Towards Automated RESTful Web Service Composition

Haibo Zhao *and* Prashant Doshi
LSDIS Lab, Dept. of Computer Science
University of Georgia
Athens, GA 30602
{zhao,pdoshi}@cs.uga.edu

Abstract

Emerging as the popular choice for leading Internet companies to expose internal data and resources, RESTful Web services are attracting increasing attention in the industry. While automating WSDL/SOAP based Web service composition has been extensively studied in the research community, automated RESTful Web service composition in the context of service-oriented architecture (SOA), to the best of our knowledge, is less explored. As an early paper addressing this problem, this paper discusses the challenges of composing RESTful Web services and proposes a formal model for describing individual Web services and automating the composition. It demonstrates our approach by applying it to a real-world RESTful Web service composition problem. This paper represents our initial efforts towards the problem of automated RESTful Web service composition. We are hoping that it will draw interests from the research community on Web services, and engage more researchers in this challenge.

1 Introduction

Introduced by Fielding [5, 6], the principles of REpresentational State Transfer (REST) has backed the development of the World Wide Web (WWW). The principles of REST include: (1) Conceptual entities and functionalities are modeled as resources identified by universal resource identifiers (URIs). (2) Resources accessed and manipulated via standardized, well-known HTTP operations (GET, POST, PUT and DELETE). (3) Components of the system communicate via these standard interface operations and exchange the representations of these resources (one resource may have multiple representations). In REST system, servers and clients typically travel through different states of resource representations by following the interlinks between resources.

By applying the principles of REST Web service (WS) development, RESTful WSs [13] are emerging as the choice for many of the leading Internet companies to expose their internal data and functionalities as URI identified resources. In contrast to the operation-centric perspective of WSDL/SOAP WSs, RESTful WSs view the applications from a resource-centric perspective. Some of the advantages of RESTful WSs include:

Light-weight: RESTful WSs directly utilizes HTTP as the invocation protocol which avoids unnecessary XML markups or extra encapsulation for APIs and input/output. The response is the representation of the resource itself, and does not involve any extra encapsulation or envelopes. As a result, RESTful WSs are much easier to develop and consume than WSDL/SOAP WSs, especially in the Web 2.0 context. Additionally, they depends less on vendor software and mechanisms that implements the additional SOAP layer on top of HTTP. RESTful WSs usually deliver better performance due to their The light-weight nature.

Easy-accessibility: URIs used for identifying RESTful WSs can be shared and passed around to any dedicated service clients or common purpose applications for reuse. The URIs and the representation of resources are self-descriptive and thus makes RESTful WSs easily accessible. RESTful WSs have been widely used to build Web 2.0 applications and mashups.

Scalability: The scalability of RESTful WSs comes from its ability to naturally support caching and parallization/partitioning on URIs. The responses of GET (a side-effect free operation) can be cached exactly the same as web pages are currently cached in the proxies, gateways and content delivery networks (CDNs). Additionally, RESTful WSs provide a very simple and effective way to support load balancing based on URI partitioning. Compared to adhoc partitioning of functionalities behind the SOAP interfaces, URI-based partitioning is more generic and flexible, and could be easier to realize.

Declarative: In contrast to imperative services from the perspective of operations, RESTful WSs take a declarative approach and view the applications from the perspec-

tive of resources. Being declarative means that RESTful WSs focus on the description of the resources themselves, rather than describing how the functions are performed. Declarative style brings the fundamental differences between RESTful WSs and WSDL/SOAP WSs to the forefront. While building services for a particular system, the declarative approach focuses on what resources needed to be exposed and how these resources can be represented; imperative approach focuses on what operations needed to be provided and what are the input/ouput of these operations. Declarative approach is considered to be a better choice [14] to build flexible, scalable and loosely-coupled SOA systems.

The characteristics of RESTful WSs mentioned above make automated RESTful Web service composition a *fundamentally different* problem than the composition problem of WSDL/SOAP WSs. Although the research community has put significant effort on automating WSDL/SOAP WSs, automated RESTful Web service composition problem, to the best of our knowledge, is less explored. In this paper, we outline the challenges of this particular problem and present our initial effort towards the problem of automating RESTful Web service composition. Our main contribution in this paper is the introduction of a formal description of the RESTful Web service composition problem. While analyzing the differences and challenges involved in this problem, we propose a formal model for classifying and describing RESTful WSs, and presents a situation calculus [10, 12] based state transition system for composing them automatically.

