Application of Business Process Execution Language to Scientific Workflows

Asif Akram David Meredith Rob Allan

CCLRC e-Science Centre, CCLRC Daresbury Laboratory, Warrington, UK WA4 4AD
Email: {a.akram; d.j.meredith; r.j.allan}@dl.ac.uk

Abstract: This paper investigates the use of the Business Process Execution Language for Web services (BPEL4WS/ BPEL) for managing scientific workflows. The complexity, unpredictability and inter-dependency of the components in a scientific workflow often demand great flexibility in a workflow-language in order to support; 1) exception handling, 2) recovery from uncertain situations, 3) user interactions to facilitate interactive steering and monitoring, 4) dynamism to adapt to the changing environment, 5) compensation handling to reverse the effects of previous activities that have been abandoned, and 6) flexibility to support dynamic selection of services at runtime and to support changing data requirements. These requirements are illustrated with examples taken from a real scientific workflow; the e-HTPX project for high throughput protein crystallography. In the context of addressing these requirements, the features of the BPEL4WS specification are discussed, which is widely regarded as the de-facto standard for orchestrating Web services for business workflows. An investigation and proposal for the use of the Web services Invocation Framework (WSIF) [38] to extend BPEL is also provided. In summary, by extending BPEL where necessary (in conjunction with standard BPEL features), workflows can be successfully adapted to fulfill all the requirements of complex scientific workflows.

Keywords: BPEL, scientific workflow, SOA, Web services, Web Services Invocation Framework.

1. Introduction

Modular software is designed to avoid failures in large systems, especially where there are complex user requirements. However, even in a modular system, inter-dependency of sub-components can make it difficult to meet changing user requirements without re-engineering code. A SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents (services and clients). Agents can be composed in different ways to meet different requirements without having to re-write code. A SOA achieves loose coupling by employing two architectural constraints: 1) a small set of well-defined interfaces to all participating software agents and, 2) ensuring the interfaces are universally available to all providers and consumers. In simple terms a service is a function that is self-contained and immune to the context or state of other services. These services can communicate with each other, either through explicit messages (which are descriptive rather than instructive), or by a number of 'master' services that coordinate or aggregate activities together. This is complementary to Object Oriented Programming. A SOA can also define a system that allows the on-demand binding of online resources as independent services. This can provide a more flexible, loosely coupled set of resources compared to traditional system architectures with the possibility to optimize for performance or functionality.

The role of a SOA it is not only limited to making software components available for on-demand consumption. SOA principles also influence the business logic of services by encouraging good design, i.e., by promoting modular and dispersed components that can be separately developed and maintained. In doing this, a SOA can help reduce IT infrastructure overhead. This also allows a greater emphasis to be placed on the core value of a service. In recent years, Web services have been established as a popular 'connection technology' for implementing a SOA. The interface required for Web services is described in a WSDL file (Web service Description Language [10]). Services exposed as Web services (such as those wrapping and exposing legacy codes) can be integrated into complex workflows that may span multiple domains and organizations.

The situation however, becomes even more complex when combining multiple geographically dispersed Web services together to form a single composite service in the form of a workflow (or sub-workflow). In this scenario, a composition language is (usually) required. There is growing interest in the use of Web services for scientific, parallel and distributed computing [5]. This interest is fuelled by the availability of different Web service toolkits that are tailored to Grid Computing, such as Globus Toolkit (GT) 3.x, GT4.x, Open Middleware Infrastructure Institute (OMII) and IBM Emerging Technologies Toolkit (ETK). Scientific applications and services that are deployed and executed using Service Oriented Grid Computing infrastructures on top of Web services are called Grid Services.

Integration of two or more services into a more complex workflow that is managed by a workflow engine is called 'Service Orchestration.' In this scenario, the workflow engine orchestrates the interactions between the partner services by acting as a broker or 'middle-man.' Service orchestration differs from 'Service Choreography' [22]. Choreography occurs in peer-to-peer style workflows where communications occur directly between partners (i.e., involving no orchestration by an intermediate workflow engine). Consequently, a set of agreed rules are required to define the roles of each peer in the collaboration and to determine which messages should be exchanged at particular

stages. This process is known as 'Service choreography.'

With the popularity of Web services, several composition languages used for service orchestration and choreography have emerged in the last few years, such as BPML [36], XLANG [26], WSFL [20], WSCDL [18], BPEL4WS [2], GSFL [28] and WS Coordination [8]. BPEL4WS is generally regarded as the de-facto standard for composing and orchestrating workflows from Web services and will be discussed in this paper. It is supported by commercial vendors and by the open-source community who have contributed good quality BPEL4WS editors and efficient BPEL execution engines.

In Section 2, the requirements of scientific workflows are discussed (illustrated with examples from a real scientific process in e-HTPX). Section 3 outlines the main BPEL architecture and built-in features provided within the BPEL4WS. Section 4 provides a more in-depth analysis of the BPEL4WS implementation in the context of addressing the requirements of scientific workflows. Section 5 outlines some of the limitations of BBEL.

2. Scientific Workflows

According to Kammer [17], "The unpredictability of business processes requires that workflow systems support exception handling with the ability to dynamically adapt to the changing environment". Traditional approaches to handling this problem have fallen short, providing little support for change, particularly once the process has begun execution. Often, this unpredictability is even more apparent in fragile or long-lasting scientific workflows where research evolution often results in constantly changing requirements. Consequently, this can directly impact; a) the changing scientific functionality of a service, b) organizational constraints and policies, and c) dependencies on underlying data and legacy codes. As a result, a lack of interface compatibility with semantically equivalent and competing services can emerge. Dynamic scientific workflows often need to be complemented with execution support at runtime, such as dynamic scheduling, dynamic resource binding, and infrastructure reconfiguration. These constraints demand flexibility in a workflow language. Traditionally, workflows were not solely designed to call different Web services in a static or dynamic manner; rather, they are supposed to provide; a) an execution environment for exception handling, b) a compensation mechanism for partially completed or failed jobs, c) dynamical invocation of services (adaptive in nature to accommodate inconsistencies among services), and d) graceful recovery from un-expected circumstances.

