Grid Service Composition in BPEL for Scientific Applications

Technical Report FIU-SCIS-2007-08-01

August 2007

Onyeka Ezenwoye, S. Masoud Sadjadi, Ariel Cary, and Michael Robinson

School of Computing and Information Sciences
Florida International University, 11200 SW 8th Street, Miami, FL 33199
{oezen001, sadjadi, acary001, mrobi002}@cs.fiu.edu

Abstract. Grid computing aims to create an accessible virtual supercomputer by integrating distributed computers to form a parallel infrastructure for processing applications. To enable service-oriented Grid computing, the Grid computing architecture was aligned with the current Web service technologies; thereby, making it possible for Grid applications to be exposed as Web services. The WSRF set of specifications standardized the association of state information with Web services (WS-Resource) while providing interfaces for the management of state data. Key to the realization of the benefits of Grid computing is the ability to integrate WS-Resources to create higher-level applications. The Business Process Execution Language (BPEL) is the leading standard for integrating Web services and as such has a natural affinity to the integration of Grid services. In this paper, we share our experience on using BPEL to integrate, create, and manage WS-Resources that implement the factory pattern. We use a Bioinformatics application as a case study to show how BPEL can be used to orchestrate Grid services. The execution environment for our case study comprises the *Globus Toolkit* as the Grid middleware and the *ActiveBPEL* as the BPEL engine. To the best of our knowledge, this work is among the handful approaches that successfully use BPEL for orchestrating WSRF-based services and the only one that includes the discovery and management of instances.

Keywords: BPEL, Grid Computing, WSRF, OGSA-DAI, Service Composition.

1 Introduction

Grid computing promises to harness the resources available on disparate distributed computing environments to create a parallel infrastructure that allows for applications to be processed in a distributed manner. The goal is to create an accessible virtual supercomputer by integrating distributed computers with the use of open standards [1]. To this end, the Open Grid Services Architecture (OGSA) [2], developed by Global Grid Forum (GGF), defines an architecture for service-oriented Grid computing; GGF no longer exists, but the OGSA architecture is still current. This architecture utilizes Web services standards such as XML [3], SOAP [4] and WSDL [5].

Under the OGSA, computational and storage resources are exposed as an extensible set of networked services that can be aggregated to create higher-function applications. These Grid services, which are sometimes transient, adhere to a set of OGSA-defined conventions for creation, lifetime management, discovery and change management [6]. Aligned with these conventions is the Web Services Resource Framework (WSRF). WSRF is a set of specifications that are defined in terms of existing Web services technologies, for modeling and management of stateful resources. The specification defines a set of interfaces that Grid services may implement. These interfaces which address issues like dynamic service creation, lifetime management, notification, and manageability, allow applications to interact with Grid services in standard and interoperable ways [7].

Key to the realization of the benefits of Grid computing is the ability to integrate basic services to create higher-level applications. We argue these higher-level applications will provide the right level of abstraction for the non-computer scientists. Thus, allowing them to concentrate on their domain specific work instead of the technical issues of integrating tools. Workflow languages permit such aggregation of services. With such languages, higher-level application can be modeled as graphs where the nodes represent tasks while the edges represent inter-task dependencies, data flow or flow control. Tasks may be performed by basic services. The Business Process Execution Language (BPEL) [8] has become the leading language for the aggregation of Web services. In this paper, we share our experience in using BPEL to compose WSRF-based Grid services to create a bioinformatics application for protein sequence matching. We show how BPEL can be used to interact with WSRF-based services that implement the factory/instance pattern [9].

The rest of this paper is structured as follows. Section 3 covers WSRF and some of its component specifications. Section 4 provides a brief overview of service orchestration and BPEL. Section 5 presents the bioinformatics application and Section 6 shows how BPEL is used to integrate Grid services to create the application. Section 7 covers the execution environment . Sections 8 and 9 provide some related work and conclusion, respectively.

2 Background on Web Services

A *Web service* is a program delivered over the Internet that provides a service described in the Web Service Description Language (WSDL) [5] and communicates with other programs, typically through SOAP messages [4]. Web services provide the desired abstraction uniformity that is needed to bridge applications regardless of the heterogeneity of platforms and implementation languages. They provide a middleware layer that is relatively lightweight and have neither the object model nor programming language restrictions imposed by other traditional middleware systems. WSDL and SOAP are both independent of specific platforms, programming languages, and middleware technologies. Moreover, SOAP leverages the optional use of the HTTP protocol, which can bypass firewalls, thereby enabling Internet-wide application integration.

