

Adapting BPEL to Scientific Workflows

Aleksander Slominski¹

Department of Computer Science, School of Informatics, Indiana University
aslom@cs.indiana.edu

Summary. The Business Process Execution Language for Web services (BPEL) has emerged as a leading workflow language for composing business Web services. In this chapter we list generic requirements for a Grid workflow language and look at how they are met by BPEL. Then we describe two example Grid workflows that capture some of dynamic aspects that are not common in business workflows but expected in Grid applications. By using those examples, we describe the Grid workflow lifecycle: workflow deployment, workflow instance creation, monitoring, and steering. The chapter finishes with identifying challenges in adapting BPEL to Grid and scientific workflows.

15.1 Introduction

In this chapter we examine to what degree a de facto standard business Web services workflow language, Business Process Execution Language for Web services (a.k.a. BPEL) can be used to compose Grid and scientific workflows. As the Grid application models (such as OGSA [188]) move towards Web services and Service Oriented Architecture (SOA) [178], supporting Web services has become a requirement for a Grid workflow language.

There is a great potential value in leveraging an established workflow language standard from the business domain as it allows for a productive sharing of workflow definitions documents, using commercial and open source tools, leveraging existing training and support, documentation, books etc. BPEL, even if it is not a primary workflow language in scientific projects, is a very good candidate for a common language for sharing workflows between different projects (This can be achieved by allowing a workflow to export and import BPEL workflows in scientific projects.) A high level overview and more details about differences between scientific and business workflows can be found in chapter 3.

In this chapter we identify the requirements that we have found to be important for scientific and Grid workflows that are not yet common in business workflows and some that may never become commonplace in business

workflows (such as an experimental approach to constructing workflows.) To this end we propose a set of additional capabilities that are needed in Grid workflows and show how they can be implemented with a concrete example.

15.2 Short overview of BPEL

The following is not meant to be a comprehensive treatment of BPEL language. Instead our goal is to highlight key features and describe parts of BPEL that are particularly important in context of scientific workflows. Additional information can be easily obtained from many online sources, books, articles, and the BPEL specification itself is the best resource for all the details.

15.2.1 Origins of BPEL

Business Process Execution Language for Web Services (BPEL4WS), when created in 2002, replaced two workflow languages created earlier by IBM and Microsoft. IBM's Web Services Flow Language (WSFL) had a graph oriented view on how to describe workflows and Microsoft's XLANG represented a more block-structured approach. BPEL merged both views and added extensive support for structural handling of errors with try/catch constructs and compensation handlers. The initial 1.0 release of BPEL was followed in 2003 by version 1.1 [75] that clarified and improved several parts of BPEL 1.0. Later that year BPEL was submitted to the Organization for the Advancement of Structured Information Standards (OASIS) and since 2004 has been standardized as WS-BPEL 2.0 [328]. The major change in version number and changed name reflect that OASIS WS-BPEL 2.0 will be a major revision and not fully compatible with 1.x versions. In this overview we will concentrate on BPEL4WS 1.1.

15.2.2 BPEL Capabilities

BPEL4WS is designed from the ground up to work with Web services and each BPEL workflow is a Web service as well. This makes BPEL an easy fit into Web services middleware and allows for easy composition of hierarchical workflows: a BPEL workflow is a Web service that can be used inside another BPEL workflow that may again be used as a Web service inside yet another BPEL workflow.

BPEL allows one to describe a blueprint of a workflow (called an "abstract BPEL") that highlights important behaviors without specifying all details. The intention is to allow the definition of publicly visible behaviors of a workflow, hiding details that may differ between implementations of a blueprint. This is like an interface or a contract in programming languages. The abstract BPEL is then implemented by a BPEL workflow that has all details filled in (this is called "executable BPEL".)

BPEL mandates support for XPath 1.0 as an expression language to manipulate XML. XML Schemas are supported as type system that is mainly used in WSDLs referenced by BPEL workflows. WS-Addressing and asynchronous conversations are supported with ability to use message correlations to flexibly relate messages that are part of a workflow execution. Those specifications provide a solid set of tools to manipulate XML messages, extract and combine parts of XML messages, describe and validate content of XML messages, and route messages to Web services.

BPEL has a strong set of control structures (loops, conditions etc.) and good support for catching and handling exceptions (faults) and reversing changes by using compensations. Compensations are particularly important for longer running workflows that need to "undo" changes in case when there are unrecoverable errors in services used by a workflow and global consistency must be restored before a workflow is finished (using traditional transactions may not be an option as long running workflows could lead to transactional locks being held for very long time.) BPEL is a control driven workflow but modeling data driven workflows that are translated to BPEL is possible (for more details in the differences between two approaches see chapter 12.)

15.2.3 Structure of BPEL workflow

The Overall structure of BPEL workflow is shown in Figure 15.1. A BPEL workflow definition is contained in a `<process>` element. This element is a container that contains a set of other elements, such as `<partnerLinks>` and `<variables>`, and one activity that is an entry point to a workflow (typically it is a `<sequence>`.)

