Grid Service Orchestration using the Business Process Execution Language (BPEL)[*]

Wolfgang Emmerich Ben Butchart Liang Chen
Bruno Wassermann Sarah L. Price

June 7, 2005

Research Note RN 05/07
Dept. of Computer Science
University College London
Gower St, London WC1E 6BT, UK

Abstract

Modern scientific applications often need to be distributed across grids. Increasingly applications rely on services, such as job submission, data transfer or data portal services. We refer to such services as grid services. While the invocation of grid services could be hard coded in theory, scientific users want to orchestrate service invocations more flexibly. In enterprise applications, the orchestration of web services is achieved using emerging orchestration standards, most notably the Business Process Execution Language (BPEL). We describe our experience in orchestrating scientific workflows using BPEL. We have gained this experience during an extensive case study that orchestrates grid services for the automation of a polymorph prediction application.

[*]The work has been funded by the UK EPSRC through grants GR/R97207/01 (e-Materials) and GR/S90843/01 (OMII Managed Programme)

1 Introduction

There is growing interest in the use of web service infrastructures for scientific parallel and distributed computing. The main reason for the uptake of web service infrastructures is the insight that scientific computing can significantly benefit from the investment the computing industry at large is committing to methods, tools and middleware in support of web service development. This interest has further been fuelled by the availability of web service infrastructures that are tailored to grid computing, such as GT3.x, GT4.x and the Distribution from the Open Middleware Infrastructure Institute (OMII). In the remainder of this paper, we refer to services that are defined, deployed and executed using these service-oriented grid computing infrastructures as *grid services*.

Service-oriented architectures developed using grid service infrastructures enable the invocation of a service remotely across the web. While of limited use in itself, the ability to define, deploy and invoke grid services remotely represents an important building block for the definition of services, such as job submission and monitoring, staging and file transfer services and data portal services. Scientists will then want to build higher-level scientific services by combining these low-level services.

(Peltz 2003) refers to such a combination that integrates the invocation of two or more services into a more complex executable workflow as *service orchestration* and contrasts this with service choreography, which tracks message exchange between different autonomous domains. Web service orchestration is supported by the Business Process Execution Language for Web Services (BPEL) (Andrews, et al. 2003) that was proposed by Microsoft, IBM, Siebel, BEA and SAP. Web service choreography is supported by the Web Services Choreography Interface (WSCI) (Arkin, et al. 2002).

Motivated by our earlier work on adopting service-oriented grid computing infrastructures for scientific computing (Butchart, et al. 2003, Nowell, et al. 2004) and for the following reasons we have developed an interest in the orchestration of scientific workflows. Firstly, scientists have a genuine need for orchestration of scientific grid services. Secondly, BPEL is emerging as the de-facto industry standard for the orchestration of services. Finally, the computing industry is much more likely to develop a scalable and robust implementation of a workflow engine than a single research group or even a research consortium. The question that we therefore set out to answer is: to what extent can BPEL be used for the orchestration, i.e. the definition and enactment, of scientific workflows?

We have chosen an experimental research method to answer this question and the main contribution of this paper is an account of the results of that experiment. We have selected a representative scientific problem, which involved both compute and data services. We have then defined the scientific workflow using BPEL. We have subjected three different BPEL implementations to execute the defined workflow in order to compare the characteristics. Based on this experiment we believe BPEL to be suitable for the execution of scientific workflows. We have found that different notations are required to support scientists in modelling workflows at the right levels of abstraction.

The next section presents our choice of experiment. We then discus the requirements for scientific workflow management in Section 3. Section 4 describes how BPEL can be used to orchestrate scientific workflows. Section 5 describes the experience we made with enacting BPEL workflows with an engine developed in-house and we compare this with results obtained from executing workflows with the Oracle and ActiveBPEL engines. In Section 6 we compare our findings to related work before we conclude the paper in Section 7.

2 The Experiment

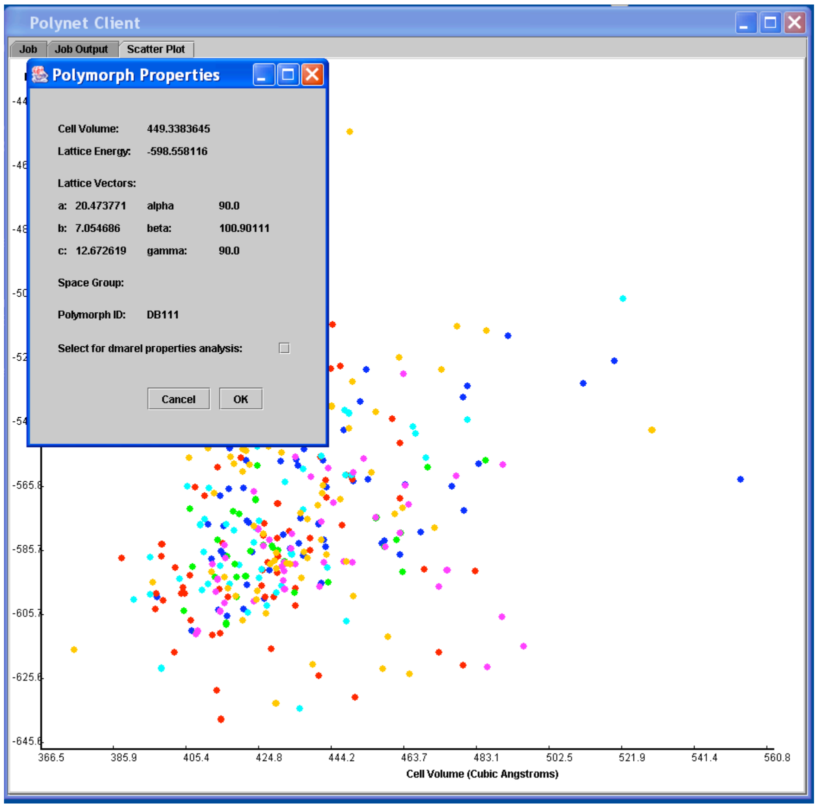
We have chosen an experimental research method to investigate the suitability of BPEL for scientific workflows. The particular questions we want to address are:

- What is the advantage of using BPEL over workflow encoded in a programming language?
- Is BPEL expressive enough to define scientific workflows?
- Will scientists be able to describe their workflow with BPEL?
- Are existing BPEL engines capable of executing large-scale scientific workflows?

2.1 The Scientific Problem

The application we have chosen for our experiments with BPEL is in the area of theoretical chemistry: more precisely in the computational prediction of organic crystal structures from the chemical diagram. This area of research is particularly challenging and worthwhile, as some organic molecules can adopt more than one crystal structure (i.e. have different polymorphs) which have different physical properties. Since new polymorphs are often discovered serendipitously after decades of work on a material, even in the pharmaceutical industry which can only patent, license and market a specific polymorph, a method of computationally predicting polymorphs and their physical properties would have considerable impact on the development of molecular materials (Price 2004).

