Composing RESTful services with JOpera

Cesare Pautasso

Faculty of Informatics, University of Lugano, Switzerland
c.pautasso@ieee.org
http://www.pautasso.info/

Abstract. The REST architectural style is emerging as an alternative technology platform for the realization of service-oriented architectures. In this paper, we apply the notion of composition to RESTful services and derive a set of language features that are required by composition languages for RESTful services: dynamic late binding, dynamic typing, content-type negotiation, state inspection, and compliance with the uniform interface principle. To show how such requirements can be satisfied by an existing composition language, we include a case-study using the JOpera visual composition language. In it, we present how to build a composite application (DoodleMap) out of some well-known, public and currently existing RESTful service APIs.

1 Introduction

RESTful services [1,2] are currently perceived as a lightweight mean to enable point-to-point integration betwen service providers and a large number of clients. RESTful services are also being more and more used to build so-called *mashups*, applications built by composing multiple Web services and Web data sources into an single, integrated user interface.

Whereas mashups have been positioned as composition done at the user-interface layer [3], the goal of this paper is to apply the notion composition to RESTful services independently of the user interface of the resulting application. Thus, considering the recursive property of software composition[1], we argue that composing a set of RESTful services should result in another RESTful service, which can later be consumed by the user interface of a mashup application, or also be invoked by other composite RESTful services.

To study the problem of composing RESTful services, in this paper we apply the traditional concepts of software composition (i.e., composition languages, component models, and composition techniques). We do so by presenting a concrete application called DoodleMap, built by composing a set of existing and popular RESTful services (e.g., Yahoo! Local, Doodle, and Google Maps) using the JOpera Visual Composition Language [5].

The paper makes the following contributions. We first give a definition of RESTful service composition in terms of the component model and the composition techniques implied by the REST architectural style. From these, we

[1] "A composition of components should itself be composable" [4].

derive a set of requirements to be satisfied by languages for RESTful service composition (support for dynamic binding, content-type negotiation, hyperlink generation, and compliance with the uniform interface principle). To show a practical example of how these requirements can be addressed with a concrete composition language, we present a detailed case study. In it, the latest version of the JOpera visual composition language is used to build a non-trivial, interactive application by means of the composition of existing RESTful service APIs by well-known Web 2.0 service providers [6]. Whereas the resulting application can be considered as a mashup, we design it following a layered approach, where the model of a composite RESTful service is described separately from the user interface of the application, to foster its reusability. Moreover, the user interface of the mashup is decoupled from changes in the component services, which only affect parts of the model of the composite service.

The rest of this paper is structured as follows. We introduce the problem of RESTful service composition in Sect. 2 and derive a set of requirements for composition languages applied to REST in Sect. 3. Section 4 introduces the case study, showing the potential usefulness of applying composition to RESTful services. The implementation with JOpera is described in the following Sect. 5. From it, we discuss a few observations on the need for iterative and interactive composition methodologies in Sect. 6. Related work is presented in Sect. 7, before we conclude the paper in Sect. 8.

2 RESTful Service Composition

Traditional software composition focused on defining languages, techniques, and models for building systems out of reusable software components [4]. As software components evolved into services [7,8], the notion of composition remained as one of the core principles of service-oriented computing [9]. With the emergence [10] of a novel abstraction (the *resource*) as defined by the Representational State Transfer (REST) architectural style [2], it becomes important to explore whether composition remains relevant, and how it can be applied to the design and implementation of application systems made out of – so-called – RESTful services [1]. The resource abstraction introduced by the Representational State Transfer (REST) architectural style [2] poses a number of challenges to existing service composition languages. In this section, while summarizing the main characteristics of REST (refer also to [1,11] for an introduction), we discuss to which extent it can be interpreted as a component model. The characteristics of such component model for RESTful services are then used to enumerate a set of specific requirements that should be taken into account during the design of languages for RESTful service composition.

Composing RESTful services amounts to constructing a new resource out of the state and functionality provided by a set of existing resources (Fig. 1). The state of the composite resource can be simply computed as a projection over the state of the component resources. In the more general case, the composite resource can also maintain its own independent state. This can be used to

cache the state of the components or to augment it with additional information. State transitions of the composite resource can trigger the interaction with its component resources, which can also change state.

Resources published by a RESTful service are exposed using a fine-grained addressing mechanism: the Uniform Resource Identifier (URI [12]). As a consequence, composite RESTful services need to be able to refer to a large and dynamic collection of URIs, identifying their component resources. This collection may change over time since, as we are going to show in the example case study, component resources may be created and deleted during the lifecycle of the composite resource. Also, following the recursive nature of software composition, a composite RESTful service itself may expose a variable number of resources to its clients.

Resources are manipulated using their CRUD-like[2] uniform interface, which provides a fixed set of four predefined actions to enable clients to: 1) initialize the state of a new child resource using `POST`; 2) read the current state of a resource using `GET`; 3) update the state of an existing resource (or initialize it if it does not exist) using `PUT`; 4) delete the state of a resource using `DELETE`.