The rest of the paper is organized as follows. We describe a motivating scenario in Section 2, and this scenario will be used as a running example to explain our modeling method for individual WSs and the proposed composition approach. In Section 3, we introduce a conceptual model for classifying and describing individual RESTful WSs. Section 4 presents in detail the theoretical framework for composing RESTful WSs. We introduce a tailored state transition system and explain the formulation of this system. We will detail how the proposed framework approaches the online shopping scenario mentioned in Section 2. We also provide a brief comparison between the composition of WSDL/SOAP WSs and RESTful WSs in Section 5. Finally, we conclude our work with a discussion of challenges involved in RESTful Web service composition and present our future research directions in Section 6.

2 Scenario: B2C online shopping

In this section, we introduce a simplified online shopping scenario. Registered customers place orders to the system, and one customer may have multiple orders in the system. Orders are not handled until the payment is received. Once the payment is verified, the system processes the order and ships the order to the customer. This system is intended to be implemented using WSs so that it can be used by external business partners or third party Web 2.0 applications. We would like to expose each of the functional components using WSs.

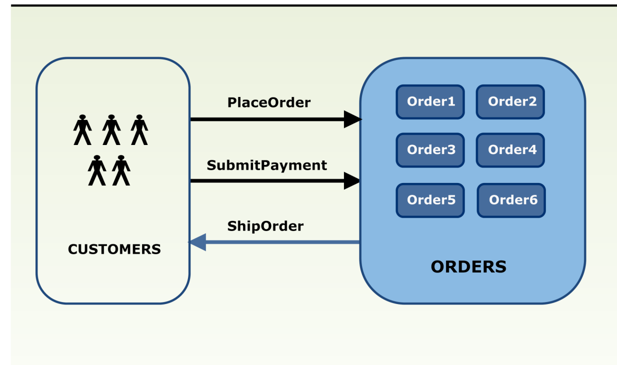

Figure 1. A simplified online shopping scenario

An imperative approach of handling this application would start with functionality decomposition. One solution could be to expose Remote Procedure Call (RPC) style WSs as below:

WSDL/SOAP WSs and Interfaces		
getCustomer	Description:	get customer informat
	Input:	customer id
	Output:	customer information
getOrder	Description:	get order information
	input:	order id
	output:	order information
placeOrder	Description:	place a new order
	Input:	customer id, order
	Output:	success or failure
SubmitPayment	Description:	submit a payment
	Input:	order id, payment
	Output:	success or failure
ShipOrder	Description:	ship the order
	Input:	order id
	Output:	success or failure

Table 1. The list of RPC style WSs

3 RESTful WSs

In this section, we introduce a classification of RESTful WSs, and present a conceptual modeling approach to describe the identified types of WSs. Unlike WSDL/SOAP

based WSs, there is no commonly recognized model or description language available for RESTful WSs. To facilitate automated composition, we present an ontology based formal model for RESTful WSs, this model is at the conceptual level and may be bounded with ontology languages.

3.1 Classification of RESTful WSs

Most resources associated with RESTful services can be directly mapped to domain resources – either a set of resources or individual resources. Besides these two types, we identify a third type of RESTful WSs – these services consume some resources or manipulate related resources, they can not be directly mapped to domain resources or resource collections. We call them *transitional* RESTful WSs. This type of RESTful WSs are less declarative than the other two types, and we should minimize the use of this type of services when we adopt a resource-centric approach to design WSs. But in some cases, we do need transitional RESTful WSs as we show using application scenario mentioned in Section 2.

Type I: Resource Set Service This type of services is mapped to a set of domain resources. In the online shopping scenario, resource related to a set of customers and a set of orders can be both considered of this type. We name them CustomerSet and OrderSet respectively. Type I RESTful Web service may be utilized to capture the concept level resources or the set of instance resources. This type of services support all four HTTP operations(GET, PUT, DELETE and POST).