Computational crystal structure research has been emerging for over a decade, since the computer power became available to consider the vast range of different possibilities for packing an organic molecule into a crystal structure. Progress has been assessed by international blind tests, such as (Motherwell, et al. 2002). These techniques rely on Fortran application programs, such as MOLPAK (Holden, et al. 1993) and DMAREL (Willock, et al. 1995) during polymorph searches. Finding polymorphs is often achieved using an exhaustive search strategy that involves generating possible molecule packings and then optimising the lattice energy in order to decide whether the resulting hypothetical crystal structure is thermodynamically likely. MOLPAK currently supports a total of 29 different packing types, each of which can generate up to 200 different molecule packings. The calculation of the physical properties for each of those packings with DMAREL are completely independent of each other, which enables solving this problem using CPUs in a computational grid without shared memory and with low bandwidth connections.



Cell Volume: 449.3383645

Lattice Energy: -598.558116

Lattice Vectors:

a: 20.473771	alpha	90.0
b: 7.054686	beta:	100.90111
c: 12.672619	gamma:	90.0

Space Group:

Polymorph ID: DB111

Select for dmarel properties analysis: ☐

Figure 1: Search result for a diasteromeric organic salt

Each search produces a large amount of data, such as the information needed to define the crystal structure, its energy, density, mechanical and spectroscopic properties etc. Scientists need to review the data at appropriate levels of abstraction. One abstraction, used initially, is a scatter plot that shows the lattice energy and cell volume per molecule for each hypothetical crystal structure generated. An example of a scatter plot is shown in Figure 1. The scientists then focus in on the properties of the structures which are thermodynamically plausible. The details of the hypothetical polymorphs are then published and stored, for example using a data portal, such as the CCLRC portal (Drinkwater, et al. 2003). This stored data has been used (Peterson, et al. 2002) to suggest possible crystal structures when new polymorphs of the molecule are discovered later, by comparison with the limited experimental evidence available.

When we first supported this application we hard coded the workflow into a user interface application that generated all the jobs required. Unfortunately, the scientists kept changing the workflow, which suggests that this problem benefits from more flexible orchestration using an explicit representation of the workflow. In that way scientists can flexibly modify the number of jobs they want to execute in parallel. Moreover, they can perform conversions between different representations of the output data. We have, for example developed a grid service that transforms the output files into the Chemical Markup Language (Murray-Rust 1997). Scientists might then also decide to include service requests that upload results for public review to a data portal.

2.2 A Typical Polymorph Search Workflow

Figure 2 shows an overview of a typical workflow that scientists might wish to execute for the polymorph search of a particular molecule. They initially need to setup the search and prepare the molecule description. They then need to choose which packing types they might wish to explore. Each of the possible 29 packing type can be analysed in parallel. Scientists then determine the degree of precision with which the exploration of each packing type occurs and this determines how many different subsequent DMAREL executions are required for the packing type. For the highest precision this may result in 200 concurrent executions of DMAREL per packing type.

The rectangles in Figure 2 represent grid services and arrows show control flow. Black bars show spawning and joining of concurrent subprocesses. Submission of MOLPAK and DMAREL computation jobs relies on the GridSAM job submission service that is available form the OMII. GridSAM implements the Job Submission Description Language (JSDL) defined by the GGF (Lee, et al. 2004). The figure does not show any data flow, which is mainly in a peer-to-peer manner by auxiliary staging grid services not shown in the figure and handling of data is limited to information required to control the workflow.

2.3 Suitability as an Experiment

The research method we have employed uses a *replicated and controlled experiment* (Zelkowitz & Wallace 1998). In particular, we replicate the polymorph search scientists have previously done manually, or using a hard-coded workflow embedded in an application, with an explicit orchestration of services with BPEL. We now discuss how

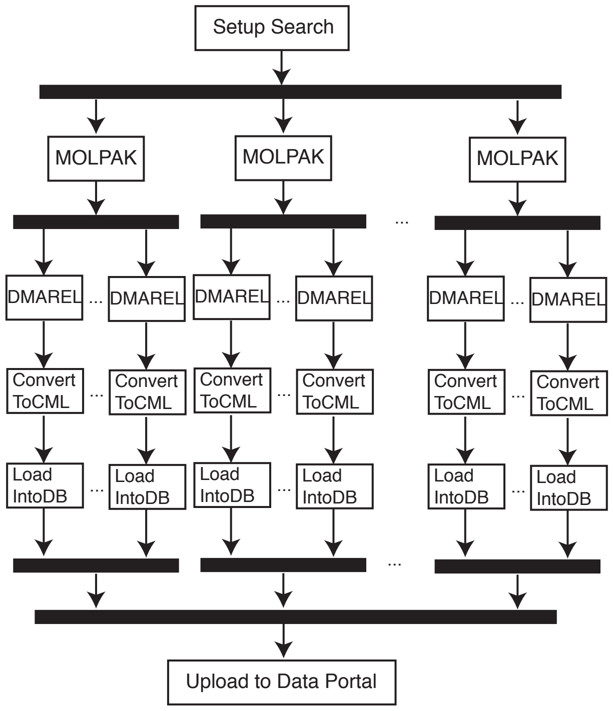
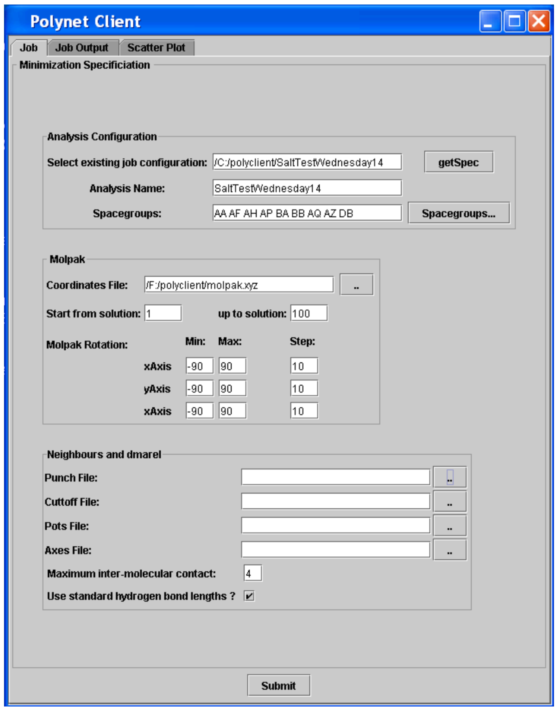


Figure 2: Overview of a Polymorph Search Workflow

this experiment will allow us to answer the questions posed at the beginning of this section.



Minimization Specificiation

Analysis Configuration

Select existing job configuration: /C:/polyclient/SaltTestWednesday14

Analysis Name: SaltTestWednesday14

Spacegroups: AA AF AH AP BA BB AQ AZ DB

Molpak

Coordinates File: /F:/polyclient/molpak.xyz

Start from solution: 1 up to solution: 100

Molpak Rotation:

	Min:	Max:	Step:
xAxis	-90	90	10
yAxis	-90	90	10
xAxis	-90	90	10

Neighbours and dmarel

Punch File:

Cuttoff File:

Pots File:

Axes File:

Maximum inter-molecular contact: 4

Use standard hydrogen bond lengths ? ✓

Figure 3: User interface to control Polymorph search

We have previously developed a user interface that scientists use to control a polymorph search. The interface is shown in Figure 3 and the scientific application is described in detail in (Nowell et al. 2004). The Polynet user interface allows scientists to define a number of search parameters, such as the molecule and different packing types they want to investigate. The interface then executes a hard-coded workflow similar to the one shown in Figure 2. The availability of this user interface gives us a comparison between a workflow orchestrated using BPEL and a workflow encoded in a programming language (Java).

