BPEL for REST

Cesare Pautasso

Faculty of Informatics, University of Lugano
via Buffi 13, 6900 Lugano, Switzerland
cesare.pautasso@unisi.ch

Abstract. Novel trends in Web services technology challenge the assumptions made by current standards for process-based service composition. Most RESTful Web service APIs, which do not rely on the Web service description language (WSDL), cannot easily be composed using the BPEL language. In this paper we propose a lightweight BPEL extension to natively support the composition of RESTful Web services using business processes. We also discuss how to expose the execution state of a business process so that it can be manipulated through REST primitives in a controlled way.

1 Introduction

With the goal of attracting a larger user community, more and more service providers are switching to REST [1] in order to make it easy for clients to consume their Web service APIs [2–4]. This emerging technology advocates a return to the original design principles of the World Wide Web [5] to provide for the necessary interoperability and enable integration between heterogeneous distributed systems. The HTTP standard protocol thus becomes the basic interaction mechanism to publish existing Web applications as services by simply replacing HTML payloads with data formatted in "plain old" XML (POX) [6].

In this context, many of the assumptions made by existing languages for Web service composition no longer hold [7]. Since most RESTful Web service APIs do not use the standard Web service description language (WSDL) to specify their interface contracts, it is not possible to directly apply existing languages, tools and techniques that are built upon this standard interface description language. Considering that REST prescribes the interaction with resources identified by URIs [8], languages that assume static bindings to a few fixed communication endpoints do not cope well with the dynamic and variable set of URIs that make up the interface of a RESTful Web service. Even the basic message-oriented invocation constructs for sending and receiving data cannot be consistently applied with the uniform interface principle of a RESTful Web service. Moreover, in some cases, also the assumption of dealing with XML data [9] may not apply when accessing resources represented in other, more lightweight, formats such as the JavaScript Object Notation (JSON [10]).

In this paper we argue that process-based composition languages can and should be applied to compose RESTful Web services in addition to WSDL-based ones. However, given the differences between the two kinds of services [11, 12], we claim that native support for composing RESTful Web services is an important requirement for a modern service composition language. To address this requirement, we show how the Business Process Execution Language (WS-BPEL [13]) standard can be extended. Also, following the recursive nature of software composition (the result of a composition should be composable [14]), we propose to apply REST principles to the design of the API of a BPEL engine so that processes themselves (and a view over their execution state) can be published through a RESTful Web service interface.

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 our extensions to the BPEL language with an example loan application process. The extensions are then specified in Section 5. The relationship between the resource and the process abstractions is discussed in Section 6. Related work is outlined in Section 7. We draw our conclusions in Section 8.

2 Motivation

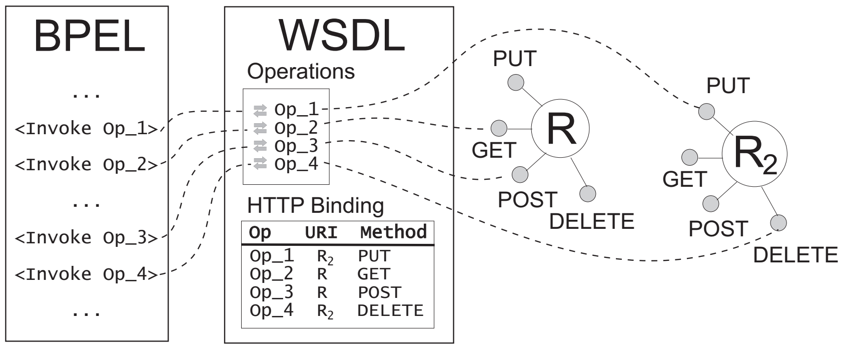
The following quote from the specification of the WS-BPEL 2.0 standard motivated the research towards the extensions presented in this paper.

> 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 [13, Section 3].

In other words, the WS-BPEL and the WSDL languages are tightly coupled. The *partner link type* construct, introduced in BPEL to tie together pairs of matching WSDL port types, forces all external interactions of a process to go through a WSDL interface. Over time, this strong constraint on the original design of the composition language has produced a set of extensions (e.g., BPEL-SPE [15] for processes invoking other subprocesses, BPEL4People [16] for interaction with human operators, or – more recently – BPEL4JOB [17] with job submission and fault handling extensions to better deal with the requirements of scientific workflow applications, BPEL-DT [18] to compose data intensive applications, and BPEL^{light} [19] proposing a complete decoupling of the two languages in the context of message-oriented interactions). Along the same direction, in this paper we propose another extension to make BPEL natively support the composition of RESTful Web services.