Scientific workflows can be more demanding in both their nature and in their requirements than business processes. In the following section, some of the requirements that may be encountered in scientific workflows are outlined (some of which are also common to business processes). The requirements are hierarchical and co-dependent and include: 1) A Modular design; 2) Effective Exception handling (this depends on a modular design); 3) Effective Compensation (which should be possible from the exception handling code); 4) Workflow flexibility and adaptivity; 5) Support for long running transactions and conversations with partner services; and 6) Support for the management and monitoring of workflow processes. Before discussing each of the requirements, the e-HTPX project is briefly introduced.

2.1 Sample Application

The e-HTPX project [1, 31] is a distributed computing infrastructure designed for structural biologists to remotely plan, initiate and monitor protein crystallography experiments from their desktop computer. A number of key services for e-HTPX are being developed by leading UK e-Science, synchrotron radiation and protein manufacture laboratories (SRS Daresbury Laboratory [35]; The European Synchrotron Radiation Facility [32]; The Oxford Protein Production Facility [33]; York Structural Biology Laboratory [40]). The services being developed for the project define the 'e-HTPX Pipeline,' which is illustrated in Fig. 1, where:

Stage 1 – Protein Production
Stage 2 – Crystallization
Stage 3 – Data Collection
Stage 4 – Phasing (data processing)
Stage 5 – Solution of digital protein structure model
Stage 6 – Submission of protein model into public database.

The pipeline covers protein crystallization, delivery of a protein crystal to a synchrotron radiation facility, data collection using X-ray diffraction methods, and HPC data processing services. Remote access to these services is implemented by a collection of Web services. The data collection and data processing stages of the e-HTPX pipeline are particularly well suited to the application of workflow technologies. The following sections discuss some of the generic scientific workflow requirements. Real examples are illustrated from e-HTPX.

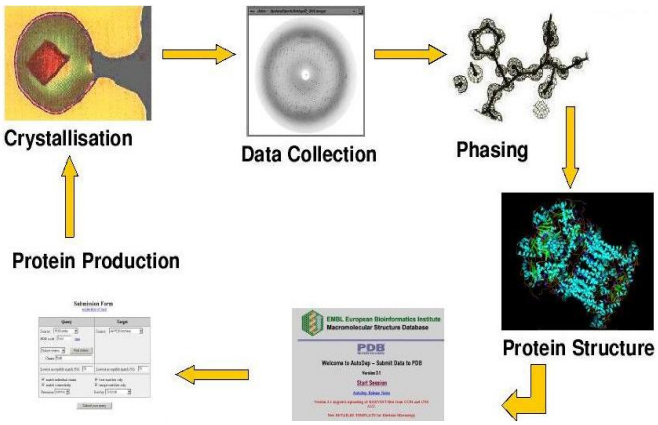

Fig. 1. e-HTPX Protein crystallography workflow.

2.2 Requirements for Scientific Workflows

2.2.1 Modular Design

The potential for complexity in a scientific workflow usually demands a thorough investigation of all the activities involved, including a detailed examination of inter-related dependencies and data. Failure of even one activity can hinder the successful execution of the whole workflow. Consequently, it is important to design modular workflows by grouping similar or related activities together and ensuring no side effects. Nutt [21] has discussed the modularization of workflows in detail in the context of: 1) Models of procedures, 2) Workflow modeling languages, 3)

Computation languages for modeling, 4) Workflow modeling systems and, 5) Issues in workflow models. These modules operate as 'black boxes' in the overall workflow, each having their own variables, computational logic, dependency constraints, security model, and event handlers. Workflow systems aim to support the organization of work and require that work activities are categorized and their relationships described in some form of process model.

The components in a large scientific workflow are arranged and connected around precise scientific requirements, not on programming logistics. A large repository of inter-connectable and re-usable components opens a wide range of possibilities for processing and relating diverse scientific data (the 'Science-Factory' concept) [34]. By modularizing workflows, module components can be made sufficiently scalable and reliable to serve as repeatable units. This helps automate the process of integrating diverse resources and scientific services across faculties (and even institutes). A workflow typically has a flexible n-tier design that facilitates the independent upgrade of individual components and tiers of the system. This helps ensure workflow sustainability, especially for future initiatives [14].

With regard to e-HTPX, modularity has been integrated from the outset and has been achieved by grouping inter-dependent processes in the e-HTPX pipeline into distinct Web service 'Categories.' Three Web service categories have been devised; 1) Protein Crystallization, 2) Data Collection (these services are tied to a particular synchrotron radiation facility, e.g., services involved with shipping crystals to a synchrotron and synchrotron specific data collection services), 3) Post-Data Collection Processing. The Web service categories are effectively standalone and can operate in isolation from each other. Each category is currently hosted at a different institution (crystallization services hosted at OPPF, data collection services hosted at specific synchrotron – SRS and ESRF, and HPC data processing hosted at Daresbury Laboratory). Modularizing related e-HTPX processes into categories offers a significant advantage to the user in that it is possible to consume services only for a single category, rather than being tied to the entire e-HTPX pipeline. In addition, modularization allows the service provider to implement fine-grained authentication mechanisms that are specific to the particular hosting institution (in conjunction to the e-HTPX wide authentication mechanism). In doing this, the institution can control access to their own resources (e.g., by facilitating special guest users/ home laboratory users).