A problem that occurred when scientists used Polynet is that they had lost control and ownership of the workflow that they previously possessed when they manually executed the search. They had to return to programmers whenever the workflow had to be modified. Scientists were therefore very keen to re-acquire control of the workflow and were willing to participate in systematic user studies to establish the usability of BPEL for workflow modelling.

The polymorph search is quite demanding from a scalability point of view. The workflow might involve up to $(29 \times 200) = 5,800$ concurrent invocations of MOLPAK and DMAREL. MOLPAK and DMAREL jobs may take any time between 5 minutes and an hour to complete. The Polynet application achieves this by generating jobs that are submitted to a compute cluster that is controlled by a scheduler. The Polynet application generates the jobs that are submitted to the scheduler in a way that is transparent to scientists. Likewise the BPEL orchestration will have to handle concurrent job submission. The polymorph search application is reasonably rich in that it not only involves massively parallel computations but it also needs to handle the amount of data that is produced during the search. The total volume of data produced during an exhaustive search of a molecule is in the region of 6 GBytes and scientists might wish to complete up to 40 studies during a month, producing a 0.25 TerraByte of data per month.

Processing these data during workflows involves conversion between the output of MOLPAK and the input format for DMAREL, transformations of results to the standardised Chemical Markup Language, enriching results with meta data about the computation prior to upload of selected search results to a data portal. This combination of parallel computation with data handling makes it a fairly representative scientific grid application.

3 Scientific Workflow Management

The aim of this section is twofold. We sketch the overall tasks that need to be accomplished during the management of scientific workflows and from that derive requirements for the management of scientific workflows that are composed by combining grid services.

3.1 Defining Scientific Workflows

From a methodological point of view, defining a scientific workflow with BPEL should start with identifying the grid services that need to be orchestrated. The workflow definition will need to refer to the type definitions used in grid service interfaces. Thus the WSDL definitions need to be available for each service that is to be orchestrated. The workflow definition formalism should have a type system. The type system should be able to import the message types that are used in grid service interface so that they can then be used to declare the types of state data maintained by the workflow. It should be possible to statically validate the orchestration of the service against the interface definitions in order to pinpoint errors in the type system as early as possible. The next step in defining a workflow involves the definition of the scientific workflow.

Basic Activities: Defining an orchestration of grid services involves the definition of both, control flow and data flow between basic activities. The activities that the workflow definition should be able to determine include invocation of a grid service, synchronisation, assembly of the message content that is to be passed on to a grid service, extraction of results from grid service responses and raising faults.

Control Flow and Data Flow: In order to determine the control flow that the workflow definition language needs to support the identification of elementary activities. It should allow for clustering of elementary activities into sequences of activities as well as repetitive and conditional execution of such sequences. For scientific computing, the ability to specify the concurrent execution of sequences of activities is of paramount importance. From a data flow perspective, scientists need to be able to declare temporary stores for the input/output data of service invocations. The scope of such declarations should be flexible and determinable as part of the workflow definition. Workflows have to define how data that is used as input to a service request is assembled and likewise how relevant parts of the output data are extracted.

Hierarchical Composition: Scientific workflow definitions may get reasonably complex. Such complexity has to be mastered. This can be achieved through the introduction of abstraction. It should be possible to treat the definition of a workflow as a single grid service so that this grid service can then be used in a further workflow. In this manner scientific workflows can be composed out of more basic workflows, which, in turn, may rely on services that are themselves workflows.

Failure Handling: It is in the nature of distributed computing that activities fail and this is certainly also true for grid services. Thus, the workflow definition should support the definition of failure handling mechanisms. These may range from simple exception detection and handling that may resubmit jobs, compensating actions that undo certain actions in case of failure to more powerful advanced transactional behaviour that may be required in case distributed state in different databases needs to be kept consistent in the presence of failures.

3.2 Deployment of Scientific Workflows

The definition of the workflow will need to be put into operation so that it can be interpreted and enacted by an interpreter that implements the operational semantics of the workflow definition language. We refer to such interpreters as *workflow engines*. A pre-condition for this to happen is that the grid services that form part of a workflow are deployed so that the workflow engine can communicate with the services using the Simple Object Access Protocol (SOAP) (Gudgin, et al. 2003).

Testing Grid Services: Before deployment and enactment of scientific workflow can start proper, we believe the designer of a workflow shout unit test the individual grid services that are involved in their workflow. Given the general absence of implementation details about the grid services, these unit tests usually take the form of black-box tests. Such black-box tests need to confirm functional correctness as well as exception behaviour in case of erroneous input conditions. Ideally the tests should be fully automated using, for example test drivers written in JUnit (Beck & Gamma 2000) and be controlled using Ant (Bailliez et al. 2005).

Deployment: The workflow should then be deployed itself. To support this, the workflow engine needs to be able to read an archive that contains all necessary descriptions of the workflow. It should be possible to *hot deploy* new workflow definitions, i.e put new workflow definitions into operation, change existing workflows and delete obsolete workflow definitions without having to restart the workflow engine. This is even more important in scientific computing as workflows are often long-running and workflow engines may be used for more than one process at a time. It would be largely inconvenient if new processes could only be deployed when all other processes were completed.

3.3 Enactment of Scientific Workflows

Enactment: The workflow engine should support the remote invocation of a workflow by exposing a WSDL interface for a workflow so that it can be invoked remotely by sending a SOAP message. Once such a message has been received, the engine should start the workflow enactment by executing the control-flow and managing the data flow described in the workflow definition.

Concurrency: The engine should also support the concurrent execution of the same or different workflows. It is quite conceivable that different scientists want to execute the same workflow with different input parameters, thus the engine should be able to create a new instance of a workflow upon receipt of an invocation. Likewise, different scientists might wish to deploy and enact different workflows at the same time.

Scalability: A single execution of the Polynet application may involve up to several thousand concurrent basic activities and tens of thousand individual SOAP messages that need to be managed. It is quite conceivable that several scientists might wish to enact several of these workflows concurrently. Thus the workflow engine needs to be designed in a manner that scales up to such load. This means in particular, careful management of the memory allocation required to hold the workflow state and smart implementation of concurrency; a naïve mapping to operating system processes or threads would not scale sufficiently.

Monitoring: Scientific workflows are potentially long running activities. It is thus of crucial importance to scientists to be able to observe and monitor the ongoing enactment of a workflow. In particular, it is helpful if the current workflow state as materialised in actual values held by variables defined in the workflow can be observed. Moreover, it is helpful to be able to see which activities of the workflow have completed, which ones are ongoing and which ones have failed. In the latter case it is helpful to be able to observe the exceptional conditions that have caused the failure.

