Data & Knowledge Engineering 68 (2009) 851–866

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

RESTful Web service composition with BPEL for REST

Cesare Pautasso

Faculty of Informatics, University of Lugano, via Buffi 13, 6900 Lugano, Switzerland

ARTICLE INFO

Article history:
Available online 10 March 2009

Keywords:
Web service composition
RESTful Web service
BPEL extension

ABSTRACT

Current Web service technology is evolving towards a simpler approach to define Web service APIs that challenges the assumptions made by existing languages for Web service composition. RESTful Web services introduce a new kind of abstraction, the resource, which does not fit well with the message-oriented paradigm of the Web service description language (WSDL). RESTful Web services are thus hard to compose using the Business Process Execution Language (WS-BPEL), due to its tight coupling to WSDL. The goal of the BPEL for REST extensions presented in this paper is twofold. First, we aim to enable the composition of both RESTful Web services and traditional Web services from within the same process-oriented service composition language. Second, we show how to publish a BPEL process as a RESTful Web service, by exposing selected parts of its execution state using the REST interaction primitives. We include a detailed example on how BPEL for REST can be applied to orchestrate a RESTful e-Commerce scenario and discuss how the proposed extensions affect the architecture of a process execution engine.

1. Introduction

The Business Process Execution Language (WS-BPEL [1]) is the current standard language for Web service composition. Its design is centered around the notion of business process used as glue between interacting services. Following the recursive nature of software composition [2], services are composed into processes, which themselves can be consumed as services. From a syntactical perspective, the service abstraction assumed by BPEL is the one provided by the Web service description language (WSDL). WSDL-based services expose a set of operations using two interaction patterns: synchronous invocation through remote procedure calls, and asynchronous interactions via messaging. These two patterns are directly reflected in the composition mechanisms supported by the BPEL language.

Recently, a different kind of service abstraction (based on the REpresentational State Transfer – REST – architectural style [3]) has appeared, challenging the assumptions made by the BPEL language. This emerging technology involves a rediscovery of the original design principles of the World Wide Web [4] to provide a new abstraction for publishing information and giving remote access to application systems: the resource. This abstraction provides the foundation for so-called RESTful Web services [5]. It can be directly mapped to the interaction primitives found in the HTTP standard protocol, making it easy to publish existing Web applications as services by – as a first approximation – replacing HTML pages with data payloads formatted in "plain old" XML (POX) [6].

Since a very large number of service providers are switching to REST [7,8] in order to make it easier for clients to consume their Web service APIs and thus grow a larger user community, it becomes important to study how this wealth of new services [9] can be reused by means of composition. The composition of RESTful Web services is also often associated with so-called Web 2.0 Mashups [10,11]. Emerging Web 2.0 APIs have gained a large following, since they use REST to manage the complexity and reduce the effort of scraping data out of plain HTML Web pages [12]. They have become popular with the

E-mail addresses: cesare.pautasso@unisi.ch, c.pautasso@ieee.org

0169-023X/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2009.02.016

mashup building community, where these services and data sources are combined and reused in novel and unexpected ways [13].

Since most RESTful Web services are not described using the standard Web service description language (WSDL), it is not possible to reuse existing languages and tools that require the presence of WSDL interface contracts. A consensus still needs to be reached in terms of how to describe the interface of such RESTful Web services. Many specific description languages for RESTful Web services – e.g., the Web Application Description Language (WADL [14]), the Web Resource Description Language (WRDL [15]), Norm's Service Description Language (NSDL [16]), or simply the Web Description Language (WDL [17]), and others – have been proposed in the past few years. As we are going to show, it is also possible to use the new HTTP binding of the latest version of the Web service description language (WSDL 2.0). In practice, most existing APIs still rely on human-oriented documentation, which includes interactive examples to help developers learn how to use a particular service. For resources represented using XML, a schema definition is often referenced from the documentation.

This unclear situation makes it challenging to define a composition language for REST as it is currently not yet possible to assume that a particular service description language will gain widespread acceptance. Thus, in this paper we do not rely upon the presence of a specific description language. Similar to the BPEL simplification presented in [18], our approach is independent from the chosen service description language. We propose BPEL extensions that directly map to the resource abstraction and natively support the corresponding interaction mechanisms and invocation patterns [19].

In this paper, we assess how this new RESTful Web service abstraction impacts the assumptions made during the design of the BPEL language. For example, REST requires clients to interact with many resources identified by URIs [20]. BPEL instead assumes to interact with services bound to a few fixed communication endpoints. Thus, it does not handle well the variable set of URIs that make up the interface of a RESTful Web service. From this and many other limitations, in this paper we identify the need for research on novel composition languages better tailored to support the specific properties and constraints of the REST architectural style [21,22].

Considering the differences between the two styles [23,24], another contribution of this paper is to show that process-based composition languages can be applied to compose RESTful Web services in addition to "traditional" WSDL-based ones. This is important, as we claim that native support for composing RESTful Web services has become a requirement that modern service composition languages can no longer ignore. Still, it is also important to keep the existing support for composing WSDL-based services. The BPEL for REST extensions we introduce enable processes to natively invoke RESTful Web service APIs and to publish a view over their execution state through a RESTful Web service interface. As sketched in Fig. 1, the goal is to enable BPEL processes to publish resources (P) created out of the composition of existing ones (R and S).

The rest of this paper is structured as follows: The motivation for our work is presented in Section 2. We continue in Section 3 giving some background on RESTful Web services and outlining the challenges involved in composing this novel kind of services. In Section 4, we introduce the extensions to the BPEL language. The design of a BPEL for REST engine reference architecture is outlined in Section 5. An example of how to apply BPEL for REST to an electronic commerce scenario is described in Section 6 (the corresponding code is listed in the Appendix). Section 7 discusses the relationship between the resource and the process abstractions more in depth. Related work is presented in Section 8, before the paper is concluded in Section 9.

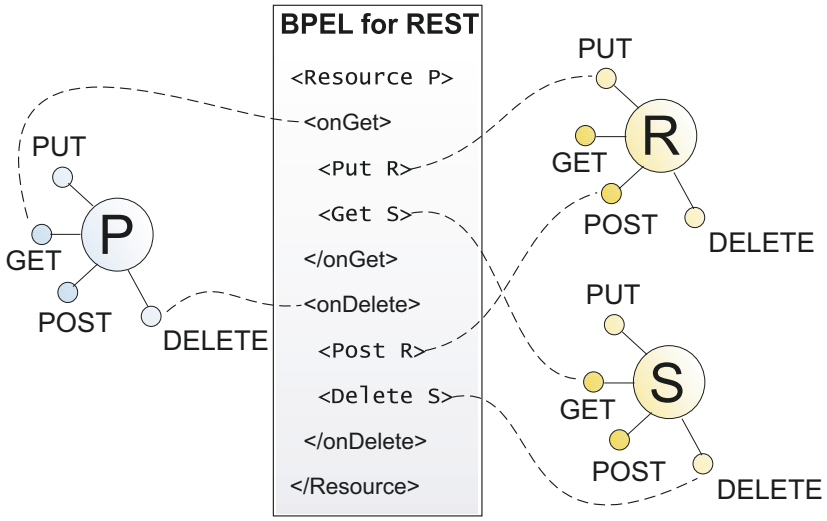


Fig. 1. Composing a RESTful Web service out of two existing ones using BPEL for REST.