As opposed to the traditional service invocation mechanism implemented by sending and receiving messages through a bus [13], the uniform interface introduces a novel composition technique. This technique builds upon the synchronous request-response interaction (similar to a remote procedure call) while making explicit some important properties of the interaction. On the one hand, it features explicit support for idempotent and reliable service invocation: `GET`, `PUT`, `DELETE` requests can be retried an arbitrary number of times without side-effects. `GET` is a safe, read-only operation. `PUT`, and `DELETE` are idempotent because they set the resource into a known state: with `PUT` the new state is given by the client, with `DELETE` the state is removed. In case of communication errors, these can be repeated as many times as necessary. Unsafe interactions (which may cause side-effects on the server) are explicitly marked with `POST` and should be dealt

[2] Create Read Update Delete

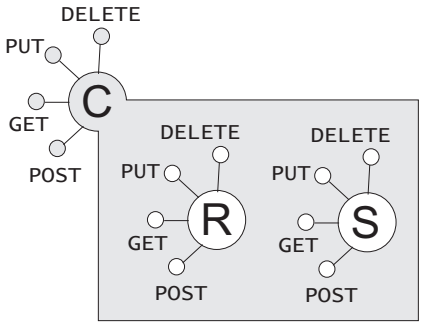


Fig. 1. A composite RESTful service (C) built out of the composition of two existing ones (R and S)

with appropriately. On the other hand, the set of possible actions is limited, well defined, and fixed to the previously described ones. This contributes to establish loose coupling between the composition and its component resources [14]. Providing explicit support for the uniform interface as a composition technique is thus important to enhance the reliability and the loose coupling properties of a composition.

The state of a resource, as it is transferred from and to the client, needs to be serialized using a standardized format. REST does not restrict the resource representation to use a specific format. This resource representation can thus be XML based, but also use other, more lightweight formats (e.g., JSON [15]). The actual format can be negotiated in order to achieve full interoperability between clients and resources without putting too many upfront constraints on the data exchange format understood by both.

REST also prescribes to use *hypermedia as the engine of application state*. In other words, resource relationships can be explicitly rendered as hyperlinks (i.e., pointers to URIs). This way, the representation of the state of a resource retrieved by a client can contain links that can be used to guide the client to interact with other, related resources. For example, the response to a query to a search engine contains a list of links to relevant Web sites. Likewise, a shopping cart resource can contain links that lead to the check out of the cart and let the client complete its purchase by following them.

3 RESTful Composition Language Requirements

The following requirements summarize the challenges for a composition language applied within the "REST component model". These requirements should be considered in addition to the ones (e.g., hierarchical aggregation of compositions, verification and testing of compositions, support for composition evolution and partial upgrade) that are independent of the properties of the actual component model [16]. In addition to the features usually found in composition languages, composition language for RESTful services should explicitly provide:

1. *dynamic late binding*. Resource URIs to be consumed may only become known at run time (for example, by following a hyperlink). Also, URIs may have to be dynamically generated to be sent to clients of the composition.
2. *uniform interface* support. Resource manipulation with `GET`, `PUT`, `DELETE`, and `POST` should be provided as a native composition technique.
3. *dynamic typing*. Resources can have multiple representation, whose type may only become known at run-time. Constraints on the expected set of types could be specified in the composition.
4. *content type negotiation*. Compositions should be able to negotiate the most appropriate representation (both with their clients and with their component services).
5. *state inspection*. Clients should be able to "bookmark" and interact with the state of a composition using the hyperlink URIs it provides them.

4 Example Mashup Case Study

As an example practical application of RESTful service composition, in this section, we present a case study called "DoodleMap". This is built by composing the RESTful service APIs of the Yahoo! Local search service and the Doodle poll service, together with the Google Map widget. Since this composition can be consumed from a Web browser and it includes an interactive user interface composing data and widgets of different sources, we can call it a mashup [17]. A screenshot of the composite application user interface is shown in Fig. 2.

The DoodleMap mashup enhances the Doodle poll service (shown in the bottom frame) to display alternative locations on a map widget. This way, the poll participants may vote after looking at the location of the meeting places (or restaurants, hotels, ski resorts, etc.) as they are positioned on a map displayed above. The most voted location is highlighted on the map. The poll is initialized with the results of a Yahoo! Local search and is closed once a predefined number of participants has voted.

The layered architecture of the mashup is shown in Figure 3. The user interface layer runs in a Web browser. It contains a map widget, which is populated with markers showing the locations of the poll alternatives, placed using the geographic locations returned by the Yahoo! Local search service. It also contains an embedded Doodle poll form, which can be directly used to vote on the preferred locations. A script running in the browser periodically updates the map with the latest results returned from the Doodle poll. Due to the browser same-origin security policies, the script may not retrieve this information from the Doodle API. Instead it has to go through the DoodleMap Poll State Proxy.

The case study illustrates a useful application of RESTful service composition. Data read from one service resource (Yahoo!) is forwarded to create a new resource in a different service (Doodle). The state of the poll is visualized on the user interface widgets and monitored by the mashup, so that the poll can be closed once it reaches a certain number of participants. The mashup itself is published as a RESTful service, with two resources (M and P). M is read from the browser to display a Web page which contains the map and also embeds the Doodle form. P is used to retrieve the current state of the poll and to periodically update the map widget to display the latest poll results.

More in detail, the Web page with the user interface is retrieved with a `GET` request to the DoodleMap mashup service. To create a new DoodleMap poll, the mashup resource accepts also `POST` requests, with the required input information, such as the parameters of the Yahoo! Local search, the poll title, description, author, and the expected number of participants. While servicing this request, the mashup `GET`s the results of a Yahoo! Local search, and uses them to initialize a new Doodle poll (with a `POST` request). The mashup also monitors the state of the poll (with `GET`) so that it can cache it for the user interface and can decide to close the poll (with a `PUT` request) once the number of participants reaches a given number. Whereas the exact sequence of these interactions is not visible from the structural representation of Figure 3, these will become clear as we describe the implementation in the next Section.