3.4 Summary

In this section we have defined the requirements for scientific workflow management. We have in particular elicited requirements for a language used to define workflows and subsequently operational requirements for a workflow engine that is used to deploy and enact workflows. In the next section we will investigate how BPEL addresses these requirements and use examples from the Polymorph search application for illustration purposes.

4 Defining Scientific Workflows with BPEL

BPEL supports the orchestration of grid services into more complex workflows, which means that all activities that are to be done during a workflow need to be exposed as web services. They need to have an interface defined in the Web Services Definition Language (WSDL) (Chinnici, et al. 2004).

We show examples of these from the polymorph search example.

4.1 The BPEL Type System

BPEL can maintain temporary data structures during execution. Those data can be queried and manipulated by a BPEL workflow in order to pass data from one service invocation to another. To facilitate such queries and manipulation, data need to be represented in XML. BPEL has a type system that relies on types that are defined in XML Schemas. In practice, the types of a BPEL workflow are defined as parameter types of grid services in the WSDL description of the orchestrated grid services. As these grid services are potentially defined by independent organisations there is a possibility for name clashes that need to be resolved. We now discuss how BPEL handles these and how we have used these typing mechanisms in the polymorph search experiment.

Relationship to WSDL: The polymorph search experiment combines a number of grid services, each of which defines its own WSDL in which types are declared that we use during the workflow. The polymorph search experiment uses types to describe JSDL from the GridSAM WSDL, it uses types to hold CML data in variables and those data types are declared in the WSDL of the data conversion grid services etc. To use those types, the designer of a workflow needs to declare the name space for each WSDL declaration that is used in a BPEL process and then map the name space in a deployment descriptor to a URI where the WSDL can be obtained from. An example of such a name space definition is shown below:

```
<process name="poly5"
 targetNamespace="http://gsproctwo"
 xmlns:jsdl="http://www.ggf.org/jsdl-1.1.xsd"
 xmlns:cml="http://cml2molpak"
 xmlns:gs="http://www.icenigrid.org/gridsam">
 ...
</process>
```

Name space Management: The name space definition is used to resolve potential conflicts of types defined by different organisations that happen to have the same name. We found several examples of such conflicts in the polymorph search experiment. For example, GridSAM and the constituent JSDL schema defined by the GGF both define a type `JobDescription`. The namespaces above allow us to distinguish between `gs:JobDescription` and `jsdl:JobDescription`.

4.2 Defining Data Flow

Variables: Types can then be used to declare local *variables*. These variables are used to manage data flow between different grid service invocations. By assigning types to these variables it then becomes possible to validate both statically and dynamically that the data passed to and from a grid service using SOAP messages are compatible with the parameters declared in the WSDL description of the service. In our experience such typing greatly reduces the possibility of errors.

We give an example of such a variable declaration below. It shows how variable `submitreq` is declared to be of type `SubmitJobRequest` as defined in the GridSAM namespace.

```
<scope name="dmarelscope">
 <variables>
  <variable
   name="submitreq"
   messageType="gs:SubmitJobRequest"
  />
 ...
</scope>
</variables>
```

Structuring Scope: In practice, it is necessary to use a great number of these variables. It is sometimes convenient to be able to use the same names for different variables, for example if they are to be used in a number of concurrent sub-processes. In order to avoid interference between these processes, BPEL provides the notion of a *scope*. For example, the polymorph application uses scopes to distinguish a large number of different variables that are all called `submitreq` in the concurrent flows for MOLPAK and DMAREL job descriptions. This avoids having to invent artificial names for them. Scopes not only prevent name clashes, but they also allow a BPEL engine to decide when variables will no longer be needed and the, potentially large, data structures stored in a variable can be released.

4.3 Basic Activities

Having discussed how type system and data flow requirements are being met we can now investigate the primitives that are available for assembling workflows. We first review basic activities and then consider how these can be combined using control flow control primitives to more complex workflows.

Service Invocation: The most important basic activity in BPEL is used to invoke services and we show an example below. It invokes `submitJob` from `JobSubmissionPartner`, which provides `JobSubmissionPortType`, a port type defined in the GridSAM WSDL interface. The binding of the partner link is outside the scope of the BPEL definition and is in practice often deferred until deployment time and then defined in a deployment descriptor. It might even be deferred until run-time if registries, such as UDDI are used to locate a suitable partner.

```
<invoke
 name="submitGS"
 partnerLink="JobSubmissionPartner"
 portType="gs:JobSubmissionPortType"
 operation="submitJob"
 inputVariable="submitreq"
 outputVariable="submitresp"
/>
```

Partners must define WSDL port types for each interface that is used in the workflow definition. In addition, BPEL requires WSDL service definitions to provide a partner link, specifying a role to each interface (portType). The BPEL workflow definition can reference these partner-link-roles to describe how an interface is used in an interaction between the workflow and a partner service, for example, to make a distinction between an interface that is needed to send requests and an interface used to send call back messages to the workflow instance.

Synchronisation: Invocation of a service can be either synchronous or asynchronous. Both are defined by the BPEL `invoke` construct. Invocations that give both input and output variables are executed synchronously and the BPEL workflow is blocked while the service executes. Therefore, the example above is a synchronous invocation, which is appropriate as `submitJob` returns control as soon as it has processed the job submission. For asynchronous execution, BPEL supports the notion of correlation sets that are used to associate replies to invocations with business process instances.

Receipt: Every BPEL process is, in fact, a web service in its own right. The service can be invoked by sending a SOAP message to a BPEL engine. We exploit this mechanism to hierarchically structure possibly complex scientific workflows, make them modular and individually reusable. The BPEL primitive used to achieve this is a *receive* statement as shown below. The statement declares that the process implements operation `runPolySearch` and a port type `PolySearchPT`. It also declares that upon receipt it stores the parameters to the message in a variable called `analysisName`. Moreover, whenever `runPolySearch` is invoked, a new process instance is created that spawns off a new BPEL process that executes concurrently with any previously created processes that have not yet terminated.

```
<receive
 partnerLink="client"
 portType="tns:PolySearchPT"
 operation="runPolySearch"
 variable="analysisName"
 createInstance="yes"/>
```

Assignments: Another important basic activity allows us to manipulate data stored in variables. This is done using *assignments*. An assignment consists of any number of *copy* statements that copy data from a source to a target destination. Source destinations can be XML data given as a literal, the result of evaluating an expression, or an XPath query that extracts data from some other variable. The destination of a copy statement denotes a particular part in a variable of a particular WSDL message type. It may optionally include a query that determines where in a complex XML data structure the element is to be copied to. As an example consider the assignment below, that assigns a name of a temporary directory as the output element of a JSDL job description. In most BPEL enactment environments, the query builds up the hierarchy of all parent elements if they are not already present in the DOM tree held in the variable.

```
<assign>
 <copy>
  <from expression="'/tmp/molpakout0'"/>
  <to
    variable="submitreq"
    part="JobDescription"
    query="/gs:JobDescription/jsdl:JobDefinition \
           /jsdl:JobDescription/jsdl:Application \
           /jsdl:Output"
  />
 </copy>
 ...
</assign>
```

We have found the degree of flexibility provided by the queries in assignments to be extremely useful. In the polymorph search a few assignments allowed us to extract energy data from a CML crystal structure that was derived from DMAREL results and create an XML input data structure for an XSLT style sheet that translates the XML input into an SVG file visualising a scatter plot as shown in Figure 1. Once uploaded to a web server this SVG file is then used to provide an overview of results at an appropriate level of abstraction.