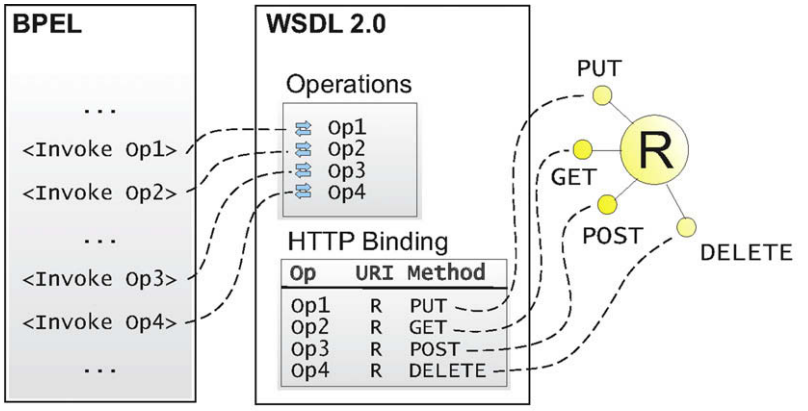
2. Motivation

A first motivation for the research presented in this paper can be found reading the specification of the WS-BPEL 2.0 standard, clearly stating one of the most important design decisions of the language [1, Section 3]:

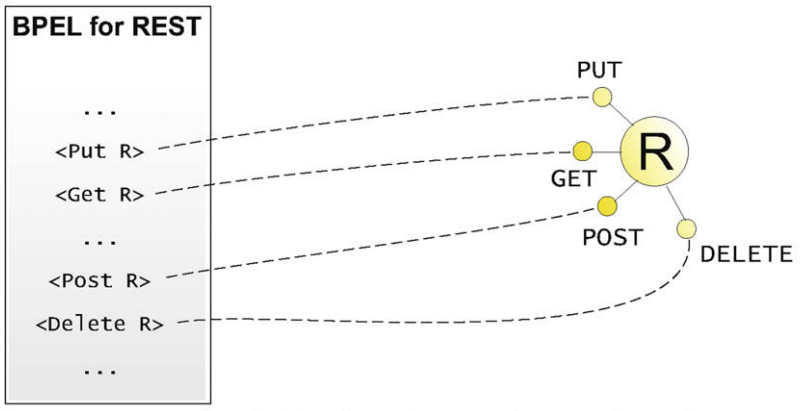
> The WS-BPEL process model is layered on top of the service model defined by WSDL 1.1. [...] Both the process and its partners are exposed as WSDL services.

This choice introduces a tight coupling between the BPEL and the WSDL languages. More in detail, BPEL uses the *partner link type* construct to tie together pairs of matching WSDL *port types*. Thus, all incoming and outgoing interactions of a process with external services must go through a WSDL interface. Whereas this originally was seen as a promising approach to effectively deal with the heterogeneity of the services to be composed, over time this constraint has spawned a set of extensions to the BPEL language. For example, BPEL-SPE [25] for processes invoking other sub-processes, BPEL4People [26] for interaction with human operators, or – more recently – BPEL-DT [27] for data intensive applications; BPEL4Chor [28] for modeling choreographies and their participants; BPEL4JOB [29] with fault handling extensions and the added support for job submission to better deal with the requirements of scientific workflow applications. Also in the Grid workflows area, [30] proposes three new BPEL activities for invoking stateful WSRF-based Grid services. Finally, BPEL^{light} [18] showed how to completely decouple the two languages in the context of message-oriented interactions. Along the same direction, in this paper we propose yet another extension to make BPEL natively support the composition of RESTful Web services.

Existing efforts towards enabling the composition of RESTful Web services with plain BPEL without extensions revolve around the new WSDL 2.0 HTTP binding. To argue against this approach, we could use the argument that WS-BPEL 2.0 does not (yet) support WSDL 2.0, but instead it has been designed targeting its predecessor WSDL 1.1. Ignoring the differences between WSDL 1.1 and 2.0, Fig. 2a shows that it would be possible to wrap a RESTful Web service behind a WSDL document using the newly introduced HTTP binding [31]. This binding enables the BPEL process to send and receive data over the HTTP protocol without using the SOAP message format. Thus, it would appear that the problem of composing RESTful Web services



(a) Wrapping RESTful Web services through the WSDL 2.0 HTTP Binding



(b) Direct invocation of RESTful Web services using the BPEL for REST extensions

Fig. 2. Comparison of two alternative solutions for using BPEL to compose RESTful Web services.

with BPEL has already been solved, at least concerning the support for their invocation. Nevertheless, this solution is not satisfactory.

From a theoretical perspective, this approach hides the resource abstraction and the corresponding RESTful interaction primitives behind a service-oriented abstraction. As we are going to discuss, the composition mechanisms (i.e., synchronous and asynchronous message exchange) provided by WSDL do not match the semantics of a "GET", "PUT", "POST", or "DELETE" request performed on a resource URI [23,32–34]. Thus, we claim that explicitly controlling the RESTful interaction primitives used to invoke a service and native support for publishing the state of BPEL processes as resources from a process would be beneficial (Fig. 2b).

In practice, WSDL 2.0 is not yet widely deployed, especially to describe existing RESTful Web services, and there is very little evidence that this will change in the future. Thus, the burden of recreating a WSDL description wrapping the RESTful Web service is shifted from the service provider to the BPEL developer [35]. Whenever the underlying RESTful Web service is updated, all clients must update their copy of the corresponding WSDL document. This contradicts existing best practices suggesting that the contract describing a service interface should be maintained by the service provider and not by the consumers of a service.

3. Composing RESTful Web services

In this section, we present the design principles and constraints of the REST architectural style [3] and introduce the notion of composite RESTful Web service. For each principle, we discuss how it challenges the standard BPEL composition language. To address these challenges we have designed the language extensions presented in the rest of the paper. More information on REST and Resource-oriented architectures can be found in [4,5,32,36,37,23].

Resource addressing through URI – The interface of a RESTful Web service consists of a set of resources, identified by URIs. URIs give a unique address identifier to make specific resources globally accessible [20]. To invoke a RESTful Web service clients need to be able to interact with a variable set of URIs, which are not always known in advance and may require some form of dynamic discovery. Thus, late binding plays a major role in RESTful Web service composition and BPEL for REST should support the dynamic binding to URI addresses that become known only at run-time. Providers of RESTful Web services publish a set of resources, manage their lifecycle, and control the evolution of their state. Thus, BPEL for REST should support the specification of how the state of arbitrary resources may evolve so that processes can be used to implement the state transition logic of composite resources [38].

Uniform interface – Once a resource has been identified, the set of operations that can be applied to it is fixed by design to the same four methods (or verbs): PUT, GET, POST, and DELETE. These CRUD-like operations apply to all resources with similar semantics, even if in some cases not every operation can be meaningful. Each of these methods is invoked on a resource using a synchronous HTTP request–response interaction round.

GET requests are used to retrieve a representation of the current state of a resource. These read-only requests are by definition idempotent, as they should not modify the state of the corresponding resource and can be safely repeated an arbitrary number of times. On the one hand, the results of such requests can be cached. On the other hand, clients should always be able to perform a GET request without prior knowledge about the semantics of the resource. As a composition mechanism, this kind of operation is useful to aggregate information from multiple sources. A GET performed on a composite service reads its current state, which in turn can be computed by getting and combining the state of its component resources.

PUT requests are used to transfer a new state onto a resource. If the resource identified by the URI of the PUT request does not previously exist, it will be initialized with the given state. Otherwise, the resource state is simply updated to the new one. Also PUT requests are idempotent, since they can be replayed with the same payload an arbitrary number of times without affecting the state of the resource. This write operation on a composite resource can be used to trigger state transitions, where clients set the resource to its next state. Composite services can route such request to update the state of one or more of their component resources.