Without loss of generality, we will ignore the differences between WSDL 1.1 and the latest WSDL 2.0 version and assume that WS-BPEL will soon be updated to support the latter. This is important because – as shown in Figure 1 – the new HTTP binding introduced in WSDL 2.0 can be used to wrap a RESTful Web service and describe its interface using the WSDL language [20]. Thus, the problem appears to be already solved: with this binding, a BPEL process could send and receive messages over the HTTP protocol without necessarily using the SOAP message format.

From a practical standpoint, however, the approach does not lead to a satisfactory solution for the following reasons. WSDL 2.0 is not yet widely deployed in the field, especially to describe existing RESTful Web service APIs, and there is very little evidence indicating that this will change in the future. Thus, at the moment the burden of going



Op	URI	Method
Op_1	R_2	PUT
Op_2	R	GET
Op_3	R	POST
Op_4	R_2	DELETE

Fig. 1. Funneling RESTful Web services through a WSDL 2.0 interface in order to invoke them from a BPEL process

through the procedure of recreating a synthetic WSDL description for the RESTful Web service interface is shifted from the service provider to the BPEL developer [21]. Every supplementary artifact introduced in a solution makes it more complex and more expensive to maintain. The additional WSDL description needs to be updated whenever changes are made to the corresponding RESTful Web service.

From a theoretical point of view, this approach hides the resource-oriented interaction primitives of REST inside the service-oriented abstractions provided by WSDL. As we are going to discuss, the invocation of a WSDL operation by means of synchronous or asynchronous message exchange does not always fully match the semantics of a "GET", "PUT", "POST", or "DELETE" request performed on a resource URI [11, 22–24]. Thus, we argue that native support for explicitly controlling the interaction based on REST primitives would be beneficial to developers trying to apply the BPEL language to specify how to compose resources, as opposed to WSDL-based services (Figure 2).

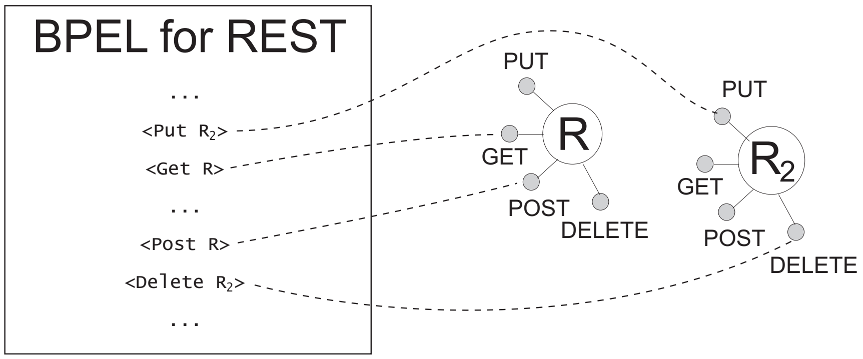


Fig. 2. Direct invocation of RESTful Web services using the BPEL for REST extensions

3 Composing RESTful Web services

The composition of RESTful Web services is typically associated with so-called Web 2.0 Mashups [25, 26], where this emerging technology helps to reduce the complexity and the effort involved in scraping the data out of HTML Web pages [27]. In this context, a consensus still needs to be reached in terms of how to describe the interface of such RESTful Web service. As we previously discussed it is not always convenient to use the HTTP binding of the latest version of the Web *service* description language (WSDL 2.0). Whereas the Web *application* description language (WADL [28]) has been recently proposed, most APIs still rely on human-oriented documentation. This typically includes interactive examples that help developers to infer how to use a particular service. If XML is employed as representation format, an XML Schema description is often associated with the textual documentation.

This uncertain 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 be used. Thus, like [19], we choose not rely upon the presence of such a description language and introduce language constructs for service invocation that directly map to the underlying interaction mechanisms and patterns [29].

In the rest of this section we give an overview over the design principles followed by RESTful Web services [1, 6] and discuss how these challenge the BPEL language in its current form. These principles and challenges have been taken into account during the design of the corresponding BPEL extensions presented in this paper.