Other basic activities: BPEL includes a number of further activities, such as waiting for a given period of time, catching faults, raising faults and compensating activities that we have used but whose description is less interesting and was therefore omitted.

4.4 Defining Control Flow

In order to gain Turing completeness a language has to introduce further primitives for determining control flow. The minimum of these include conditional execution and iteration / recursion of a sequence of activities. BPEL provides these primitives in a straightforward manner by providing *sequence*, *while*, *switch* and *pick* structured activities. Sequence, while and switch have the conventional semantics. Pick allows definition of non-deterministic choice. An example of how they are used is shown below. The fragment defines how a process is waiting for a job submission to complete. It executes a sequence of an operation that waits for 30 seconds and then invokes a job status operation from a job monitoring port type to poll for the result. The loop exits if there are two property elements contained in the status response variable, indicating that it has been submitted and completed.

```
<while
  name="waitForGS"
  condition="count(bpws:getVariableData(
                    'statusresponse',
                    'JobStatus',
                    '//gs:Property'))
                   &lt; 2">
 <sequence>
  <wait for="'PT30S'" name="wait4Property"/>
  <invoke
    name="statusGS1"
    partnerLink="JobMonitoringPartner"
    portType="gs:JobMonitoringPortType"
    operation="getJobStatus"
    inputVariable="statusrequest"
    outputVariable="statusresponse"/>
 </sequence>
</while>
```

A more elegant way to re-synchronise the process with the termination of a job would have been to use a call back mechanism, such as BPEL message correlation or WS-Addressing. Neither are yet supported by the GridSAM service. If they were supported we would have used the message correlation primitives and a receive statement to wait for notification messages from GridSAM and then exited the loop if the notification message indicated termination.

Concurrent Execution: An important primitive for scientific workflows is the ability to execute a sequence of activities in parallel. This is supported in BPEL using *flows*. A sequence may contain a number of flows. The activities contained in each flow will then be executed concurrently with activities contained in the other flows. We show an example flow below. It determines that four MOLPAK job submissions are to be done in parallel to each other. In our polymorph application each of these flows then contains a sequence with another 200 flows each of which executes a DMAREL job submission.

```
<sequence>
 <flow name="molpak1">
  <scope>
   <variables>
    <variable name="submitreq".../>
    <variable name="submitres".../>
    <variable name="statusreq".../>
    <variable name="statusrep".../>
   </variables>
   <sequence>
    ...
    <invoke operation="submitJob".../>
    ...
   </sequence
  </scope>
 </flow>
 <flow name="molpak2" .../>
 <flow name="molpak3" .../>
 <flow name="molpak4" .../>
</sequence>
```

In summary, BPEL satisfies a great number of requirements for defining scientific workflows that we have discussed earlier. It includes a static type system that relies on types defined in WSDL documents of constituent grid services. It supports the definition of both data flow by way of assign activities that modify variable content and use variables as input/output parameters of grid service invocations. BPEL provides Turing complete control flow primitives and in particular addresses concurrent execution of parallel activities.

4.5 Lessons learned

We now discuss the experience we made when using BPEL to define the polymorph search workflow. The main result is that we were able to express the workflow for the polymorph search application in full using BPEL.

The experiment of using BPEL for this polymorph search replicated two different previous methods used by scientists. Before we started our collaboration, scientists used to submit jobs manually. This involved a large number of mundane manual tasks, such as converting data representations, transfer of files, synchronisation of jobs and analysis of results. Both the hard-coded workflow in the Polynet application shown in Figure 3 and the BPEL workflow represent a significant improvement as these searches can now be completed in hours rather than months and relieve scientists of laborious tasks.

In comparison to the Polynet application, this experiment shows a number of further advances that BPEL can achieve. Firstly, the workflow is now explicit and no longer buried inside Java code. This means that the steps taken in a search are explicitly documented. Such documentation provides important meta data to accompany the search results as it enables other scientists to repeat a search. Secondly, it is now easier to modify the workflow as the orchestration can be manipulated more easily. For example, when we introduced data extraction previously we had to handle XSLT and XPath processors and ourselves manipulate DOM trees and write style sheets. With the XPath query capability of BPEL the extraction is done with a few assignment operators that abstract away the underlying XML processing machinery. Thus in our experience it is considerably easier to modify the resulting workflow than it was when the workflow was buried in hand-written code.

However, BPEL is not as well suited as it could be to defining computational workflows and we now discuss how these shortcomings can be overcome while retaining a standard BPEL engine as the execution environment.

The first problem we struggled with is that flows are still not expressive enough. In order to achieve concurrency between activities one has to enumerate all the concurrent flows. This leads to BPEL workflows that are larger than can be understood by humans and is hard to maintain as it contains significant redundant information. In the case of the polymorph search, expressing a precise search for four packing types requires 4×200 flows and this leads to a BPEL file that is approx. 5 MBytes large.

We have overcome this problem by introducing indexed flows. An index is a variable of a scalar type that has a start value s and end value e. An indexed flow then executes e-s number of flows in parallel. The sequence of activities may use the index variable to distinguish between the individual concurrent flows. Indexed flows can be implemented fairly easily, for example, by a style sheet that expands all flows and then creates standard BPEL. Thus we do not loose the ability to use standard BPEL engines to execute scientific workflow.

One of the main aims of extracting the workflow from scientific applications is to empower scientists to define and modify the workflow themselves. BPEL as sketched above does not meet that objective. In a usability study we found that computational chemists were not able to write BPEL and thus we have failed to return ownership and give them the ability to manipulate their workflows themselves.

From this study, we conclude that a visual language with appropriate tool support is required. Indeed commercial BPEL environments, such as the Collaxa tool suite now distributed by Oracle and the ActiveBPEL environment provide such graphical notations. We have defined such a graphical notation and built a visual editor as a plug-in for Eclipse (Gamma & Beck 2004) that will be distributed by the OMII. The detailed discussion of both graphical language and tool support is subject to a companion paper.

The next observation is that a significant number of BPEL activities are needed for what a scientist might deem to be an elementary activity (such as a job submission). In our polymorph example, these BPEL activities include preparation of the JSDL file, staging of input files prior to job submission, synchronisation of job result with the main body of the process and subsequent extraction of relevant data from the job submission.

To address this problem, we have introduced a mechanism to define domain specific extensions of BPEL. These extensions are then expanded by a preprocessor into standard BPEL so that we continue to retain the ability to use an off-the shelf BPEL engine for the enactment of scientific workflows.

5 Deploying and Enacting Scientific Workflows

When we started using BPEL in early 2003 there were no suitable implementations available and we decided to implement the subset of BPEL described above ourselves. The aim of this implementation was to enable the experimentation with BPEL while we were waiting for commercial engines to appear. The subset did not implement asynchronous invocations and message correlation. It did not have any facilities for compensations and error handling either. It took approx. nine person months to complete that implementation and functionally it was capable of executing the workflow described above.