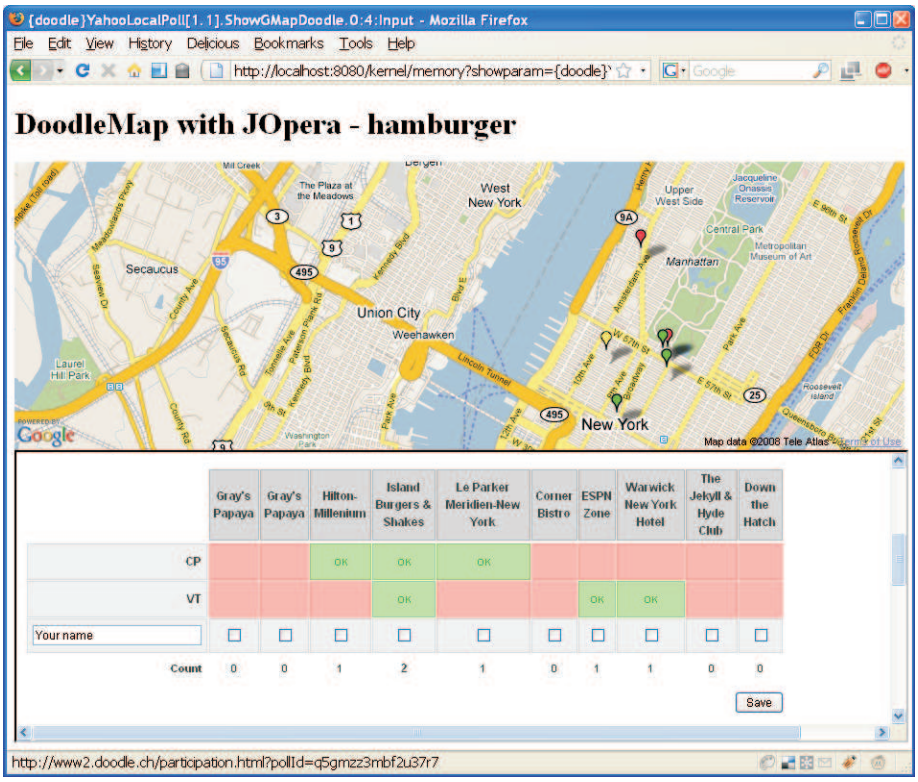
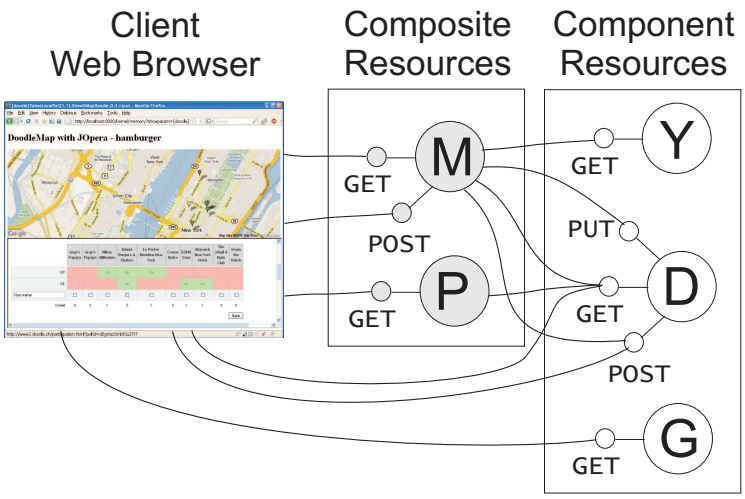


Fig. 2. Screenshot of the DoodleMap example case study

5 JOpera Implementation

The "DoodleMap" mashup has been implemented using the JOpera visual composition language. This section describes the composition code in detail. A discussion on the iterative construction methodology used to produce the composition starting from the available RESTful services can be found in the next section.

The JOpera visual composition language provides a graphical notation to model workflows in terms of control flow dependencies and data flow transfer graphs [5]. Each node of the graph represents tasks (or basic execution steps) and their input and output parameters. Tasks can be dynamically bound to specific service invocation adapters that allow the composition language to be applicable to a variety of composition techniques [18]. In this paper, we focus on the new adapters for invoking external RESTful services using the HTTP protocol, as well as on "glue" adapters to perform local computations used mainly for data transformation. A large collection of adapters (including support for traditional



Legend:

Resource	—	RESTful Service
Y	—	Yahoo! Local Search
G	—	Google Maps API
D	—	Doodle API
M	—	DoodleMap Mashup
P	—	DoodleMap Poll State Proxy

Fig. 3. Layered architecture of the DoodleMap example case study.

WS-* services) is available and more can be easily provided with a plug-in based extensibility mechanism that does not affect the basic composition language [19].

The JOpera for Eclipse rapid composition environment provides an integrated development tool supporting the entire lifecycle of a service composition. It features a design perspective, with tools for managing a library of reusable services, a visual, drag, drop and connect, environment for composing them into workflows. Workflows are compiled to Java bytecode for efficient execution and can be incrementally tested and debugged by executing them in the Monitor perspective. Once workflows are completed, they can be deployed on a remote execution engine to be published as a reusable service (both accessible using REST and WS-* interfaces). JOpera can be freely downloaded from [20].