Resource addressing through URI – The interface of a RESTful Web service consists of a set of resources, identified by URIs. From the client perspective, the interaction with a RESTful Web service requires to interact with a dynamic and variable set of URIs, which are not necessarily known nor identifiable in advance. From the service provider viewpoint, it should be possible to use languages such as BPEL to manage the lifecycle of arbitrary resources and to implement the state transition logic of composite resources [30].

Uniform interface – The set of operations available for a given resource is fixed by design to the same four methods: PUT, GET, POST, and DELETE. These CRUD-like operations apply to all resources with similar semantics. POST creates a new resource under the given URI; GET retrieves the representation of the current state of a resource; PUT updates the state of a resource (and may also initialize the state of a new resource if none was previously found at the given URI); DELETE frees the state associated with a resource and invalidates the corresponding URI. Each of these methods (or verbs) is invoked on a resource using a synchronous HTTP request-response interaction round. Thus, there is no need for the asynchronous (one-way) message exchange patterns supported by BPEL/WSDL. Also, since the set of operations is fixed to the four methods, there is no apparent need to explicit enumerate them using an interface description language.

Self-descriptive messages – Thanks to meta-data, clients can negotiate with service providers the actual representation format of a resource. Thus, it is not always possible to statically assign a fixed data type to the payload of an HTTP response. Also, since RESTful Web services may not always exchange XML data, BPEL variables need to accomodate a larger set of possible data representation formats. Additionally, meta-data is used to perform client and server authentication, access control, compression, and caching. Thus, from within a BPEL process it should be possible to access and control this meta-data, in a way similar to how SOAP message headers are configured.

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. Through hyperlinks, valid future states of the interaction can be embedded in the representation of a resource and discovered interactively by clients. From this principle, it follows that resource URIs may be dynamically generated by a service. Thus, a mechanism for extracting URIs from response messages is needed together with a construct for dynamic binding of activities to target resource URIs.

4 Example

Before giving a complete specification of the BPEL for REST extensions, in this section we informally introduce them within the example process illustrated in Figure 3.

The Loan application process exposes one resource, the `loan` that represents the state of a loan application. Clients can initiate a new loan application with a PUT request on this resource and retrieve the current state of their application with GET. A DELETE request immediately cancels the application if it is still in progress, otherwise it triggers the execution of a different loan cancellation process. The lifecycle of a loan application goes through several stages as specified by the sequence of activities triggered by the PUT request. In particular, the process invokes the RESTful Web services of two different banks to gather available interest rates and payment deadlines. It will then wait for the client to make a decision choosing between the two competing offers. Once the client has chosen an offer, the BPEL process will confirm the acceptance with a POST request that is dynamically bound to a URI provided by the bank service (indicated with `$accept` in the Figure) together with the offer.

For clarity, in the following we highlight the BPEL for REST extension activities in **`boldface`**. Also, to enhance the readability of the XML code, we have omitted namespace declarations and taken some liberty with the syntax of the `<assign>` activity (this simplified syntax is not part of the proposed language extension). Variable interpolation is indicated by prefixing the name of variables with the `$` sign.

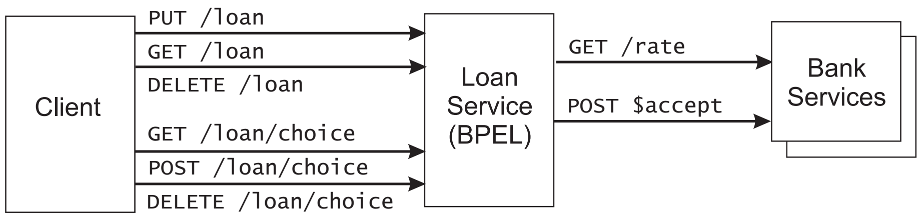


Fig. 3. Loan Application Process Example

```
<process name="LoanApplication">
<resource uri="loan">
<!-- State variables of the resource -->
 <variable name="name"/>
 <variable name="amount"/>
 <variable name="rate"/>
 <variable name="bank"/>
 <variable name="start_date"/>
 <variable name="end_date"/>
```

These variable declarations store the state of the `loan` resource declared within the `LoanApplication` BPEL process. Associated with the resource, the process specifies three request handlers: `onPut`, `onGet`, `onDelete`.

```
<!-- PUT /loan request handler -->
<onPut>
 <if><condition>$request.amount &gt 100000</condition>
  <then>
   <response code="400">
    Requested amount too large
   </response>
   <exit/>
  </then>
  <else>
   <sequence>
    <assign>
          name = $request.name;
        amount = $request.amount;
    start_date = $request.start_date;
    </assign>
    <response code="201">
     Processing loan application
    </response>
```