In 2004, Active Endpoints released an open source BPEL implementation called ActiveBPEL. Also in 2004 Oracle bought Collaxa, a company that had produced a BPEL environment. In October 2004 we switched from using our own implementation to these two environments.

We now discuss the experience made during these endeavours. For the industrial strength implementation we focus on the ActiveBPEL engine as it is available as open source form and will therefore be of more interest to the scientific community than an engine that attracts license costs.

5.1 Use of In-House BPEL engine

From a functional point of view the BPEL engine we built only addressed a subset of the requirements that we outlined in Section 3.This meant that scientists that used our engine had to restart the engine whenever they modified the workflow, which proved difficult. Thus the construction and subsequent use of the engine only revealed some of the requirements.

Moreover, it was not possible to monitor the progress of a workflow other than by observing the results produced within. This made it very hard to establish that the workflow was working correctly and also to find any errors in the workflow definition.

Our workflow engine also did not provide any mechanisms for statically establishing the correctness of the workflow beyond validation against the BPEL schema. Again this made it difficult to establish that workflows are correctly defined as we had to rely on code reviews and inspections.

When testing the scalability of the engine we encountered some serious problems that prevented us from completing a full polymorph search. The first problem was that we implemented BPEL flows using Java threads. While this seemed elegant at first, in retrospect this choice was rather naïve. During a full polymorph search, depending on synchronisation of job completion, in the worst case we might have up to 29×200 active flows. This means that there are 6000 concurrent threads, which is more than JVMs and operating systems can generally handle. The next problem was that we implemented only synchronous invocation as we deemed the message correlation required for asynchronous execution to be too hard to implement. Our engine implemented invocations by opening connections to the constituent web services, kept them open while it was waiting for the results. The problem was again that Solaris does not support 6000 concurrently open network sockets. Our engine also did not scale to more than one concurrent user as we had not implemented the notion of process instances. Finally, the memory management in our BPEL engine was insufficient and we ran out of memory with only four packing types. The problem here was that we kept references to BPEL variables on the engine stack even if they had gone out of scope. This prevented the Java garbage collector from freeing the memory. Even after we rectified this problem by explicitly assigning null values to the references the problem persisted. The problem was only resolved after we tuned the Java garbage collector and instructed it to run in parallel with the BPEL engine.

In summary, we were able to start experimentation with BPEL using our own engine and this was very valuable. But we have concluded that it is too hard to write a production strength BPEL enactment environment. This is because there are important requirements regarding validation, monitoring, deployment and scalability that are too hard to be addressed by a single research group endowed with only a modest amount of research money. What we did achieve though was to gain confidence that BPEL has all the right abstractions to express scientific workflows and this allowed us to switch to an industrial-strength BPEL engine.

5.2 Use of ActiveBPEL

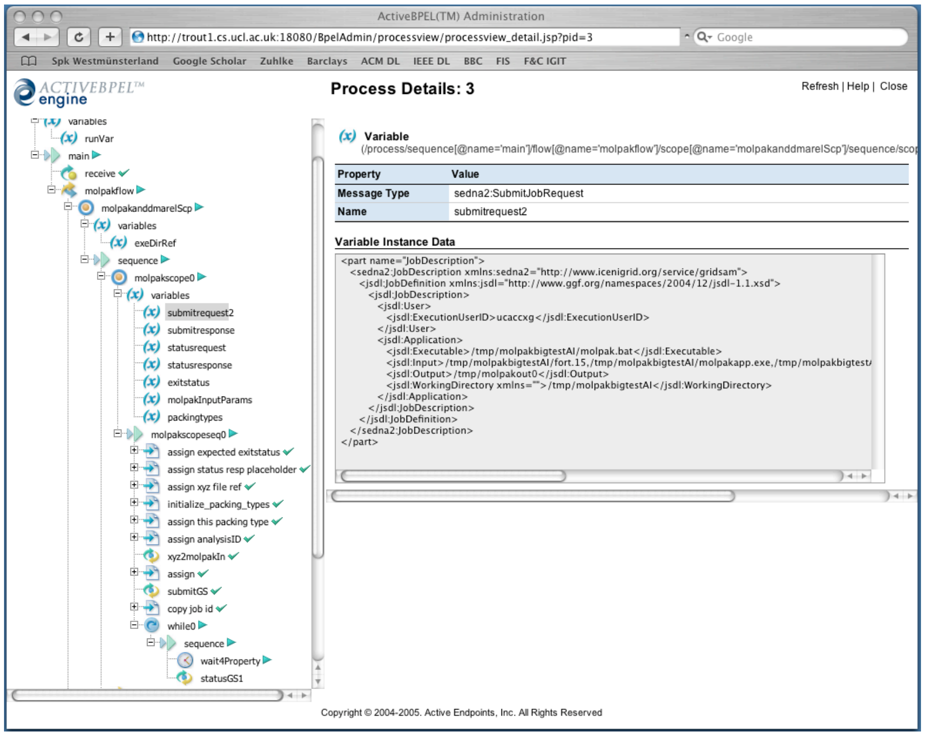
Active BPEL is deployed as a servlet into Apache's Jakarta Tomcat container. It has extensive documentation and has been released by Active Endpoints into the public domain under a Lesser Gnu Public License. It provides a full implementation of the BPEL 1.1 specification and Active Endpoints provide professional services, such as training and consultancy. We now discuss how ActiveBPEL implements our validation, deployment and enactment requirements.

The ActiveBPEL distribution does not contain a static validation tool that would check the type system of BPEL. Instead, Active Endpoints host a free validation service on their web site that can be used to find flaws. We have found this service to be extremely useful as it has helped us to define correct BPEL prior to attempting deployment into the Engine.

For deployment, ActiveBPEL uses a format that archives the BPEL process description together with a deployment descriptor and any WSDL declarations that are not available on-line. The archive is compressed, which simplifies transfer to and from the engine. We found that a 5 MByte BPEL file with associated deployment descriptors and WSDL files is archived in a mere 57 KBytes. Once an archive is stored in the deployment directory ActiveBPEL hot deploys the process. Likewise, if an archive is deleted from the deployment directory it removes the file and if a file is changed it updates the deployed workflow. This allows relatively straightforward integration of process deployment into any grid environment as secure file transfer mechanisms can be used to authenticate and facilitate deployment.

The workflow of a full polymorph search contains approx. 64,300 basic activities.

The ActiveBPEL engine was able to execute the entire workflow for a full search and we did not encounter any scalability problems with the engine itself. We did, however, encounter scalability issues elsewhere. The first problem was related to the service that was in charge of performing the data conversion to CML and staging the input files for DMAREL. When two MOLPAK packing type calculations finished at the same time, the workflow invoked the subsequent conversion and staging service to prepare the DMAREL inputs 400 times within 10 seconds. One such execution takes longer than 10 seconds and therefore all 400 executions of the service happened concurrently. Both the workflow engine and the container hosting the staging service coped fine with this degree of concurrency, but the operating system imposed a soft limit of 1024 file descriptors that users could open at the same time. As each of the service invocations had to open about 10 files the limit was exceeded and the services malfunctioned. The problem was easily rectified by increasing the file descriptor limit. But the lesson we learned from this is that to achieve scalable workflows it is important that services involved are more likely to falter under load than the engine that creates the load in the first place.