JOpera provides three views over a service composition model: control flow dependencies, data flow transfers, and service bindings. In the following we describe each of the views in detail.

5.1 Control-Flow Dependencies

JOpera uses an unstructured, graph-based approach to visualize the partial execution order of the tasks of a workflow [21]. Tasks are shown as the nodes of a graph linked by edges representing control flow dependencies. Dependencies

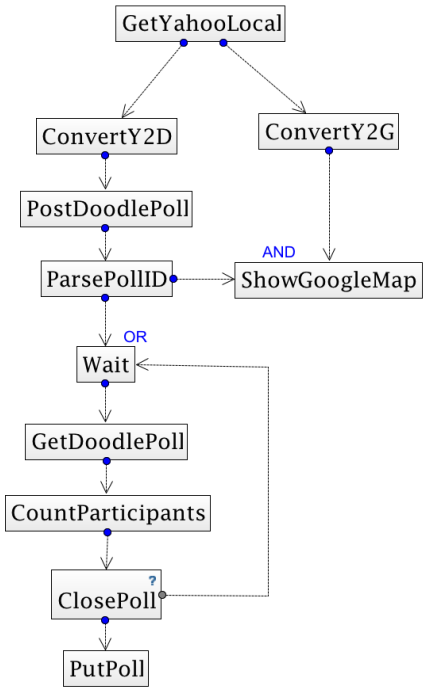


Fig. 4. DoodleMap: Control Flow Dependency Graph

can fire when tasks reach a given execution state. All but one dependency in the example in Fig. 4 are triggered by successful task completion. Only the edge used to close the loop is triggered when the `ClosePoll` task found at the exit of the loop is not executed (when the loop exit condition associated with the task — marked with a `?` icon — is not yet satisfied). Multiple incoming edges on a task indicate a synchronization point in the control flow. The `ShowGoogleMap` uses an `AND` synchronization, as it waits for both predecessors to finish, while the `Wait` task uses an `OR` synchronization in order to be started as the workflow enters the loop as well as when the loop is repeated.

As shown in Fig. 4, the workflow begins with a `GET` request to the Yahoo! Local search service, represented by the `GetYahooLocal` task. Before the results of the search can be used in the mashup they are converted to a format that is suitable to be represented on a Google Map (`ConvertY2G`) and to be used for creating a new Doodle poll (`ConvertY2D`). The two conversion tasks are executed in parallel since there is no control flow dependency between them. Once the results have been converted, the execution continues with a `POST` request on the Doodle API to create a new poll (`PostDoodlePoll`). The headers of the response returned by Doodle are parsed to extract the hyperlink identifying the newly created poll resource, as well as the authorization key to administer it. Once

this information is available, the ShowGoogleMap task is ready to be executed, as all the information required to create the user interface of the mashup is available.

The second part of the workflow is used to monitor the state of the poll and close it once enough participants have responded. This is done with a loop of tasks that, Wait a given amount of time, perform a GET request to retrieve the current state of the poll resource (GetDoodlePoll), and count how many participants have responded (CountParticipants). The loop is repeated if there are not enough participants (this condition triggers the edge from the ClosePoll back to the Wait task). Otherwise, execution continues to the ClosePoll task, which changes the local copy of the state of the poll, and finally ends after the PutDoodlePoll task has transferred the modified state back to the Doodle service.

5.2 Data Flow Transfers

The graph defining how data flows between the various input and output parameters of the workflow tasks is shown in Fig. 5. JOpera provides a separate representation of this view over a composition model due to its complexity. This way, it is possible to visualize the coarse-grained order of execution of tasks separately from their fine-grained data exchanges. The two views are not orthogonal, as a data flow transfer implies a control flow dependency (but not vice-versa). The JOpera editor helps to keep the two views in synch automatically. Concerning the syntax of the data flow graph, tasks are shown with input and output parameters floating around them and linked to the task with incoming (input parameters) and outgoing (output parameters) white-headed edges. Parameters of tasks are shown in white, while parameters of the adapters bound to tasks are shown in grey (and labeled with the SYS prefix). Black-headed edges represent a data transfer operation between output and input parameter of tasks, which is executed as a task is about to be started. As specified in [5], the data flow graph may contain loops.

At the top of Fig. 5 the input parameters of the whole workflow are attached to the shape labeled with DoodleMap. The values of these parameters are set at the beginning of the workflow execution. The composition workflow can be executed multiple times with different input parameters. These define a separate DoodleMap poll, which can have a specific title, description, name of the author, number of alternative location results for a given query topic within a U.S. zip code. Part of these parameters are transferred to build the Yahoo! Local query URI within the GetYahooLocal task. Others are used to initialize the state of the new poll created by the PostDoodlePoll task. The time parameter is used to configure the refresh rate of the user interface (represented by the ShowGoogleMap task) and the same is also used to specify the polling interval of the workflow loop.