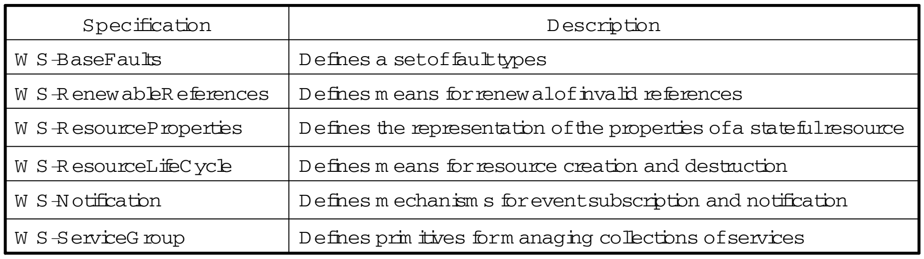
Web services, which are typically used to represent reusable business functions (*e.g.*, flight reservation), can be the building blocks of more complex business processes. Although complex business processes can be developed in general-purpose languages such as Java and C++, such languages do not provide high-level constructs to readily define workflow processes that represent composite Web services. Business Process Execution Language (BPEL) is a high-level workflow language that can be used to create coarse-grained business processes that constitute a number of related business functions [8, 10, 11]. By representing a workflow that coordinates activities among other Web services, BPEL allows for the creation of coarse-grained Web services by wiring together activities that can invoke other Web services, manipulate data and handle exceptions. BPEL and Web services make it possible for organizations to deploy flexible Service Oriented Applications (SOA) [12, 13]. SOA-based integration permits the discovery and use of existing resources and thereby reducing the cost and speed with which applications can be developed.

3 Web Services Resource Framework

In 2003, the Global Grid Forum, a working group for the standardization of Grid computing, released the specification for the Open Grid Services Infrastructure (OGSI). As of 2006, GGF was merged with the Enterprise Grid Alliance to form the Open Grid Forum - a global standarization entity for both industrial and academic grid computing. OGSI is a set of conventions and extensions on the use of Web Service Definition Language (WSDL) and XML Schema to enable the modeling and management of stateful Web services. The ultimate goal of the specification is to encourage openness of the Grid infrastructure by aligning the Grid computing model with the emerging Web services architecture. The OGSI specification addresses issues concerning creation and management of the lifetime of instances of services, declaration and inspection of service state data, notification of service state change and standardiztion of service invocation faults.

In 2004, the Organization for the Advancement of Structured Information Standards (OASIS) refactored the OGSI specification into the Web Services Resource Framework (WSRF). This framework standardizes the concept of Web Services Resource (WS-Resource) [14], which is, the association of a state component with a Web service. This association permits, through *standardized interfaces*, the manipulation of the *named typed* state component as part of the execution of the Web service. This creates the impression of statefulness of that Web service. The stateful component is seen as implicit input in the execution message exchange of the associated Web service (*implied resource pattern*). The term pattern is used because the relationship between the Web service and the stateful resource is defined using a set of conventions on existing Web services technologies [14].

WSRF addresses some of the criticisms [15] of OGSI such as; the specification was too monolithic and did not allow for flexible incremental adoption and exten-



Specification	Description
WS-BaseFaults	Defines a set of fault types
WS-RenewableReferences	Defines means for renewal of invalid references
WS-ResourceProperties	Defines the representation of the properties of a stateful resource
WS-ResourceLifeCycle	Defines means for resource creation and destruction
WS-Notification	Defines mechanisms for event subscription and notification
WS-ServiceGroup	Defines primitives for managing collections of services

Fig. 1. WSRF component specifications.

sions to WSDL 1.1. Also, the and non-standard use of XML schema meant that OGSI did not work well with existing Web services and XML tooling. OGSI was seen as too object-oriented by coupling the service and the stateful resource it acts upon as one entity. WSRF maintains all the functions of OGSI but incorporates some existing Web services technologies. WSRF partitions its functionality into distinct component specifications, namely, *WS-RenewableReferences*, *WS-ResourceProperties*, *WS-ResourceLifeCycle*, *WS-Notification*, *WS-ServiceGroup* and *WS-BaseFaults*. With this separation, developers can now choose which of the specifications to use. Figure 1 contains a brief description of the specifications. WSRF now supercedes the initial OGSI specification, thereby rendering it obsolete. Figure 2 depicts the relationship between the Web services technologies, OGSI and the WSRF specifications. The WSRF leverages the other specifications in the stack.

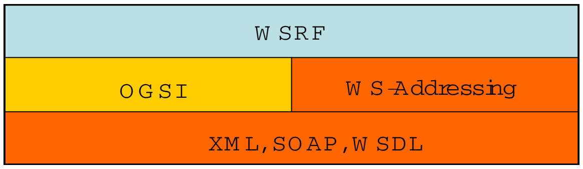


Fig. 2. The relationship between WSRF, OGSI and Web services technologies.

An instance of a stateful resource may be created by the use of a WS-Resource factory. This factory is any Web service capable of instantiating the stateful component. The factory implies a use pattern that may be implemented in several Web service operations. To instantiate a stateful resource, the Web service has to create a new stateful resource, assign an identity to that resource and create the association between the resource and its Web service. The factory returns an *endpoint reference*, which contains the identifier that refers to the new stateful resource.

A thorough description of the component specifications of WSRF is beyond the scope of this paper. However, a brief overview of *WS-ResourceProperties* and *WS-Addressing* helps to provide context for the material that follows.

3.1 Web Services Resource Properties

Resource properties refer to the individual components of a Web service state. This state is described in a WS-Resource properties document. Each resource property is represented as an XML element in this document. The *WS-Resource-Properties* specification standardizes the way properties of a resource can be defined as part of a Web service interface [14]. It also defines mechanisms for querying and modifying those resource properties through Web service interfaces. When defining the interface to a Web service, the *portType* component, which describes a set of operations and messages of a service (WSDL 1.1), can be associated with at most one resource property document, any Web service that implements this portType is therefore associated with the stateful resource defined by the resource property document [14]. Service users may determine the resource type and ways to access or modify its properties by retrieving the WSDL definition of the associated Web service.

