Journal of Grid Computing (2006) 3: 283–304
DOI: 10.1007/s10723-005-9015-3

Grid Service Orchestration Using the Business Process Execution Language (BPEL)*

Wolfgang Emmerich[1,**], Ben Butchart[1], Liang Chen[1]
Bruno Wassermann[1] and Sarah L. Price[2]
[1]*London Software Systems, Department of Computer Science, UCL, London, WC1E 6BT, UK*
E-mail: w.emmerich@cs.ucl.ac.uk
[2]*Department of Chemistry, UCL, London, WC1E 6BT, UK*

Received 1 June 2005; accepted in revised form 6 December 2005

Key words: Grid computing, Grid services, workflow, business process execution language

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. Using this example, we explain the extent with which the BPEL language supports the definition of scientific workflows. We then describe the reliability, performance and scalability that can be achieved by executing a complex scientific workflow with ActiveBPEL, an industrial strength but freely available BPEL engine.

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 web 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 Internet. While of limited use in itself, the ability to define, deploy and invoke Grid services remotely represents an important building block for job submission and monitoring, staging, file transfer and data portal services. Scientists will then want to build higher-level scientific services by combining these low-level services.

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

(Peltz, [26]) 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) [2] that was proposed as a standard by Microsoft, IBM, Siebel, BEA and SAP to the Organisation for the Advancement of Structured Information Standards (OASIS). Web service choreography is supported by the Web Services Choreography Interface (WSCI) [3].

Motivated by our earlier work on adopting service-oriented Grid computing infrastructures for scientific computing [6, 23], 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 will be able to build. 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 used an open source BPEL implementation to execute the defined workflow in order to measure the performance, scalability and reliability 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 discuss 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 using the freely available ActiveBPEL engine. 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 suitabilitys of BPEL and its implementation for scientific workflows. The two questions we want to address are:

- Is BPEL expressive enough to define scientific workflows?
- Are the performance, scalability and reliability characteristics of a freely available BPEL engine sufficient to execute 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) that 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 [28].

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 [21]. These techniques rely on Fortran application programs, such as MOLPAK [15] and DMAREL [31]. 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 thermody-

namically likely. MOLPAK currently supports a total of 38 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.

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

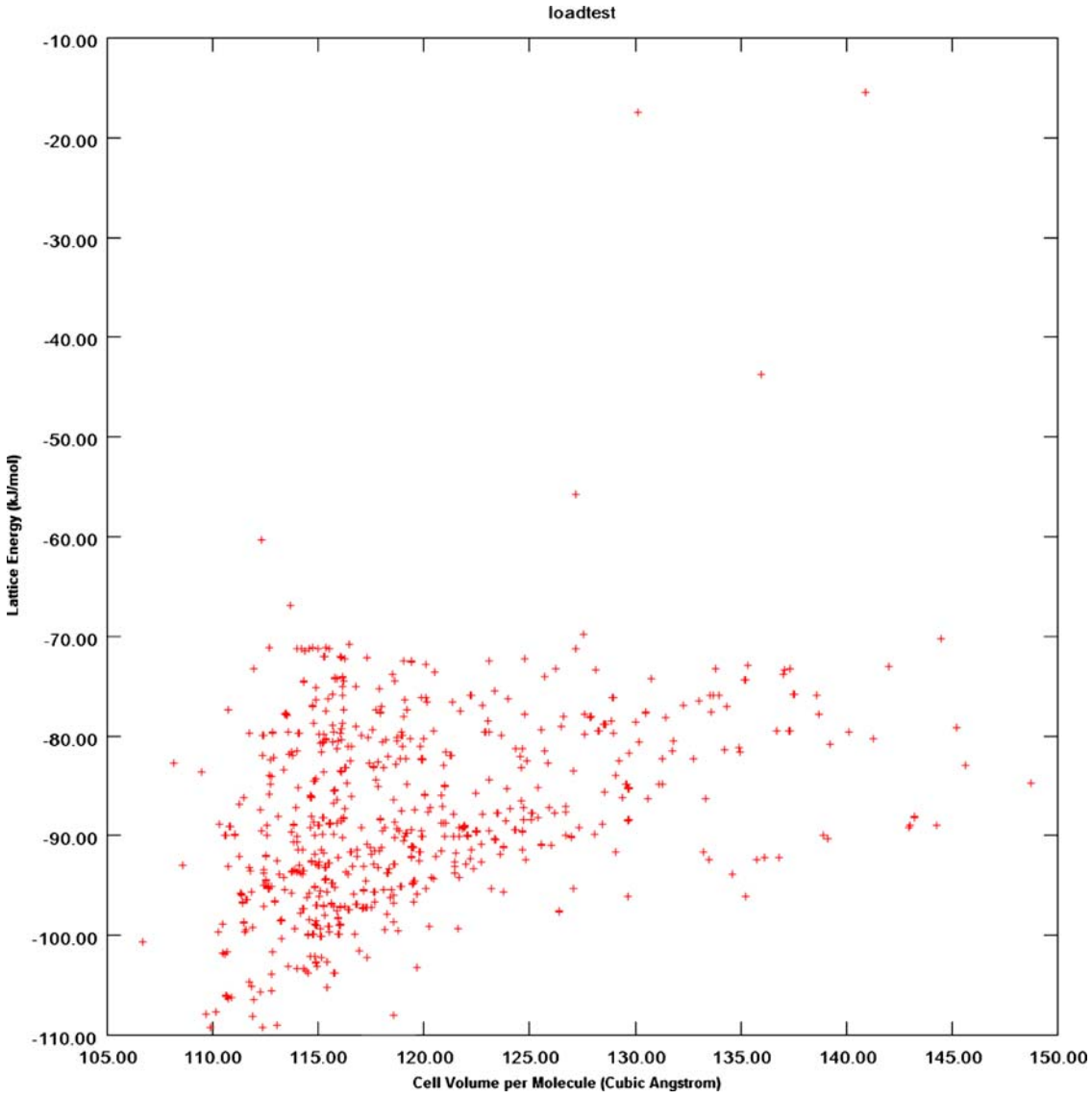

Figure 1. Scatterplot showing the energies and volumes of hypothetical crystal structures of 5-azauracil, a biologically active compound. The known crystal structure corresponds to the lowest energy structure.