The PUT request is used to initialize the state of the `loan` resource. However, if the requested loan amount is too large, the client will be informed with a `response` carrying the HTTP code 400 (Bad Request) and the resource will be immediately deleted using the BPEL `<exit>` activity. Otherwise the state of the resource is initialized from the BPEL for REST predefined variable called `$request` which stores the input payload of the HTTP PUT request. After informing the client that the resource could be created (HTTP response code 201), the `onPut` request handler continues with the next step of the loan application process, when two different RESTful Web services are invoked to retrieve the available interest rates for the given loan request.

```
<!-- Get rates from two different bank services -->
 <scope>
  <variable name="ubs_response"/>
  <variable name="cs_response"/>
  <variable name="url_accept"/>
  <variable name="accept_response"/>
  <flow>
   <get uri="http://www.ubs.ch/rate?chf=$amount&from=$start_date"
        response="ubs_response">
   <get uri="http://www.cs.ch/rates?amount=$amount&start=$start_date"
        response="cs_response">
  </flow>
```

The two services are invoked in parallel using the BPEL `<flow>` activity. They are invoked using a GET request on a URI that is constructed using the current state of the `loan` resource. The response of the services are stored in the corresponding variables, if the invocation is successful.

At this point, the client should decide which loan rate is preferred. To do so, the process dynamically creates a new resource at the URI `loan/choice`. A GET on this new resource will retrieve the current offers (represented in JSON) while a DELETE request can be used to reject both offers and cancel the entire loan application process.

```
<!-- Let client choose the preferred bank -->
     <while>
      <condition>TRUE</condition>
      <resource uri="choice">
       <onGet>
<!-- Return the rates offered by the banks -->
        <response code="200">
         <header name="Content-Type">application/json</header>
      [ {  bank:"cs",
           rate:"$cs_response.rate",
           end_date:"$cs_response.until" },
        {  bank:"ubs",
           rate:"$ubs_response.rate",
           end_date:"$ubs_response.end" } ]
        </response>
       </onGet>
       <onDelete>
<!-- Reject the offer and cancel the loan application -->
        <sequence>
         <response code="200"/>
         <exit/>
        </sequence>
       </onDelete>
       <onPost>
<!-- Store the client choice and continue -->
       <sequence>
        <assign>bank = $request.choice;</assign>
        <if><condition>bank == "cs"</condition>
         <then><assign>rate = $cs_response.rate;
                      end_date = $cs_response.until;
                      url_accept = $cs_response.accept</assign></then>
         <else><assign>rate = $ubs_response.rate;
                      end_date = $ubs_response.end;
                      url_accept = $ubs_response.accept</assign></else>
        </if>
        <response code="200"/>
        <activeBPEL:break/>
       </sequence>
      </onPost>
     </resource>
    </while>
```

To continue the execution of the process, the client must communicate its choice using a POST request. The process will update the state of the loan resource with the client decision and the corresponding information from the chosen offer: the `rate` and `end_date` of the loan and the URL to use in order to confirm the acceptance of the offer with the bank.

A successful (code 200) response is then returned to the client and the execution continues by exiting the `<while>` loop[1]. Once the execution exits the `scope` of the resource `choice` declaration, such resource is no longer available to clients. Further requests to its URI will result in a 404 (Not Found) code being returned by the BPEL for REST engine. To conclude the `onPut` request handler, the chosen bank service is informed by sending the client's name with a POST request dynamically bound to the acceptance URL that was returned with the terms of the offer.

```
<!-- Accept the loan offered by the chosen bank -->
    <post uri="$url_accept" request="$name" response="accept_response">
   </scope>
  </sequence>
  </else>
 </if>
</onPut>
```

It is important to point out that once the execution of the `<onPut>` request handler is completed, the state of the newly created `loan` resource is not discarded, but it will remain available to clients until the corresponding DELETE request is issued. As illustrated in the final part of the example, clients can retrieve such state at any time using a GET request. To cancel the loan application, clients may issue a DELETE request. However, depending on the state of the loan resource, canceling it may not always be possible and may require to execute the corresponding loan cancellation business process.

```
<!-- GET /loan request handler -->
<onGet>
<!-- Return the state of the loan application -->
</onGet>

<!-- DELETE /loan request handler -->
<onDelete>
 <if> <condition>bank == null</condition>
  <then>
   <response code="200"/>
   <exit/>
  </then>
  <else>
<!-- Start the loan cancellation process -->
   <invoke...>
  </else>
 </if>
</onDelete>
</resource>
</process>
```

[1] For simplicity, the process calls the `<break>` activity (a non-standard BPEL extension introduced by the ActiveBPEL engine). However it would also be possible to do so by setting the appropriate flag to be tested in the loop condition.