3.2 Web Services Addressing

The Web services Addressing (WS-Addressing) [16] specification standardizes the endpoint reference construct. The specification was designed to provide an interoperable and transport-neutral way of encoding addressing information. Here, *transport* refers to the transport layer of the network reference model. This enables messaging systems to support message transmission through networks. WS-Addressing defines two constructs to convey the necessary information, they are: endpoint reference and message information header.

Endpoint Reference. The identifier for a stateful resource may be dynamically associated with a Web service during message exchange execution. This dynamic association eliminates the need to know the identity and location of the resource encapsulated by the Web service. The identifier is encoded in an Endpoint Reference (EPR) and is used to address a target WS-Resource. An Endpoint Reference [16] is an XML structure that encapsulates the information needed to route messages to their destination Web service. This Endpoint References may be returned by the factory that creates a new WS-Resource and may contain, in addition to the address, other metadata such as *reference properties*. It is within the reference properties that the key for identifying a resource instance is encapsulated. Figure 3 shows a WS-Addressing endpoint reference as used within the conventions of WS-Resource. The endpoint reference contains two components: (1) The *Address* component encapsulates the network transport-specific address of the Web service; (2) the *ReferenceProperties* component contains a stateful resource identifier.

Message Information Header. Message information headers carry end-to-end message characteristics including addressing for source and destination endpoints as well as message identity [14]. A request message directed to a service using an endpoint reference must include the stateful resource identifier. Since WS-Addressing specifies that ReferenceProperties elements must appear as SOAP

```
  <EndpointReference>
    <Address>http://host/wsrf/Service</Address>
    <ReferenceProperties>
      <ResourceKey>8807d620</ResourceKey>
    </ReferenceProperties>
  </EndpointReference>
```

Fig. 3. A WS-Resource-qualified Endpoint Reference.