XML is a very verbose language so in the interest of keeping examples readable we will use a simplified notation instead of XML. In this notation text indentation indicates a level of nesting of an XML element and XML attributes are simply listed after an element name as `name=value` pairs (possibly on multiple lines.) We will also omit details that are not important for a given example, such as 'messageType' attribute for variables.

By using this compact notation the example from Figure 15.1 can be rewritten in a shorter form as shown in Figure 15.2.

15.2.4 The Most Common BPEL Activities

Basic activities. BPEL provides a set of simple constructs for sending and receiving messages. Typical BPEL workflow will start with a `<receive>` activity and will end with a `<reply>` activity that sends a reply message to whoever send initial messages that was received. It is easy to send a message to other Web services (they are called partners in BPEL) by using `<invoke>` activity. There are two versions of `<invoke>`: the one-way version when only inputVariable is present and the request-response version when both inputVariable and outputVariable are present in `<invoke>`.

```

<process name="BpelProcessName" targetNamespace="..."
  xmlns="http://schemas.xmlsoap.org/ws/2004/03/business-process/">
  <partnerLinks>
    <partnerLink name="partnerA" partnerLinkType="wsdl:parnterALinkType"
      myRole="myRoleInRelationToPartner"/>
    ...
  </partnerLinks>
  <variables>
    <variable name="varA" messageType="wsdl:MessageA"/>
    ...
  </variables>
  <!-- this is executable part of workflow -->
  <sequence>
    <receive partnerLink="partnerA" portType="wsdl:parnterALinkType"
      operation="doSomething" variable="varA" />
    <assign>
      <copy>
        <from>$varA.someParameter</from>
        <to>$varB.anotherInfo</to>
      </copy>
    </assign>
    <invoke partnerLink="partnerB" portType="pb:anoptherPartnerPT"
      operation="doSomethingElse" inputVariable="varB"
      outputVariable="varC" />
    ... <!-- here something more happens -->
    <reply partnerLink="partnerA" portType="wsdl:parnterALinkType"
      operation="doSomething" variable="results"/>
  </sequence>
</process>

```

Fig. 15.1: Outline of BPEL process in XML

Data manipulation. All messages in BPEL are contained in variables. Variables are passed between BPEL activities. To copy and change content of variables `<assign>` activity can be used - it supports XPath language to select and modify XML content (other data manipulation languages may be used as extensions to BPEL but only XPath is required.)

Structured activities BPEL has a set of structural activities similar to what is available in procedural languages. Loops (`<while>`) and conditions (`<switch>` and `<if>` in BPEL 2.0) are supported. In addition to block level construct - `<sequence>` - BPEL also supports starting multiple threads of execution in parallel by using `<flow>`.

Graph based workflows This last capability is a key to support graph based composition. It is easy to start many activities in parallel with `<flow>` and BPEL allows one to define graph-like dependencies between activities. Each activity (a node in a graph) may have a set of incoming and outgoing links. For

```

process name="BpelProcessName"
  variables $varA, $varB, ...
  partnerLinks "partnerA", "partnerB", ...
  # this is executable part of workflow
  sequence
    receive partnerLink="partnerA"
      operation="doSomething" variable="varA"
    assign copy from $varA.someParameter
      to $varB.anotherInfo
    invoke partnerLink="partnerB"
      operation="doSomethingElse" inputVariable="varB"
      outputVariable="varC"
    # here something more may be added
  reply partnerLink="partnerA"
    operation="doSomething" variable="results"

```

Fig. 15.2: Outline of BPEL process without XML

an activity to start its execution all incoming links must be enabled. When an activity is finished all of its outgoing links will be enabled and that will enable related incoming links for other activities and so on (additional details can be found in BPEL specification.) This capability allows to build any graph in BPEL and the interesting part is that BPEL allows the programmer to mix structured and graph approaches in one workflow.

15.2.5 Limitations of BPEL.

BPEL does not have a parallel loop. This is particularly important for scientific code. If the number of iterations is constant it is possible to use `<flow>` to start multiple activities in parallel but this approach does not work if the number of iterations is depending on an input to a workflow. A parallel loop can be simulated with non-blocking invocations of a Web service (that is a BPEL sub-workflow) but such invocations are hard to track and in general establishing communication channels between sub-workflows and main workflow may be difficult (such as detecting when all sub-workflows finished successfully execution.)

This and some other limitations of BPEL 1.1 (such as limited capabilities of `<assign>` activity) may be fixed in upcoming OASIS WS-BPEL when work on it is finished.

15.3 Goals and Requirements for Scientific Workflows in Grids

Based on our experience we identified a set of requirements that are desirable for a scientific workflow language and a workflow execution environment

(typically called a "workflow engine") for Grids. Those requirements can be used to evaluate any Grid workflow language and later we will use them to see how BPEL meets requirements for a Scientific Workflow language in Grids.