ActiveBPEL(TM) Administration
http://trout1.cs.ucl.ac.uk:18080/BpelAdmin/processview/processview_detail.jsp?pid=3
Google
Spk Westmünsterland
Google Scholar
Zuhlke
Barclays
ACM DL
IEEE DL
BBC
FIS
F&C IGIT
ACTIVEBPEL™ engine
Refresh | Help | Close
Copyright © 2004-2005. Active Endpoints, Inc. All Rights Reserved

Process Details: 3

```
variables
  runVar
main ▶
  receive ✓
  molpakflow ▶
    molpakanddmarelScp ▶
      variables
        exeDirRef
      sequence ▶
        molpakscope0 ▶
          variables
            submitrequest2
            submitresponse
            statusrequest
            statusresponse
            exitstatus
            molpakInputParams
            packingtypes
          molpakscopeseq0 ▶
            assign expected exitstatus ✓
            assign status resp placeholder ✓
            assign xyz file ref ✓
            initialize_packing_types ✓
            assign this packing type ✓
            assign analysisID ✓
            xyz2molpakIn ✓
            assign ✓
            submitGS ✓
            copy job id ✓
            while0 ▶
              sequence ▶
                wait4Property ▶
                statusGS1
```

Variable
(/process/sequence[@name='main']/flow[@name='molpakflow']/scope[@name='molpakanddmarelScp']/sequence/scop

Property	Value
Message Type	sedna2:SubmitJobRequest
Name	submitrequest2

Variable Instance Data

```
<part name="JobDescription">
  <sedna2:JobDescription xmlns:sedna2="http://www.icenigrid.org/service/gridsam">
    <jsdl:JobDefinition xmlns:jsdl="http://www.ggf.org/namespaces/2004/12/jsdl-1.1.xsd">
      <jsdl:JobDescription>
        <jsdl:User>
          <jsdl:ExecutionUserID>ucaccxg</jsdl:ExecutionUserID>
        </jsdl:User>
        <jsdl:Application>
          <jsdl:Executable>/tmp/molpakbigtestAI/molpak.bat</jsdl:Executable>
          <jsdl:Input>/tmp/molpakbigtestAI/fort.15,/tmp/molpakbigtestAI/molpakapp.exe,/tmp/molpakbigtest
          <jsdl:Output>/tmp/molpakout0</jsdl:Output>
          <jsdl:WorkingDirectory xmlns="">/tmp/molpakbigtestAI</jsdl:WorkingDirectory>
        </jsdl:Application>
      </jsdl:JobDescription>
    </jsdl:JobDefinition>
  </sedna2:JobDescription>
</part>
```

Figure 4: User interface for monitoring workflows

The BPEL environment provides a servlet that calculates a web-based user interface for monitoring the state of the engine. The servlet supports investigating the set of deployed BPEL processes, the state of the deployment and it provides access to all process instances using a web browser. For each of these process instances it lists the set of basic activities, the variables that are declared and the content of these variables. Figure 4 shows a screen dump of the state of a polymorph search workflow. The left-hand pane shows all activities and variables declared and the right-hand pane shows the JSDL document stored in `submitreqeust2`.

A problem we encountered with this monitoring solution is again related to scalability. The left-hand pane shows several icons and a text box per activity. With more than 65,000 activities in the full polymorph workflow there are several hundred thousand icons to be transferred and even after an hour of waiting this did not scale. The engine, however also logs the progress of the workflow and we found this log to be fully scalable and sufficient to understand the progress for large-scale workflows.

In summary, we found the ActiveBPEL engine to satisfy the needs of orchestrating grid services into scientific workflows. The engine itself is robust and scalable even though the scalability of the monitoring solution could be improved.

6 Related Work

Over the last 20 years there has been a great interest in both research and industry in systematically defining, reasoning about and enacting processes and workflows. The first application domain was probably software processes (Osterweil 1987) and a great number of software process modelling and enactment environments emerged at the end of the 1980s. For an overview and critical appraisal of this work we refer the interested reader to (Fuggetta 2000). Interest in commercial business processes and workflows emerged in the early 1990s and resulted in the development of a number of workflow management environments, such as FlowMark (Leymann & Roller 1994). FlowMark was the predecessor of the WebSphere MQ WorkFlow product of IBM and greatly influenced WSFL (Leymann 2001). WSFL was then later merged with XLANG (Thatte 2001) in a development that resulted in BPEL. The stance taken in this paper is to build on this long stream of work and to explore the extent with which BPEL is usable for scientific applications.

Most other work on managing scientific workflow takes a radically different approach in that they developed workflow systems from scratch. To do so is indeed necessary as long as grid computing is not properly aligned with web services.

DAGMan (Frey 2002), Taverna (Oinn, et al. 2004), GridFlow(Cao, et al. 2003) and GridAnt (Amin, et al. 2004) support scientific workflow outside the scope of the web service framework. DAGMan is used in the Condor Scheduling system in order to manage dependencies between jobs. DAGMan and Condor are tightly coupled and controlling workflows with DAGMan that do not involve Condor is not straightforward. Taverna was developed primarily for Bioinformatics applications. It has only limited support for Web Service invocation as it does now allow data caching and manipulation constructs like WSDL derived variables and assignments. This means even simple data conversions require the creation and invocation of customised local transformation services. GridAnt is a relatively lightweight environment for the definition and monitoring of workflows based on Ant (Bailliez et al. 2005). The Ant language, however, does not have built-in primitives for handling concurrency and synchronisation of asynchronous job submission and also does not support handling invocations to grid services very well. GridFlow focuses on resource allocation and to some extent replicates resource allocation systems, such as Condor or the Globus GRAM. Instead our work strictly separates the notion of workflow from resource allocation, which is delegated instead to a job submission grid service. In comparison to these approaches the use of BPEL does not yield any advantages when the basic activities that need to be combined are not web or grid service invocations. If on the other hand all activities are web services, as will be increasingly the case with the emergence of service based Grid infrastructures, industrial strength BPEL engines will be a far superior execution environment for scientific workflow.

Triana (Taylor, et al. 2003, Majithia, et al. 2004) and GSFL (S. Krishnan & v. Laszewski 2002) are two home grown workflow engines for service oriented grid applications. As described above, building a truly scalable and robust production-strength workflow system is very hard and published work does not include any discussion of the scalability and reliability of these approaches. We can therefore not compare their robustness to the one we obtain with BPEL. We agree with (Majithia et al. 2004) that it should indeed be possible to map Triana and GSFL workflows onto BPEL and this paper has outlined the benefits of doing so, namely the availability of robust industry-strength engines for workflow enactment.

(Deelman, et al. 2003) describes an interesting approach to workflow management. Their work distinguishes abstract (i.e. domain specific) and concrete workflows. Abstract workflows contain domain specific concepts, while concrete workflows use a particular workflow technology, DAGMan in this case. Our experience confirms that this approach is promising. Again this paper suggests that the same approach can be achieved using BPEL.