CCLRC portal [10]. This stored data has been used [27] 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. This was, in part, due to the emergence of vast computational power on a Campus Grid at UCL that enabled interactive exploratory searches at lower precision that could then be followed by higher precision searches in an over-night workflow execution. These changes in scientific method demanded changes in 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 order and number of jobs they want to execute in parallel. They added 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 (CML) [22] and another service that renders such CML data into the scatterplot format shown in Figure 1. Once likely polymorphs are found scientists want to specify workflows that automate the upload of 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. Subject to resource availability, each of the possible 38 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.

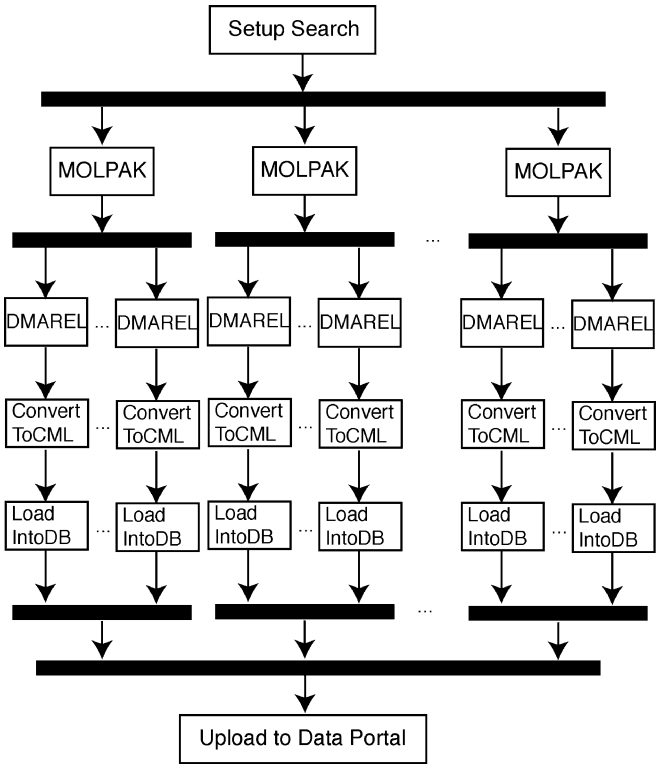

Figure 2. Overview of a polymorph search workflow.

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 from the OMII. GridSAM implements the Job Submission Description Language (JSDL) defined by the GGF [17]. GridSAM is able to implement data staging needs that may be caused by firewalls and only partial visibility of input and output data to the other services.

2.3. *Suitability as an Experiment*

The research method we have employed uses a *replicated and controlled experiment* [32]. 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 this experiment will allow us to answer the questions posed at the beginning of this section.

Before using systematic Grid computing techniques, scientists were using a batch queue on a Silicon Graphics server with four CPUs. The searches lasted between one to four months. Apart from the limited CPU power available a substantial reason was the high-degree of manual interaction for job submission, job monitoring, data conversion and data analysis.

We then implemented a user interface that scientists used to control a polymorph search. The interface is shown in Figure 3 and the scientific application is described in detail in [23]. 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 fully automates the search and 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 $(38 \times 200) = 7,600$ concurrent invocations of MOLPAK and DMAREL. MOLPAK and DMAREL jobs may take any time between 5 min and 1 h 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 GB and scientists might wish to complete up to 40 studies during a month, producing a 0.25 TB 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 Lan-

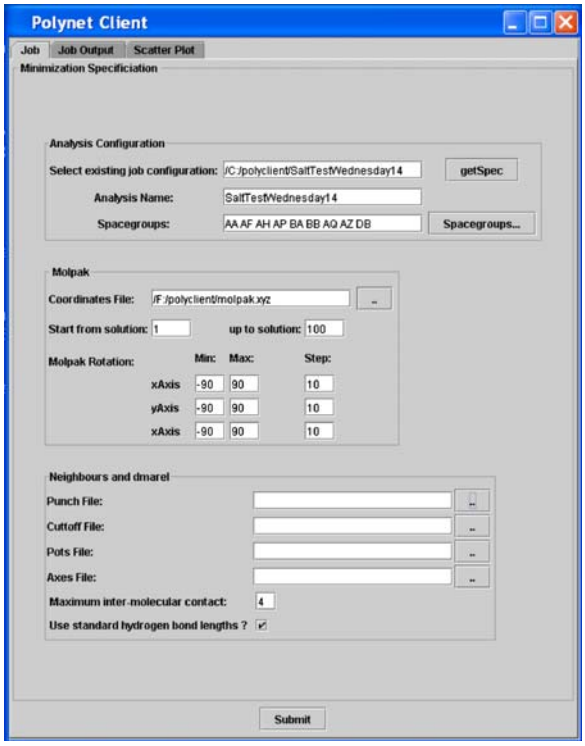

Figure 3. User interface to control polymorph search.

guage, extraction of data for visualisation purposes, purging unnecessary interim results and enriching selected results with meta data prior to upload 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 WSDL interfaces [8]. 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 using 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 signalling and responding to faults.

Control Flow and Data Flow: In order to determine control flow should allow for clustering of basic 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*

Workflows can be executed on the same machine where they have been defined. This is appropriate if the workflows are short in duration and are changed very frequently. If workflows are services that are used by other workflows, they are long running, and need to be highly scalable, they might be better executed on some dedicated server. They then need to be transferred from the machine where they were defined to the server where they can be interpreted and enacted by an interpreter that implements the operational semantics of the workflow definition language. We refer to this transfer as *deployment* and as the workflow language interpreters as *workflow engines*.

Testing Grid Services: Before deployment and enactment of scientific workflow can start proper, we

believe the designer of a workflow should 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 using test drivers written in JUnit [5] and be controlled using Ant [4].

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 Grid services.

4.1. *Defining Data Flow*

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. We note that quite often the bulk of

the scientific data themselves are passed directly between services without any involvement of the BPEL engine as typically only a small subset of those data are required to control the overall workflow.

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 that are to be orchestrated. As these Grid services are potentially defined by independent organisations there is a possibility for name clashes that are resolved using XML's namespace mechanism.