DELETE requests are used to delete an existing resource and invalidate the corresponding URI. After the request completes, the service provider will no longer be able to successfully respond to GET requests to the URI of the deleted resource. Also DELETE requests are idempotent, because after a resource has been deleted once, repeating the request should not have any other side-effect on the service. Deleting a composite service can be interpreted as a cancellation request, that can be used to interrupt a partially executed process and undo its effects on its component resources.

POST requests are used for the creation of *subordinate* resources. As we will show in the example section, a POST request on the URI `/order` can be interpreted as the request to create a new order resource, which should be identified with a URI such as `/order/42`. Analogous to PUT, this introduces an apparent redundancy regarding resource creation. However, POST lets the service provider decide how to identify the newly created resource, while with PUT requests the client is responsible for choosing a unique identifier for the new resource. This mechanism is also applicable to the instantiation of new composite resources. The notion of resource and its subordinates fits very well the relationship between a process template and its corresponding instances. Thus, in case of BPEL for REST it can be used

as a standard instantiation mechanism for new processes. It can also help to simplify the correlation of client requests with the corresponding process instances, as it is possible to assign a unique URI to identify each instance of a process.POST requests can also be *overloaded* and used for expressing arbitrary interaction semantics with a resource, when the previous three methods do not fit [5, p. 101]. In general, POST is the only kind of requests that produces side effects on the service. Thus, only in this case, results cannot be cached and it is not safe to replay POST requests. This catch-all role opens up the possibility of using POST as a generic interaction mechanism to expose arbitrary functionality provided by the composite resource.

Self-descriptive messages – Services provide multiple representations for a given resource so that clients can negotiate to use the most suitable format and encoding of the information. This valuable flexibility makes it difficult to statically determine the actual data type of a given HTTP response payload. In some cases, also the assumption of dealing with XML data [39] may not hold when accessing resources represented in other, more lightweight, formats such as the JavaScript Object Notation (JSON [40]). Thus, the assumption that BPEL variables may only contain SOAP message payloads or, in general, XML data, may no longer apply when using them to store resource representations. Also, it should be possible to serve the state of a composite resource – implemented using a BPEL process – represented using alternative formats, adapted to the context of each client.In addition to content-type negotiation, REST uses meta-data to control many properties of the interaction, such as client and server authentication, access control, data compression, and caching. Thus, it should not only be possible to specify from a BPEL process the method and the URI of an HTTP request, but also to control the meta-data associated with it. Programmatic access to the HTTP request and response headers is therefore a requirement for the BPEL for REST extensions.

Hypermedia as the engine of application state – Whereas every interaction is kept stateless using self-contained request and response messages, stateful interactions are based on the concept of explicit state transfer. Valid future states of the interaction can be embedded in the representation of a resource as hyperlinks. These are dynamically discovered and followed interactively by clients that are guided by the RESTful Web service along a correct interaction path. From this principle, it follows that resource URIs may be dynamically generated by a service and embedded into resource representations. Thus, BPEL for REST should provide a mechanism for processes to generate new resource URIs and to send these back to their clients. Also, the language should enable processes to extract URIs from response messages and provide constructs for performing the late binding of invocation activities to dynamically discovered target resource URIs.

4. BPEL for REST extensions

The extensions to BPEL for composing RESTful Web services are of three different kinds. First, BPEL processes should perform a direct invocation of RESTful Web service. Second, a view of the execution state of a BPEL process should be published as a resource. Third, the execution semantics of some BPEL constructs needs to be revised to fit with the REST design principles. To do so, in this section we introduce a set of invocation activities, request handlers, and constructs that are directly related to the REST design principles.

4.1. Invoking RESTful Web services

To invoke a RESTful Web service from a BPEL process, we propose to add the following four activities: <get>, <post>, <put>, <delete>.

As summarized in Fig. 3, the four activities use the `uri` attribute to specify the address of the targeted resource. The URI can be a constant value, but also be computed out of data stored in the process variables. This way, parameter values can be interpolated into the URI for GET requests. Additionally, a variable may store the complete target URI to be invoked. With this simple extension, BPEL for REST supports dynamic late binding to invoke resource URIs that are only known at runtime. The only constraint on the URI is that it should address a resource using the HTTP or the HTTPS schemes.

Following the convention of the existing BPEL <invoke> activity, the data for the request and response payloads is stored in variables that are referenced from the corresponding `request` and `response` attributes. In case of <get> and <delete> activities, there is no request payload as these REST primitives operate on the resource URI only. For <put> and <delete> the response attribute is optional, as some services may return an empty payload when invoked using these two methods. Only the <post> activity requires both request and response attributes to be set.

The meta-data sent with the HTTP request headers can be specified using the <header> child elements of each of the four invocation activities. Also in this case, header values can be set to constant values but also computed from information stored in BPEL variables. The headers found in the response are discarded unless a variable for storing them is specified in the optional `response_headers` attribute.

Similar to standard BPEL <invoke> activities, also <get>, <post>, <put>, and <delete> are equipped to deal with invocation failures. In particular, if an HTTP status code indicating an error (i.e., 4xx or 5xx) is detected in the response, the activity will fail and raise the corresponding BPEL `fault` that can be caught by a standard fault handler. As shown in Fig. 3, fault handlers in BPEL for REST can be associated with specific HTTP status codes. Unless a specific fault handler is specified, all

```
<get uri="" response="" response_headers=""?>
  <header name="">*value</header>
  <catch code="">*...</catch>
  <catchAll>?...</catchAll>
</get>

<post uri="" request="" response="" response_headers=""?>
...
</post>

<put uri="" request="" response=""? response_headers=""?>
...
</put>

<delete uri="" response=""? response_headers=""?>
...
</delete>
```

Fig. 3. BPEL for REST extensions to invoke a RESTful Web service.

other HTTP codes (e.g., like 3xx used to indicate a redirection, or 2xx indicating a successful interaction) are transparently managed by the BPEL engine and do not raise a fault. However, if necessary they may also be handled using the `catch` construct. If a variable to store the response headers is specified in the `response_headers` attribute, it will also be used to store the status code of the response, listed among the header fields with the name `Status-Code`. This way the outcome of an invocation can be preserved for later use in the process without necessarily having to deal with it immediately using a fault handler.

4.2. Publishing processes as RESTful Web services

To declaratively publish resources from a BPEL process we introduce the <resource> container element (Fig. 4). This construct allows processes to dynamically publish resources to clients depending on whether their declarations are reached by the execution of the BPEL process. Once the control flow of a process reaches the <resource> element, the corresponding URI is published and clients may start issuing requests to it. Once execution leaves the `scope` where the <resource> is declared, its URI is no longer visible to clients that instead receive an HTTP code 404 (Not found). Resources that are declared as top-most elements in a BPEL process never go out of scope. They become visible to clients as soon as the BPEL process is deployed for execution. Resource declarations can be nested. The URI of nested resources is computed by concatenating their `uri` attribute with the usual path (`/`) separator.

Like the BPEL <scope>, a <resource> may contain a set of <variable> declarations that make up the *state* of the resource found at a given `uri`. These state variables are only accessible from the BPEL code found within the resource declaration.