5 BPEL for REST extensions

As shown in the previous example, in this paper we propose two kinds of extensions. First, it should be possible to invoke a RESTful Web service directly from a BPEL process. Also, we propose a declarative construct to expose parts of the execution state of a BPEL process as a resource. To do so, we choose the introduce a set of activities, constructs and handlers that are directly related to the REST uniform interface principle.

5.1 Invoking RESTful Web services

To invoke a RESTful Web service using the HTTP (or HTTPS) protocol from a BPEL process, we add these four activities: `<get>`, `<post>`, `<put>`, `<delete>`.

As shown in Figure 4, the four activities use the `uri` attribute to specify the target resource address. The URI can be a constant value, but also be computed out of data currently stored in the process variables. Thus, BPEL for REST supports dynamic binding to invoke resource URIs that are only known at runtime. The only constraint on the structure of the URI is that it should target a resource accessible using the HTTP or the HTTPS protocols.

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 with these two methods.

The headers sent with the HTTP request can be controlled using the `<header>` child elements of each of the four invocation activities. Also in this case, their values can be set to constant values but also computed from information stored in BPEL variables.

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

```
<get uri="" response="">
  <header name="">*value</header>
  <catch code="" faultName=""?>*...</catch>
  <catchAll>?...</catchAll>
</get>
<post uri="" request="" response=""> ... </post>
<put uri="" request="" response=""?> ... </put>
<delete uri="" response=""?> ... </delete>
```

Fig. 4. BPEL for REST extensions to invoke a RESTful Web service

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

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

Fig. 5. BPEL for REST extensions to declare resources within a process

5.2 Publishing processes as RESTful Web services

To declaratively publish certain sections of a BPEL process as a resource we introduce the `<resource>` container element (Figure 5). This construct allows to dynamically publish resources to clients depending on whether their declarations are reached by the execution of the BPEL process. Once 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 and they are immediately published once the BPEL process is deployed for execution. As shown in the example, resource declarations can be nested. The URI of nested resources is computed by concatenating their `uri` attribute with the usual path (`/`) separator.

Similar to 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 visible from within the resource declaration.

Like the BPEL `<pick>`, the `<resource>` contains a set of 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>`, `<onDelete>`. If a request handler for a given verb is not declared, requests to the resource using such verb will be answered by the BPEL engine with HTTP code 405 (Method not allowed). At least one request handler must be included in a resource declaration and a handler for a given request 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. If 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. Since only POST requests are not meant to be *idempotent*, the `<onPost>` handler may be flagged to guarantee proper isolation[2] with respect to the access of the resource state variables. To ensure that GET requests on a resource are indeed *safe*, the `onGet` request handler only has read-only access to the state variables of the corresponding resource.

The behavior of a request handler can be specified using the normal BPEL structured activities (i.e., `<sequence>`, `<flow>`, etc.). However, control-flow `links` across 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 `<response>` 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 invocation of a RESTful Web service. 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 `response` 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 `response` element should specify the HTTP code and the headers of the response. The connection with the client will be closed after the last `response` element is executed. As shown in the `onPut` request handler of the example, a `response` does not need to be placed at the end of the request handler, as the handler execution may continue even after the response has been sent to the client.

5.3 Minor BPEL extensions and changes

In this section we discuss a few minor extensions and small changes to the semantics of existing BPEL activities to round off the support for REST in the language.

As shown in the example, 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 remain accessible to clients even after the normal execution of the process has completed.

Given the absence of a static 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 [13, SA025]. In particular, 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.

[2] Similar to the BPEL isolated scope [13, Section 12.8]

6 Discussion

In BPEL for REST, the concept of business process is augmented with notion of resource. We have introduced the `resource` declarative construct to specify that at a certain point of the execution of a process, parts of its state can be published to clients using a RESTful Web service interface. Thus, it is important to understand what is the relationship between the state of a BPEL process instance and the state of such resources.