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 (indicated with prefix `gs`).

```
<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.

4.2. *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 primitives to define 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, an XPath query that extracts data from some other variable, or the result of calling a procedure in a programming language, such as Java or JavaScript that can be incorporated using the BPEL extension mechanism. 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 XML document 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 eXtensible Style Language Transformation (XSLT) between the XML input data and a Scalable Vector Graphics (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 that are less interesting and were omitted from this discussion.

4.3. *Defining Control Flow*

In order to gain Turing completeness a language has to introduce further primitives for determining control flow. 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 provides for non-deterministic choice.

An example of a loop 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 sec 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>
```

The non-deterministic control flow primitive pick awaits the occurrence of a set of events and then executes the corresponding activity associated with that event. The two types of possible events are the arrival of a message or the occurrence of a pre-specified timeout. One use of the pick activity is illustrated in the example below. This example shows a pick with three branches; two branches will execute upon receipt of a particular message and the third one will execute, if none of these messages have been received within a specified time limit. In the example the pick waits for the addition of some data to a list of data items on the first branch. This allows the process to collect data from various sources, until it is told that all required data items have been gathered. The second branch is activated upon receipt of a message indicating that data submission has finished and processing of this data can now start. Finally, the last branch models a timeout that will execute, if no message has been received within 5 min.

```
<pick>
 <onMessage
  partnerLink='addDataItemPartner'
  portType='tns:gsSubmit_addDataPT'
  operation='addDataItem'
  variable='in_dataVar'>
   <!-- increment index variable and
        assign new data item to list -->
   <assign name='incrementIndex'...
   ...
 </onMessage>
 <onMessage
  partnerLink='conversionPartner'
  portType='tns:gsSubmit_convert2CMLPT'
  operation='convertMListToCML'
  variable='submitDataVar'>
   <!-- reset index variable and
        list of data items -->
   <assign name='resetIndex'...
   <!-- invoke conversion service -->
   <invoke name='listConversion'...
 </onMessage>
 <onAlarm for='PT5M'>
   <!-- reset list and index variable -->
   <assign name='restList'...
   <assign name='resetIndex'...
 </onAlarm>
</pick>
```

The switch statement allows for the conditional execution of activities by specifying case and otherwise branches. The simple example below shows the conditional assignment of some data to a list of crystals. The switch checks in its condition whether the data we are interested in is actually present in the result of some computation that has been carried out. If the relevant data is found, we assign the resulting data structure to a particular position in a list (the index is indicated by `crystal[#d#]` in this case). There is a default otherwise branch that will execute an empty action in case none of the specified conditions hold.

```
<switch>
 <case
  condition="count(bpws:getVariableData(
           'parseCrystalRespMessageVar',
           'body','//cml:crystal/@z'))>0">
  <assign name="crystalArray"">
  <copy>
   <from variable
   ="parseCrystalRespMessageVar"
         part="body"
         query="//cml:crystal"/>
   <to variable
   ="invokedmarelindicesResponseVar"
```

```
   part="body"
   query="/cml:PackingTypeResult/cml:
    crystal[#d#"]/>
  </copy>
  </assign>
 </case>
</switch>
```