2.2.2 Exception Handling

Even the best-planned workflows will most probably encounter exceptions during their execution. Managing the error involves handling the exception and rejoining the normal workflow so that it may proceed correctly. The occurrence of exceptions is a fundamental part of organizational processes [25]. Exceptions can result from a number of different sources such as inconsistent data [12], divergence of tasks from the underlying workflow model [6], unexpected contingencies [23] and un-modeled changes in the environment [27]. A workflow model should provide exception management with the ability to detect the occurrence of exceptions, or even the possibility of anticipating an exception thus improving the ability to react and recover within the minimum amount of time.

A well-designed workflow provides exception handling at different levels throughout its execution. Exception handling can be categorized into three distinct types; 1) Global Exception Handlers, which are defined at the core workflow level, 2) Scoped Exception Handlers, which are defined at the modular level and are related to a set of similar and/ or inter-dependent activities, and 3) Inline Exception Handlers, which are tightly coupled to specific activities or tasks and are limited in their scope and re-usability. Bartherness and Wainer [4] have classified exceptions into two main categories; 1) events modeled by the system which deviate from the normal flow are called 'expected exceptions' (described as 'variations' by Saastamoinen [23]) and, 2) un-modeled events which cannot be handled by the actual activity/ task are called 'unexpected exceptions.' Global exception handlers are generic in nature and are used mainly to handle unexpected exceptions. Inline exception handlers are specific to the problem and process and therefore handle expected exceptions/ variations. Scoped exception handlers are hybrid in nature; they are generic but relate only to a set of activities and tasks within the module. Kammer [17] suggests that artifact recovery and exception handling should be localized if permissible rather than propagating the exception up through the exception hierarchy.

For e-HTPX, the modularity of the services allows each service category to handle its own scoped exception types (i.e., exceptions that are related to that specific service category). The most problematic exceptions for e-HTPX are those of the 'Unexpected Global' type. Unexpected errors occur mainly due to the physicality of the tasks that are invoked by a service and their unexpected failure (usually related to physical problems). This type of exception can occur during data collection if a sample-changing robot fails to operate correctly (causing a "robot collision"), or a crystal sample is damaged in transit to the synchrotron. Because these errors are physical in nature, they are difficult to forecast by the service and handle elegantly in the service software. Furthermore, because these types of error cannot be anticipated by the workflow-engine at design time, workflows are often poorly equipped to handle these types of exception. In such cases, it may be necessary to restart the workflow from a specific breakpoint and roll back to a point where all prior activities were successfully completed.

2.2.3 Compensation Mechanism

The goal of compensation is to reverse the effects of previous activities that have been abandoned. A workflow should call a compensating operation from an exception handler if exceptions are thrown (expected or unexpected - it must be noted however, that not every exception requires compensation). An effective compensating mechanism is more dependent on the participating services than on the workflow itself. Nevertheless, all situations requiring compensation should still be modeled for the workflow at design time in conjunction with exception handling. Compensation can be defined as 'undoing steps in the business process that have already completed successfully,' and is one of the most important concepts in business processes. Compensation behavior must be explicitly defined by the services to ensure reversal of a specific operation

when required. Workflows should be able to call those compensating operations in order to release the specified resource. Reversal is however not always possible.

Compensation handlers can be more important to business processes where there are many different stakeholders with financial interests compared to scientific workflows. Nevertheless, scientific workflows often require compensation, especially when considering long running processes that make use of scarce resources (e.g., compute resources, synchrotron radiation resources). For e-HTPX, compensation is important for the data-processing stages of the e-HTPX pipeline (e.g., the Bulk Molecular Replacement service which utilizes HPC resources). Following submission of a compute job, the remote user can monitor the results as they are being generated. During the early stages of the compute job, the user may wish to cancel because poor and/ or un-expected results are being generated (a function of data quality). In this scenario, the compute resources can be released as a result of compensation functionality.

2.2.4 Flexibility and Adaptivity

Scientific applications can often be highly dynamic and subject to constant evolution. Consequently redesign and optimization of an existing process often becomes indispensable in a rapidly changing environment [15]. During these redesigns, workflows have to be adjusted repeatedly. In contrast, workflows modeled for traditional systems are often static and are not designed to cope with a rapidly and dramatically changing environment. In order to accommodate rapidly changing and dynamic environments, workflows have to be flexible and adaptive.

A workflow is flexible if it is subject to modification and change, usually in terms of the data types involved and service endpoint location. Flexibility is often required during the early stages of a workflow implementation when the services are immature. With regard to flexibility in data, this can be achieved by applying generic 'base' or 'loose' data types that may be extended at a later date to accommodate new or more tightly constrained fields (in contrast to data types that are 'narrow' or 'tightly' constrained to the operation at the outset). With regard to service endpoint location, most workflows integrate services having static locations; they refer to single external process selected by the developer at design time. This approach is appropriate for highly constrained systems. However, in more flexible workflows, the integration of additional and/ or re-located services has to be accounted for. Consequently, potential callouts and logic for selecting which partner service to integrate has to be built into the business process. Flexibility in the location of services can be achieved through WS-Addressing [7], and by dynamically binding to services from a service selection. Service endpoint location is usually specified in an external configuration file or by user interaction. The only assumption is that all the services have 'similar' interfaces, which may not always be the case. Nevertheless, it may be possible to specify a different workflow path for different services at the cost of increasing workflow complexity (the technical details will be discussed in next section).