The lifecycle of typical BPEL process instances begins with the execution of a so-called *start activity*, which may be either an *instantiating receive* or a `pick` configured with a `createInstance="Yes"` attribute. 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 resources should follow the REST uniform interface principle. Resources are created (or initialized) with a POST/PUT request. Once a resource has been created, clients may read its current state using GET requests, update its state using PUT requests, and discard its state using DELETE requests.

In BPEL for REST, the state of a resource is accessed and manipulated from within the resource 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 [31, 30], 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., `/loan`) corresponding to a process model, if a POST request is answered with a 201 (Created) status code, the engine detects this and adds a `Location: /loan/i` redirection header (where `i` is a unique identifier of the newly created instance). Further GET, PUT, and DELETE requests from clients will have to be directed to the specific resource instance URI `/loan/i`. Nested resource URIs are still constructed by concatenating their `uri` attributes, but should now also include the resource instance identifier. A GET request to the original resource URI `/loan` could be then used to return hyperlink references to all active process instances managed by the engine.

BPEL for REST distinguishes between two aspects of a resource that can be published from a process: its URI and its state stored in the variables declared within the resource. 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 particular 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 also using the REST uniform interface.

All in all, the goal of the BPEL for REST extensions is to follow a declarative approach to embedding resources within processes so that developers do not have to worry about correlating requests with resource instances, a feature that should be handled transparently by the engine.

7 Related Work

BPEL for REST builds upon several existing research contributions within the area of Web service composition languages [32, 33]. Also, from the practical side, ad-hoc support for invoking RESTful Web services is currently being discussed for some BPEL engines.

In [30], BPEL is proposed as a suitable language to model the internal state of the resources published by RESTful Web services. To do so, BPEL scopes are used to represent different states of a resource and POST requests trigger the transition between different scopes/states. 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 exactly, the implementation of a resource state transition logic using BPEL).

BPEL^{light} [19] is an attempt to remove the tight coupling between BPEL and 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 take a similar, but less generic, approach, that introduces a specific set of *resource-oriented* activities to provide native and direct support for the interaction with RESTful Web services.

The idea of a RESTful engine API to access the execution state of workflow instances has been described in [34]. In this paper we provide explicit language support to control which parts of a process becomes exposed through a similar kind of API.

Bite [31] (or the IBM Project Zero assembly flow language) can be seen as a BPEL with a reduced set of activities specifically targeting the development of composite Web application workflows. As in 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 condensed within a single `<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). Still, the `<receive-reply>` activity in Bite can seen as a form of the combination `<resource><onPost>` in BPEL for REST, since it makes processes wait for client POST requests on a particular URI.

Within the Apache Orchestration Director Engine (ODE) project, a wiki-based discussion regarding RESTful BPEL extension has been recently started[3]. The proposed extension 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 declaring and instantiating resources is provided through extensions of the `<onEvent>` and `<receive>` activities. It is worth noting that this solution does not follow the one based on the WSDL 2.0 HTTP binding presented in Section 2. In this paper we proposed a different approach that clearly separates the RESTful activities from the standard ones used to interact with WSDL-based services.

8 Conclusion

This paper contributes to the ongoing discussion on how to best capture the notion of stateful resource in Web service composition languages originally based on the concepts of business processes and of message-based service invocation. It focuses on the research problems that stem from the interaction between two current emerging technologies: WS-BPEL and RESTful Web services. Given their lack of formally described interfaces and the possibility of not always using XML messages, RESTful Web services are challenging to compose through the WSDL-based invocation abstractions required by WS-BPEL. The paper presents in detail using the classical "loan application" example a new extension to the BPEL standard with the goal of providing native support for the composition of RESTful Web services. The extension turns the notion of "resource" and the basic RESTful interaction primitives (GET, POST, PUT, and DELETE) into first class language constructs. With these, a BPEL process can directly interact and manipulate the state of external resources and declaratively publish parts of its state through a RESTful Web service API.

Acknowledgements

This work is partially supported by the EU-IST-FP7-215605 (RESERVOIR) project. The author would also like to thank Domenico Bianculli and the anonymous reviewers for their insightful comments.

[3] Linked from `http://ode.apache.org/bpel-extensions.html`, last checked on June 13th, 2008.