header elements in the message, the stateful resource identifier is mapped to this message information header. Figure 4 illustrates the use of a SOAP header to propagate the stateful resource identifier. The target Web service will extract this identifier from the SOAP message and use it to locate the stateful resource needed for the execution of the request. The `<Action>` element of the header identifies the verb of the message.

```
<Envelope>
  <Header>
    <MessageID>...</MessageID>
    <To>http://host/wsrf/Service</To>
    <ResourceKey>8807d620</ResourceKey>
    <Action>...</Action>
    <From>...</From>
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

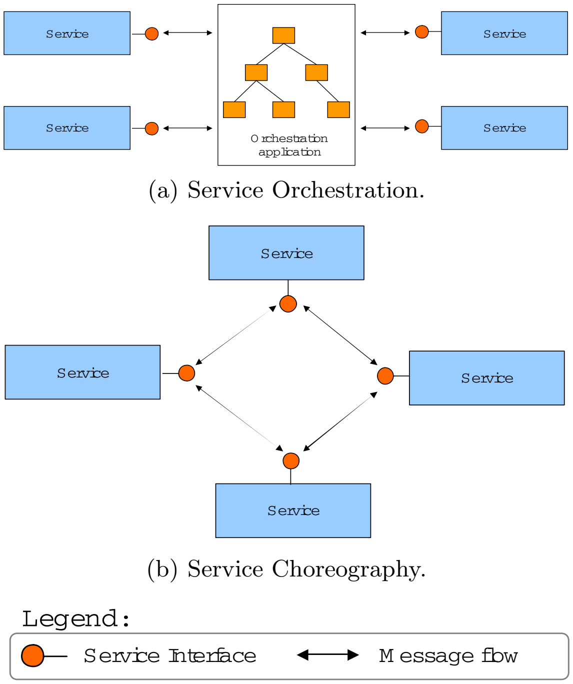
Fig. 4. A SOAP request message carrying a resource identifier in its information header.

4 Service Orchestration

Service-Oriented Architecture implies that distributed software services can be loosely integrated through clearly defined interfaces. This integration is often achieved through service *orchestration* or *choreography* [17]. In service *orchestration*, integration is achieved through a central application. This application (usually an executable workflow) models the interaction between the integrated services. The application is aware of the interfaces of the services and controls their execution order and message exchange. In service *choreography*, there may not exist a central controlling process, rather interacting services are aware of each other and each service knows of its participation in the message exchanges of the interaction. Figure 5 illustrates the difference between orchestration and choreography. To create service orchestration, a workflow language such as Business Process Execution Language (BPEL) is often used.

4.1 Business Process Execution Language (BPEL)

BPEL has become the leading standard for Web service orchestration. With BPEL, Web services can be integrated, using some XML-grammer, to create



(a) Service Orchestration.

(b) Service Choreography.

Fig. 5. Comparing orchestration and choreography.

a higher-level application (business process). The XML-grammer that defines a BPEL process is interpreted and executed by an orchestration engine which exposes the process as a Web service. BPEL, which is built on XML-Schema, WSDL and XPath, weaves together *basic* and *structured* activities to create the logic of the process. BPEL provides many constructs for the management of process activities, including loops, conditional branching, fault handling and event handling (such as timeout). It allows for activities to execute sequentially or in pararrel.

A *basic activity* is a primitive BPEL activity that performs some atomic action that interacts with other Web services outside of the BPEL process. These activities include the `invoke` activity that performs an operation on an external Web service partner. Some others are, the `receive` which waits for external input from a partner and the `reply` activity which send back a message to the partner that invoked the `receive` operation. *Structured activities* are those activities that derived from a combination of several activities. Structured activities aim to specify the order in which the combined activities execute. They provide support for asynchronous interactions which is important for efficiency and scalability. For instance, activities that are meant to execute concurrently or sequentially are enclosed in `flow` and `sequence` tags, respectively. Other structure activities include `switch` for conditional branching, `pick` for alternative choices and a `while` loop construct. Structured activities can be nested and the execution order between blocks of activities can be defined with the use of links.

BPEL processes can manage exceptions generated from service invocations. Faults can be generated externally by invoked Web service partners or internally within the process. Faults are caught with code defined inside fault handlers. In addition, BPEL activities can be grouped into a single transaction with the `scope` tag. The `scope` construct provides a context for a subset of activities. It can contain fault and event handler for activities nested within it. The `scope` allows the activities enclosed within it to be managed as a logical unit. With the need for scientists to create application that integrate multiple Grids service and the alignment of the Grid service paradigm with Web services technologies, BPEL is now poised to play an important role in the orchestration of Grid services.

5 WSRF Services for Bioinformatics

In Bioinformatics, one of the most critical jobs is Computational Biology, which is, the analysis of biological data. In this sections, we use a Grid application that we developed for computational biology as the case study to demonstrate the use of BPEL in the orchestration of WSRF-based Grid services. This application attempts to match protein sequences [18]. This matching of sequences can be computationally intensive depending on the size of the sequences processed. Figure 6 shows the high-level architecture of the application. Below, we explain some of the components of this architecture.

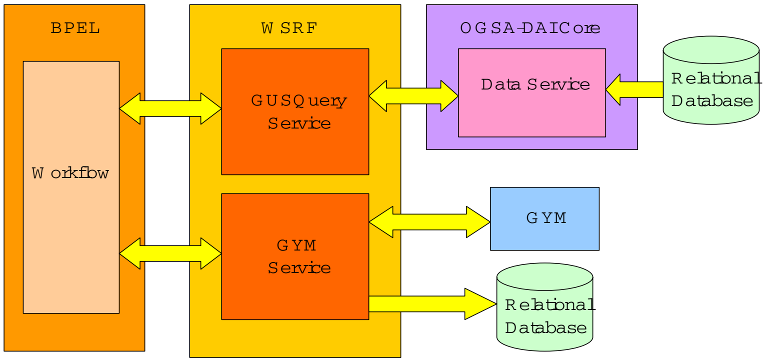


Fig. 6. The high-level architecture of the application.

OGSA-DAI. Biological data is stored in a variety of formats such as flat files, relational databases and XML repositories. Some of these data repositories are platform and/or language dependent, which makes it difficult to integrate in new environments. So, there is a need to seamlessly access these disparate sources of information and integrate them into the Grid for further processing. OGSA-DAI [19] is middleware that facilitates the access and integration of data from separate sources in a Grid computing environment. OGSA-DAI makes data sources accessible via Web services (Data services).

GYM. GYM is a biological application for processing protein data sequences. Here, GYM is used to detect Helix-Turn-Helix (HTH) Motifs [18] in protein sequences. Proteins with HTH motifs are usually responsible for binding with DNA [18]. The GYM program is a legacy application written in C. It takes as input, a sequence of protein data. GYM instances are run on Grid nodes to process the sequences available from the sources.

GUSQuery Service. This is a WS-Resource. The resource in this case is the protein sequence data obtained from an OGSA-DAI data service. This service contains a search method which takes as input a range of sequences and the location of the data service through which to get those sequences. In adherence to the WSRF specification, this service is accompanied by a *factory* service. The factory service, called *GUSQueryFactory*, contains a create method which creates an instance of the GUSQuery Service. The factory service returns an endpoint reference which identifies the created instance.

GYM Service. This is also a WS-Resource. It contains a method which takes as input a range of protein sequence data. This data is then processed locally using a GYM application. The result from the GYM application is stored in a database. This service also has a factory service called *GymFactory*.

Workflow. This BPEL process weaves together the interaction between the GUSQuery service and the GYM service. This executable workflow, which is exposed as a Web service, is also responsible for creating the instances of those services through their respective factory services. It uses the endpoint references returned by those factories to identify the specific service instances. It passes the protein sequence data from the GUSQuery service to the GYM service. Figure 7 is a sequence diagram that shows the interaction between the workflow and the services. The first step is to use the GUSQueryFactory to create an instance of GUSQueryService, this operation returns the endpoint reference identifier for the instance. The GUSQueryService instance is then invoked to retrieve a set of protein sequences. These sequences are then sent to the GymService for processing (step 4) , just after an instance of that service is created (step 3). The last step retrieves the result of that sequence processing.

As stated in Section 4, BPEL is an *orchestration* language and which means, it has a centralized architecture. Thus, all message and data exchange between collaborating services go through the BPEL process. Although this presents a limitation, especially in a Grid environment where it might be necessary to move large amounts of data, it is possible to implement a BPEL process that combines service orchestration and choreography. To achieve this hybrid, rather than moving large amounts of data through the BPEL process, the reference to the data is passed. The services in the collaboration can then transfer the data amongst themselves using applications such as GridFTP.

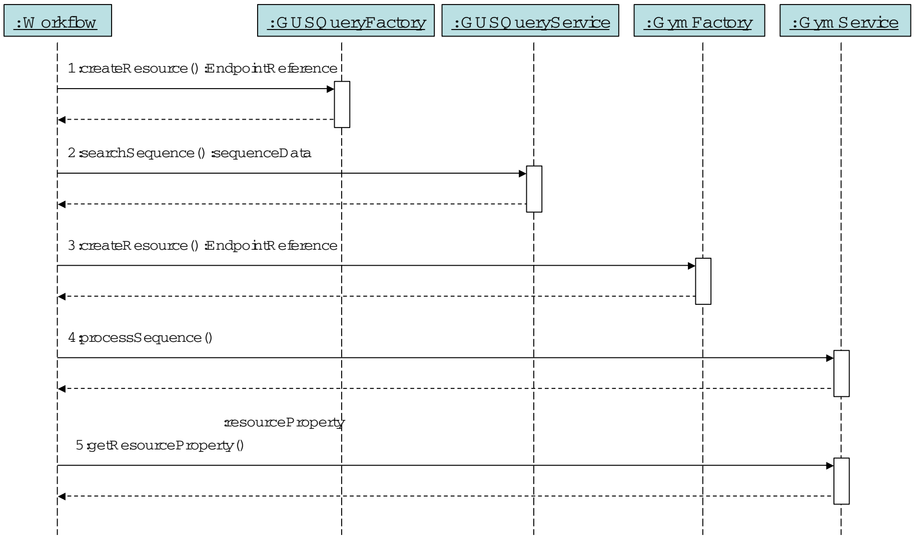


Fig. 7. The sequence diagram for workflow.

6 WSRF with BPEL

To fully immerse BPEL as part of any Grid environment, it is necessary not only to use BPEL to integrate WSRF-based Grid services but also to expose the BPEL process itself as a WS-Resource. Although trying to expose a BPEL process as a WS-Resource does present some obstacles, since WSRF is not considered in the current BPEL specification, this can be achieved through language extensions and mediators [20] [21]. In this paper, BPEL is used solely for the integration of WSRF-based Grid services.

In this section, we show how the interaction with the WSRF-based services (shown in figure 7) is achieved in BPEL. The details about the definition of the services themselves, is outside the scope of this paper. Where necessary, some code have been simplified for brevity.

6.1 Partner links

The partner links define the different services that interact with the BPEL process. The `<partnerLinks>` section in the code listing below shows the four services that interact with the workflow.