However they will vary in different domains. For example see chapter 17 where requirements for semantic workflows are discussed and chapter 27 with requirements identified in SEDNA scientific modeling environment.

Generic Design Goals

Use of Standards. Standards help to increase the reuse of workflows and help us share parts of whole workflows. We believe that using an industry standard Web services workflow language is beneficial to scientific workflows. Besides greater reuse and sharing of tooling it also allows us to leverage existing knowhow in tutorials, documentation, and other resources available on the Internet. Only when a standard workflow language does not meet requirement of a scientific workflow (either for a generic or a specific scientific domain) and such a language can not be extended to meet requirements (or extensions are too complicated), a new workflow language should be created. BPEL is the current de-facto standard for Web services based workflows in business environments and therefore it is a good candidate for a standard based Scientific Workflow language for Grids.

Integration with Web architecture. In addition to running workflows a Grid workflow engine should follow the general design of a Web Architecture [373, 429] In particular using URIs simplifies integration of information resources maintained in a workflow engine with portals, scientific notebooks, data management systems, and any other scientific or Grid tools. Using URIs allows us to reference workflows (and their parts) already stored in a workflow engine. In particular, this makes it easier for us to integrate a workflow engine with emerging Semantic Web standards [453] that use URIs to identify everything and it make such semantically enriched information machine-understandable.

Integration with portals. A workflow engine should be easy to integrate into an existing scientific portal. At minimum a workflow engine should expose a set of monitoring and administrative operations that can be accessed by portals as Web services. It would be also beneficial if a workflow engine used Semantic Web data standards [453] and was easy to integrate with scientific data management systems such as MyGrid [37] and myLEAD [363].

Requirements Specific to Scientific Workflows

Integration with legacy code. It should be easy to use in scientific workflows components that are not Web services. This requirement can be met by either directly adding support for specific legacy or special execution capabilities or it can be achieved by taking advantage of WSDL's flexibility. Both choices are common. However using WSDL as a common abstraction to describe a

”service” that is not necessarily a Web service provides a uniform and an elegant abstraction. A service accessed from a workflow can be anything from a ”real” SOAP-based Web service over HTTP to a service that is just an executable running locally. This is advantageous as it simplifies a workflow language - it needs only to describe the orchestration of services described in WSDLs. Also using WSDLs makes a workflow description more abstract and resilient to minor changes and allows that service implementation and location is determined at the moment when workflow needs to access a WSDL-described service. Apache WSIF [168] is an example of a runtime environment that allows seamless access to any service that is described in WSDL and available over SOAP/HTTP, SOAP/JMS, as a local Java object, EJB, and even as embedded scripts. The other possibility is to embed actual code that interacts with legacy functionality into BPEL as an extension (for example proposed BPELJ [239] which allows one to embed Java code snippets into BPEL.)

Experimental flexibility. A scientific workflow language and a workflow runtime environment should support a scientific laboratory notebook paradigm. They should allow a user to construct and develop a workflow incrementally, add and remove steps in a running workflow, modify existing workflow activities, allow to re-execute workflow parts, modify its structure during execution, allow ”branching” a running workflow by cloning its state, and other operations that may come up when creating and running experiments. The exact set of capabilities depends on what is expected by a particular group of users that will be using Grid workflows.

History and provenance. A workflow execution environment for scientific workflows should automatically record the history of a workflow execution. A history log should have enough information to reproduce the workflow execution. That may include, but is not limited to, a time ordered list of what services were executed (with enough information to uniquely identify the service instances used), what input and output messages were passed, record any modifications to workflow state, etc. This information should be used to construct a full provenance record by an external service. It is also helpful if a workflow execution environment integrates with external provenance tracking services.

Reuse and Hierarchical Composition. To encourage workflow reuse, it is important that workflows can be used as parts in bigger workflows. This can be enabled if workflows are Web services themselves that can be part of other workflows. A workflow engine should support such composition by exposing each workflow as a Web or Grid service.

Support very long running processes. We expect that some workflows will be used to orchestrate Web and Grid services that may take very long periods of time to complete. Therefore it is very important that a workflow engine will

not only run and store state of such workflows (so they can survive intermittent failures) but that it will be also easy to find, monitor, and manage such workflows.

Support running very large number of workflows. In some scientific domains running experiments involves starting very large number of short lived workflows. A workflow engine must provide capabilities to track all workflows started and make easy to control them.

Grid Specific Requirements

Accessing Grid resources. As it was mentioned before (in case of legacy code) it is possible to use WSDL abstraction to hide implementation details of a service. The same approach can be applied to accessing Grid services from a workflow language. In the case when a WSDL abstraction is not used, a workflow language needs to have Grid specific extensions to interact with specific grid protocols to use Grid resources. Emerging standards such as WSRF [257] provide a promising set of common and reusable WSDL protocol bindings specifically geared for Grids.