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>
```

Hierarchical Composition: As mentioned earlier, hierarchical composition of workflows enables us to deal with the complexity of large workflows and provide for reuse of workflows by exploiting the fact that each workflow is itself expressed as a Web service, which can be invoked. Figure 4 illustrates how our case study application, the polymorph search, has been decomposed into a number of smaller sub-workflows. The main workflow or driver of the overall workflow is indexedmolpakdmarel. This workflow gathers input data and then invokes the invokemolpak sub-workflow, which in turn generates a JSDL description of its job requirements and invokes the gsSubmit workflow. The invokemolpak workflow is responsible for the generation of molecule packing types. The gsSubmit workflow, which is reused by several workflows, submits jobs to GridSAM Grid service and subsequently monitors the status of submitted jobs. It eventually informs its client of the status of the submitted jobs. Once an invocation to invokemolpak has returned successfully, the main workflow uses these data to commence the calculation of the physical properties of a molecule packing by invoking invokedmarelindices. This sub-workflow again expresses its job requirements in JSDL and makes use of the gsSubmit workflow for actual submission of the jobs. Finally, the main workflow submits the results of invokedmarel-indices to the visualiser workflow, which creates graphical and tabular representations of the resulting data. Even though this discussion omits any details about the parallel execution of workflow instances, which actually takes place in the polymorph search workflow, it should become apparent that hierarchical composition substantially simplifies the design and testing of large workflows.

Asynchronous Invocation and Message Correlation: Support for asynchronous invocations is needed in order to prevent client workflows from blocking and waiting for a response from long-running operations. BPEL supports asynchronous interactions between workflows and services in terms of one-way operations, which only accept an input and return immediately. The requester of an asynchronous service initiates the interaction by invoking a one-way operation on the service provider (we call this operation initiate). At a later point in time, the service provider returns the result of this operation by invoking a one-way callback operation on the requester (we call this operation onResult). An important concept in asynchronous invocations is that of message correlation. The service provider needs a means of specifying which process instance a particular callback message is intended for. BPEL supports message correlation through the definition of correlation sets. We illustrate these concepts further by considering an asynchronous version of our gsSubmit workflow, shown in Figure 5.

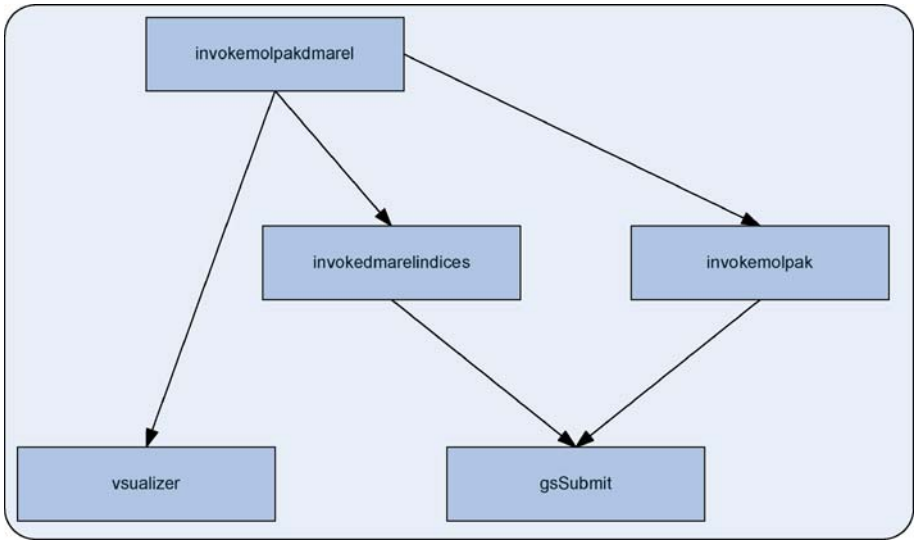

Figure 4. Decomposition of polymorph search into five BPEL processes.

The left-hand side shows a sample requester workflow that submits jobs and monitors the status of these jobs. The right-hand side shows the asynchronous gsSubmit workflow that provides job submission and monitoring to the requester. The bold elements of the diagram emphasize the elements comprising an asynchronous interaction among the two workflows. The requester and provider workflow are instantiated by the main workflow. The requester workflow then submits its jobs to the provider in a synchronous manner and receives the corresponding job identifier as a response. Next, the requester uses the provider's initiate operation (encapsulated in the middle branch of the pick activity) to register its interest in updates to the status of its jobs. For correlation purposes, the requester specifies that a property accessible to both workflows called JOB_ID and initialised to the previously returned job identifier shall be used for correlating any responses to the requester. The registration for updates is an asynchronous operation. That is, the client is free to continue with other activities, whilst the provider stores the job id in a list and queries another service for the status of the jobs in its list. The provider determines whether useful job status information is available for a particular job identifier and returns status updates to the registered client using the requester's onResult operation. It uses the previously agreed value for the correlation set (i.e., JOB_ID) in order to enable the BPEL engine to determine the correct process instance that is to receive the update. As we can see in the diagram, the requester workflow synchronises by using a receive operation accepting input via its onResult operation. Receipt of a status update at the requester then completes the asynchronous interaction between these two workflows.

The example of our asynchronous gsSubmit workflow demonstrates an important benefit of asynchronous interactions for the overall performance of a workflow. By offering a service in an asynchronous manner, the same gsSubmit process instance can support a number of requesters and supply them with results as they become available reducing any overhead caused by a requester blocking in order to wait for a response. More importantly, we avoid the need for creating a new process instance for each requester and thus the overall workflow becomes more scalable.

Compensation Handling: Compensation handling is required in order to allow workflows to return to a consistent state upon detection of any kind of failure such as a failed invocation or a variable returned from an invocation, whose value is outside the valid range as defined by a particular application. BPEL supports compensation by the definition of compensation handlers. Compensation handlers can be defined at the scope or process level and facilitate definition of sequences of activities that undo the effect of some previously executed activities. It is the responsibility of an application to define any custom

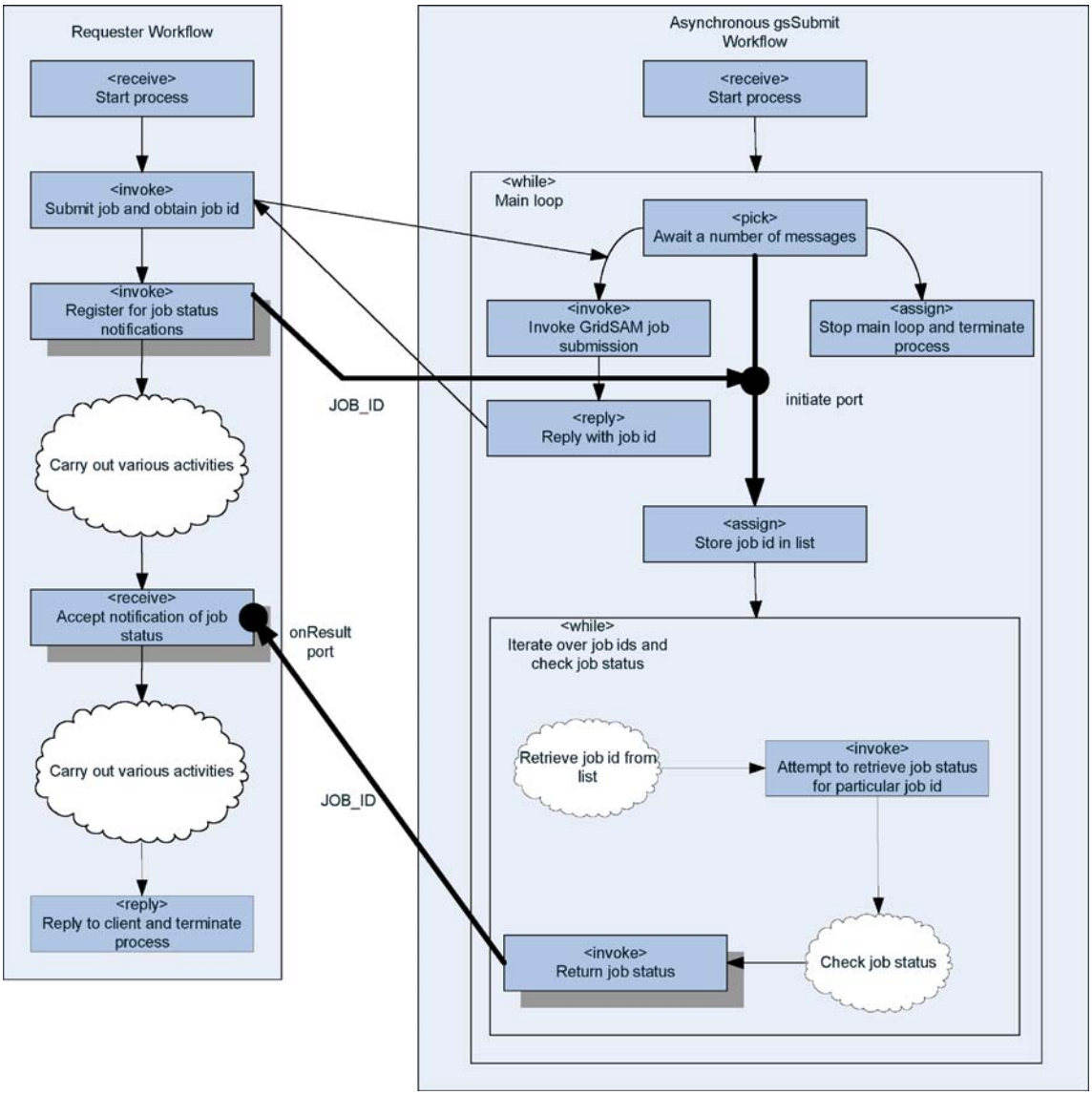

Figure 5. Asynchronous invocation and correlation.

compensation behaviour. The example below shows a compensation handler that is used to delete a job from a queue in case a failure occurs during submission and submit the job to another queue. The compensating activity uses the variable received by its associated invoke (i.e., JobSubmitResponseVar) to invoke an operation called CancelSubmit on the same partner and thereby reverses the previously completed job submission operation. Then, the handler invokes a submission operation on another partner called BackupQueue.