The XML results retrieved from the Yahoo! Local service are stored in the SYS.page output parameter of the HTTP adapter bound to the GetYahooLocal task. These results are copied into the SYS.InputXML parameter of the XSLT

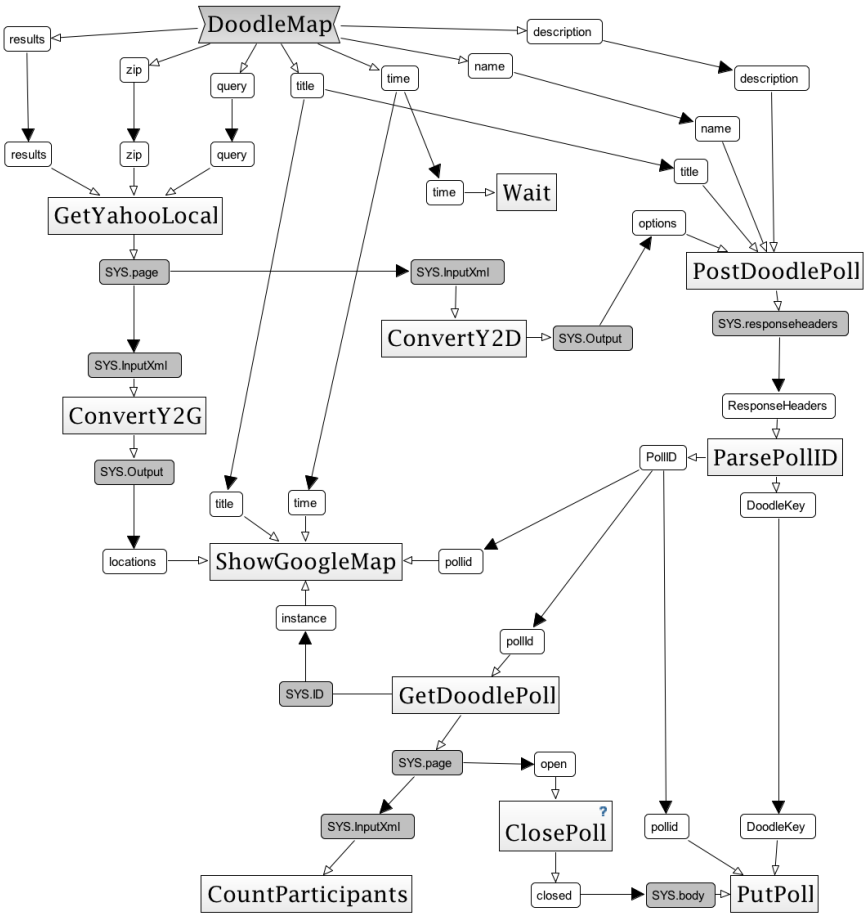


Fig. 5. DoodleMap: Data Flow Transfers Graph

adapter bound to the two conversion tasks. The results of the transformations are stored in the corresponding `SYS.Output` parameters and are now in a format more suitable for initializing the `options` of the new poll and configuring the `locations` of the markers to be displayed on the map.

Once the poll has been created, the Doodle API returns a hyperlink to identify the new poll resource and an access key which needs to be used to administer the poll. These are found resp. in the `Location` and `x-DoodleKey` headers of the HTTP response. All headers are stored in the `SYS.responseheaders` output parameter of the `HTTP` adapter bound to the `PostDoodlePoll` task. The following task `ParsePollID` is responsible for extracting the values of the two response headers and storing them in its `PollID` and `DoodleKey` output parameters. The link to the poll resource is sent to three tasks (`ShowGoogleMap`, `GetDoodlePoll`,

PutPoll), which use it to display the poll in the user interface, retrieve the current state of the poll for monitoring the number of participants, and for updating the state of the poll once it is closed. Only for the latter task, also the access key is required.

The data flow between the tasks that make up the monitoring loop is used to transfer the poll resource XML representation to the CountParticipants task which is bound to the XPATH adapter and uses a simple XPath query to count the number of participants. Also, the same XML representation if passed as input to the ClosePoll task, which toggles the state of the resource from open to closed. The result is passed to the input SYS.body parameter of the HTTP adapter bound to the PutPoll task.

In order to implement the visualization of the current state of the poll on the map, we publish part of the state of the compositions as a resource and generate a hyperlink referring to it. This is then passed to the ShowGoogleMap task, which embeds it into the user interface. Once this is loaded into a Web browser, a script will use the hyperlink to retrieve the necessary data from the workflow published as a resource and update the map widget. More concretely, this is realized by identifying the task that stores the required information (in our case the GetDoodlePoll task, which retrieves the state of the poll resource and stores it in the mashup) and by connecting its SYS.ID identifier property to the task which contains the user interface code. This way, once the workflow is instantiated, the web page produced by the ShowGoogleMap task will contain a link that can be used to retrieve the state of the poll resource cached in the workflow.

5.3 Service Bindings

In the following, we open up a few of the tasks of the composition and present how they are bound to the corresponding service invocation adapter. Depending on the specific composition technique, each adapter defines a set of input (and output) parameters, which need to be configured in order to enable the execution of the task. JOpera provides an open set of predefined adapters that allow tasks to call: Java snippets, local Java methods, local UNIX programs, remote SSH commands, human operators, remote WS-* services, and — as we have anticipated in the previous sections — remote RESTful services through HTTP, and local XPath queries and XSLT transformations.