```
<partnerLinks>
   ...
   <partnerLink name="gym"
      partnerLinkType="GymPortTypeLink"
      partnerRole="GymPortTypeProvider"/>
   <partnerLink name="GusFactory"
      partnerLinkType="GusFactoryPortTypeLink"
      partnerRole="GusFactoryPortTypeProvider"/>
   <partnerLink name="gus"
      partnerLinkType="GUSQueryPortTypeLink"
      partnerRole="GUSQueryPortTypeProvider"/>
   <partnerLink name="GymFactory"
      partnerLinkType="GymFactoryPortTypeLink"
      partnerRole="GymFactoryPortTypeProvider"/>
</partnerLinks>
```

The partnerLinks here correspond to the GYM service (gym), as well as the factory for that service (GymFactory), the GUSQuery service (gus) and its factory service (GusFactory). The partner link type and a role name in each partner link identifies the functionality that must be provided by the by the partner services, that is, the interface (portType) that the partners need to implement [22].

6.2 Variables

The code below shows some of the message variables used by the BPEL process. The variables are defined in terms of WSDL message types, XML Schema simple types, or XML Schema elements. These variables are used in messages exchanged with partner services.

```
<variables>
   ...
   <!-- some variables GUSQuery service & factory -->
   <variable messageType="ns4:CreateResourceRequest"
      name="CreateResourceRequest"/>
   <variable messageType="ns4:CreateResourceResponse"
      name="CreateResourceResponse"/>
   <variable messageType="ns2:searchSequenceRequest"
      name="gsuRequest"/>
   <variable messageType="ns2:searchSequenceResponse"
      name="gusResponse"/>
   <!-- some variables GYM service & factory -->
   <variable messageType="ns5:GetResourcePropertyRequest"
      name="GetResourcePropertyRequest"/>
   <variable messageType="ns5:GetResourcePropertyResponse"
      name="GetResourcePropertyResponse"/>
   <!-- variable for endpoint reference -->
   <variable element="wsa:EndpointReference"
      name="DynamicEndpointRef"/>
</variables>
```

6.3 Creating a Web service instance

Creating a new Web service resource instance involves making a call to the *createResource* operation of designated factory service. This is achieved by using BPEL's service invocation mechanism. The `<invoke>` construct allows a BPEL process to invoke a one-way or request-response operation on a portType (interface) offered by a partner service [22]. The code listing below shows the invocation to the `createResource` operation of the GUSQuery factory service.