7 Conclusions

It is genuinely hard to build industrial strength middleware and only very few research groups have succeeded in doing so. A robust and scalable workflow engine falls into this category of middleware systems. Engines developed in an industrial setting are much more likely to have the required quality characteristics than one developed in a research group. The emergence of BPEL as the de-facto industry standard for web service orchestration is significant because it means that a number of BPEL engines will be available.

Our work is important for two reasons, we have firstly explained how BPEL can be used to combine grid services. Based on a number of case studies, one of which we have explained in detail in this paper, we have confidence that BPEL can be used more generally for such grid service orchestration. Secondly, our work has confirmed that both the ActiveBPEL and the Oracle Collaxa engine have the required quality attributes to be able to handle large-scale scientific workflows.

We will bundle ActiveBPEL and the Eclipse-based scientific workflow development environment and then release it through the Open Middleware Infrastructure Institute later this year. We are currently extending the user base. The e-Minerals project (Calleja, et al. 2004) have recently used the environment successfully to define a workflow for modelling absorption of pollutants in soil. We have started to assist a number of further discussions to model processes with BPEL.

References

K. Amin, et al. (2004). 'GridAnt: A Client-Controllable Grid Workflow System'. In D. King & A. Dennis (eds.), *Proc. of the* 37^{th} *Annual Hawaii International Conference on System Sciences (HICSS'04) – Track 7*, p. 70210c. IEEE Computer Society Press.

T. Andrews, et al. (2003). *Business Process Execution Language for Web Services Version 1.1*. OASIS, http://ifr.sap.com/bpel4ws.

A. Arkin, et al. (2002). *Web Service Choreography Interface (WSCI) 1.0*. W3C, http://www.w3.org/TR/wsci.

S. Bailliez et al. (2005). *Ant User Manual*. Apache Jakarta, http://jakarta.apache.org/ant.

K. Beck & E. Gamma (2000). 'Test-infected: programmers love writing tests'. In *More Java Gems*, pp. 357–376. Cambridge University Press, New York, NY, USA.

B. Butchart, et al. (2003). 'OGSA First Impressions: A Case Study using the Open Grid Service Architecture'. In S. Cox (ed.), *Proceedings of the UK E-Science All Hands Meeting, Nottingham*, pp. 810–816. EPSRC.

M. Calleja, et al. (2004). 'Grid tool integration within the eMinerals Project'. In S. Cox (ed.), *Proc of the 2004 UK E-Science All Hands Meeting, Nottingham*, pp. 812–817. UK Engineering and Physical Science Research Council.

J. Cao, et al. (2003). 'GridFlow: Workflow Management for Grid Computing'. In 3^{rd} *Int. Symposium on Cluster Computing and the Grid*, pp. 198–205. IEEE Computer Society Press.

R. Chinnici, et al. (2004). 'Web Services Description Language'. W3c working draft, W3C, http://www.w3.org/TR/wsdl20.

E. Deelman, et al. (2003). 'Mapping Abstract Complex Workflows onto Grid Environments'. *Journal of Grid Computing* **1**(1):25–39.

G. Drinkwater, et al. (2003). 'The CCLRC Data Portal'. In S. Cox (ed.), *Proc. of the UK e-Science All Hands Meeting*, pp. 540–547. UK EPSRC. ISBN 1-904425-11-9.

J. Frey (2002). 'Condor DAGMan: Handling Inter-Job Dependencies'. Tech. rep., University of Wisconsin, Dept. of Computer Science, http://www.cs.wisc.edu/condor/dagman.

A. Fuggetta (2000). 'Software process: A Roadmap'. In *The Future of Software Engineering*, pp. 25–34, New York, NY, USA. ACM Press.

E. Gamma & K. Beck (2004). *Contributing to Eclipse: principles, patterns and plug-ins*. Addison-Wesley.

M. Gudgin, et al. (2003). 'Simple Object Access Protocol'. W3c recommendation, W3C, http://www.w3.org/TR/soap12-part1.

J. R. Holden, et al. (1993). 'Prediction of possible crystal structures for C-, H-, N-, O-, and F-containing organic compounds'. *Journal of Computational Chemistry* **14**(4):422–437.

W. Lee, et al. (2004). 'A Standard Based Approach to Job Submission through Web Services'. In S. Cox (ed.), *Proc. of the UK e-Science All Hands Meeting, Nottingham*, pp. 901–905. UK EPSRC. ISBN 1-904425-21-6.

F. Leymann (2001). 'Web Services Flow Language'. Tech. rep., IBM.

F. Leymann & D. Roller (1994). 'Business process management with FlowMark'. In *Compcon Spring '94: Digest of Papers*, pp. 230–234. IEEE Computer Society Press.

S. Majithia, et al. (2004). 'Triana: A Graphical Web Service Composition and Execution Toolkit'. In *Proc. of the* 4^{th} *Int. Conference on Web Services*, pp. 514–524. IEEE Computer Society Press.

W. D. S. Motherwell, et al. (2002). 'Crystal structure prediction of small organic molecules: a second blind test'. *Acta Crystallographica Section B-Structural Science* **58**:647–661.

P. Murray-Rust (1997). 'Chemical Markup Language'. *World Wide Web Journal* **2**(4):135–147.

H. Nowell, et al. (2004). 'Increasing the Scope for Polymorph Prediction using e-Science'. In S. Cox (ed.), *Proc of the 2004 UK E-Science All Hands Meeting, Nottingham*, pp. 968–971. UK Engineering and Physical Science Research Council.

T. Oinn, et al. (2004). 'Taverna: A tool for the composition and enactment of bioinformatics workflows'. *Bioinformatics Journal* **20**(17):3045–3054.

L. J. Osterweil (1987). 'Software Processes are Software Too'. In *Proc. of the* 9^{th} *Int. Conf. on Software Engineering*, pp. 2–13. IEEE Computer Society Press.

C. Peltz (2003). 'Web services orchestration and choreography'. *IEEE Computer* **36**(10):46–52.

M. L. Peterson, et al. (2002). 'Iterative high-throughput polymorphism studies on acetaminophen and an experimentally derived structure for form III'. *Journal of the American Chemical Society* **124**(37):10958–10959.

S. L. Price (2004). 'The Computational Prediction of Pharmaceutical Crystal Structures and Polymorphism'. *Advanced Drug Delivery Reviews* **56**(3):301–319.

P. W. S. Krishnan & G. v. Laszewski (2002). ‘GSFL: A workflow framework for grid services’. Technical Report Preprint ANL/MCS-P980-0802, Argonne National Laboratory.

I. Taylor, et al. (2003). ‘Triana Applications within Grid computing and Peer to Peer Environments’. *Journal of Grid Computing* **1**(2):199–217.

S. Thatte (2001). ‘XLANG: Web Services for Business Process Design’. Tech. rep., Microsoft.

D. J. Willock, et al. (1995). ‘The relaxation of molecular crystal structures using a distributed multipole electrostatic model’. *Journal of Computational Chemistry* **16**(5):628–647.

M. Zelkowitz & D. Wallace (1998). ‘Experimental models for validating technology’. *IEEE Computer* **31**(5):23–31.