The new HTTP adapter models the invocation of a RESTful service with four parameters: Method, URI, Body, and the optional request headers (headin). For GET and DELETE requests, the Body is not used. Values for these parameters can be bound at design-time to constant values, but also be dynamically bound at run-time to the input parameters of a task with a variable interpolation mechanism. For example, the URI of the GetYahooLocal poll task is set to the following URI template:

```
http://local.yahooapis.com/LocalSearchService/V2/localSearch?
      appid=X&query=%query%&zip=%zip%&results=%results%
```

Table 1. Service Binding Table for the Doodle RESTful service

Task `GetDoodlePoll` (`HTTP` adapter)
Method parameter: `GET`
URI parameter: `http://www2.doodle.com/api1/polls/%pollId%`

Task `PutDoodlePoll` (`HTTP` adapter)
Method parameter: `PUT`
URI parameter: `http://www2.doodle.com/api1/polls/%pollId%`
headin parameter: `x-DoodleKey:%DoodleKey%`

Task `PostDoodlePoll` (`HTTP` adapter)
Method parameter: `POST`
URI parameter: `http://www2.doodle.com/api1/polls/`
Body parameter:

```
<?xml version="1.0" encoding="UTF-8"?>
<poll xmlns="http://doodle.com/xsd1">
 <type>TEXT</type>
 <extensions/>
 <hidden>false</hidden>
 <levels>2</levels>
 <state>OPEN</state>
 <title>%title%</title>
 <description>%description%</description>
 <initiator><name>%name%</name></initiator>
 <options>%options%</options>
</poll>
```

The placeholder labels found between % sign (e.g., *%zip%*) will be replaced with the actual values of the input parameters of the task, before the HTTP request is performed. A similar approach is used for the other tasks bound to the `HTTP` adapter, as summarized in Table 1. In order to update the state of the poll, the `PutDoodlePoll` task uses the HTTP request header (`headin` parameter) to transfer the authentication key required by the Doodle service. The `PostDoodlePoll` task also requires the `body` parameter to be configured with the payload of the HTTP `POST` request (also shown in the Table). The payload consists of an XML document skeleton into which the task input parameter values are inserted within the corresponding XML elements.

In order to provide the necessary glue between the service invocations, the composition contains a number of tasks dedicated to perform small (and local) computations used to transform the data retrieved from one service so that it can be transferred to the next one. As a composition language, JOpera does not include any native support for performing such computations. Instead, like [22], it provides a variety of adapters so that the most suitable data transformation language can be chosen without polluting the main composition language. In the example, we use Java snippets (in the `ClosePoll` and `ParsePollID` tasks), XSLT transformations (in the `ConvertY2D` and `ConvertY2G` tasks) and one XPath query in the `CountParticipants` task. For completeness, Table 2 also includes

Table 2. Service binding table for some of the glue tasks

Task `ConvertY2D` (`XSLT` adapter)
InputXSLT parameter:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0" xmlns:yl="urn:yahoo:lcl">
<xsl:output method="xml" indent="no"/>
<xsl:template match="//yl:Result">
 <option><xsl:value-of select="yl:Title/text()"/></option>
</xsl:template>
<xsl:template match="text()"></xsl:template>
</xsl:stylesheet>
```

Task `CountParticipants` (`XPATH` adapter)
InputXpath parameter: `count(//ns:participant)`
Namespaces parameter: `ns:http://doodle.com/xsd1`

Task `ClosePoll` (`JAVA.SNIPPET` adapter)
Script parameter:

```
closed = open.replaceAll("<state>OPEN</state>", "<state>CLOSE</state>");
```

the actual code used to implement a representative task bound to each kind of adapter.

6 Discussion

Due to the lack of a standardized (and machine-readable) interface description language, composing RESTful service is far from trivial. Even for a simple mashup application as the one described in the case study, a significant amount of time and effort needs to be devoted to interpreting the human-oriented documentation associated with each service. Also, it is difficult to catch minor errors (e.g., concerning the usage of whitespace in the XML payloads, or the absence of required parameters in a URI) in the configuration of the service invocation adapters at compile-time. Instead, the services need to be carefully tested to understand their behavior and the semantics of their data representation formats. When errors occur, there is very little debugging information available beyond the HTTP 40x and 50x status codes returned by the RESTful service.

To alleviate some of these problems, JOpera provides a composition environment that supports an iterative methodology for composition development. Thanks to its interactive debugging and testing tools, JOpera allows developers to capture and analyze the results of a failed execution within the context of the original composition design. Thus, the effort to feed back the information about a service that has been learned from a failed test into an improved composition is reduced.

To develop the example of the case study, the composition process can start bottom-up by dragging a pair of services (e.g., Yahoo and Doodle) into the data flow view of a new workflow and by connecting directly their parameters. Execution of such composition will fail, as Doodle will reject the data originating from Yahoo!. Additional "glue" tasks can thus be added to solve the problem supplying the missing transformation logic. This needs to be developed top-down [23], using – for example – the data samples collected by JOpera during the execution of the failed workflows to create test cases. After the glue is completed, it can be tested with the original services and the composition can be further extended. The result of this incremental and interactive approach is visible in the structure of the composition: in the data flow graph shown in Fig. 5 tasks bound to invoke RESTful services are interleaved with tasks bound to the local computations used to provide the necessary adaptation.

With respect to the requirements outlined in Sect. 3, we have demonstrated in the case study that JOpera – extended with the functionality of the `HTTP` adapter – provides some degree of support for all of them.

1. *dynamic late binding.* Through the variable interpolation mechanism used to form URI strings passed to the `HTTP` adapter, it is possible to dynamically select a URI and bind it to a task at run time. This has been used in the example both to follow a hyperlink returned by a previous service invocation, as well as to encode parameter values provided by other tasks.
2. *uniform interface.* The `Method` parameter of the `HTTP` adapter complies with the REST uniform interface principle, as it allows to select one among the `GET`, `POST`, `PUT`, and `DELETE` methods used to manipulate a resource.
3. *dynamic typing.* Similar to variables of scripting languages, also data flow parameters of a JOpera workflow can store data of any type. Therefore, they can be used to transfer data of a type that will only become known at run time.
4. *content type negotiation.* The `HTTP` adapter already allows to read any response header and write any request header, thus providing low-level support for content-type negotiation. We plan to make this feature more accessible in a future version of the adapter.
5. *state inspection.* By exposing the state of a running workflow instance as a resource, and by providing a language construct for generating resource identifiers associated with the tasks of a workflow, we have shown that it is possible to provide hyperlinks that enable the interaction of clients with specific parts of the composition. Also, new workflow instances can be started with a `POST` request carrying the values of the workflow input parameters. The corresponding response includes a hyperlink that enables clients to get the results of the workflow once it has completed as well as to access a subset of its internal state while it is still running. The workflow state will be kept until a `DELETE` request arrives.

7 Related Work

The work presented in this paper can be located at the intersection of three research areas in which software composition plays a major role: service composition, mashup development languages and environments, and REST – seen as an emerging alternative service technology platform [24].

The current standard technology for service composition is represented by the Web Services Business Process Execution Language (WS-BPEL [25]). As summarized in Table 3, the statically typed language lacks support for dynamic late binding to a variable set of URIs, it does not support the composition mechanism provided by the uniform interface, nor it supports content type negotation or state inspection. In [26], we have proposed a lightweight extension to the WS-BPEL standard called BPEL for REST to address these limitations. The extension is based on adding a concrete set of activities for invoking RESTful services to the WS-BPEL language so that it can support the missing composition techniques. In this paper we have explored an alternative approach, where the original composition language does not require any extension to be applied to a new component model using a different composition technique. This result validates some of the original claims associated with the JOpera service composition language [5], in particular regarding the generality of its service abstraction [19]. Whereas the language had originally been proposed for composing WS-* services in 2003, this paper presents how the same language can be used to effectively compose RESTful services in 2009.

REST has been described as the right architectural style to enable *serendipitous reuse by means of composition* [27]. The idea of RESTful service composition has also been explored in the Bite project [28], where a simplified version of the BPEL language targeting REST has been proposed. The Bite language however only partially addresses the requirements we have identified in Sect. 3, as it lacks support for content-type negotiation and provides only limited compliance with the uniform interface principle (`PUT` is not supported [29]). In [30], the state transition logic of a RESTful service has been designed using a Petri-net formalism, which could also potentially be used for composition purposes. However, due to the lack for modeling data flow aspects, it is unclear how Petri-nets could be used to implement the case study example presented in this paper. The use of workflow languages for composing RESTful services has also been proposed

Table 3. Service composition languages comparison summary

Requirement	**WS-BPEL**	**BPEL for REST**	**Bite**	**JOpera**
1. *dynamic late binding*	No	Yes	Yes	Yes
2. *uniform interface*	POST only	Yes	Partial	Yes
3. *typing*	Static	Dynamic	Dynamic	Dynamic
4. *content type negotation*	No	Yes	No	Yes
5. *state inspection*	No	Yes	Yes	Yes

in [31], where a tag-based solution to address the state inspection requirement is proposed.