Type II: Individual Resource Service Individual domain resources can be modeled with this type of services that represent the individual resources in the resource set. For example, in the scenario, individual customer and individual order are mapped to this type. Individual payment and shipment are also considered of this type associated with orders and customers. Type II RESTful Web service may be utilized to capture instance level resources, and it supports three HTTP operations(GET, PUT, DELETE). Operation POST is not applicable here since the URI identified individual resource is already created.

Type III: Transitional Service Although most of RESTful WSs are mapped to the domain resources or resource sets, some of the services are more transition or transformation oriented. The functionality of this type of services is loosely defined as services that consume resources, create resources and update or transform the states of the related resources. For example, SubmitPayment and ShipOrder in the scenario are of this type. When we invoke SubmitPayment with the order information, we create a new resource payment associated with this order, and we update the isPaid property, denoting the payment status, of the order resource from "false" to "true". Similar steps need to be done for the Web service ShipOrder as well. Type III Web service may be utilized to capture transition-oriented functionalities, and it only supports POST operation.

In the next sub-section, we provide a detailed modeling approach to describe these types of WSs identified above.

3.2 Modeling RESTful WSs

By identifying these types of RESTful WSs, we take a resource-centric look at the original scenario. A list of the declarative WSs needed to model the online shopping scenario are presented in Figure 2. In the rest of this section, we propose an ontology based conceptual model to describe these identified RESTful WSs. This model will be used to build our automated composition framework in Section 4.

To expose resources as RESTful WSs for a particular domain, it is intuitive to create the association between Web service resources and domain ontology resources. The idea is that a declarative RESTful approach models each service as a resource, so the description of WSs is essentially the description of the resources and the "state transfer" of these resources. As we classify RESTful WSs into three classes, we explain the specific modeling for them respectively. Generally speaking,

(1) Type I RESTful Web service is a set of ontology instances of the same concept, and the "set" itself could be also considered as a resource as well. While applying GET, DELETE and PUT operations to Type I RESTful Web service, it will fetch, remove and update the representation of this concept resource respectively; Applying POST operation will add into the resource set a new instance resource of this concept.

(2) Type II RESTful Web service is directly mapped to an ontology instance. Type I and Type II RESTful WSs are modeled directly using mapped resources in ontology. Applying GET, DELETE and PUT operations to Type II RESTful Web service will fetch, remove and update the representation of the corresponding instance resource based on the standard semantics of these HTTP operations. POST operation is not applicable for Type II service.

(3) Type III RESTful Web service is transition-oriented, and we describe them as "state transfer" of resources using transition rules. We adopt Semantic Web Rule Language (SWRL) [8] to formally describe these rules. SWRL is a formal language to describe rules based on OWL and RuleML. It has been widely used to describe semantic rules in the ontology context. In our modeling framework, services of Type I and II are mapped to ontology resources; services of Type III are described by the transition rules of ontology resources. Consequently, SWRL becomes an appropriate choice to describe the rules associated with Type III RESTful WSs.

To define the formal semantics of these three types

RESTful WSs		
CustomerSet	TYPE:	Type I
	URI:	http://some.com/Customers
	Supported Operations:	GET, PUT, DELETE, POST
OrderSet	TYPE:	Type I
	URI:	http://some.com/Customers/[Customer_id]/orders
	Supported Operations:	GET, PUT, DELETE, POST
Customer	TYPE:	Type II
	URI:	http://some.com/Customers/[Customer_id]
	Supported Operations:	GET, PUT, DELETE
Order	TYPE:	Type II
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]
	Supported Operations:	GET, PUT, DELETE
Payment	TYPE:	Type II
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/payment
	Supported Operations:	GET, PUT, DELETE
Shipment	TYPE:	Type III
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/shipment
	Supported Operations:	POST
SubmitPayment	TYPE:	Type III
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/submitpayment
	Supported Operations:	POST
ShipOrder	TYPE:	Type III
	URI:	http://some.com/customers/[customer_id]/orders/[order_id]/shipOrder
	Supported Operations:	POST

Figure 2. Identified RESTful WSs

of WSs, we define two new classes: WSRESOURCE, RESTWS. WSRESOURCE describes the resources RESTful WSs represent. WSRESOURCE is either mapped to an individual ontology instance (Type II) or a set of ontology instances (Type I). RESTWS is used to describe the RESTful WSs, RESTWS contains its associated WSRESOURCE. Type I & II RESTful WSs has only one associated WSRESOURCE, Type III RESTful Web service may have multiple associated WSRESOURCEs.