```
<invoke
   name="InvokeGusFactory"
   partnerLink="GusFactory"
   portType="GusFactoryPortType">
   operation="createResource"
   inputVariable="CreateResourceRequest"
   outputVariable="CreateResourceResponse"
</invoke>
```

The invoke activity which makes a synchronous call to the factory service, contains the `portType` of the operation as well as the `inputVariable` and `outputVariable` variables. If the invocation is successful, the `outputVariable` will contain the endpoint reference of the created instance. Below is an example SOAP message returned after a successful invocation to the factory service.

```
<soapenv:Envelope>
   ...
   <soapenv:Body>
      <createResourceResponse xmlns="...">
         <wsa:EndpointReference>
            <wsa:Address>
               http://.../wsrf/services/GUSQueryService
            </wsa:Address>
            <wsa:ReferenceProperties>
               <ns1:GUSQueryResourceKey xmlns:ns1="...">
                  8807d620-acb3-11db-9abe-b9e88f054119
               </ns1:GUSQueryResourceKey>
            </wsa:ReferenceProperties>
            ...
         </wsa:EndpointReference>
      </createResourceResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

In this message can be seen the `EndpointReference` element which contains the `Address` and resource key (`GUSQueryResourceKey`) to the created service instance.

6.4 Invoking the Web service instance

Since the identifier of a WS-Resource instance is obtained at runtime, any message to this instance must contain the resource identifier in its SOAP header (as described in Section 3.2). Note that, although the instance of the service is only known at runtime, the service would have been declared as a partner at development time (see Section 6.1). The BPEL specification allows for the actual service endpoint of a partner to be dynamically defined within the process. The specification however, does not make provisions for how dynamically obtained information such as resource identifiers can be define for those endpoints. This type of information needs to be mapped to the headers of the SOAP messages for the target endpoint. Because the BPEL specification is deficient in this regard, the method mapping desired information to SOAP headers depends on the specific implementation of the BPEL execution engine. The method me describe below is suited for the ActiveBPEL Engine (see Section 7.2)

To dynamically associate an endpoint reference to a service, the WS-Addressing endpoint reference [16] is used to represent the dynamic data required to describe a partner service endpoint [22]. To achieve the association of a partner with its service endpoint, an endpoint reference has to be assigned to the declared partner link within the process. As shown below, we use the `copy` operation of an assignment activity to copy *literally* an endpoint reference to a variable (DynamicEndpointRef).

```
<copy>
   <from>
      <wsa:EndpointReference xmlns:s="...">
         <wsa:Address/>
         <wsa:ServiceName PortName="GUSQueryPortType">
            s:GUSQueryService
         </wsa:ServiceName>
         <wsa:ReferenceProperties>
            <!--Elements to be mapped to the SOAP Header-->
            <wsa:Action/>
            <wsa:To/>
            <wsa:From/>
            <ns2:GUSQueryResourceKey/>
         </wsa:ReferenceProperties>
      </wsa:EndpointReference>
   </from>
   <to variable="DynamicEndpointRef"/>
</copy>
```

This endpoint reference contains an Address element that will hold the service endpoint address. The ReferenceProperties of the endpoint reference contains some WS-Addressing message information header elements and a GUSQueryResourceKey element. The GUSQueryResourceKey element will hold the resource identifier for the WS-Resource. Values for the endpoint reference will be assigned at run time. The message information header elements and the GUSQueryResourceKey will be mapped, by the BPEL engine to the invocation SOAP message for the partner Web service, which in this case is GUSQueryService.

The WS-Resource identifier information required for the endpoint reference is copied from the reply message of their respective factory services. The copy operation below copies the service endpoint address from the factory response message (CreateResourceResponse) to the endpoint variable (DynamicEndpointRef). The query attribute of the <from> and <to> clauses are XPath [23] queries. XPath queries are used to select a field within a source or target variable part.

```
<copy>
   <from variable="CreateResourceResponse"
      part="response"
      query="/ns4:createResourceResponse
               /wsa:EndpointReference/wsa:Address"/>
   <to variable="DynamicEndpointRef"
      query="/wsa:EndpointReference
               /wsa:ReferenceProperties/wsa:To"/>
</copy>
```

A similar mechanism is used to assign the service endpoint address to the <wsa:Address> property of the endpoint reference variable. The BPEL engine needs this address to determine the destination of the invocation message for the service. The <wsa:To> component of the message information header is used by the service to determine the endpoint of the required service instance. Here, we use the same address returned by the factory because for this application, the address of a service and its instance are the same.

```
<copy>
```

```
   <from variable="CreateResourceResponse"
      part="response"
      query="/ns4:createResourceResponse
               /wsa:EndpointReference/wsa:Address"/>
   <to variable="DynamicEndpointRef"
      query="/wsa:EndpointReference
               /wsa:Address"/>
</copy>
```

The name of the operation to be invoked on the WS-Resource instance needs to be assigned to the `Action` part of the SOAP header. To achieve this, an XPath expression to write the name as a string to the endpoint reference variable. An XPath expression, which is specified in an expression attribute in the `<from>` clause, is used to indicate a value to be stored in a variable. The string that represents the operation, is in the for of a URI that includes the target namespace of the WSDL document for the WS-Resource and the associated portType. Thus in the listing below, `http://GUSQueryService_instance` is the namespace, `GUSQueryPortType` is the portType and `searchSequence` is the operation.

```
<copy>
   <from expression="string('
      http://GUSQueryService_instance
      /GUSQueryPortType/searchSequence')"/>
   <to variable="DynamicEndpointRef"
      query="/wsa:EndpointReference
      /wsa:ReferenceProperties/wsa:Action" />
</copy>
```