References

1. Fielding, R.: Architectural Styles and The Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
2. Vinoski, S.: Serendipitous reuse. IEEE Internet Computing **12**(1) (2008) 84–87
3. O'Reilly, T.: REST vs. SOAP at Amazon. (April 2003) `http://www.oreillynet.com/pub/wlg/3005`.
4. Programmable Web: API Dashboard. (2007) `http://www.programmableweb.com/apis`.
5. Fielding, R.: A little REST and Relaxation. The International Conference on Java Technology (JAZOON07), Zurich, Switzerland. (June 2007) `http://www.parleys.com/display/PARLEYS/A\%20little\%20REST\%20and\%20Relaxation`.
6. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly (May 2007)
7. 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`.
8. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): generic syntax. IETF RFC 3986. (January 2005)
9. Florescu, D., Gruenhagen, A., Kossmann, D.: XL: An XML programming language for Web service specification and composition. In: Proc. of the 11th International World Wide Web Conference (WWW2002), Honululu, Hawaii, USA (May 2002)
10. Crockford, D.: JSON: The fat-free alternative to XML. In: Proc. of XML 2006, Boston, USA (December 2006) `http://www.json.org/fatfree.html`.
11. 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)
12. Haas, H.: Reconciling Web services and REST services. In: Proc. of ECOWS 2005, Växjö, Sweden (November 2005) Keynote Address
13. OASIS: Web Services Business Process Execution Language (WSBPEL) 2.0. (2006)
14. Assmann, U.: Invasive Software Composition. Springer (2003)
15. IBM, SAP: WS-BPEL Extension for Sub-Processes. (October 2005)
16. Active Endpoints, IBM, Oracle, SAP: WS-BPEL Extension for People. (August 2005)
17. Tan, W., Fong, L., Bobroff, N.: BPEL4JOB: A fault-handling design for job flow management. In: Proc. of the 5th International Conference on Service-Oriented Computing (ICSOC 2007), Vienna, Austria (2007)
18. Habich, D., Richly, S., Preissler, S., Grasselt, M., Lehner, W., Maier, A.: BPEL-DT - data-aware extension of BPEL to support data-intensive service applications. In: Emerging Web Services Technology, Vol. II, Birkhäuser (September 2008)
19. Nitzsche, J., van Lessen, T., Karastoyanova, D., Leymann, F.: BPEL^{light}. In: Proc. of the 5th International Conference on Business Process Management (BPM 2007). (September 2007)
20. Chinthaka, E.: REST and Web services in WSDL 2.0. (May 2007) `http://www.ibm.com/developerworks/webservices/library/ws-rest1/`.
21. Pasley, J.: Using Yahoo's REST services with BPEL. Cape Clear. (2008) `http://developer.capeclear.com/video/httpwizard/httpwizard.html`.
22. Snell, J.: Resource-oriented vs. activity-oriented Web services. IBM developerWorks. (October 2004) `http://www-128.ibm.com/developerworks/webservices/library/ws-restvsoap/`.
23. Vinoski, S.: Putting the "Web" into Web services: Interaction models, part 1: Current practice. IEEE Internet Computing **6**(3) (May-June 2002) 89–91

24. Vinoski, S.: Putting the ”Web” into Web services: Interaction models, part 2. IEEE Internet Computing **6**(4) (July 2002) 90–92
25. Wikipedia: Mashup (web application hybrid). `http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)`
26. Maximilien, M., Nielsen, D., Tai, S., eds.: 1st International Workshop on Web APIs and Services Mashups. (September 2007)
27. Schrenk, M.: Webbots, Spiders, and Screen Scrapers. No Starch Press (2007)
28. Hadley, M.J.: Web Application Description Language (WADL). (2006) `http://wadl.dev.java.net/`.
29. Barros, A., Dumas, M., ter Hofstede, A.H.: Service interaction patterns. In: Proc. of the 3rd International Conference on Business Process Management. Volume 3694 of LNCS., Nancy, France, Springer (2005)
30. Overdick, H.: Towards resource-oriented BPEL. In: 2nd ECOWS Workshop on Emerging Web Services Technology. (November 2007)
31. 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)
32. Dustdar, S., Schreiner, W.: A survey on web services composition. International Journal of Web and Grid Services (IJWGS) **1**(1) (2005) 1–30
33. Pautasso, C., Alonso, G.: From Web Service Composition to Megaprogramming. In: Proc. of the 5th VLDB Workshop on Technologies for E-Services (TES-04), Toronto, Canada (August 2004) 39–53
34. zur Muehlen, M., Nickerson, J.V., Swenson, K.D.: Developing web services choreography standards - the case of REST vs. SOAP. Decision Support Systems **40**(1) (July 2005) 9–29