Similar to the BPEL <pick> construct, the <resource> contains a set of request handlers that are triggered when the process receives the corresponding HTTP request. As opposed to <pick>, which contains one or more <onMessage>/<onAlarm> handlers, the request handlers found within a <resource> directly stem from the REST uniform interface principle. They are: <onGet>, <onPost>, <onPut>, and <onDelete>. If a request handler for a given verb is not declared, requests to the

```
<resource uri="">
 <variable>*
 <onGet>?                          ... </onGet>
 <onPut>?                          ... </onPut>
 <onDelete>?                       ... </onDelete>
 <onPost isolated="false"?>?  ... </onPost>
</resource>

<respond code=""?>
 <header name="">*value</header>
  payload
</respond>
```

Fig. 4. BPEL for REST extensions to declare resources within a process.

```
<extensions>
  <extension namespace="http://jopera.org/bpel/rest/2008"
             mustUnderstand="yes" />
</extensions>
```

Fig. 5. BPEL for REST extensions namespace declaration.

resource using such verb should be implicitly answered by the BPEL engine with HTTP status code 405 (Method not allowed). At least one request handler must be included in a resource declaration and a handler for a given method may appear at most once.

Another difference with <pick> is that there is no limit on the number of times one such handler may be concurrently activated during the lifetime of the resource it is attached to. For example, when multiple clients of a BPEL process issue in parallel a GET request on a resource declared from within the process, the execution of the corresponding onGet request handler will not be serialized and should be repeated for each request. To ensure that GET requests on a resource are indeed *safe*, the onGet request handler has read-only access to the state variables of the corresponding resource. As we have previously discussed, only POST requests are not meant to be *idempotent*. Only the <onPost> handler may therefore be flagged using the optional isolated boolean attribute.[1] When the attribute is set to true, proper isolation with respect to concurrent access to the resource state variables should be guaranteed.

The behavior of the process within a request handler can be specified using the normal BPEL structured activities (i.e., <sequence>, <if>, <flow>, <while>, etc.). However, control-flow links across activities that belong to different handlers are not supported.

To access the data sent by clients with the request payload, a pre-defined variable called $request is available from within the scope of the request handler. Likewise, a variable called $request_headers gives read-only access to the HTTP request headers.

Results can be sent back to clients using the BPEL for REST <respond> activity. Its code attribute is used to control the HTTP response status code that is sent to clients to indicate the success or the failure of the request handler. The response headers can be set using the same header construct introduced for the previously described RESTful invocation extensions. The payload of the response is embedded within the body of the element, but could also be precomputed in a variable (i.e., by inlining a reference to the $variable in the body of the element). Whereas at least one respond element should be found within a request handler, in more complex scenarios it could be useful to include more than one (e.g., to stream back to clients over the same HTTP connection multiple data items as they are computed by the BPEL process). In this case, only the first respond element should specify the HTTP code and the headers of the response. The following ones, should only contain the payload to be appended. The connection with the client will be closed after the last respond element is executed. A respond activity does not need to be placed at the end of the request handler, as the handler execution may continue even after the final response has been sent to the client.

4.3. Minor BPEL extensions and changes

To complete the support for composing RESTful Web services, in this section we discuss a few minor extensions and small changes to the semantics of existing BPEL activities. Fig. 5 lists the namespace for the proposed extensions that can be used to identify processes that make use of them as specified in [1, Section 5.1].

The BPEL <exit/> activity – in addition to completing the execution of the process – has the additional effect of discarding the state of all resources that were associated with the process. Since nested resources are implicitly discarded as execution moves out of their declaration scope, exit has only an effect on the state of the top-level resources, which would otherwise remain accessible to clients even after the normal execution of the process has completed.

Given the absence of an explicitly defined interface description for RESTful Web services, and the lack of strong typing constraints on the data to be exchanged, BPEL for REST is a dynamically typed language. Thus, static typing of <variable> declarations becomes optional [1, SA025]. The attribute messageType – being directly dependent on WSDL – is not used, while the type or element attributes may still be used in the presence of an XML schema description for the RESTful Web service.

5. Reference architecture

In this section, we discuss the impact of the proposed BPEL for REST extensions on the architecture of a BPEL engine. The extensions can be integrated into existing BPEL execution engines using available extensibility mechanisms.

As shown in Fig. 6, as part of the service invocation back-end of the engine, the proposed RESTful invocation activities can be seen as BPEL extension activities, for which a specific execution handler is provided. This handler maps the invocation of a RESTful service down to an HTTP request–response interaction bound to the specified URI of the remote service provider.

[1] Similar to the BPEL isolated scope [1, Section 12.8].

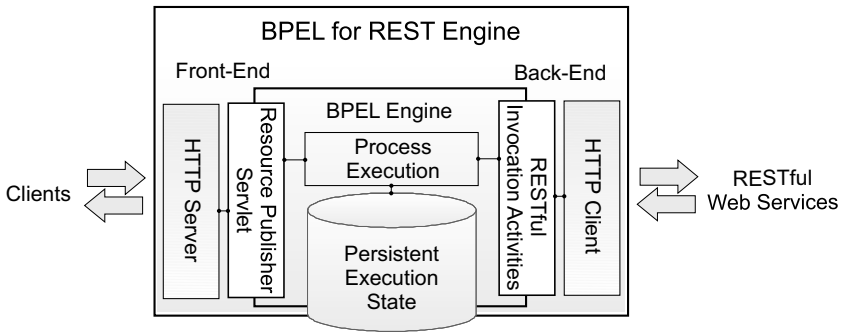


Fig. 6. Reference architecture for the BPEL for REST extensions.

Concerning the management of the state of a composite resource, the proposed extension can be implemented reusing existing state management mechanisms already present in any BPEL engine, because the state of a composite resource is stored using standard BPEL variables. The publishing of composite resources and their URIs can be managed with a servlet-like mechanism deployed together with the BPEL engine in the corresponding application server container. The servlet devoted to resource publishing in the front-end of the engine handles HTTP requests from clients by routing them to the corresponding request handler found in the corresponding BPEL process. Its responsibilities include the correlation of client requests with the corresponding process instance as well as the transfer of the data computed in a process request handler back to the client waiting for the response.

6. Example

After having introduced the BPEL for REST extensions and presented how they can be implemented, in this section we show how the extensions can be applied in practice to compose RESTful Web services in the context of the e-Commerce scenario illustrated in Fig. 7.

The scenario involves the collaboration between a shop service, several product catalog services, a payment service, and a shipping service. Clients can browse product catalogs and request price quotes. If they agree with the offered quotes, they forward them to the shop service to place an order. Once an order has been created, clients may add or remove line items from it and check the total value of the order computed by the shop service (which may offer additional discounts). Clients may confirm an order by uploading the associated payment information. Once an order has been confirmed it can no longer be modified by clients. The shipping service periodically polls the shop service for confirmed orders, gathers their corresponding products and ships them. Clients can track the status (i.e., to check when the shipment has occurred) of their order through the shop service and may also cancel an order as long as it has not yet been shipped. The shop should refund the payment if the canceled order was already confirmed.