Dynamic service selection from a service pool makes it possible to schedule their execution dynamically at runtime. For computationally extensive tasks, sub-tasks can be allocated to different services executing in parallel. Flexible workflow modeling addresses the following issues [21]: 1) Too prescriptive. Often a workflow program over constrains the solution by providing coordination or computation constraints; 2) Too vague. The work specification does not provide enough information to the workflow engine to enable them to perform the desired work by invoking participating services in required manner; 3) Exceptions. Exceptions occur because the procedure is an incomplete specification of the intended processing, because the intended procedure itself is incomplete, or because the procedure is completely inappropriate for the work [3]; 4) Changes. A workflow procedure is specified, but the model must change frequently due to changes in the nature of application/ experiments or the computing environment; 5) Workflow as fiction. The workflow specification assumes everything perfect and works 'by the book.' Assumptions according to an arbitrary operational model are always too rigid and inefficient; 6) Loss of Informal Information. When a manual procedure is encoded as a workflow procedure, the encoding will almost certainly fail to encode all the information flow associated with manual processing; 7) Loss of informal processing. The logical extension of ineffectiveness due to informal information loss is ineffectiveness due to the loss of informal work. In summary, modeling of workflows to accommodate flexibility provides both efficient and adaptive workflows since flexible workflows can adapt according to the requirements.

Workflow flexibility is certainly a requirement for e-HTPX service interaction. This is because the services often undergo upgrades/ modifications in response to the advancing nature of the research that underpins the service. A workflow must be capable of selecting and consuming preferred services from a service-pool at run-time. This is especially relevant to the data collection services, which are duplicated across multiple synchrotron facilities. Data flexibility is also required. The e-HTPX Web services adopt a comprehensive messaging model implemented in XML Schema. It is anticipated that if the e-HTPX services are widely adopted at multiple synchrotron facilities, the original 'parent' e-HTPX XSD may be further refined to suit the particular needs of that service implementation.

The adaptivity of a workflow is closely related to flexibility. Workflow adaptivity can be static or dynamic depending on whether changes and necessary adjustments can be accommodated at design time or runtime respectively (adaptivity and flexibility are closely related). The simplest means in which workflow users adapt to change is by responding to exceptions. These exceptions allow current workflow implementations to break a pre-defined process in an attempt to solve the problem. Workflow adaptivity is further accommodated-for by adding support for extensibility. For workflows, this often involves defining 'empty' activities that act as placeholders for future extensions and adaptations. These are also requirements of agile approaches to developing distributed systems. The Workflow Management Coalition (WfMC) [36], as part of its standards effort, has created Interface Four, a mechanism that governs the hand-off of work to other processes and messaging systems in an attempt to mediate how exceptions will be handled. According to the literature [36], the requirements of dynamic

adaptive workflows include; 1) Contingency Management and Hand-off, which provide mechanisms for dealing with and recovering from expected and unexpected divergence from the intended process; 2) Partial Execution, which supports creating and executing processes and process fragments (modules) as they are needed, rather than requiring the entire process be rigidly specified ahead of time; 3) Dynamic Behavior in terms of both execution model and object behaviors provides flexibility to modify workflow paths and executed behaviors at run-time independent of object data and, 4) Reflexivity, which allows a workflow component to programmatically examine, analyze, create and manipulate its own process and data.

2.2.5 Support for Long Running Processes / Transactions

Scientific workflows can be long running, often requiring days (or even months) to complete. In e-HTPX, both shipping crystals to a synchrotron facility and performing data collection experiments occurs over a period of 2~3 days. In the context of workflows, long running workflow processes / instances are managed by the workflow engine. Different workflow vendors provide workflow-engine specific support to persist workflow state during power-outs, hardware failure and software malfunction. For example, Oracle BPEL Process Manager has a notion of 'hotspots.' A hotspot is a marked stage in the workflow when the state of the workflow instance is persisted.

In order to support long running conversations and transactions between partners, both the workflow instance and partner services have to maintain conversational state in the message exchange. For workflows, this involves the addition of process identifiers that are embedded and exchanged between partners during a conversation (process identifiers are termed 'correlation properties' in the context of workflows). This concept is similar to maintaining state in a http session.

2.2.6 Management of Workflow

Workflow management involves the addition of breakpoints, addition of steering capabilities, and provision of monitoring mechanisms. Workflow management is complicated and, in practice, workflows can be managed in many different ways. Effective management depends on the underlying structure of the workflow engine and on the environment in which the workflow executes. Nevertheless, there should still be support for managing workflows in terms of specifications. Workflow management is crucial for compute-intensive and long-running workflows and services. The following points address the main workflow management concepts:

Breakpoint/ Checkpoint: A breakpoint is an arbitrary location in the workflow that divides different stages that are crucial to the overall functioning of the workflow. Persistence of state is useful at breakpoints for a number of reasons; to enhance fault tolerance, to allow re-execution with different experimental data sets/ bindings, and to facilitate inspection/ editing of intermediate data values.

Steering: Steering further generalizes breakpoints to accommodate rollback, restart, and modification to the processes at runtime. A good management mechanism should permit the re-execution of a disrupted workflow, permit workflow re-starts at arbitrary midpoints, and allow rollback to a previous internal state (invoked by computer or user). Kaiser [16] summarizes some of the main steering concepts as: reset, re-schedule, restart, undo, complete, abort, recover, ignore or jump operations in the process interpreter.