Dynamic resources. Support for on-demand creation of resources such as Grid services is essential. In addition to using WSDL abstractions to hide access protocols one should be able to dynamically create Grid services when they are needed (for example GFac [255]).

Designed for scalability. Nothing in the language design should prevent a scalable implementation of a workflow engine.

Integration with grid security. One of the most important and fundamental aspects of Grids is a requirement for strong and flexible authentication and authorization. There are many approaches that are popular. Therefore a workflow language and engine should not mandate one particular security model but be flexible and open so that it can incorporate security capabilities as extensions.

15.4 Illustrative Grid Workflow Example

The LEAD (Linked Environments for Atmospheric Discovery [274]) is a National Science Foundation Large Information Technology Research (ITR) project that is creating an integrated, scalable cyberinfrastructure for mesoscale meteorology research and education. Crucial to the success of LEAD is the ability not only to compose services and data sources into applications but make them dynamically adaptive. This requirement is described in LEAD as Workflow Orchestration for On-Demand, Real-Time, Dynamically-Adaptive Systems (WOORDS [275]). Some of the desired capabilities include the ability to change configuration rapidly and automatically in response to

weather, continually be steered by new data, respond to decision-driven inputs from users, initiate other processes automatically, and steer remote observing technologies to optimize data collection for the problem at hand. Those goals can be expressed as a more generic capability: workflows that are driving LEAD applications must be responsive to events and be able to adapt their future execution paths (more details on workflow in LEAD can be found in chapter 10.)

Many typical scientific workflows are long running workflows and are composed of many steps such as data acquisition, decoding, processing, and visualization. Those steps may need to be repeated and run in parallel for many hours or days before final results are available.

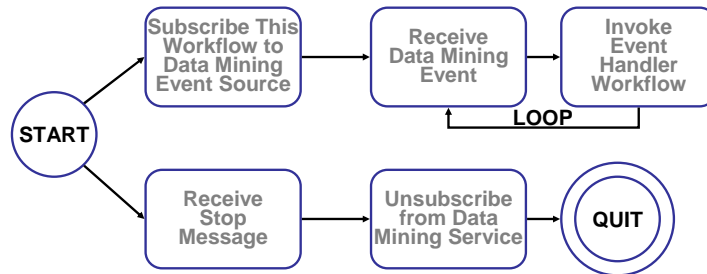


Fig. 15.3: Persistent workflow that is monitoring data mining events

As an example we take two workflows illustrating types of workflows that LEAD plans to use and describe them in a simple scenario. Let assume that we have a data mining service that monitors real-time data streams and detects potentially interesting patterns such as the formation of a tornado. When such an interesting condition is detected the mining service publishes an event to a message bus service (that may support standards such as WS-Eventing or WS-Notification.) A user may choose to run a permanent and persistent workflow that subscribes to data mining events. A simplified graph of such workflow is shown in Figure 15.3 and in Figure 15.4 we show an outline of a BPEL process for that workflow. The BPEL document has a list of declared variables and a list of partner links. Each partner link represents a Web service that is either using the workflow or is used by the workflow (or both). BPEL does not specify how the location of the partner is established and typically this is left to a workflow deployment phase. However more dynamic, per-execution or even during execution, locating of partners is possible (this is discussed in more detail later when workflow lifecycle is described.) When an instance of this sample workflow is started, the first activity executed is `<sequence>`. Then each activity inside sequence is executed beginning with the first assignment. We have used short notation for `<assign>` (`$running = true`) to show that true is assigned to variable named "running". The second

```

process name="PersistentMonitoringWorkflowForUserFoo"
  variables $running, $stopMsg, $workflowName, $subscribeMsg, ...
  partnerLinks "WorkflowUser", "EventBus", "DataMining", "WorkflowEngine" ...
  sequence
    $running = true
    assign from partnerLink="DataMining"
      endpointReference="workflowEventConsumer"
      to "$subscribeMsg/wse:DeliveryTo/wse:NotifyTo"
    invoke name="SubscribeToEvenService" partnerLink="EventBus"
      portType="wse:EventSource" operation="subscribe"
      inputVariable="subsrcibeMsg" outputVariable="subscribeResponse"
  flow
    sequence
      receive name="ReceiveStopMessage" partnerLink="WorkflowUser"
        variable="stopMsg"
      $running := false
      sequence name="RunSequence"
        while condition $running
          $workflowName := "EventHandlinkWorkflow"
          receive name="ReceiveEvent" partnerLink="DataMining" variable="event"
          invoke name="StartEventHandlerWorkflow" partnerLink="WorkflowEngine"
            portType="wse:EventSource" operation="startNewWorkflowInstance"
            inputVariable="workflowName" outputVariable="workflowLocation"
          assign from $workflowLocation to partner "EventHandlerWorkflow"
          invoke name="InvokeEventHandlerWorkflow"
            portType="wse:UserWorkflow" operation="processEvent"
            inputVariable="event"
        exit
  exit