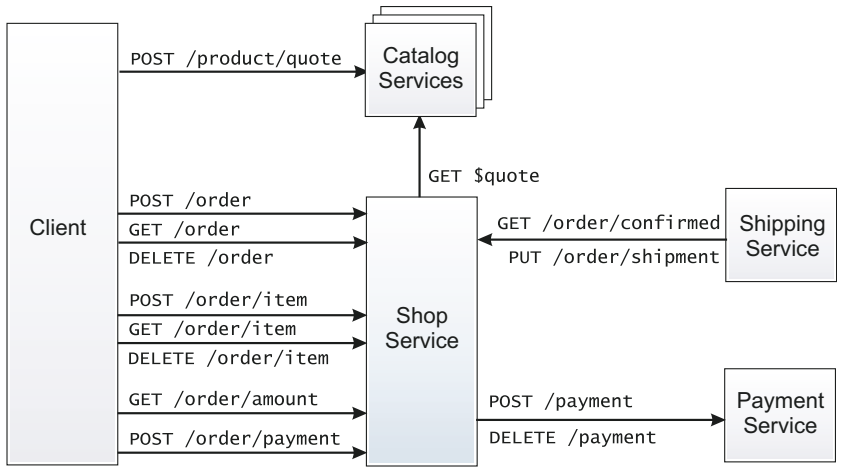


Fig. 7. RESTful e-Commerce Scenario.

Table 1
Example RESTful services API summary.

Verb	URI	Parameters	Description
Shop Service			
GET	`/order`		Get the list of orders.
GET	`/order/confirmed`		Get a list of confirmed orders.
GET	`/order`	*oid*	Read information about the order *oid*.
GET	`/order/amount`	*oid*	Retrieve the total amount for order *oid*.
POST	`/order`	*cid*	Create a new order for customer *cid*
DELETE	`/order`	*oid*	Cancel order *oid*
GET	`/order/item`	*oid*	Get the list of line items for order *oid*.
GET	`/order/item`	*iid*	Read information about the line item *iid*.
POST	`/order/item`	*oid*	Create a new line item for order *oid*
DELETE	`/order/item`	*iid*	Remove line item *iid* from its order
POST	`/order/payment`	*oid*	Update order *oid* with payment information and proceed to checkout.
PUT	`/order/shipment`	*oid*	Update order *oid* with the shipment information once its items have been shipped.
Catalog Service			
GET	`/product`		Get the list of products of the catalog service.
GET	`/product`	*pid*	Read description of product *pid*.
POST	`/product/quote`	*pid*, *qty*	Request a quote for ordering a certain quantity *qty* of product *pid*.
GET	`/quote`	*qid*	Read the product *pid*, *price*, and *qty* associated with a certain quote *qid*.
Payment Service			
POST	`/payment`		Perform a payment.
DELETE	`/payment`	*vid*	Refund a previously charged *vid* payment.

Each service exposes a RESTful API, featuring the resources and the methods listed in Table 1. The table defines the semantics of each method applied to each resource with the given parameters. In the following we give a more detailed description of how the services interact. This interaction can be implemented using the BPEL for REST code for the shop service found in the Appendix.

New orders are created using `POST/order` and are assigned a unique *oid*. This is used by clients to retrieve information about them, i.e., the list of ordered items (`GET/order/item`) and the amount for the order (`GET/order/amount`). To order specific items, clients may use a `POST/order/item` request passing a reference to a quote obtained from the product catalog service. The shop service will retrieve the corresponding information (the product, the price, and the ordered quantity) and store the item – identified with a unique *iid* – as part of the state of the order. Clients can remove ordered items using a `DELETE/order/item` request. Once clients update the order with the payment information (`POST/order/payment`), the shop service contacts the payment service to charge the order's amount to the customer (also with a `POST/payment` request, augmented with the amount to be charged for the particular order). If this interaction succeeds, the order becomes confirmed. Its items can no longer be modified by clients and the shipping service can proceed with the shipment. To do so, the shipper service periodically performs a `GET /order/confirmed` request to download the confirmed orders from the shop service and handle their shipment. Once an order has been shipped, its state is updated with a `PUT/order/shipment` request from the shipping service. Clients may cancel their orders as long as they have not yet been shipped. Canceling a confirmed order will cause the shop service to send a `DELETE/payment` request to the payment service to refund the amount charged to the customer.

The order resource published by the shop service follows the state machine shown in Fig. 8. State transitions are triggered by specific requests from clients of the shop service. This state machine can be implemented with a BPEL process that uses the extensions presented in this paper. In the rest of this section we describe the most relevant BPEL for REST usage patterns and language constructs referring to the code found in the Appendix.

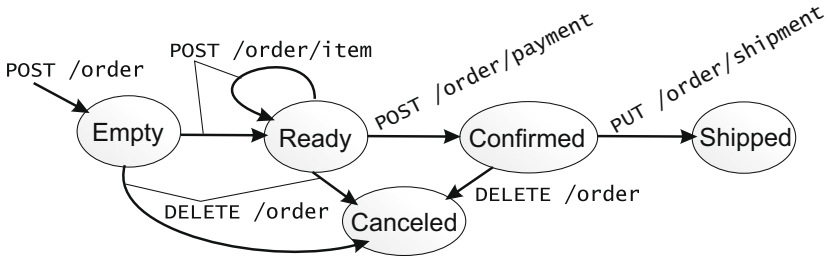


Fig. 8. State of an order managed by the shop service.

Lines 1–9 declare the state variables of a purchase order managed by the shop service BPEL process. These are: *oid*, the order unique identifier; *state*, the current state of the order (following the state machine of Fig. 8); *customer*, a container for the customer information relative to the order; *items*, the list of line items; *payment*, the payment information; *shipment*, the shipment tracking information (set by the shipping service); *amount*, the total value of the order.

The POST handler starting on line 1 initializes a new order resource by first checking the validity of the submitted customer information. Once the state of the resource has been initialized with the `<assign>` activity found on lines 21–26. The following `respond` activity informs the client of the *oid* of the newly created order resource. Invalid requests are answered with HTTP status code 400 (Bad Request).

After an order has been initialized, the process begins waiting (line 31) for clients to update it with the actual products to be ordered. To do so, the process publishes the `/item` resource and declares its state variables (lines 35–39). The following POST request handler, receives a product quote URL from the customer and uses it to GET the corresponding product information from the referenced catalog service (line 44). Once the first item has been added, the order is no longer empty and its state can transition to "Ready" (line 50).

Lines 61–79 show the GET request handler for the `item` resource. Depending on whether the *iid* referring to a specific resource is set, the handler returns the entire list of items for an order or the information describing a specific one. If the requested item cannot be found, a 404 status code is returned.

The POST request handler for the `/order/payment` is found on lines 107–135. It first computes the amount to be paid from the list of items associated with the order. Then it forwards the payment information received from the client to the payment service (line 115). It uses the `catch` construct to detect whether the payment has been successful. In this case, it updates the order's state to "Confirmed" and notifies the client about it (lines 117–127). The following `catchAll` element instead is used to forward to the client the erroneous response returned by the payment service. Once the payment has been completed successfully, the process exits the loop on line 137, publishing the `/order/shipment` resource and deactivating the `/order/payment` and the other resources declared within the loop's scope. Thus, it is no longer possible to modify the line items of a confirmed order and once an order becomes confirmed, the shipping service can update it with the shipment tracking information, as shown in the PUT request handler in lines 140–150.

The GET request handler for the `/order/amount` is implemented twice, depending on the state of the order. Lines 94–103 show that – as long as the order can be modified by clients – the *amount* must be computed from the line items information before it can be returned. Once the state of the order changes to "Confirmed", the requested amount can be directly returned by reading the cached value in the *amount* variable (lines 154–158).

Finally, lines 165–200 implement the order cancellation state transition triggered by a DELETE request on the `/order` resource. In particular, response status code 405 (Method not allowed) is returned if this request is performed on an already shipped order (line 167). The refund of the payment is implemented in lines 171–187, where a DELETE `payment` request is performed on the payment service. In this case, the state of the order is set to "Canceled" only if the payment service could successfully refund the payment.