Monitoring: Workflow monitoring is crucial for optimization and analysis of internal state, especially with regard to data flow, inspection of values, re-scheduling of tasks and re-ordering of tasks for parallel execution.

3. BPEL4WS

The goal of the BPEL4WS specification is to provide a notation for specifying business process behavior based on Web services. BPEL4WS builds on top of the Web service stack and is defined entirely in XML, combining various technologies including WS-Reliable Messaging, WS-Addressing [7], WS-Coordination [8] and WS-Transactions [11] (BPEL however, is not strictly limited to only these technologies). BPEL4WS also provides a rich vocabulary for the description of business processes supporting two different types of process declaration (Abstract and Executable Processes).

3.1 Abstract Process

An 'Abstract Process' specifies the message(s) exchanged between participating services without exposing the internal details of the process. The objective of an abstract business process in BPEL is therefore to specify only the externally observable aspects of process behavior, also known as 'public process behavior' without revealing their internal implementation. Abstract business processes are not executable and must be explicitly declared as 'abstract.' An abstract process usually represents an executable process. Abstract processes only provide the complete description of external behavior that is relevant to a partner or several partners involved in the workflow. There are two good reasons to separate the public aspects of business process behavior from internal or private aspects. 1) To hide the internal complexity and decision making of the businesses process from partners, 2) Separating public from private processes provides the freedom to change private aspects of the process implementation without affecting the observable behavior. Observable behavior must clearly be described in a platform independent manner and captures behavioral aspects that may have cross-enterprise significance.

Abstract processes have restricted access to the syntax and semantics available to executable BPEL processes. Although BPEL abstract processes are not executable, they can indirectly impose behavioral regulations upon private processes executed by the BPEL orchestration server. For example, a BPEL abstract process may impose sequencing rules for invoked operations within the BPEL private process. These rules can define inter-dependencies between executable operations. The BPEL orchestration server should throw an exception when any attempt is made to run these operations out of order.

In summary, systems integration requires more than the ability to conduct interactions by using standard protocols.

The full potential of Web services as an integration platform will be achieved only when applications and business processes are able to integrate their complex interactions through standard integration models. The abstract process provides this missing process integration model, which can be tailored and customized in the executable process. Often however, the use of abstract processes is overlooked when designing workflows with BPEL.

3.2 Executable Process

An 'Executable Process' allows specification of the exact details of an abstract process workflow. Executable processes extend abstract processes and different executable processes can extend the same abstract process. The BPEL specification defines various building blocks, known as 'activities.' These activities can be used to specify an executable process's business logic to any level of complexity. The use of these activities can be further controlled through corresponding attributes, which gives the developer more flexibility. These activities can be grouped into different categories based on their usage (Basic, Structured and Special):

3.2.1 Basic Activities

invoke: Calls a Web service; which can be either one-way or request-response.
receive: Receives an incoming Web service call.
reply: Sends a response to a received Web service call.
assign: Used for copying and manipulating data using XPath. An assign activity has the sub element:
- *copy:* which itself wraps two elements 'from' the source and 'to' the sink.

throw: Used to throw an exception when the process drifts from its designated path.
rethrow: To forward a fault from inside a fault handler.
empty: A 'no operation' activity used as an extensibility feature.
wait: Pauses the flow of a process for a specified period (time-out) or until a particular time has passed (alarm).

3.2.2 Structured Activities

sequence: Executes one or more activities sequentially.
flow: Executes one or more activities in parallel.
switch: Executes one of several paths based on the value of a condition.
while: Executes a specified activity as long as a condition is true.
RepeatUntil: Repeats execution of contained activity until the condition becomes true. (Feature of BPEL 2.0)
ForEach: Executes contained activity exactly N+1 times where N equals the final value minus the start value. (Feature of BPEL 2.0)
pick: Waits for a message to arrive; it can time out, raise an alarm or throw an exception.
variables: Defines global variables used by the process or scoped variables within the scope.

3.2.3 Special Activities

validate: To validate XML data stored in variables.
scope: Defines a new scope for variables, fault handlers, and compensation handlers.
compensate: Invokes compensation on an inner scope that has previously completed.
faultHandlers: Used to catch exceptions thrown by the 'throw' activity during the execution of the process.
terminate: Immediately terminates a business process instance.
extensionActivity: A wrapper for domain specific language extension.
exit: To immediately terminate execution of a business process instance.

3.3 BPEL Process Execution

In BPEL, the main workflow is an executable process. The BPEL engine can manage the lifecycle of multiple workflow instances. The engine navigates through the workflow and invokes required partner Web services. Partner services can also invoke workflow instances. A Web service can participate by implementing a role according to a service 'partner type.' This is achieved by referencing a port-type(s) contained in a common WSDL interface that is supported by all other partners. A service that implements this port-type interface can be a partner in the process. The BPEL specification also provides constructs to handle remote and long-running transactions (LRT) and exceptions using a 'correlation-set' (also used by other related standards, e.g., WS-Coordination and WS-Transaction). The correlation-set is a token in a Web service message that identifies the specific workflow instance. A partner service must maintain the same token in an ensuing request/response message, which can be used by the workflow engine to route the message to the correct workflow instance. This enables features like exception handling and compensation in long running activities. A concrete example of a correlation-set is further discussed and illustrated in Section IV. In BPEL, the <invoke> activity performs or invokes a single operation that involves the exchange of input and output messages. The <invoke> and <receive> activity both support correlation-sets. The dependency between the workflow process and the orchestrated services is established using partner 'link types.'

Within the BPEL workflow process itself, different activities pass information among themselves using global data variables. The BPEL specification had built-in support for local transactions, exception handling and recursive decomposition.