```

Fig. 15.4: Outline of BPEL document describing example workflow

<assign> in the sequence is used to copy location ("endpoint reference") of the workflow Web service (as mentioned before when a BPEL workflow is started it becomes a Web service) to "subscribeMsg" variable. This variable holds content of a message that is sent to the data mining service to subscribe for events. Sending the message is accomplished by the <invoke> operation. This is request-response invocation (it has both input and output variable) and it is a blocking operation, i.e. further workflow execution of this thread is stopped until a response arrives. The response may be either a response message and then its content is copied to the output variable or it may be a fault message. BPEL has sophisticated support for handling faults but in this example it is not needed and the default behavior works well. By default, if a fault happens the workflow instance is terminated with an error and the workflow execution environment may notify a user about an abnormal termination of the workflow.

The next activity executed in the sequence is `<flow>`. It splits execution into two parallel threads of execution. The first one will immediately block on `<receive>`. When this workflow Web service receives "stopMsg" then this thread will unblock and set "running" variable to false. Since this is the last activity in the flow sequence, this thread will be terminated. The other thread started in the flow is more persistent. We have a `<while>` that keeps executing until "running" variable becomes false. In this loop the `<receive>` will block until an event is received from the data mining service. If more than one event is received and the workflow is busy then events are put into a queue and no event is lost. The next activity in the loop creates a new workflow instance by calling a workflow execution service (workflow engine) to create a workflow instance identified by "EventHandlingWorkflow" string. When the workflow instance is created it may be further configured (as explained later in the description of workflow lifecycle) but in this example we just use the new workflow location to invoke it. This invocation is one-way (no output variable) so there is no need to wait for the result of the invocation and the loop can continue. When "running" variable becomes false (after receiving the stop message in other thread) the loop will be exited. This is not an optimal solution as the loop may be still blocked, waiting to receive an event. Unfortunately, BPEL does not have the capability to interrupt blocking waits (still some BPEL implementations may allow one to configure timeouts for blocking receive/invoke, and, in such case, a workflow will eventually finish). For simplicity we could just use `<exit>` in the thread that received a stop message (that is what is shown in Figure 15.3) but in this example we show how multiple threads inside a BPEL workflow instance can communicate by using shared variables (as it is an interesting capability to have in more complex workflows.)

When an event is received the workflow will start other workflow ("EventHandlingWorkflow") such as one depicted in Figure 15.5. This event handler workflow may finish quickly (when the even is deemed "uninteresting") or it may continue running for long time to determine if anything interesting happens. That may lead to generation of other events that may trigger execution of other workflows and eventually sending of an urgent notification to a user that something like a tornado is happening with a high probability.

In Figure 15.6 we have an example of BPEL code to implement the workflow graph showed in Figure 15.5. As we see in those examples BPEL is capable of describing complex workflows but more than a workflow language is needed. An important part of a workflow execution is monitoring. Users should be able to determine the state of the workflows they started. Users may want to know what workflows are waiting for services, what are the intermediary results, etc.

When something interesting is noticed in a workflow then a user should be able not only to steer the workflow execution (start, stop, pause), but also to modify either the state of one particular workflow or a whole group of similar workflows. This is an important requirement for a workflow execution envi-

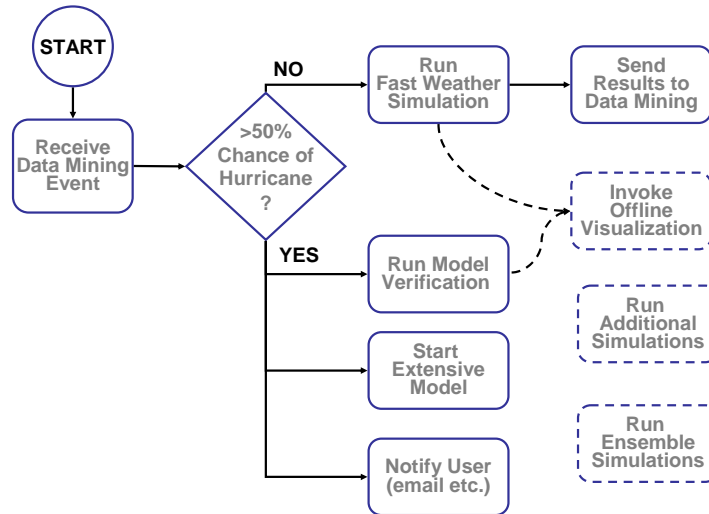


Fig. 15.5: Workflow instance launched in response to a data mining event

ronment in LEAD: workflows are built incrementally and can be modified by a user even when they are running (we depict some of possible modifications in the second workflow by drawing them with dashed lines on the Figure 15.5.) The User can add new steps or rearrange existing steps to meet new requirements. Workflows are frequently changing, reflecting what the user wants to get done. This experimental flexibility fits well into the scientific lab notebook paradigm mentioned in requirements. For example the user may add a new visualization step to the second workflow or modify the first workflow to launch another experimental workflow on a dedicated resource in response to events under some conditions. This experimental capability is part of a workflow engine and not a workflow language (BPEL) but nonetheless is important for running scientific workflows in Grids.