7. Discussion

The main design decision of the BPEL for REST extensions is to enable developers to compose services at the level of abstraction defined by REST. Thus, the language is extended with activities that reflect the RESTful interaction primitives and give direct support for composition of resources. In this context, it is important to discuss the relationship between the concept of business process and the resource abstraction.

With BPEL for REST, the `resource` construct enables a process to declaratively publish an arbitrary set of resources. Also, the state transition logic of a specific resource can be realized using a business process, which defines under which conditions the resource should change state and how it should react to client requests. Thus, as we have shown in the example, using BPEL for REST is possible to define a relationship between the state of a BPEL process instance and the state of its resources.

The lifecycle of a BPEL process instance begins with the receipt of a message triggering the execution of an *instantiating receive* or a `pick` having a `createInstance` attribute set. Once a process has been instantiated, its state consists of the values assigned to its variables together with an "instruction pointer" indicating which subset of its activities are currently active. During execution, all messages exchanged are correlated with a particular process instance based on their content and on the correlation sets declared in the process. Execution of a process instance proceeds until it reaches an `exit` activity or the process simply runs out of activities that can be executed. The state of a process instance is typically discarded once execution has completed.

The lifecycle of a resource begins with a POST/PUT request to initialize its state. Once a resource has been created, clients may read its current state using GET requests, update its state using PUT requests, and discard it using DELETE requests.

BPEL for REST does not impose any constraint on whether resources are instantiated as part of the process execution or whether new process instances are created upon a particular request to a published resource. Both approaches are supported. If a resource is declared as a top-level element of a BPEL process, clients can interact with its URI as soon as the BPEL code is deployed for execution, no matter whether a process instance has yet been started. If a resource is declared from within a local scope of the process, its URI is published only once the execution of a process instance reaches the particular

scope. By introducing this distinction between top-level and local resource declarations, BPEL for REST supports a pure resource-oriented style of composition, where the result of a BPEL process is a resource that is accessed through the REST uniform interface and can be instantiated multiple times. Nevertheless, BPEL processes can also be instantiated using standard compliant *start activities* and publish resources during their execution that expose part of their state to clients as a RESTful Web service.

In BPEL for REST, the state of a resource is manipulated using its attached request handlers. A new resource instance is created by initializing the resource state variables from within the `<onPut>` or `<onPost>` request handler. To let clients identify a specific resource instance, in the simplest case, an HTTP Cookie can be automatically generated by the BPEL engine handling PUT requests. The engine may intercept responses carrying the HTTP status code 201 (Created) and add the cookie with a unique identifier. Clients will send the cookie for all future interactions (e.g., GET, PUT, and DELETE) with the resource URI and the engine will use the cookie to correlate the requests with the correct state of the resource instance.

As proposed in [38], a cookie-free solution based on URI rewriting that involves the template-based generation of resource identifiers is also possible. This would work as follows: given a top-level URI resource (e.g., `/order`), if a POST request is answered with a 201 (Created) status code, the engine emits a `Location: i` redirection header (where `i` is a unique identifier of the newly created instance) so that clients can direct further GET, PUT, and DELETE requests to the specific resource instance URI `/order/i`. Nested resource URIs are still constructed by concatenating their `uri` attributes, which should now also interleave the resource instance identifier. A GET request to the original resource URI `/order` will retrieve hyperlinks to all active resource instances published by the active processes.

Similar to how HTTP sessions management is transparently supported by most existing Web development languages and infrastructures, the choice between these (or other) request correlation mechanisms should not affect the design of the language. This way, the most suitable correlation mechanism can be chosen when a process is deployed for execution.

8. Related work

This paper builds upon many research contributions within the area of Web service composition languages [41,42].

One of the first proposals for using BPEL to model the internal state of resources can be found in [38]. To do so, BPEL scopes are used to represent different states of a resource and POST requests trigger the transition between different scopes. GET, PUT, and DELETE are mapped to `<onMessage>` event handlers that access, update and clear the values of the BPEL variables declared within the currently active scope. The XPath language embedded in BPEL `assign` activities is extended with functions to compute URI addresses. Unlike the extensions presented in this paper, the resulting "resource-oriented" BPEL does not support the invocation and the composition of external RESTful Web services, but only the publishing of a BPEL process as a resource (or, more precisely, the implementation of a resource state transition logic using BPEL).

Using a RESTful API for a workflow engine to publish the execution state of workflow instances has been proposed by [43]. In this paper, we provide explicit language support to control which parts of a process are published through a similar kind of API. Also, the idea of publishing a "link to a process instance" [44] can be traced back to the Simple Workflow Access Protocol (SWAP [45]), originally proposed by the Workflow Management Coalition (WfMC) in 1998. More recently, the idea of mapping the state of a workflow instance to a "resource"-like abstraction, as defined by the WS-Resource Framework has been explored in [46].

BPEL^{light} [18] advocates to decouple BPEL from WSDL by identifying the BPEL activities that are closely dependent on WSDL abstractions and subsuming them with a generic messaging construct (the `<interaction-Activity>`). We follow a similar, but less generic, approach, when we propose a specific set of *resource-oriented* activities to natively support the composition of RESTful Web services.

Bite [47] (or the IBM Project Zero assembly flow language) can be seen as a simplified variant of BPEL with a reduced set of activities specifically targeting the development of collaborative Web application workflows [48]. Similar to BPEL for REST, the language supports the invocation of RESTful Web services and the corresponding runtime allows to automatically publish processes as resources. Unlike BPEL for REST, the Bite language does not give a direct representation of the REST interaction primitives, as those are abstracted in one `<invoke>` activity, which – as opposed to the one from BPEL – can be directly applied to a URI. Also, with Bite it is not possible to dynamically declare resources from within a process, so that clients may access their state under different representations in compliance with the REST uniform interface principle (e.g., the PUT verb is not supported).

From a practical standpoint, ad-hoc support for invoking RESTful Web services is currently being discussed for some BPEL engines and development environments [35]. Within the Apache Orchestration Director Engine (ODE) project, a proposal for RESTful BPEL extension has been recently published.[2] Unlike BPEL for REST, it overrides the semantics of existing BPEL activities (i.e., `<invoke>`) to handle the invocation of RESTful Web services. Non-standard attributes are introduced to store the required metadata and bindings to URIs. The ability of publishing resources is provided through extensions of the `<onEvent>` and `<receive>` activities. The professional version of ActiveBPEL uses a different approach to provide "implicit"

[2] Linked from http://ode.apache.org/bpel-extensions.html, last checked on October 6th, 2008.

support for REST. Existing BPEL <invoke> and <receive> activities are bound to predefined WSDL document used by the engine to transform external REST interactions so that they can be handled from existing BPEL constructs [49]. This way – for example – existing BPEL processes can be deployed to be started with a GET request issued from a Web browser. BPEL for REST follows a different approach, giving a clear separation between the RESTful activities and the standard ones used to interact with traditional WSDL-based services. Thus, developers of BPEL processes can work at a level of abstraction more suitable to resource composition.

9. Conclusion

As REST gains traction and the number of published RESTful Web service APIs grows, it becomes important to understand how the concept of business process can be applied as the glue to compose "resources" in addition to traditional "services". In this paper, we have proposed an extension to the WS-BPEL standard process modeling language to support the native composition of RESTful Web services. The extensions promote the resource abstraction to become a first-class language construct. This way, the interaction primitives (GET, POST, PUT, and DELETE) stemming from the REST uniform interface principle can be directly used from within a BPEL process as new service invocation activities. A RESTful Web service APIs can also be implemented using BPEL with the proposed declarative constructs for publishing resources as a view over the process variables. With a detailed example, we have applied the proposed extensions to an e-Commerce scenario, showing how BPEL for REST can be used in practice to specify the behavior of a composite resource. A description of the design of a reference architecture to implement the proposed extensions on top of a generic BPEL engine is also included.