The listing below shows how we use the `copy` operation and XPath queries to copy the resource instance key (`GUSQueryResourceKey`) from the factory response message to the endpoint reference variable.

```
<copy>
   <from variable="CreateResourceResponse" part="response"
      query="/ns4:createResourceResponse
      /wsa:EndpointReference/wsa:ReferenceProperties
      /ns2:GUSQueryResourceKey"/>
   <to variable="DynamicEndpointRef"
      query="/wsa:EndpointReference/wsa:ReferenceProperties
      /ns2:GUSQueryResourceKey"/>
</copy>
```

The `<wsa:From>` property of the message information header identifies the source of the meassage, this property can be set with the WS-Addressing ”anonymous” endpoint URI [16]. This anonymous endpoint URI can be used because the invocation to the resource instance is synchronous in this case and the underlying message layer takes care of delivering replies to the source.

```
<copy>
   <from>
      <From xmlns="...">
         <Address>
            http://schemas.xmlsoap.org/ws/2004/03
                           /addressing/role/anonymous
         </Address>
      </From>
   </from>
   <to variable="DynamicEndpointRef"
      query="/wsa:EndpointReference
               /wsa:ReferenceProperties/wsa:From"/>
</copy>
```

After assigning values to all the necessary parts of the endpoint reference variable, an association is now made with this variable and the desired partner link. As shown below, a `copy` operation is used to copy the endpoint reference variable (`DynamicEndpointRef`) to the predefined partner link. An invocation can now be made to the Web service (WS-Resource) partner (GUSQuery service). The information carried in the SOAP message header of the invocation is used to identify the appropriate instance of this service.

```
<copy>
   <from variable="DynamicEndpointRef"/>
   <to partnerLink="gus"/>
</copy>
```

6.5 Accessing resource properties

The *WS-ResourceProperties* specification includes a set of port types for querying and modifying the state of a WS-Resource. The Gym service (Section 5) implements the *GetResourceProperty* port type of this specification. We use this port type and its operation (also called *GetResourceProperty*) to access the result from the Gym application (Section 5). Prior to invoking the *GetResourceProperty* operation, some initialization needs to be made to the variable of its input message. This initialization includes the name of the resource property to which we want to retrieve the value. In our case, this resource property is called *result*. The listing below shows how we initialize the *GetResourcePropertyRequest* in the BPEL process. The `<from>` clause includes (as attributes) the target namespace of the WSDL documents that contain the definitions for the *GetResourceProperty* port type and the *result* resource property.

```
<copy>
   <from>
      <GetResourceProperty
      xmlns="http://docs.oasis-open.org/wsrf/2004/06
      /wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
      xmlns:ns3="http://GymService_instance">
         ns3:result
      </GetResourceProperty>
   </from>
<to variable="GetResourcePropertyRequest"
      part="GetResourcePropertyRequest"/>
</copy>
```

The *WS-ResourceProperties* specification also includes the *GetMultipleResourceProperty* port type for retrieving the values of multiple resource properties. To make an invocation the operation of this port type, the variable message initialization must include the list of all the resources properties, each encapsulated within a `</ResourceProperty>` element.

```
<copy>
   <from>
      <GetMultipleResourceProperty
         xmlns="http://docs.oasis-open.org/wsrf/2004/06
            /wsrf-WS-ResourceProperties-1.2-draft-01.xsd">
         <ResourceProperty
            xmlns:ns3="http://GymService_instance">
               ns3:result
         </ResourceProperty>
         <ResourceProperty xmlns:ns1="...">
            ...
         </ResourceProperty>
      </GetMultipleResourceProperty>
   </from>
   <to .../>
</copy>
```

Because we are trying to access the resource properties of a WS-Resource instance, assignments need to be made to all parts of the message header necessary for identifying the instance. The way to do this is described in Section 6.4, the only difference now is in the URI that specifies the verb of the invocation message.

```
<copy>
   <from expression="string('http://docs.oasis-open.org
         /wsrf/2004/06/wsrf-WS-ResourceProperties
         /GetResourceProperty')"/>
   <to variable="DynamicEndpointRef"
         query="/wsa:EndpointReference
         /wsa:ReferenceProperties/wsa:Action"/>
</copy>
```

Although we do not implement all the WSRF port types, the examples shown above is sufficient to enable the orchestration of WSRF-based services that implement the factory/instance pattern. In this section, we showed how the interaction with the WSRF-based services (shown in figure 7) is achieved using BPEL as the composition language.

7 Execution Environment

The Grid service that make up our application are deployed under the Globus Toolkit version 4 [24]. The BPEL engine we use is AcitveBpel version 3.0 [25]. In the rest of this section, we present a brief overview of these tools.

7.1 The Globus toolkit

The Globus Toolkit is an open source toolkit for building Grids. Version 4.0 [24] of this toolkit implements the WSRF specification. The software is a set of components that can be used to develop Grid applications. The toolkit includes software services and libraries that address issues like resource monitoring, discovery, management, security, data management, fault detection, etc. It allows users to access remote resources as if they were located locally. Globus Toolkit has become the de facto standard for building Grid solutions.