```
<scope name="submitScope">
<compensationHandler>
 <invoke partnerLink="Queue"
       portType="SP:JobSubmission"
       operation="CancelSubmit"
       inputVariable="getResponse"
       outputVariable="getConfirmation">
 </invoke>
 <invoke partnerLink='BackupQueue'
       portType='alt:JobSubmission'
       operation='SyncSubmit'
```

```
      inputVariable='JobSubmitRequestVar'
      outputVariable='JobSubmitResponseVar'>
 </invoke>
</compensationHandler>
<invoke partnerLink="Seller"
      portType="SP:JobSubmission"
      operation="SyncSubmit"
      inputVariable="JobSubmitRequestVar"
      outputVariable="JobSubmitResponseVar">
 </invoke>
</scope>
```

Compensation handlers can be invoked explicitly by using the *compensate* activity or implicitly due to the occurrence of a fault. Faults can be communicated by a partner via its reply activity and received by invoke activities. In general, faults can be thrown anywhere in a workflow. A BPEL fault handler catches faults and can handle them by, for example, calling a suitable compensation handler. The example below shows the definition of a simple faultHandler, which, upon catching a fault indicating that the submitted job has not been scheduled for execution within a given deadline invokes the above compensation handler to cancel the submission and instead submit to an alternative queue.