Mashups are a novel kind of Web application which combine data sources and Web services of different providers [17,32]. In the past few years, a number of mashup composition languages and tools have appeared (e.g., Yahoo! Pipes [33], Microsoft Popfly [34], IBM Swashup [35]) targeting a wide community of end-users mashup developers. As we have demonstrated in the case study, the JOpera visual composition language can also be used to build a similar kind of applications. In particular JOpera is focused on the integration logic layer of a mashup, and only provides limited support for building the user interface layer, where multiple widgets should be composed: in the case study, the construction of the user interface was "concentrated" within a single task of the workflow. An example of a complementary composition tool focused at the user interface layer [3] is Mixup [36].

8 Conclusion

As more and more RESTful services become available [6], the Web is shifting from an open medium for publishing data to a programmable platform for building composite applications by means of innovative assembly of existing RESTful services and data sources. To enable the vision of a programmable Web, it is necessary – in addition to the design of composition languages satisfying the requirements presented in this paper – also to design a suitable toolchain for building compositions. JOpera represents an example of such a composition tool, but it is only a first step towards helping developers effectively deal with the dynamic, flexible, stateful and reflective nature of RESTful services.

To give a concrete perspective on the kind of problems that can be encountered while composing RESTful services we have presented the DoodleMap case study, introducing a non-trivial application built out of public and currently widely used RESTful services. The main limitations of the current approach, which will need further research to be fully solved, concern how the abstractions provided by a composition language for RESTful services can be mapped to the properties of the corresponding runtime environment. In particular, it should be possible to enhance the reliability of the composition by taking into account the idempotency associated with some of the uniform interface. This should be done transparently and should not require any additional programming effort as it is currently the case. Also, a more declarative approach to provide support for content type negotiation would raise the level of abstraction of the composition language. Likewise, further work is needed to deal with state managent, security and scalability concerns.

Acknowledgements

The author would like to thank the anonymous reviewers for their invaluable suggestions and also several participants of a recent Dagstuhl seminar on Soft-

ware Service Engineering for their feedback on the concept of RESTful service composition.

References

1. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly (May 2007)
2. Fielding, R.: Architectural Styles and The Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
3. Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R., Casati, F.: Understanding UI integration: A survey of problems, technologies, and opportunities. IEEE Internet Computing **11**(3) (May-June 2007) 59–66
4. Assmann, U.: Invasive Software Composition. Springer (2003)
5. Pautasso, C., Alonso, G.: The JOpera visual composition language. Journal of Visual Languages and Computing (JVLC) **16**(1-2) (2005) 119–152
6. Programmable Web: API Dashboard. (2009) `http://www.programmableweb.com/apis`.
7. Szyperski, C.: Component technology - what, where, and how? In: ICSE '03: Proc. of the 25th International Conference on Software Engineering, Portland, Oregon (2003) 684693
8. Sessions, R.: Fuzzy boundaries: Objects, components, and web services. ACM Queue **2**(9) (December/January 2004-2005)
9. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall (2005)
10. Laskey, K., Hgaret, P.L., Newcomer, E., eds.: Workshop on Web of Services for Enterprise Computing, W3C (February 2007) `http://www.w3.org/2007/01/wos-ec-program.html`.
11. Fielding, R., Taylor, R.N.: Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology **2**(2) (2002) 115–150
12. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): generic syntax. IETF RFC 3986. (January 2005)
13. Chappell, D.: Enterprise Service Bus. O'Reilly (2004)
14. Pautasso, C., Wilde, E.: Why is the web loosely coupled? a multi-faceted metric for service design. In: Proc. of the 18th World Wide Web Conference, Madrid, Spain (April 2009)
15. Crockford, D.: JSON: The fat-free alternative to XML. In: Proc. of XML 2006, Boston, USA (December 2006) `http://www.json.org/fatfree.html`.
16. Nierstrasz, O., Meijler, T.D.: Requirements for a composition language. In: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems. (1994) 147–161
17. Wikipedia: Mashup (web application hybrid). `http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)`
18. Pautasso, C., Alonso, G.: Flexible binding for reusable composition of web services. In: Proc. of the 4th Workshop on Software Composition (SC 2005), Edinburgh, Scotland (April 2005)
19. Pautasso, C., Alonso, G.: From web service composition to megaprogramming. In: Proceedings of the 5th VLDB Workshop on Technologies for E-Services (TES-04), Toronto, Canada (August 2004)

20. Pautasso, C.: JOpera: Process support for more than Web services. http://www.jopera.org.
21. Eshuis, R., Grefen, P.W.P.J., Till, S.: Structured service composition. In: Proc. of the 4th International Conference on Business Process Management (BPM2006). Volume 4102 of LNCS., Vienna, Austria (2006) 97–112
22. Oberleitner, J., Gschwind, T., Jazayeri, M.: The Vienna component framework enabling composition across component models. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon (2003) 25–35
23. Gschwind, T.: Type based adaptation: An adaptation approach for dynamic distributed systems. In: Proc. of the Third International Workshop on Software Engineering and Middleware (SEM 2002), Orlando, FL (May 2002) 130–143
24. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful Web Services vs. Big Web Services: Making the right architectural decision. In: Proc. of the 17th World Wide Web Conference, Beijing, China (April 2008)
25. OASIS: Web Services Business Process Execution Language. (April 2007) http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.
26. Pautasso, C.: BPEL for REST. In: 7th International Conference on Business Process Management (BPM08), Milan, Italy (September 2008)
27. Vinoski, S.: Serendipitous reuse. IEEE Internet Computing **12**(1) (2008) 84–87
28. Rosenberg, F., Curbera, F., Duftler, M.J., Kahalf, R.: Composing RESTful services and collaborative workflows. IEEE Internet Computing **12**(5) (September-October 2008) 24–31
29. Curbera, F., Duftler, M., Khalaf, R., Lovell, D.: Bite: Workflow composition for the web. In: Proc. of the 5th International Conference on Service-Oriented Computing (ICSOC 2007), Vienna, Austria (2007)
30. Decker, G., Luders, A., Schlichting, K., Overdick, H., Weske, M.: RESTful petri net execution. In: 5th International Workshop on Web Services and Formal Methods, Milan, Italy (September 2008)
31. Xu, X., Zhu, L., Liu, Y., Staples, M.: Resource-oriented architecture for business processes. In: Proc of the 15th Asia-Pacific Software Engineering Conference (APSEC2008). (December 2008)
32. Descy, D.E.: Mashups..with or without potatoes... TechTrends **51**(2) (December 2007) 4–5
33. Trevor, J.: Doing the mobile mash. Computer **41**(2) (February 2008) 104–106
34. Microsoft: Popfly. http://www.popfly.ms/.
35. Maximilien, E.M., Wilkinson, H., Desai, N., Tai, S.: A domain-specific language for Web APIs and services mashups. In: Proc. of the 5th InternationaL Conference on Service-Oriented Computing (ICSOC 2007), Vienna, Austria (September 2007) 13–26
36. Yu, J., Benatallah, B., Casati, F., Daniel, F., Matera, M., Saint-Paul, R.: Mixup: A Development and Runtime Environment for Integration at the Presentation Layer. In: 7th International Conference on Web Engineering (ICWE07). Volume 4607 of LNCS., Como, Italy, Springer (2007) 479