7.2 The ActiveBPEL engine

The ActiveBPEL engine [25] is a Java-based open source implementation of a BPEL engine. It reads BPEL process definitions (and WSDL files) and exposed the BPEL process as a Web service. Incoming messages may trigger start activities which causes the engine to create new instances of the process. The engine manages persistence, queues, alarms, and other execution details on behalf of the process. The version of ActiveBPEL engine we used runs under an Apache Tomcat servlet container.

8 Related Work

Duscher [26] present the architecture of an execution environment for mathematical services using Web service technologies. WSRF is used for the architecture and BPEL is used to represent the workflow that specifies the interactions with the mathematical services. The execution environment comprises a mathematical service (GAPService) and a BPEL engine. The GAPService implements both a service and factory interface. A part of the GAPService manages the lifecycle of the instances. This work differs from ours because, although the creation of service instances is done through the BPEL workflow, the discovery and management of those instances is not. Also, their Web services do no execute as part of a Grid environment.

Various works [27, 28] have been done to address the issue of using BPEL to orchestrate Grid services. However, these works do not cover WSRF-based Grid services and the implied resource pattern. Others [20, 21], discuss the issues involved in exposing BPEL processes as WSRF-based Grid services.

9 Conclusion

In this paper, we discussed and explained how BPEL can be used as a language for integrating WSRF-based Grid services. In a case study, we demonstrated how some WSRF-based Grid services can be integrated to create a Bioinformatics application that is used to detect Helix-Turn-Helix motifs in protein sequences. The integrated WSRF services implement the factory pattern [9]. We showed how BPEL can be used to create, discover and manage WS-Resource instances. The centralized nature of data movement in BPEL presents a problem for high-performance computing, however, this limitation can be remedied by using techniques that enable the direct transfer of data between partner services. Also, the BPEL specification does not make provisions for how dynamically obtained information such as resource identifiers, usernames and passwords can be specified within SOAP message headers. The method of achieving this is left open to the implementation of the various BPEL engines. Therefore, there is a need for standardization in this regards for BPEL process to remain portable and assume its place as the language for orchestrating Grid services.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: Enabling scalable virtual organizations. Lecture Notes in Computer Science **2150** (2001)
2. The Open Grid Services Architecture, http://www.globus.org/ogsa/.
3. Yergeau, F., Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E.: Extensible Markup Language (XML) 1.0 (Third Edition). W3C. (2004)
4. M. Gudgin et al.: SOAP Version 1.2. W3C. 1.2 edn. (2003)
5. R. Chinnici et al.: Web Services Description Language (WSDL) Version 2.0. W3C. 2.0 edn. (2004)
6. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: Grid services for distributed system integration. Computer **35**(6) (2002) 37–46
7. Web Services Resource Framework, http://www.globus.org/wsrf/.
8. Weerawarana, S., Curbera, F.: Business process with BPEL4WS: Understanding. Online article (2002)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY (1995)
10. Sherman, D.: Business flows with BPEL4WS. Online article (2005) URL: http://xml.sys-con.com/read/39780.htm.
11. Leymann, F., Roller, D., Schmidt, M.T.: Web services and business process management. IBM Systems Journal **41**(2) (2002)
12. Kreger, H.: Web Services Conceptual Architecture (WSCA 1.0). IBM Software Group. (2001) Available at URL: http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf.
13. D. Booth et al. : Web Services Architecture. W3C. (2004)
14. Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Sedukhin, I., Snelling, D., Storey, T., Vambenepe, W., Weerawaran, S.: Modeling stateful resources with Web services. (2004)
15. Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Maguire, T., Snelling, D., Tuecke, S.: From OGSI to WS-Resource framework: Refactoring and evolution. (2004)
16. Web Services Addressing (WS-Addressing), http://www.w3.org/Submission/ws-addressing/.
17. Peltz, C.: Web services orchestration and choreography. IEEE Computer **36**(10) (2003) 44–52
18. Narasimhan, G., Bu, C., Gao, Y., Wang, X., Xu, N., Mathee, K.: Mining for motifs in protein sequences. Journal of Computational Biology **9**(5) (2002) 707–720
19. The OGSA-DAI Project, http://www.ogsadai.org.uk/.
20. Leymann, F.: Choreography for the grid: towards fitting BPEL to the resource framework: Research articles. Concurr. Comput. : Pract. Exper. **18**(10) (2006) 1201–1217
21. Slominski, A.: On using BPEL extensibility to implement OGSI and WSRF grid workflows. In: GGF10 Workshop on Workflow in Grid Systems, Berlin, Germany (2004)
22. Andrews, T., Curbers, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thattee, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services version 1.1. (2003)
23. XML Path Language (XPath), http://www.w3.org/TR/xpath.
24. The Globus Toolkit, version 4, http://www.globus.org.

25. ActiveBPEL 3.0, `http://www.activebpel.org`.
26. Duscher, A.: An execution environment for mathematical services based on WSRF and BPEL. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria (2005)
27. Kuo-Ming Chao, Muhammad Younas, N.G.: BPEL4WS-based coordination of grid services in cooperative design. Computers in Industry **57** (2006) 778–786
28. Emmerich, W., Butchart, B., Chen, L., Wassermann, B., Price, S.L.: Grid service orchestration using the business process execution language BPEL. Journal of Grid Computing **3**(3-4) (2005) 283–304