4. Using BPEL4WS to Implement Scientific Workflows

The richness of the BPEL specification stems from the range of different activity types. Only a few of these activities have been outlined in the previous section. Those presented comprise the main building blocks of a workflow. In general, the BPEL vocabulary is tailored more to the requirements of business processes, which often have different requirements to scientific workflows. Nevertheless, different BPEL activities can still be used (sometimes in non-conventional ways) to address the requirements of scientific workflows. In this section we provide a more in-depth analysis of the BPEL4WS implementation in the context of the requirements

of scientific workflows.

4.1 Modular Design

A modular design is the first requirement of a scalable and maintainable workflow. A modular design facilitates grouping similar or inter-related activities together to define complex algorithms. These modules can be atomic in nature fulfilling the requirements of ACID transactions (Atomicity, Consistency, Isolation, and Durability). An ACID transaction either successfully executes the whole module, or if an error occurs, the compensation handlers of the already completed sub-tasks are invoked. The BPEL specification provides variety of structured activities for implementing modularity in a workflow (sequence, flow, scope, while, switch). These structured activities can have any number of sub activities to model the logic of the overall workflow.

A 'scope' activity defines its own local variables, exceptions, compensation mechanisms and event handlers. A scope activity is independent of the main process and is used to model subordinate workflows within the main process. An example of a scope activity that can be defined within a BPEL script is shown in Fig. 2. A scope activity propagates its outcome to other structured activities within the workflow by updating global variables. Global variables are declared at the beginning of the BPEL workflow script and are accessible from any structured and non-structured activity.

```
<scope>
    <variables>
      <!-- Variables definitions local to
scope -->
    </variables>
  …..
    <faultHandlers>
      <!-- Fault handlers local to scope -->
    </faultHandlers>

    <compensationHandler>
      <!-- Compensation handlers discussed
later -->
    </compensationHandler>

    <eventHandlers>
      <!-- Event handlers are discussed later
in this article -->
    </eventHandlers>

    <!-- Activities structured or non-
structured -->
  </scope>
```

Fig. 2. Layout of the BPEL4WS scope activity.

A 'sequence' activity allows the user to define a set of activities that will be executed sequentially. These activities could include invocation of partner services, exception handlers, conditional-statements or even empty activities.

The 'flow' activity is used to group different activities or tasks that are executed in parallel. There are different scenarios when the 'flow' activity is helpful, particularly when the outcome of multiple independent tasks is required for next stage and the sequential execution of these independent activities would be less efficient.

A 'while' activity is used to define an iterative activity that is performed until the specified Boolean condition evaluates to false. A 'while' activity is often useful in scientific workflows involving parameter sweeps, where iterations are executed with different sets of parameters. The Boolean conditional logic of the while activity can involve complex logic if necessary using logical operators (e.g., NOT, AND, OR).

The last structured activity is the 'switch' activity, which expresses conditional behavior. It consists of one or more conditional branches defined by a 'case' activity, followed by an optional 'otherwise' element. The 'case' or 'otherwise' activity wraps the set of inter-related tasks to make a single module.

All the structured activities can have child structured-activities, resulting in extremely flexible workflows. Inter-related tasks can be further grouped together based on their dependency, requirements, and domain or common variables. For the most part however, the best approach to achieve effective modularity is to group sub-tasks based on the variables they share, as this isolates and confines the local variables within controlled boundaries. Sub modules within the module improve the maintenance and scalability of the workflow as they can be easily replaced as required to accommodate change in the business process, or run in parallel to improve the efficiency of the main workflow.

4.2 Exception Handling

The BPEL specification provides a set of in-built exceptions for different situations; 'createInstance' exceptions are thrown when the workflow script cannot be instantiated or loaded, 'initializePartnerRole' exceptions are thrown when partner services cannot be invoked according to the role in the script, 'messageExchange' exceptions are thrown when the request or response message format is not consistent with the WSDL interface and 'queryLanguage' exceptions are thrown when the response or request cannot be queried by sub elements during an assignment activity.

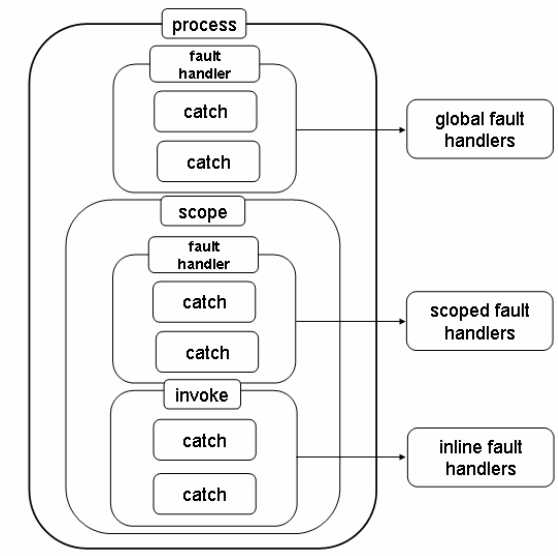

Fig. 3. Different exception handling mechanisms in the BPEL4WS.