```
<faultHandlers>
 <catch faultName="SP:scheduleTimeOut"
  faultContainer="error">
    <sequence name="fault-sequence">
    <compensate scope="submitScope"/>
      ...
```

These mechanisms enable our workflows to deal with exceptional and unexpected events (through implicit compensation handlers) and to continue execution even after a part of it had to be reversed.

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. BPEL supports both synchronous and asynchronous invocations and provides compensation handling mechanisms that are inspired by SAGAS [14], which have been an advanced transaction mechanism for long-running activities.

4.4. *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 metadata to accompany the search results when archived at a data portal as it enables other scientists to repeat a search. Also these other scientists are more likely to be able to understand and use a widely adopted standard than a proprietary workflow language.

Secondly, it is now easier to modify the workflow as the orchestration can be manipulated more easily. To test this we set up an experiment that we replicated with BPEL and the hard-coded client workflow. The experiment consisted in adding functionality for extracting the data from the CML files and converting them into a user presentable format. This involved adding the invocation of two new web service to do the transformation. For the dedicated client it took approximately two days to code the data extraction and add the orchestration of the additional web services and test the client. It took another two days to install the updated code on all 10 PCs that the scientists use. The construction of a BPEL workflow that performs the data extraction

and visualisation was done in half a day and deployment was a matter of minutes to perform the sftp upload to the BPEL engine.

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 approximately 5 MB 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. Version 2.0 of the BPEL language will address this shortcoming and have a parallel for each operator that has the same semantics as our indexed flows.

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 commercially available ActiveBPEL environment provide such graphical notations. We have defined such a graphical notation and built a visual editor as a plug-in for Eclipse [13] 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 plug-ins. We have used these plug-ins in the example above to define the submission of a DMAREL or MOLPAK job submission as a single element. The semantics of each of these plug-in types are defined by a Java class that creates standard BPEL. The classes are then incorporated into Eclipse using its plug-in mechanism and are executed during deployment of a graphical workflow definition so that the deployed code remains standard BPEL. In this way we retain the ability to use an off-the-shelf BPEL engine for the enactment of scientific workflows. Again the detailed description of this mechanism is beyond the scope of this paper and will be subject to the companion paper mentioned above.

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 a subset of BPEL 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 and did not have any fault–tolerance mechanisms. It took approximately 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 2005 IBM integrated a BPEL engine into their Websphere product family and Microsoft released a version of BizTalk Server that handles BPEL. We then switched from using our own implementation to these industrial environments.

We believe that the freely available ActiveBPEL engine will be of more interest to the academic community and therefore discuss our experience with 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 focus the discussion of our experience on the extent with which ActiveBPEL meets the functional and non-functional requirements delineated above.

5.1. *Functional Requirements*

Validation: 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 mistakes. We have found this service to be extremely useful as it has helped us to correct our BPEL workflows prior to attempting deployment into the engine.

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 MB BPEL file with associated deployment descriptors and WSDL files is archived in a mere 57 KB. 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.

Monitoring: 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 6 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 84,000 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 present any results. Hierarchical decomposition of the workflow significantly ameliorated this problem but the performance for large component processes is still poor. The engine, can however also log the progress of the workflow and we found this log to be fully scalable and sufficient to understand the progress for large-scale workflows.

5.2. *Reliability*

Reliability and fault–tolerance are important for scientific workflows as these are often long running activities. There are many different kinds of faults the engine has to tolerate. These include the ability to handle failures of orchestrated services, failures in communication with orchestrated services, hardware failures of the node that executes the BPEL engine itself or even deliberate shutdowns imposed by administrators. We now discuss how ActiveBPEL copes with all of these faults.

As described above, ActiveBPEL fully implements BPEL 1.1 including compensation handlers, which can be used to define in a workflow how the engine should behave if one of the orchestrated services fails.

For the handling of the remaining fault classes ActiveBPEL can be configured to persist its process state. ActiveBPEL persists the details of every active process, it persists the scoping information, including variable values, it also persists the content of its alarm queue and message correlation data. ActiveBPEL relies on the availability of an existing relational database management system (RDBMS) and supports persistence mappings to Oracle, SQLServer, DB2 and MySQL. As this project is part of the OMII Managed Programme, which has adopted and includes a distribution of PostgresQL as its database management system, we have added persistence support for PostgresQL to ActiveBPEL. This was possible as we had access to the open source distri-

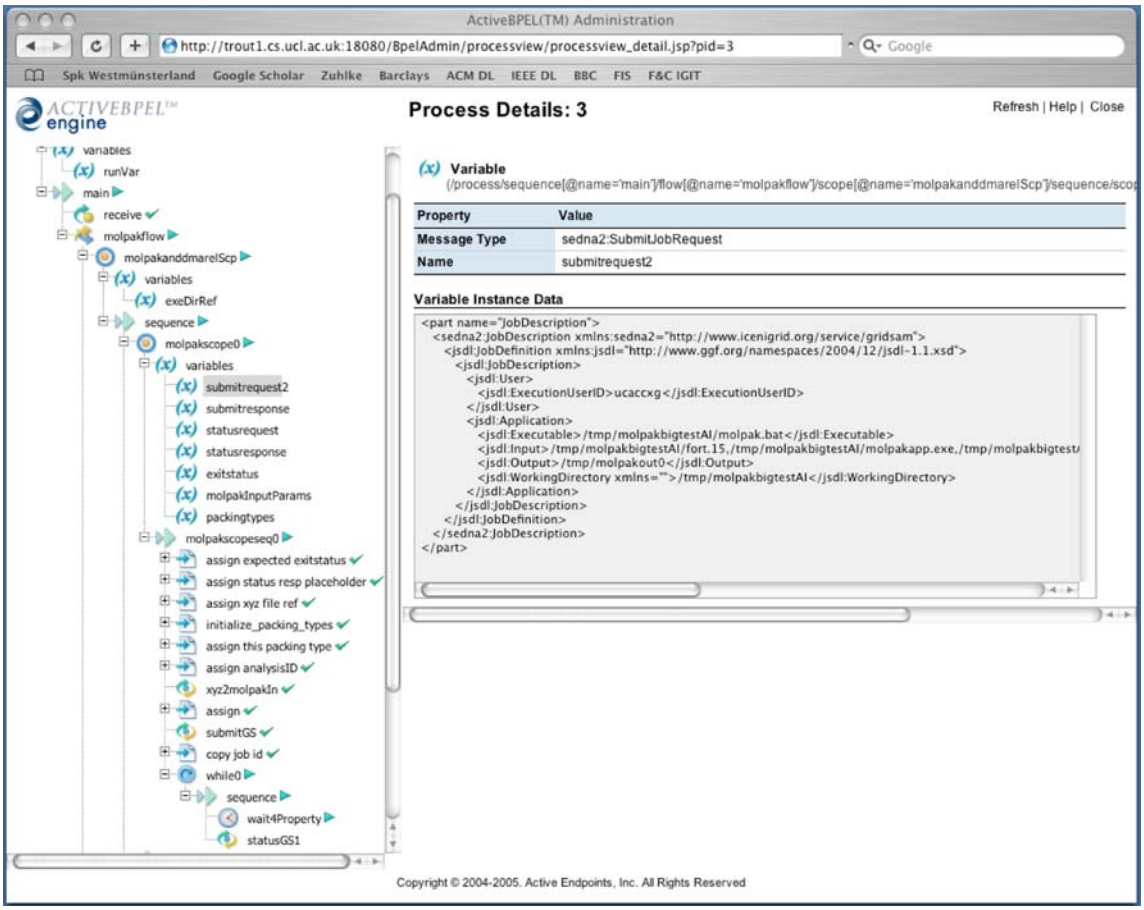

Figure 6. User interface for monitoring workflows.

bution. It took us about a week to modify the DDL and add amend some queries to work on PostgresQL.

Unlike MySQL, Postgres supports the notion of transactions, which allows ActiveBPEL to perform updates to persistent process state under transaction control. Thus any assignment, receive or invoke effectively performs a database transaction to modify the persistent state in addition to the state change in memory. This means that should a failure occur during process execution the restart will happen at the state of the last completed transaction.

We have relied on these fault tolerance mechanisms in practice when executing the polymorph search. The worker nodes in our Condor pool are rebooted every evening at 7 p.m. and by using compensation handling and relying on the fault tolerance mechanisms embedded in GridSAM and Condor our process was able to survive these downtimes and continue the execution. Moreover, once a week the cluster that executes the ActiveBPEL engines in production is rebooted at the end of a night. Ongoing polymorph searches were able to survive these reboots by relying on the persistence mechanism described above.

5.3. *Performance*

We performed a number of performance tests of the ActiveBPEL engine to measure execution speed. We have used two simple benchmarks for this purpose. The first one forks a variable number of concurrent flow to see how many flows can be created and how fast these flows can be performed. The second benchmarks iterates in a loop for a variable number of times and increments a counter (Table 1).

We have performed these benchmarks on low-end equipment, i.e., a single CPU Dell Server with hyper-threading, 2 GB of main memory and 8 GB of virtual

Table 1. Performance of loops and flows.

	Time [ms]			Time [ms]	