15.5 Workflow Lifecycle on example of GPEL Engine

Overview. We will now continue to delve into our example to see how aforementioned goals and requirements can be met. To make the description very concrete we use the Grid Process Executions Language (GPEL) environment developed at Indiana University. Following the requirements for standards and reuse, we use BPEL. GPEL consist of two parts. First part is GPEL language defined as a subset of BPEL 1.1 language. We are gradually expanding the supported subset with the goal of supporting the final version of WS-BPEL standard in future versions of GPEL. However as BPEL is still under a stan-

```

process name="EventHandlinkWorkflow"
sequence
  receive name="ReceiveEvent" partnerLink="WorkflowCaller" variable="event"
  if condition $event.probability < 50.0
  then
    sequence
      invoke name="WeatherSim" partnerLink="WeatherSimulationExecution"
        portType="fw:FastWeatherSim" operation="runFastCheck"
        inputVariable="event" outputVariable="runResults"
      invoke name="SendResults" partnerLink="DataMiningService"
        portType="dm:DataMining" operation="runDataMining"
        inputVariable="runResults" outputVariable="sendStatus"
    else
      flow # start 3 parallel activities
        invoke name="ModelVerification" partnerLink="ModelVerification"
          portType="fw:ModelVerification" operation="verify"
          inputVariable="event" outputVariable="verificationResults"
        invoke name="WeatherSim" partnerLink="WeatherSimulationExecution"
          portType="fw:WeatherSim" operation="runExtensiveModel"
          inputVariable="event" outputVariable="runModelResults"
        sequences
          $notifyMsg/userName = "foo"
          $notifyMsg/event = $event
          invoke name="NotifyUser" partnerLink="NotificationService"
            portType="dm:UserNotificationService" operation="notifyUser"
            inputVariable="notifyMsg"
      exit

```

Fig. 15.6: Outline of BPEL document describing example event handling workflow

standardization process in OASIS, for now we provide a stable set of semantics by freezing set of BPEL constructs in GPEL namespace.

When compared to BPEL, GPEL adds support for Grid oriented lifecycle and workflow management operations (those were intentionally left out of BPEL standardization scope.) The GPEL API has a set of standard XML messages that can be used to find capabilities of a workflow engine, deploy workflows, start them, and control their execution. This workflow lifecycle is described in details in following sections.

15.5.1 Workflow Composition

There are many tools that can be used to prepare BPEL workflows. They range from simple or advanced XML editors (sometimes with XML Schema support to assist in XML creation) to graphical tools that provide an intuitive GUI to compose workflows by connecting Web services in a graphical way by

hiding from users the XML text of the BPEL process and generating XML automatically when needed. Because graphical tools operate on a higher level of abstraction, they usually support only a subset of BPEL language and provide functionality that is specialized for certain groups of users. For example Sedna (see chapter 27) provides a convenient GUI to manipulate high level abstractions such as an indexed flow construct (a representation of a parallel loop construct that is not available in BPEL) and supports visual macros and plug-ins to reuse fragments of BPEL code. In LEAD we developed XBay Workflow Composer [386] that provides an intuitive GUI tool to compose Web services and generate BPEL or GPEL workflows. XBay provides an extensible library of LEAD services and allows a user to drag-and-drop services and connect them together. In addition to workflow composing XBay allows to monitor and visualize workflow execution (for example visual cues, such as colors, are used to show state of services during execution.)

15.5.2 Workflow Engine Introspection

The way a client discovers the capabilities of a workflow engine differs greatly from one implementation to another. Typically there is no a mechanism to discover capabilities of a workflow engine but it is known beforehand what are capabilities of a particular workflow runtime installation. In the GPEL API we specified the discovery process by defining an extensible way to do a workflow engine introspection. This makes it easier for clients to interact with different GPEL implementations and to discover additional capabilities. The discovery is performed by obtaining, typically using HTTP, an introspection XML document. This document describes capabilities of a GPEL engine. For example one of the capabilities is a location where new documents can be created inside the GPEL engine. When a workflow deployment tool (such as the XBay Workflow Composer) is deploying a workflow to a GPEL engine it must first obtain an introspection document to find a location where the deployment documents can be created (see next section for details.) The location of introspection document can be found in multiple ways. It can be hardcoded into the client software but a more flexible approach is to allow a user to specify location of a workflow engine. This location may point to a web page that contains a link to the actual GPEL introspection document