We describe the formal semantics of the universal HTTP operations (GET, POST, PUT and DELETE) on these three types of RESTful WSs. We use *onto* as the name space for the associated domain ontology, and *swrl* as the name space for SWRL.

Modeling Type I Web service Type I Web service is mapped to a set of instances of the same concept in ontology.

Def WSRESOURCE:	
has_name:	onto:wsresource#name
has_description:	onto:wsresource#description
map:	onto:set(onto:resource)

Def RESTWS I:	
has_name:	onto:restws#ws-name
has_description:	onto:restws#description
has_URI:	onto:restws#uri
has_type:	onto:restws#type
has_wsresource:	onto:wsresource
onGET:	onto:opn:Supported
onPUT:	onto:opn:Supported
onDELETE:	onto:opn:Supported
onPOST:	onto:opn:Supported

Let's take the service serving the list of ORDERS as an example. The description of its corresponding WSRESOURCE and RESTWS is as follows:

Example: WSRESOURCE-ORDERSET

has_name: order set
has_description: the wsresource representing order set
map: {onto:instance | isA(onto:instance, onto:ORDER)}

Example: RESTWS-ORDERSET

has_name: orderset-restws
has_description: RESTful WS serving the order set
has_URI: http://some.com/[customer_id]/orderset
has_type: type I
has_wsresource: onto:#WSRESOURCE-ORDERSET
onGET: onto:opn:Supported
onPUT: onto:opn:Supported
onDELETE: onto:opn:Supported
onPOST: onto:opn:Supported

Modeling Type II Web service Similar to defining Type I WSs, we define the WSRESOURCE first in the ontology, then associate the Web service with the defined WSRESOURCE. WSRESOURCE of type II Web service is mapped to a particular ontology instance.

Def RESTWS II:

has_name:	onto:restws#ws-name
has_description:	onto:restws#description
has_URI:	onto:restws#uri
has_type:	onto:restws#type
has_wsresource:	onto:wsresource
onGET:	onto:opn:Supported
onPUT:	onto:opn:Supported
onDELETE:	onto:opn:Supported
onPOST:	onto:opn:NotSupported

Let's take the service providing ORDER information as an example. The description of its corresponding WSRESOURCE and RESTWS are as follows:

Example: WSRESOURCE-ORDER

has_name: order-wsresource
has_description: the wsresource representing [order_id]
map: onto:#[order_id]

Example: RESTWS-ORDER

has_name: order-restws
has_description: Web service serving order infor
has_URI: http://some.com/[customer_id]/[order_id]
has_type: type II
has_wsresource: onto:WSRESOURCE-ORDER
onGET: onto:opn:Supported
onPUT: onto:opn:Supported
onDELETE: onto:opn:Supported
onPOST: onto:opn:NotSupported

Modeling Type III Web service transitional RESTful WSs support only POST operation which causes "state transfer" between resources. In other words, the state of the resources will change themselves based on the request and state of other related resources, guided by certain rules. We use SWRL [8] rules to describe the functionality of this type of RESTful Web service. SubmitPayment and ShipOrder in the scenario(Figure 2) are of this type.

Def RESTWS III:

has_name:	onto:restws#ws-name
has_description:	onto:restws#description
has_URI:	onto:restws#uri
has_type:	onto:restws#type
has_wsresource:	onto:set(onto:wsresource)
onGET:	onto:opn:NotSupported
onPUT:	onto:opn:NotSupported
onDELETE:	onto:opn:NotSupported
onPOST:	swrl:rule

For the instance of the service ShipOrder,

Example: WSRESOURCE-SHIPMENT

has_name: shipment-wsresource
has_description: wsresource [shippment_id]
map: onto:#[shippment_id]