BPEL also supports the different fault handler types that were outlined in the previous section, i.e., 'Global,' 'Local' and 'Inline' exception handlers. This provides the BPEL

workflow developer with an elegant recovery mechanism. Fig. 3 summarizes different level of fault handlers supported by the BPEL4WS specification. Fault handlers specified at different levels filter exceptions so that only generic exceptions are propagated up through the exception hierarchy. Fig. 4 shows an example of a global or process level exception handler, defined as a child of the root <process> element. Process exceptions include those caused by network and remote Web service errors. The process level fault handlers catch all un-caught exceptions thrown by all activities, including nested and non-nested tasks. The main role of the process level fault handlers is to handle the un-caught exceptions propagated from the modules. In the workflow script there can be several <catch> activities that handle specific faults and <catchAll> activities to handle all other faults. To specify which fault should be handled by a catch block, the developer must either specify a 'faultName' or a 'faultVariable.'

```
<process ...>
   ……
<faultHandlers>
      <catch faultName="ns:xxx" >
         <!-- Perform an activity -->
      </catch>
      <catch faultVariable="anyVariable" >
         <!-- Perform an activity -->
      </catch>
      ...
      <!-- catchAll is optional -->
      <catchAll>
         <!-- Perform an activity -->
      </catchAll>
   </faultHandlers>
……
<process ...>
```

Fig. 4. Process (Global) level fault handlers in the BPEL4WS script. The process level fault handlers are children of the <process> element.

```
<process ...>
  ……
<scope>
    ……
<faultHandlers>
      <catch faultName="ns:scope" >
         <!-- Perform an activity -->
      </catch>
      <catch faultVariable="anyVariable" >
         <!-- Perform an activity -->
      </catch>
      ...
    <!-- catchAll is optional -->
      <catchAll>
         <!-- Perform an activity -->
      </catchAll>
   </faultHandlers>
….
  <!-- Different activities -->
</scope>
……
<process ...>
```

Fig. 5. Scope level fault handlers in the BPEL4WS script. The scope level fault handlers are defined within the scope.

As discussed earlier, the 'scope' activity can define its own local variables, exception handlers and event handlers and nested sub-tasks. These scoped exceptions are declared within the 'scope' activity as shown in Fig. 5.

Fault handlers attached to a particular 'scope' handle faults of all the nested activities defined within the 'scope'. These exceptions are more specific to the business logic of the module in which they are defined. Faults that are not caught within the scope are re-thrown to the enclosing scope, until they are caught by an enclosing scope activity or are propagated to a global fault handler. Scopes in which faults have occurred are considered to have ended abnormally even if a fault handler has caught the fault successfully. In this case, execution control is passed to the enclosing scope, which can either take an alternative approach, such as invoking other semantically equivalent partner services, or can retry after delay.

The BPEL specification also provides inline exceptions that can be handled within the 'invoke' primitive activity. In general, an invoke primitive activity should define inline exception handlers rather than being wrapping within its own scope activity and defining scoped handlers. The syntax of an inline fault handler defined within a invoke activity is similar to that of a 'faultHandler' block (this is shown in Fig. 6).

```
<invoke ... >

      <catch faultName=" ns:xxx" >
         <!-- Perform an activity -->
      </catch>

      <catch  faultVariable=" anyVariable " >
         <!-- Perform an activity -->
      </catch>
      ...

      <catchAll>
         <!-- Perform an activity -->
      </catchAll>

   </invoke>
```

Fig. 6. Inline fault handlers in the BPEL4WS script defined within the invoke activity.

4.3 Compensation Handler

BPEL supports compensation of errors by defining compensation handlers, which are normally specific to a particular scope and are often related to 'Long-Running Transactions' (LRT). The compensating mechanism supported in the BPEL specification is similar to the fault handling approach, allowing three types of 'compensate activity' (process level, scope level and inline). The compensate activity uses the 'scope' attribute to identify which scope to compensate. Once any activity (e.g., 'invoke') has run successfully, its compensation handler awaits for a call to execute. It may be invoked explicitly, arising from a running 'compensate' activity, or implicitly due to a fault. Normally, compensation handlers are invoked from fault handlers. It is important to understand that compensation handlers are only called for successfully completed scopes and activities. If an error occurs in another scope / activity, the compensation handlers of the already completed scopes / activities are invoked to perform any necessary compensation (e.g., cleanup). It is also important to understand that in a BPEL script, a compensation handler can only be invoked by another compensation handler or from a fault occurrence that is defined in a higher scope or process. In other words, a compensation handler in a lower scope can only be invoked

by the occurrence of a higher (parent) scope fault or from a parent scope compensate activity.

Fig. 7 illustrates the compensation mechanism in the BPEL4WS process. The main process has two independent scopes, scopeA and scopeB. Both scopes have one invoke activity. The invokeA activity in the scopeA has inline compensation handler, whereas the invokeB activity in the scopeB has inline fault handler. The main process calls the activity invokeA (1) and after successful execution of invokeA, the process calls invokeB (2). The invokeB activity throws an exception and is caught by the inline fault handler (3). The inline fault handler re-throws the exception, which is caught by the fault handler in the parent activity (4) (in the main process). The global fault handler performs optional logging, notifies the user via server specific features and then calls the compensation handler of invokeA (5). The inline fault handler of invokeB cannot call the compensation handler of invokeA as both are at the same level in the workflow hierarchy.

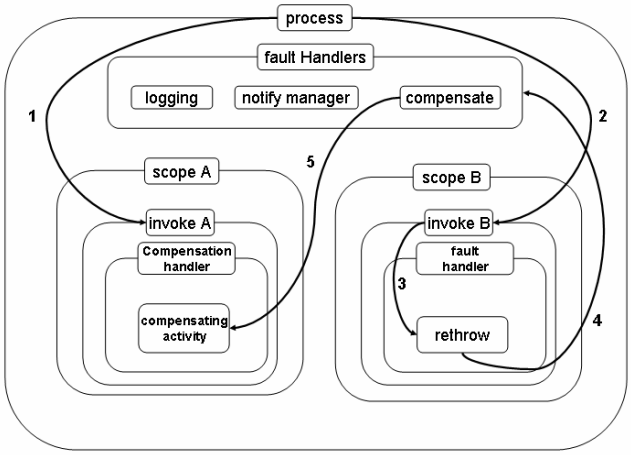

Fig. 7. Compensation mechanism in BPEL4WS. Refer to text for an explanation to the sequence of actions.