Acknowledgements

This work is partially supported by the EU-IST-FP7-215605 (RESERVOIR) project. The author would also like to thank Monica Frisoni, Domenico Bianculli, Tammo van Lessen, and the anonymous reviewers of the BPM2008 Conference for their invaluable comments.

Appendix. Sample code

The following code gives the BPEL for REST implementation of the example scenario described in Section 6. To enhance the readability of the XML code, we highlight the BPEL for REST extension activities in boldface instead of prefixing them with the rest XML namespace corresponding to the extension. Also, to save space we have taken some liberty with the syntax of the <assign> activity (this simplified JavaScript-like syntax is not part of the proposed language extension). Variable interpolation is indicated by prefixing the name of variables with the $ sign.

```
<process name="ShopService">
<resource uri="order">
 <variable name="oid"/>
 <variable name="state"/>
 <variable name="customer"/>
 <variable name="items"/>
 <variable name="payment"/>
 <variable name="shipment"/>
 <variable name="amount"/>
<!-- POST /order request handler -->
<onPost>
 <if><condition>$request.customer == null</condition>
  <sequence>
   <respond code="400">
    Cannot create a new order: missing customer information
   </respond>
   <exit/>
  </sequence>
  <else>
   <sequence>
    <assign>
           oid = new ID("/order");
      customer = $request.customer;
         state = 'Empty';
         items = [];
    </assign>
    <respond code="201">
     <header name="Location">$oid</header>
```

```
      Order $oid has been created
     </respond>
     <scope><while>
      <condition>
       ($state == 'Empty') || ($state == 'Ready')
      </condition>
      <resource uri="item">
      <variable name="iid"/>
      <variable name="product"/>
      <variable name="quantity"/>
      <variable name="price"/>
<!-- POST /order/item request handler -->
      <onPost>
       <sequence>
        <variable name="quote_response"/>
        <get url="$request.quote" response="quote_response"/>
        <assign>
            iid = new ID("/order/item");
        product = quote_response.product;
          price = quote_response.price;
       quantity = quote_response.quantity;
          state = 'Ready';
   items[$iid] = {iid: $iid, product: $product,
                  price: $price, quantity: $quantity};
        </assign>
        <respond code="201">
         <header name="Location">$iid</header>
         Item $iid has been added to order $oid
        </respond>
       </sequence>
      </onPost>
<!-- GET /order/item request handler -->
      <onGet>
       <if><condition>$request.iid == null</condition>
        <respond code="200">
         <header name="Content-Type">application/json</header>
         $items
        </respond>
        <elseif><condition>$items[$request.iid] == null</condition>
         <respond code="404">
          Requested order item $request.iid not found.
         </respond>
        </elseif>
        <else>
         <respond code="200">
          <header name="Content-Type">application/json</header>
          $items[$request.iid]
         </respond>
        </else>
       </if>
      </onGet>
<!-- DELETE /order/item request handler -->
      <onDelete>
       <sequence>
        <assign>
         items[$request.iid] = null;
        </assign>
        <respond code="200">
         Order item $request.iid deleted
        </respond>
       </sequence>
```

```
      </onDelete>
     </resource>
     <resource uri="amount">
<!-- GET /order/amount request handler -->
      <onGet>
       <sequence>
        <assign>
         amount = computeTotalOrderAmount($items);
        </assign>
        <respond code="200">
         $amount
        </respond>
       </sequence>
      </onGet>
     </resource>
     <resource uri="payment">
<!-- POST /order/payment request handler -->
      <onPost>
       <sequence>
        <variable name="payment_response"/>
        <assign>
         amount = computeTotalOrderAmount($items);
         payment = $request.payment;
         payment.amount = $amount;
        </assign>
        <post url="https://www.visa.com/payment"
              request="payment" response="payment_response">
         <catch code="200">
          <sequence>
           <assign>
            state = "Confirmed";
            payment.id = payment_response.confirmation;
           </assign>
           <respond code="200">
            Payment has been accepted: $payment_response
           </respond>
          </sequence>
         </catch>
         <catchAll>
          <respond code="500">
           Could not complete payment: $payment_response
          </respond>
         </catchAll>
        </post>
       </sequence>
      </onPost>
     </resource>
     </while></scope>
     <resource uri="shipment">
     <!-- PUT /order/shipment request handler -->
      <onPut>
       <sequence>
        <assign>
         shipment = $request.shipment;
         state = "Shipped";
        </assign>
        <respond code="200">
         Order has been shipped
        </respond>
       </sequence>
      </onPut>
```

```
151       </resource>
152       <resource uri="amount">
153       <!-- GET /order/amount request handler -->
154        <onGet>
155         <respond code="200">
156          $amount
157         </respond>
158        </onGet>
159       </resource>
160       </sequence>
161       </else>
162      </if>
163     </onPost>
164     <!-- DELETE /order request handler -->
165     <onDelete>
166     <if><condition>state == 'Shipped'</condition>
167      <respond code="405">
168       Cannot cancel orders already shipped.
169      </respond>
170      <elseif><condition>state == 'Confirmed'</condition>
171       <delete uri="https://www.visa.com/payment/$payment.id">
172        <catch code="200">
173         <sequence>
174          <assign>
175           state = 'Canceled';
176          </assign>
177          <respond code="200">
178           Order has been canceled and payment has been refunded.
179          </respond>
180         </sequence>
181        </catch>
182        <catchAll>
183         <respond code="500">
184          Cannot refund payment. Order will not be canceled.
185         </respond>
186        </catchAll>
187       </delete>
188      </elseif>
189      <else>
190       <sequence>
191        <assign>
192         state = 'Canceled';
193        </assign>
194        <respond code="200">
195         Order has been canceled.
196        </respond>
197       </sequence>
198      </else>
199     </if>
200     </onDelete>
201    </resource>
202   </process>
```

References

[1] OASIS, Web Services Business Process Execution Language (WSBPEL) 2.0, 2006.
[2] U. Assmann, Invasive Software Composition, Springer, 2003.
[3] R. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. thesis, University of California, Irvine, 2000.
[4] R. Fielding, A little REST and relaxation, in: The International Conference on Java Technology (JAZOON07), Zurich, Switzerland, June 2007, <http://www.parleys.com/display/PARLEYS/A\%20little\%20REST\%20and\%20Relaxation>.
[5] L. Richardson, S. Ruby, RESTful Web Services, O'Reilly, 2007.
[6] K. Laskey, P.L. Hgaret, E. Newcomer (Eds.), Workshop on Web of Services for Enterprise Computing, W3C, 2007, <http://www.w3.org/2007/01/wos-ec-program.html>.
[7] S. Vinoski, Serendipitous reuse, IEEE Internet Computing 12 (1) (2008) 84–87.
[8] T. O'Reilly, REST vs. SOAP at Amazon, April 2003, <http://www.oreillynet.com/pub/wlg/3005>.
[9] Programmable Web, API Dashboard, 2007, <http://www.programmableweb.com/apis>.
[10] Wikipedia, Mashup (web application hybrid), <http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid>).
[11] C. Pautasso, S. Tai, M. Maximilien (Eds.), Second International Workshop on Web APIs and Services Mashups (Mashups'08), 2008, <http://www.icsoc-mashups.org/>.
[12] M. Schrenk, Webbots, Spiders, and Screen Scrapers, No Starch Press, 2007.
[13] D.E. Descy, Mashups with or without potatoes, TechTrends 51 (2) (2007) 4–5.
[14] M.J. Hadley, Web Application Description Language (WADL), 2006, <http://wadl.dev.java.net/>.
[15] P. Prescod, Web Resource Description Language, 2002,<http://www.prescod.net/rest/wrdl/wrdl.html>.