Example: RESTWS-SHIPORDER

has_name: ShipOrder-restws
has_description: shipment service
has_URI: http://.../[shippment_id]/shiporder
has_type: type III
has_wsresource: {onto:WSRESOURCE-ORDER, onto:WSRESOURCE-SHIPMENT}
onGET: onto:opn:NotSupported
onPUT: onto:opn:NotSupported
onDELETE: onto:opn:NotSupported
onPOST:

```
 <ruleml:Imp>
  <ruleml:body rdf:parseType="Collection">
    <swrl:ClassAtom>
      <swrl:classPredicate
                     rdf:resource="#onto:Order"/>
      <swrl:argument1 rdf:resource="#o" />
    </swrl:ClassAtom>
    <swrl:IndividualPropertyAtom>
      <swrl:propertyPredicate
                    rdf:resource="#onto:isPaid"/>
      <swrl:argument1 rdf:resource="#o" />
    </swrl:IndividualPropertyAtom>
  <uleml:body>
  <ruleml:head rdf:parseType="Collection">
    <swrl:IndividualPropertyAtom>
      <swrl:propertyPredicate
                 rdf:resource="#onto:isShipped"/>
      <swrl:argument1 rdf:resource="#o" />
    </swrl:IndividualPropertyAtom>
  </ruleml:head>
</ruleml:Imp>
```

The rule associated with action shipOrder is that: if the input is an order and the order has been paid, the state of the

order should be updated as "isShipped" after this action is completed succesfully.

4 RESTful Web Service Composition

By establishing the model for describing RESTful WSs as ontology resources and "state transfer" of ontology resources, we form a conceptual model which can be used to facilitate automated composition of RESTful WSs. In this section, we present a situation calculus [10, 12] based sate transition system (STS) to automate the composition process of RESTful WSs.

4.1 Situation Calculus based STS

Situation calculus is a first order logic based framework for representing changes and actions, and reasoning about them. It uses *situations* to represent the state of the world, and *fluents* to describe the changes from one situation to the other caused by the actions. We briefly explain the components of situation calculus:

• **Actions** are first order terms, $A(\vec{x})$, each consisting of an action name, A, and its argument(s), $\vec{x}$. In our STS, the actions are the HTTP operations (GET, PUT, POST and DELETE) applied to RESTful WSs. We use service name and HTTP operation (serviceName_operation) to represent the action name, and the argument(s) of the HTTP operations as the argument(s) of the action. We list the possible operations in the table below:

Actions:
TYPE I RESTWS + GET
TYPE I RESTWS + PUT
TYPE I RESTWS + POST
TYPE I RESTWS + DELETE
TYPE II RESTWS + GET
TYPE II RESTWS + PUT
TYPE II RESTWS + DELETE
TYPE III RESTWS+ POST

For example,

(1)The action of getting the order information is denoted as *RESTWS-ORDER_GET()*.

(2)The action of placing a new order is denoted as *RESTWS-ORDERSET_POST(o)*, where o is a variable denoting the resource representation of order.

(3)The action of submitting payment information to an order can be denoted as *RESTWS-SUBMITPAYMENT_POST(p, o)*, where p is a variable denoting the resource representation of payment.

(4)The action of cancel a payment to an order can be denoted as *RESTWS-PAYMENT_DELETE(p, o)*

• A **situation** is a sequence of actions describing the state of the world and usually represented by symbol $do(a, s)$. For example,

(1)$do\ (RESTWS - ORDERSET_POST(o), s_0)$ denotes the situation obtained after performing $RESTWS - ORDERSET_POST(o))$ in the initial situation s_0 (s_0 is a special situation which is not represented using a do function).

(2)$do\ (RESTWS - SHIPORDER_POST(o), do(RESTWS - SUBMITPAYMENT_POST(p, o), do(RESTWS - ORDERSET_POST(o), s_0)))$ represents the situation obtained after performing the action sequence $(RESTWS - ORDERSET_POST(o), RESTWS - SUBMITPAYMENT_POST(p, o), RESTWS - SHIPORDER_POST(o))$ in s_0.

• **Fluents** are situation-dependent relations and functions whose truth values vary from one situation to another. For example, $isPaid(o, s)$ represents if the payment has been submitted to the order denoted by o in situation s. $isShipped(o, s)$ means the order identified in o has been shipped in s. In our formal model,

$$isPaid(o, s) \equiv onto : ORDER(o).isPaid$$
$$isShipped(o, s) \equiv onto : ORDER(o).isShipped$$

• **Action precondition axioms:** For each action we may define one axiom, : $poss(a(\vec{x}, s) \equiv \Pi(\bar{(x)}, s)$, which characterizes the precondition of the action. For example, the precondition axiom of action $RESTWS - SHIPORDER_POST(o)$ is:

$$isPaid(o, s) \Rightarrow$$
$$Poss(RESTWS - SHIPORDER_POST(o), s)$$

where $Poss$ denotes the possibility of performing the action.

