. Melia, P. Dolog and O. Diaz (2010), *RESTful, Resource-Oriented Architectures: A Model-Driven Approach*, 1st International Symposium on Web Intelligent Systems & Services, Hong Kong, China, pp. 282-294.
46. S. Schreier (2011), *Modeling RESTful applications*, Second International Workshop on RESTful Design, Hyderabad, India, pp. 15-21.
47. S. Charlton (2010), *Building a RESTful Hypermedia Agent, Part 1.*, `http://www.stucharlton.com/blog/archives/2010/03/building-a-restful-hypermedia`.
48. M.H. Alalfi, J.R. Cordy and T.R. Dean (2009), *Modeling methods for web application verification and testing: state of the art.*, Software Testing, Verification & Reliability archive, Vol.19, No.4, pp. 265-296, John Wiley and Sons Ltd. Chichester, UK.
49. J. Dargham and S. Al-Nasrawi (2006), *FSM Behavioral Modeling Approach for Hypermedia Web Applications: FBM-HWA Approach*, Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services, Guadeloupe, French Caribbean, pp. 199-199.
50. P.D. Stotts, R. Furuta and C.R. Cabarrus (1998), *Hyperdocuments as Automata: Verification of Trace-Based Properties by Model Checking*, ACM Transactions on Information Systems, Vol.16, No.1, pp. 1-30.
51. X. Zhao, E. Liu and G.J. Clapworthy (2011), *A Two-Stage RESTful Web Service Composition Method Based on Linear Logic*, 9th European Conference on Web Services, Lugano, Switzerland.
52. B. Kamara, G. von Bochmann, M. Dincturk, G.V. Jourdan, I. Onut (2011), *A Strategy for Efficient Crawling of Rich Internet Applications*, 11th International Conference on Web Engineering, Paphos, Cyprus, pp. 74-89.
53. *Twitter API documentation*, `https://dev.twitter.com/docs`
54. T. Berners-Lee, R. Fielding, L. Masinter (2005), *Uniform Resource Identifier (URI): Generic Syntax*,IETF standard, `http://tools.ietf.org/html/rfc3986`
55. *Internet Assigned Numbers Authority - Registered Media Types*, `http://www.iana.org/assignments/media-types/index.html`.
56. J. Gregorio, R. Fielding, M. Hadley, M. Nottingham and D. Orchard (eds.) (2011), *URI Template*, IETF draft, `http://tools.ietf.org/html/draft-gregorio-uritemplate`.
57. M. Chatti, M. Jarke, Z. Wang and M. Specht (2009), *SMashup Personal Learning Environments*, 2nd Workshop on Mash-Up Personal Learning Environments (MUPPLE'09), Nice, France, pp. 6-14.
58. *GitHub API documentation*, `http://developer.github.com/v3/`.
59. M. Trybulec (2009), *Labelled State Transition Systems*, Formalized mathematics, Vol.17, No.2, pp. 163-171.
60. E. Castellani (2003), *Symmetry and equivalence*, Symmetries in Physics Philosophical Reflections, Cambridge University Press, pp. 425-436.
61. J. Webber, S. Parastatidis and I. Robinson (2010), *REST in Practice: Hypermedia and Systems Architecture*, O'Reilly Media.
62. I. Robinson (2011), *RESTful domain application protocols*, REST: From research to practice, Springer, pp. 61-92.
63. R. France and B. Rumpe (2007), *Model-driven development of complex software: A research roadmap*, Future of Software Engineering, IEEE Computer Society, pp. 37-54.
64. M. Han and C. Hofmeister (2006), *Modeling and verification of adaptive navigation in web applications*, 6th international conference on Web engineering (ICWE '06), Palo Alto, California, USA, pp. 329-336.
65. E. Di Sciascio, F.M. Donini, M. Mongiello and G. Piscitelli (2002) *AnWeb: a system for automatic support to web application verification*, 14th international conference on Software engineering and knowledge engineering, Ischia, Italy, pp. 609-616.

66. F.M. Donini, M. Mongiello, M. Ruta and R. Totaro (2006), *A Model Checking-based Method for Verifying Web Application Design*, Electronic Notes in Theoretical Computer Science, Vol.151, No.2, pp. 19-32.
67. R. Fielding (2008), *REST APIs must be hypertext-driven*, `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`.
68. M. Broy (2011), *Can Practitioners Neglect Theory and Theoreticians Neglect Practice?*, IEEE Computer, Vol.44, No.10, pp. 19-24.
69. S. Parastatidis, J. Webber, G. Silveira and I.S. Robinson (2010), *The role of hypermedia in distributed system development*, 1st International Workshop on RESTful Design, Raleigh, North Carolina, pp. 16-22.
70. *The RESTAgent library*, `http://restagent.codeplex.com/`
71. B. Lerner, B. Burg, H. Venter and W. Schulte (2011), *C3: An Experimental, Extensible, Reconfigurable Platform for HTML-based Applications*, 2nd USENIX Conference on Web Application Development.