Number iterations	Total	Per iteration	Number flows	Total	Per flow
0	750	n/a	0	750	n/a
10	750	0	10	750	0
100	750	0	100	750	0
1,000	1,050	0.6	1,000	752	0.002
10,000	3,510	0.24	10,000	3,680	0.29

memory running Linux RedHat Enterprise 3.0. ActiveBPEL and Postgres were installed on a local RAID array. The figures show the average elapsed real time of five identical executions in milliseconds. They were measured using `/usr/bin/time` on the client side. Persistence and Logging of the ActiveBPEL engine were disabled. Moreover the engine had an empty thread pool, which means that ActiveBPEL had to start a new Java and OS thread for each process invocation. The results are shown in Table 1. The first column shows the total number of iterations. The second column shows the elapsed time of the entire operation and the third column shows the time it takes to complete a single iteration. The fourth, fifth and sixth column show the same data for a flow.

The first row shows the time it takes to complete a remote invocation of a workflow that immediately returns, which is the overhead of a BPEL process invocation. This is 750 ms and includes the time for the web service client to create the SOAP message, for the message to be transmitted across a 1 GBit/sec local area network, for the BPEL engine to parse the message, to create the BPEL process instance, which includes starting an operating system thread, to assign the payload of the SOAP message to the input variable, to compose the reply data structure, to transform the reply data into a SOAP message, to terminate the operating system thread, to transmit the SOAP response to the client and for the client to parse the response. For the purpose of calculating the time it takes to execute a single loop iteration and to start a single flow we subtract this overhead from the total elapsed time and divide the difference by the number of iterations of the loop or the number of flows.

We note that the performance of an iteration is independent of the size of the flow or the loop. This is not the case for starting a flow. We attribute this to the size of the BPEL processes that grow linear in the number of flows (more below on how this can be overcome in the future).

In absolute terms the performance shown above means that the ActiveBPEL engine is unlikely to be a performance bottleneck as we believe it is perfectly acceptable to have an overhead of under 4 s, for example when submitting 10,000 jobs as concurrent flows. The ActiveBPEL engine achieves this performance by avoiding the use of Java/OS threads for the implementation of flows and instead it implements its own scheduling mechanism that implements interleaved concurrency of flows.

5.4. *Scalability*

The workflow of a full polymorph search at the highest possible precision contains approximately 84,000 basic activities. The ActiveBPEL engine was unable to deploy such a large workflow in one process. The XML representation of the workflow had a size of just under 50 MB, which was larger than the engine could parse during deployment. The limit given the hardware configuration described above lies around 10 MB. By establishing the hierarchical decomposition of the overall workflow into four components and one composite workflow we achieved a considerably smaller memory footprint of about 1.5 MB and were able to deploy the entire workflow.

For reasons of simplicity, we chose to use synchronous communication between composite and component processes initially. This meant that for every invocation of a component workflow a new BPEL process was created. The creation of a new BPEL process requires two OS threads. One thread is used by Tomcat and Axis to handle the correlation

between invocation and reply. The other one is created by the ActiveBPEL engine to interpret the BPEL process. At the peak, our search therefore attempted creation of $38 \times 200 \times 2 = 15200$ threads, which is well beyond what most operating systems can handle. Our Linux kernel was configured to support 2,048 threads (1 k per hyperthreaded CPU) and use 2 MB of stack memory for each thread. As a result we ran out of stack memory, the process table overflowed and the workflow aborted. We could have reconfigured the Linux kernel to allocate less stack memory and have a larger process table, but this would have just moved the scalability limit and not presented a more generalisable solution.

We then limited the size of the thread pool of Tomcat and the ActiveBPEL engine to 750 threads each in an attempt to execute the process using the available threads, but this resulted in deadlocks. These deadlocks were due to the fact that the processes communicate with each other and one process could not progress if its partner was stuck in the thread queue and the process would then be blocked itself. A detailed discussion of deadlock avoidance in orchestrated scientific workflows is beyond the scope of this paper and will be subject to another companion paper.

There remain three real solutions to this scalability problem. The first solution takes advantage of the fact that we would not have the computational resources to execute 15,000 jobs in parallel anyway. Our workflow submits jobs to a Condor pool and the largest number of nodes the Condor scheduler of our pool would allocate to the user at any point in time is about 200. Thus it is sufficient to ensure that about 400 jobs are being held in the Condor queue at any point in time. We have achieved this by modifying the composite workflow to artificially constrain the degree of concurrency. Table 2 shows the memory usage depending on the number of active BPEL processes. The drawback of this solution is that the workflow no longer only reflects the logical composition of the scientific process but also encodes constraints due to the available computational resources. Changes to the workflow would then be required when the resource allocation policy changes or more nodes become available. But then these changes are now very easy to achieve.

The second solution is to distribute the workflow across multiple BPEL engines. This is possible as each of the component processes are web services in their own right and it is entirely transparent to the workflow definition whether they execute on the same or a different machine. The disadvantage of this solution is that there is considerable administrative overhead involved in setting up the engines and distributing the workflow deployment across them.

The most elegant solution is to change the communication model and use asynchronous invocation with message correlation for the communication between the processes as was described in Section 4 above. Using this asynchronous communication model we only need 40 BPEL processes and 95 threads to execute the full polymorph search.

For a workflow to be truly scalable not only is the scalability of the workflow engine important, but any of the orchestrated Grid services needs to be sufficiently scalable and well engineered.

When services are orchestrated by a BPEL process it is fairly likely that the same service will be executed concurrently. In our example this occurred for the data extraction service and the transformation service that produces the results web page from an input CML data structure. This means that these web services must be thread safe. They must protect access to any state information that they encapsulate through monitors or semaphores.

Even stateless services need to consider concurrency and scalability. Our polymorph search uses a service that is in charge of performing the data conversion between MOLPAK output and DMAREL input. When two MOLPAK packing type calculations finish at the same time, the workflow invoked the subsequent conversion 400 times within 10 s. One such execution takes longer than 10 s and therefore all 400 executions of the service happen concurrently. Both the workflow engine and the container hosting the staging service coped fine with this degree of

Table 2. Relationship between active processes, virtual memory use and number of threads.

Number of processes	Virtual memory [MB]	Number of threads
21	248	56
51	280	114
101	336	211
201	460	417
403	589	810
604	690	1,213
805	783	1,610