• **Successor state axioms** are axioms that describe the effects of actions on fluents. Hence there is one such axiom for each fluent. For example, the successor state axiom for fluent $ReceiveOrder(o)$ is:

$$Poss(a, s) \Rightarrow isPaid(o, do(a, s)) \Leftrightarrow$$
$$a = RESTWS - SUBMITPAYMENT_POST(p) \vee$$
$$(isPaid(o, s) \wedge$$
$$a \neq RESTWS - PAYMENT_DELETE(p, o)$$

In other words, the order is paid in the situation that results from performing the action if and only if we performed the $RESTWS - SUBMITPAYMENT_POST(p)$ action or we already have it in the current situation and do not perform an action $RESTWS - PAYMENT_DELETE(p, o)$ that will delete the payment.

• **Regression:** Regression is a mechanism for proving consequences in situation calculus. It is based on expressing a sentence containing the situation $do(a, s)$ in terms of a sentence containing the action a and the situation s, without the situation $do(a, s)$. The regression of a sentence φ through an action a is φ' that holds prior to a being performed iff φ holds after a. Successor state axioms support regression in a natural way [3]. Suppose that a fluent F's successor state axiom is $F(\vec{x}, do(a, s)) \Leftrightarrow \Phi_F(\vec{x}, a, s)$, we inductively define the regression of a sentence whose situation arguments all have the form $do(a, s)$:

$$Regr(F(\vec{x}, do(a, s))) = \phi_F(\vec{x}, a, s)$$

For other properties of regression, see [3].

$$Regr(F(\neg\psi)) = \neg Regr(\psi)$$
$$Regr(F(\psi_1 \wedge \psi_2)) = Regr(\psi_1) \wedge Regr(\psi_1)$$
$$Regr(F(\exists x\psi)) = (\exists x)Regr(\psi)$$

After representing the initial state s_0 and goal s using situation calculus, the composition problem is converted into a well-formed state transition problem. And the transition problem can be solved with regression mentioned above, more specific details about solving a situation calculus based state transition problem can be seen in [10, 12].

5 Related Work

To the best of our knowledge, there is no previous attempts towards a formal modeling of RESTful WSs in terms of facilitating automated service composition. The following three topics are loosely related to our presented work, although none of these discusses the specific issue of automated RESTful Web service composition.

5.1 RESTful Web service description languages

Although the Web service community is still debating if RESTful WSs really need a formal description language due to its self-declarative nature, we do need a formal, machine-understandable description language to enable automated RESTful Web service composition. Indeed, we might not need a formal description document like WSDL to write a client program consuming RESTful WSs, but if we seek to automate the process of composing RESTful Web service from the pool of vast amount of candidate RESTful WSs, a formal machine-understandable description model is needed.

For RESTful WSs, proposed languages include WADL [7], WSDL2.0 [4] and SA-REST [9]. But these languages are strongly influenced by existing imperative service description languages and do not capture well the resource-centric nature of RESTful WSs. They have focused on the descriptions of input/output as traditional service description languages do, but ignored the description of the resources and the transitions of these resources. As a consequence, these languages remain at the interface description level and are not capable of capturing the "state transfer" between resources. Thus, they can not be directly used to facilitate automated composition of individual RESTful WSs.

5.2 WSDL/SOAP Web service composition

In the past decade, much research effort [11] has been put on automated approaches to WSDL/SOAP Web service composition. Since WSDL/SOAP WSs and RESTful Web service adopt differing styles (imperative against declarative) and view the services from two different perspectives (operation-centric against resource-centric), the automated composition problem of these two kinds of WSs are very different.

WSDL/SOAP Web service composition predominately uses AI planning approaches, and these approaches focus on functional composition of individual WSs. That is, how to compose a new functionality out of existing component functionalities. However, RESTful WSs model the system from the perspective of resources. The composition of RESTful WSs focuses on the resource composition and "state transfer" between candidate WSs.