15.5.3 Workflow Deployment

Before a workflow can be started it needs to be first deployed. The deployment process defines how to associate Web services (described in WSDLs) and the actual workflow process definition (BPEL/GPEL) together. There may be additional deployment specific options such as security (who can start workflows) that must be specified. This process is not standardized in the BPEL specification as it was declared out of the scope of BPEL. As a consequence the way the deployment is accomplished in different BPEL engines

varies greatly between implementations. This is actually good for Grids as it allows us to define a deployment process that fits dynamic requirements of Grid environments.

```
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>GPEL template for Workflow Foo</title>
  <summary>GPEL template for Workflow Foo.</summary>
  <content type="application/x-gpel+xml">
    <gpel:template xmlns:gpel="http://schemas.gpel.org/2005/grid-process/" />
  </content>
  <link rel="http://schemas.gpel.org/2005/wsd1"
    href="http://gpel.example.org/foo.wsd1"/>
  <link rel="http://schemas.gpel.org/2005/gpel"
    href="http://gpel.example.org/foo.gpel"/>
</entry>
```

Fig. 15.7: An Example GPEL Workflow Template

The deployment process in BPEL engines is implemented by sending a set of XML documents, which includes, at minimum, a definition of BPEL workflow but also typically includes WSDL files for all partners and related partner link types. Sometimes instead of sending documents only their locations (URLs) are sent during deployment. There are many protocols that can be used in BPEL engine for deployment and they are ranging from simple HTTP POST and SOAP over HTTP to specialized binary protocols. A particular BPEL engine may provide a programmatic API to do the deployment but there may be also no way to do programmatic deployment if the deployment is done from a GUI application or a servlet that uses proprietary mechanisms for deploying workflows.

We believe there is a very simple way to do BPEL workflow deployment and that it may have a chance to be supported in multiple BPEL implementations eventually. It seems that the simplest way to do deployment is to use HTTP POST and send all workflow related document to the workflow engine. And that is how we defined deployment for GPEL: first a client application needs to send all documents to a GPEL engine (i.e. WSDL and BPEL/GPEL process definitions.) The documents are stored in the GPEL engine and each one gets a unique URL. Using URLs simplifies the linking of documents (and is consistent with the requirement of using Web Architecture.) When the document is stored in the GPEL engine it is validated (so no invalid BPEL or GPEL workflow definitions can be executed and errors should be reported as early as possible.) The last step of the deployment is to create a simple XML document that describes how to link different documents into a workflow template (see Figure 15.7.) The GPEL workflow template has all the information that is necessary to create workflow instances. The GPEL engine will

check that inside template document is a link to workflow document (BPEL or GPEL) and will validate that all required WSDL port types and partner link types are present (actual service bindings and locations can be set later during workflow instance creation.) This step finishes deployment.

15.5.4 Workflow Instance Creation

We recommend separating the workflow creation step from actual workflow execution. This is different from what is described in the BPEL specification where workflow instances are created implicitly when a message marked as "createInstance" is received. Making the process explicit allows for a fine grained control over a workflow instance execution environment. However both approaches can be supported in one workflow engine.

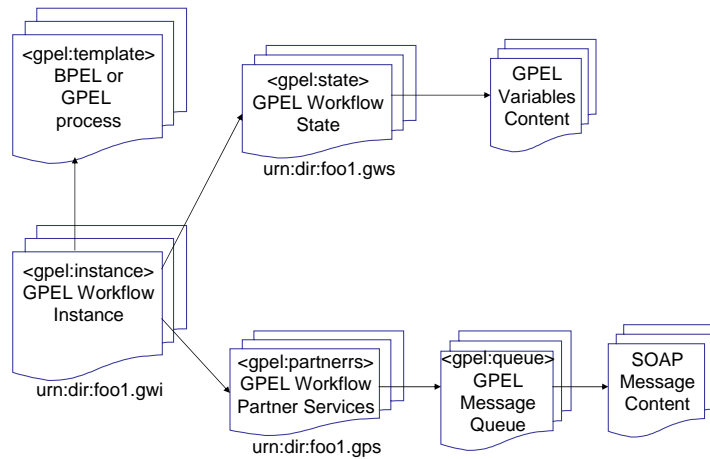


Fig. 15.8: GPEL Workflow Instance State

The separate step of workflow instance creation allows one to setup the workflow instance to use specific Grid or Web service instances. This is very important in Grid environments where a workflow instance may be part of a bigger application and will run on dedicated Grid resources requiring a special workflow setup for each execution (such as creation of security credentials and allocation of cluster nodes).

The GPEL Workflow Instance document, similarly to the GPEL template document, is deployed by using HTTP POST and essentially contains a set of links. The most important link in the workflow instance document is to the workflow template that this workflow instance "implements". A user must replace abstract WSDLs (if any) with concrete WSDLs (or EPRs) and can replace any WSDL used in deployment with a new version that points to a service instance to use just for this workflow instance.