[16] N. Walsh, Norm's Service Description Language, 2005, <http://norman.walsh.name/2005/03/12/nsdl>.
[17] D. Orchard, Web Description Language, 2005, <http://www.pacificspirit.com/Authoring/WDL/>.
[18] J. Nitzsche, T. van Lessen, D. Karastoyanova, F. Leymann, BPEL^{light}, in: Proceedings of the Fifth International Conference on Business Process Management (BPM 2007), 2007, pp. 214–229.
[19] A. Barros, M. Dumas, A.H. ter Hofstede, Service interaction patterns, in: Proceedings of the Third International Conference on Business Process Management, LNCS, vol. 3694, Springer, Nancy, France, 2005, pp. 302–318.
[20] T. Berners-Lee, R. Fielding, L. Masinter, Uniform resource identifier (URI): generic syntax, in: IETF RFC 3986, January 2005.
[21] G. Decker, A. Luders, K. Schlichting, H. Overdick, M. Weske, RESTful petri net execution, in: Fifth International Workshop on Web Services and Formal Methods, Milan, Italy, 2008.
[22] X. Xu, L. Zhu, Y. Liu, M. Staples, Resource-oriented business process modeling for ultra-large-scale systems, in: Proceedings of the Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS'08), Leipzig, Germany, 2008, pp. 65–68.
[23] C. Pautasso, O. Zimmermann, F. Leymann, RESTful web services vs. big web services: Making the right architectural decision, in: Proceedings of the 17th World Wide Web Conference, Beijing, China, 2008, pp. 805–814.
[24] H. Haas, Reconciling web services and REST services, in: Proceedings of ECOWS 2005, Växjö, Sweden, 2005, p. Keynote Address.
[25] IBM, SAP, WS-BPEL Extension for Sub-Processes, October 2005.
[26] Active Endpoints, IBM, Oracle, SAP, WS-BPEL Extension for People, August 2005.
[27] D. Habich, S. Richly, S. Preissler, M. Grasselt, W. Lehner, A. Maier, BPEL-DT- data-aware extension of BPEL to support data-intensive service applications, in: Emerging Web Services Technology, vol. II, Birkhäuser, 2008, pp. 111–128.
[28] G. Decker, O. Kopp, F. Leymann, M. Weske, BPEL4Chor: Extending BPEL for modeling choreographies, in: Proceedings of the IEEE 2007 International Conference on Web Services (ICWS 2007), Salt Lake City, Utah, USA, 2007, pp. 296–303.
[29] W. Tan, L. Fong, N. Bobroff, BPEL4JOB: A fault-handling design for job flow management, in: Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007), Vienna, Austria, 2007.
[30] T. Dörnemann, T. Friese, S. Herdt, E. Juhnke, B. Freisleben, Grid workflow modelling using grid-specific BPEL extensions, in: Proceedings of the German e-Science Conference (GES2007), Baden-Baden, Germany, 2007.
[31] E. Chinthaka, REST and Web services in WSDL 2.0, May 2007, <http://www.ibm.com/developerworks/webservices/library/ws-rest1/>.
[32] J. Snell, Resource-oriented vs. activity-oriented web services, IBM developerWorks, October 2004, <http://www-128.ibm.com/developerworks/webservices/library/ws-restvsoap/>.
[33] S. Vinoski, Putting the web into web services: interaction models, Part 1: Current practice, IEEE Internet Computing 6 (3) (2002) 89–91.
[34] S. Vinoski, Putting the web into web services: interaction models, Part 2, IEEE Internet Computing 6 (4) (2002) 90–92.
[35] J. Pasley, Using Yahoo's REST Services with BPEL, Cape Clear, 2008, <http://developer.capeclear.com/video/httpwizard/httpwizard.html>.
[36] R. Fielding, R.N. Taylor, Principled design of the modern web architecture, ACM Transactions on Internet Technology 2 (2) (2002) 115–150.
[37] H. Overdick, The resource-oriented architecture, in: Proceedings of the 2007 IEEE Congress on Services, Salt Lake City, USA, 2007, pp. 340–347.
[38] H. Overdick, Towards resource-oriented BPEL, in: Second ECOWS Workshop on Emerging Web Services Technology, 2007.
[39] D. Florescu, A. Gruenhagen, D. Kossmann, XL: An XML programming language for Web service specification and composition, in: Proceedings of the 11th International World Wide Web Conference (WWW2002), Honululu, Hawaii, USA, 2002.
[40] D. Crockford, JSON: the fat-free alternative to XML, in: Proceedings of XML 2006, Boston, USA, 2006, <http://www.json.org/fatfree.html>.
[41] S. Dustdar, W. Schreiner, A survey on web services composition, International Journal of Web and Grid Services (IJWGS) 1 (1) (2005) 1–30.
[42] C. Pautasso, G. Alonso, From web service composition to megaprogramming, in: Proceedings of the Fifth VLDB Workshop on Technologies for E-Services (TES-04), Toronto, Canada, 2004, pp. 39–53.
[43] M. zur Muehlen, J.V. Nickerson, K.D. Swenson, Developing web services choreography standards – the case of REST vs. SOAP, Decision Support Systems 40 (1) (2005) 9–29.
[44] K.D. Swenson, Workflow and web service standards, Business Process Management Journal 11 (3) (2005) 218–223.
[45] G.A. Bolcer, G. Kaiser, SWAP: leveraging the web to manage workflow, IEEE Internet Computing 3 (1) (1999) 85–88.
[46] T. Heinis, C. Pautasso, G. Alonso, Mirroring resources or mapping requests: implementing WS-RF for Grid workflows, in: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid2006), Singapore, 2006.
[47] F. Curbera, M. Duftler, R. Khalaf, D. Lovell, Bite: workflow composition for the web, in: Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007), Vienna, Austria, 2007.
[48] F. Rosenberg, F. Curbera, M.J. Duftler, R. Kahalf, Composing RESTful services and collaborative workflows, IEEE Internet Computing 12 (5) (2008) 24–31.
[49] ActiveBPEL Designer Online Help, Using a REST-based Service, 2008, <http://www.activebpel.org/infocenter/ActiveVOS/v50/index.jsp?topic=/com.activee.bpel.doc/html/UG22-1.html>.

Cesare Pautasso is assistant professor in the new Faculty of Informatics at the University of Lugano, Switzerland. Previously he was a researcher at the IBM Zurich Research Lab and a senior researcher at ETH Zurich, where he earned his Ph.D. in computer science in 2004. His research focuses on building experimental systems to explore the intersection of model–driven software composition techniques, business process modeling languages, and autonomic/Grid computing. Recently he has developed an interest in Web 2.0 Mashups and Architectural Decision Modeling. He is the lead architect of JOpera, a powerful rapid service composition tool for Eclipse. His teaching and training activities both in academia and in industry cover advanced topics related to Web Development, Middleware, Service Oriented Architectures and emerging Web services technologies. For more information refer to www.pautasso.info.