5.3 Mashup

Mashup is a Web application that combines data from multiple data sources into a single integrated application. A mashup site must access third party data using APIs, and should add value to these data during the integration. Data could come from local databases or various sources across the Internet via different protocols including HTTP, RSS [1], ATOM [2] and RESTful WSs. Compared to RESTful Web service composition, a mashup is restricted at the data-level integration, and most uses of RESTful WSs in mashup are limited to fetching data from remote sources. It usually does not involve updating or manipulating remote data sources or other resources.

6 Discussion

Due to the declarative nature and other characteristics of RESTful WSs such as being light-weight, easily accessible and scalable, we argued that RESTful WSs have some unique advantages over traditional WSs in terms of service composition, especially in the context of building Web 2.0 applications. While RESTful WSs have been widely used in building mashup applications, we believe RESTful WSs will be playing an increasingly important role in the context of SOA, where WSDL/SOAP WSs are dominant.

The RESTful approach represents a very promising way of building WSs. Although it has been considered as an important technology to realize programmable Web in the industry, and potentially adopted as widely as WSDL/SOAP Web service composition, we did not see research effort towards RESTful Web service composition because it is a relatively new technology. In this paper, we discuss two perspectives of modeling a system using WSs. We introduce a formal conceptual model for describing individual RESTful WSs (identified as three types), and present an automated composition framework based on this model. This paper represents our initial efforts towards the problem of automated RESTful Web service composition. We are hoping that it will draw some interests from the research community on WSs, and engage more researchers in this challenge.

We outline below the challenges we encountered towards automated RESTful Web service composition and present some of the future lines of work.

- Resource-centric perspective of building services is relatively new, and most of the claimed RESTful WSs do not fully adhere to REST principles.
- Lack of formal modeling or machine-understandable description languages for RESTful WSs.
- While integrating RESTful WSs (resources) from multiple parties, data heterogeneity may become a major obstacle.

Our current work focuses on the service composition approach and leaves the description of RESTful WSs at a conceptual level. As the future work,

(1) We will seek to ground our conceptual model of describing RESTful WSs with a formal language. Most uses of RESTful WSs in the industry follow an ad-hoc approach by looking at the informal service description document. The informal nature of these documents is a big obstacle to applying automated service compositions. To further realize the potential of RESTful WSs, especially in terms of facilitating automated composition, a formal, machine-understandable description language is needed.

(2) RESTful WSs and WSDL/SOAP WSs have shown advantages and disadvantages in different application situations. It would be interesting to study how we can automate the service composition process in an environment mixed with these two kinds of WSs.

Acknowledgment

This research is supported in part by grant number R01HL087795 from the National Heart, Lung, And Blood Institute. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Heart, Lung, And Blood Institute or the National Institutes of Health.

References

[1] Rss 2.0. http://validator.w3.org/feed/docs/rss2.html, 2002.

[2] Atom 1.0. http://www.atompub.org/2005/07/11/draft-ietf-atompub-format-10.html, 2006.

[3] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order mdps. pages 690–700.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language specification 2.0. http://www.w3.org/TR/wsdl20/, 2007.

[5] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architecture*. PhD thesis, University of California, Irvine, 2000.

[6] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *ICSE*, pages 407–416, 2000.

[7] M. J. Hadley. Web application description language (wadl) specification. https://wadl.dev.java.net, 2006.

[8] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml. http://www.w3.org/Submission/SWRL/, 2004.

[9] J. Lathem, K. Gomadam, and A. P. Sheth. Sa-rest and (s)mashups : Adding semantics to restful services. In *Proceedings of the First IEEE International Conference on Semantic Computing*, pages 469–476, 2007.

[10] J. McCarthy. Situations, actions and causal laws. Technical report, AI Laboratory, Stanford University, 1963.

[11] J. Rao and X. Su. A survey of automated web service composition methods. In *Semantic Web Services and Web Process Composition*, pages 43–54, 2004.

[12] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamic Systems*. MIT Press, 2001.

[13] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2007.

[14] E. Wilde07. Declarative web 2.0. In *Proceedings of the IEEE International Conference on Information Reuse and Integration*, pages 612–617, 2007.