concurrency, but the operating system imposed a soft limit of 1,024 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 is that to achieve scalable workflows it is important that orchestrated services need to be built to scale too.

5.5. *Summary*

In summary, we found the ActiveBPEL engine to satisfy the needs of orchestrating Grid services into scientific workflows. The engine itself is robust and fault tolerant, it executes workflows efficiently and its scalability is limited only by the available hardware resources.

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 [25] 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 [12]. 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 [18]. FlowMark was the predecessor of the WebSphere MQ WorkFlow product of IBM and greatly influenced WSFL [19]. WSFL was then later merged with XLANG [30] 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 [11], GridFlow [7] and GridAnt [1] 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 [24] was developed primarily for the *ad hoc* composition of Bioinformatics applications. It has 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 for which it provides an integration with programming and scripting languages. Taverna workflows are executed within the modelling environment client-side, while our workflows are executed on dedicated servers. GridAnt is a relatively lightweight environment for the definition and monitoring of workflows based on Ant [4]. 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 [29, 20] and GSFL [16] 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 [20] 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., [9] describe 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 the ActiveBPEL engine, which is freely available to all readers of this article has the required quality attributes to be able to handle large-scale scientific workflows.

We have bundled ActiveBPEL and the Eclipse-based scientific workflow development environment and have released the environment on http://sse.cs.ucl.ac.uk/omii-bpel. Future work will concentrate on improving the usability of our Eclipse-based modelling environment by integrating it with plug-ins for WSDL and Schema definition.

References

1. K. Amin, G.v. Laszewski, M. Hategan, N.J. Zaluzec, S. Hampton and A. Rossi, "GridAnt: A client-controllable Grid workflow system", in D. King and A. Dennis (eds.), *Proc. of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) – Track 7*. IEEE Computer Society, 2004, p. 70210c.
2. T. Andrews, F. Cubera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic and S. Weerawarana, Business Process Execution Language for Web Services Version 1.1. OASIS, http://ifr.sap.com/bpel4ws, 2003.
3. A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagi, I. Trickovic and S. Zimek, Web Service Choreography Interface (WSCI) 1.0. W3C, http://www.w3.org/TR/wsci, 2002.
4. S. Bailliez, et al., "Ant User Manual", Apache Jakarta, http://jakarta.apache.org/ant, 2005.
5. K. Beck and E. Gamma, "Test-infected: Programmers love writing tests", in *More Java Gems*. Cambridge University Press, New York, New York, USA, pp. 357–376, 2000.
6. B. Butchart, C. Chapman and W. Emmerich, "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*, EPSRC, 2003, pp. 810–816.
7. J. Cao, S.A. Jarvis, S. Saini and G.R. Nudd, "GridFlow: Workflow management for Grid computing", in: *3rd Int. Symposium on Cluster Computing and the Grid*, IEEE Computer Society, 2003, pp. 198–205.
8. R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer and S. Weerawarana, Web Services Description Language. W3c working draft, W3C, http://www.w3.org/TR/wsdl20, 2004.
9. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh and S. Koranda, Mapping abstract complex workflows onto Grid environments. *Journal of Grid Computing* Vol. 1, No. 1, pp. 25–39, 2003.
10. G. Drinkwater, K. Kleese, S. Sufi, L. Blanshard, A. Manandhar, R. Tyer, K. O'Neill, M. Doherty, M. Williams and A. Woolf, "The CCLRC data portal", in S. Cox (ed.), *Proc. of the UK e-Science All Hands Meeting*. EPSRC, UK, 2003, pp. 540–547. ISBN 1-904425-11-9.
11. J. Frey, "Condor DAGMan: Handling Inter-Job Dependencies", Technical report, University of Wisconsin, Dept. of Computer Science, http://www.cs.wisc.edu/condor/dagman, 2002.
12. A. Fuggetta, "Software process: A Roadmap", in *The Future of Software Engineering*, ACM Press New York, New York, USA, 2000, pp. 25–34.
13. E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns and Plug-in*. Addison-Wesley, 2004.
14. H. Garcia-Molina and K. Salem, "SAGAS", in *Proc. of ACM SIGMOD Annual Conference*, 1987, pp. 249–259.
15. J.R. Holden, Z. Du and H.L. Ammon, "Prediction of possible crystal structures for C-, H-, N-, O-, and F-containing organic compounds", *Journal of Computational Chemistry* Vol. 14, No. 4, pp. 422–437, 1993.
16. S. Krishnan, P.W. and G. v. Laszewski, "GSFL: A workflow framework for Grid services", Technical Report Preprint ANL/MCS-P980-0802, Argonne National Laboratory, 2002.
17. W. Lee, S. McGough, S. Newhouse and J. Darlington, "A standard based approach to job submission through web

services", in S. Cox (ed.) *Proc. of the UK e-Science All Hands Meeting, Nottingham*. EPSRC, UK, 2004, pp. 901– 905. ISBN 1-904425-21-6.

18. F. Leymann and D. Roller, "Business process management with FlowMark", in *Compcon Spring '94: Digest of Papers*. IEEE Computer Society, 1994, pp. 230–234.
19. F. Leymann, "Web services flow language", Technical report, IBM, 2001.
20. S. Majithia, M. Shields, I. Taylor and I. Wang, "Triana: A graphical web service composition and execution toolkit", in *Proc. of the 4th Int. Conference on Web Services*. IEEE Computer Society, 2004, pp. 514–524.
21. W.D.S. Motherwell, H.L. Ammon, J.D.D.A. Dzyabchenko, P. Erk, A. Gavezzotti, D.W.M. Hofmann, F.J.J. Leusen, J.P.M. Lommerse, W.T.M. Mooij, S.L. Price, H. Scheraga, B. Schweizer, M.U. Schmidt, B.P. v. Eijck, P. Verwer and D.E. Williams, "Crystal structure prediction of small organic molecules: A second blind test", *Acta Crystallographica Section B-Structural Science*, Vol. 58, pp. 647–661, 2002.
22. P. Murray-Rust, "Chemical markup language", *World Wide Web Journal*, Vol. 2, No. 4, pp. 135–147, 1997.
23. H. Nowell, B. Butchart, D.S. Coombes, S.L. Price, W. Emmerich and C.R.A. Catlow, "Increasing the scope for polymorph prediction using e-Science", in S. Cox (ed.) *Proc of the 2004 UK E-Science All Hands Meeting, Nottingham*, Engineering and Physical Science Research Council, UK, 2004, pp. 968–971.
24. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat and P. Li, "Taverna: A tool for the composition and enactment of bioinformatics workflows", *Bioinformatics Journal* Vol. 20, No. 17, pp. 3045–3054, 2004.
25. L.J. Osterweil, "Software processes are software too", in *Proc. of the 9th Int. Conf. on Software Engineering*. IEEE Computer Society, 1987, pp. 2–13.
26. C. Peltz, "Web services orchestration and choreography", *IEEE Computer*, Vol. 36, No. 10, pp. 46–52, 2003.
27. M.L. Peterson, S.L. Morissette, C. McNulty, A. Goldsweig, P. Shaw, M. LeQuesne, J. Monagle, N.N. Encina, J. Marchionna, A.A. Johnson, J. Gonzalez-Zugasti, A.V. Lemmo, S.J. Ellis, M.J. Cima, and O. Almarsson, "Iterative high-throughput polymorphism studies on acetaminophen and an experimen tally derived structure for form III", *Journal of the American Chemical Society*, Vol. 124, No. 37, pp. 10958–10959, 2002.
28. S.L. Price, "The computational prediction of pharmaceutical crystal structures and polymorphism", *Advanced Drug Delivery Reviews*, Vol. 56, No. 3, pp. 301–319, 2004.
29. I. Taylor, M. Schields, I. Wang and O. Rana, "Triana applications within Grid computing and peer to peer environments", *Journal of Grid Computing*, Vol. 1, No. 2, pp. 199–217, 2003.
30. S. Thatte, "XLANG: Web Services for Business Process Design", Technical report, Microsoft, 2001.
31. D.J. Willock, S.L. Price, M. Leslie and C.R.A. Catlow, "The relaxation of molecular crystal structures using a distributed multipole electrostatic model", *Journal of Computational Chemistry*, Vol. 16, No. 5, pp. 628–647, 1995.
32. M. Zelkowitz and D. Wallace, "Experimental models for validating technology", *IEEE Computer*, Vol. 31, No. 5, pp. 23–31, 1998.