Workflow Instance State. A workflow when running is stateful and its state is similar to a state in a typical program: there is a set of threads and each thread has a set of variables. A BPEL engine needs to maintain a set of variables that are scoped (and in this way similar to local variables in a thread), list of active threads of execution, and what each thread is doing: what activity is executing, is it blocked waiting for a response, etc (see Figure 15.8.)

Using XML is a very convenient way to expose workflow state. This allows us not only to monitor the state of a workflow instance execution but a user or an automatic tool (such as a case based reasoning system or a semantic agent) may modify a running workflow by simply modifying XML documents describing workflow state. If both the workflow process definitions (BPEL document) and a workflow instance state document are modified by a user then this is not just a simple modification of variables or what activity the workflow is executing (like in a debugger) but this can be a structural change to workflow (such as adding new activities.)

15.5.5 Workflow Execution

At this point after a workflow composition, deployment, and creation of a workflow instance we have a running workflow. The workflow execution is the part that is the most important to take full advantage of Grid resources. A Grid workflow engine must be able to request and create Grid resources on demand. This can be accomplished by leaving the decision about what service to use to the very moment when the workflow engine needs to send a message to a Grid service. At that point the service may be created on the best available resource and used by the Grid workflow engine.

GPPEL Workflow Instance Control The workflow instance state document contains all information pertaining to a workflow execution. An interesting consequence is that a user is able to go back-in-time to any previous state of workflow and continue execution from that moment by requesting the GPPEL engine to use a previously stored workflow instance state document. This is particularly useful to allow "cloning" of workflow execution: a user can explore possible execution paths by storing a workflow instance document and creating a workflow instance clone to experiment with an alternative execution path (this capability is limited by level of support from services used by workflow instances - in particular services used by workflow may need to support checkpointing.) In a more traditional sense the workflow state can be monitored to do debugging and, in particular, to request a step-by-step execution of the workflow instance. This is a very useful capability that can be used even by non-programmers when a suitable, high level user interface is provided. For example the metaphor of a VCR remote Start/Pause/Resume/Stop buttons may be used. In our example the persistent workflow (Figure 15.3) when started will continue running until a stop message is received. At any point a user can request the workflow engine to pause the workflow execution and

then examine the workflow state, make modifications, and either resume or step through the workflow execution.

The state of a workflow execution is not complete without knowing what messages were received and sent to Web services used during a workflow execution. A user should be able to view and modify messages and location of Web services used in a workflow instance and request resending of a message to a failing service.

In our example, the second workflow that is launched to handle a data mining event is more experimental in its nature. The intention is that a user may tailor a workflow execution to particular needs related to an event received by that workflow instance. As an example a user may want to steer what the workflow instance is doing or even add new activities to the workflow (such as invoking a visualization service.)

15.6 Challenges in using BPEL in Grids

BPEL meets generic requirements identified in section 15.3 quite well: it is becoming a leading standard for Web Services workflows and it can be well integrated with Web architecture and with portals. The current limitations of BPEL, such as poor support for running large number of parallel sub-workflow, are either addressed in OASIS WSBPEL or can be overcome by providing a higher level language that is then translated into BPEL XML for workflow execution.

Other goals and requirements are independent of choice of BPEL as a scientific workflow language - they have more to do with actual implementation of a workflow engine. First there are performance goals (such as scalability, clustering, administrative interface, etc.) that are generally desirable and they become even more important in Grid environments that require support for dynamic resources and Grid security. As many scientific workflows may take long time to complete and scientific experiments may require running large number of workflows therefore persistence is a desirable feature of a workflow engine implementation. A scientist should be able to start workflows and not to worry that if a machine running the workflow engine is rebooted all work is lost (and may need to be redone.)

Some requirements are specific to scientific workflows. One is supporting history and provenance tracking. The other is experimental flexibility - many scientific workflows may never be "finalized" but need to be incrementally refined and modified during their execution. This capability is particularly important for long running workflows where restarting (and losing all results) is not a good way to make changes in a workflow.

We hope that we showed that BPEL is a viable choice for a Grid workflow language but additional capabilities are needed in a BPEL workflow engine to meet requirements common in Grids. To this extent we shown, using the GPEL API, how to define a set of simple XML documents that can be used to

control lifecycle of a workflow and, in particular, allow to monitor and steer a running workflow instance. By defining set of simple XML documents we hope to increase the chances that such workflow engine API will be used in different middleware applications (including portals) and that it may be implemented by other scientific workflow engines used in Grids.

The main challenges are in area of interactions with legacy scientific code and Grid services. Approaches such as WSIF or BPELJ can help with making BPEL workflows interact with non Web Services but only time will tell how well they meet requirements of scientific workflows. BPEL supports extensibility so it is possible that in future some extensions may become "de facto" standards for Scientific BPEL in Grids.