The compensation mechanism in a workflow should not be considered a remedy for deviations from the planned process unless the partner services can provide sufficient compensation. It is the responsibility of the partner service to implement these compensation strategies. It must be noted however, that often Web service designers concentrate only on the implementation of the core service with little consideration of compensation. For example, many scientific workflows are often long running, but offer no provision to release unwanted resources. The compensating features provided by BPEL are often under-utilized due to poor service design and over reliance on RPC style Web services. Any sort of compensation is only possible for non-blocking style Web services, implemented using the asynchronous Message Exchange Pattern (MEP).

4.4 Data Flexibility, Workflow Flexibility and Adaptivity

As previously discussed, scientific services are often subject to change, especially with regard to the data types and service-endpoint locations. Data flexibility is supported using ‘generic’ or ‘loose’ Web service data types. A ‘loosely typed’ Web service means the WSDL file does not contain a tightly constrained XML schema in the type system to fully define the format of the messages. Instead the WSDL defines generic types to encapsulate data within messages (i.e., encoded Strings, CDATA sections, base64 encoded sections, SOAP attachments, xsd:anyType, xsd:any). The main advantage of loose types is that different data formats can be encapsulated without having to update the WSDL interface (e.g., different document types can be transmitted as SOAP attachments or different markup fragments such as XML can be encoded in strings). This allows Web service components to be replaced in a very flexible ‘plug-and-play’ fashion. It must be noted however, that the loosely typed approach requires extra negotiation between providers and consumers in order to agree on the format of the data that is encapsulated by the generic type. Consequently, an understanding of the WSDL alone is usually not sufficient to consume the service. Fig. 8 shows an example of a loose / generic data type. The generic XML data type ‘any’ is a wrapper for arbitrary XML fragments. Consequently, this type can be mapped to different implementations, for example, either as a javax.xml.soap.SOAPElement for JAX RPC Web services or as an org.apache.axis.message.MessageElement for Apache Axis Web services. This technique produces a greater work-load for the Web service that retrieves the XML fragment from the message, but provides flexibility to replace optional values with default values rather than generating errors.

```
<complexType>
      <sequence> <any maxOccurs="1"/>
</sequence>
</complexType>
```

Fig. 8. The <xsd:any> loose data type.

Workflow flexibility provides dynamic selection of services at runtime from a pool of equivalent services, and can be implemented in different ways. A common practice is to use an exception handler to call an appropriate compensation handler through a switch statement in response to an exception resulting from an unavailable service. The compensation handler may then invoke an alternative service. However, this is a non-conventional approach, and may make the workflow hard to manage and maintain. This is illustrated in Fig. 9.

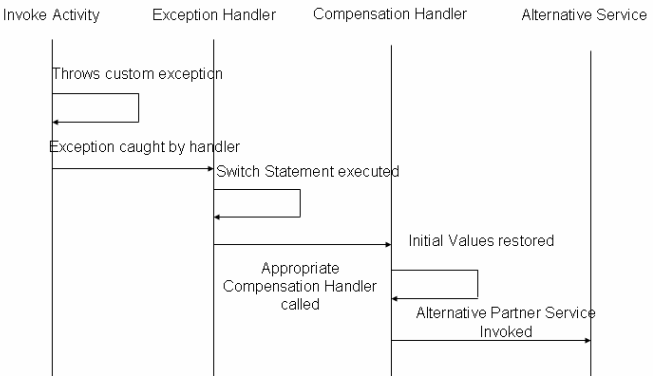

Fig. 9. A non-conventional approach to compensation handling in a flexible workflow. The logic used to call an alternate service may become overly-complex when using custom or non-conventional techniques.

A more conventional approach involves dynamically binding the partner Web services. Dynamic binding allows the workflow to invoke alternative services through configuration of inputs at run-time. This approach eliminates the need for static management of the partner services during the development phase. Dynamic binding assigns WS-Addressing [7] compliant endpoint references (EPR) to partner Web services (known as 'links') during execution of the workflow. An EPR can encapsulate an endpoint defined for any protocol (e.g., http, jms, https, smtp). All services replaceable through a dynamic binding must have the same (WSDL) interface. Fig. 10 shows an example of a BPEL4WS script that dynamically binds the partner services. The variable 'dynamicPartnerReference' can be updated at runtime based on user input, querying the online status of a Web service or from a configuration file.

Workflow adaptivity is mainly facilitated by the 'empty' activity, which means 'no operation.' An 'empty' activity can be used anywhere in the workflow as a 'place-holder' for extension, and can be replaced by any functional activity as required. For example, when a partner service provides a new compensation service, an 'empty' activity can be replaced by an actual 'invoke' activity to call the new service.

Another provision provided by the BPEL specification that facilitates workflow flexibility is the 'link' activity, which defines dependencies between different workflow activities. The 'link' activity defines the 'source' and 'target' activities. The target activity can be executed only after the source activity has successfully completed. A single target activity can have multiple source activities, in which case the target is executed only after successful completion of all source activities. Source and target activities can belong to different 'structured' activities, and belong to different parallel executing blocks. Consequently, links can be used to create very flexible dependency chains if required. A workflow engine also manages the links between activities simplifying development. Links can have associated transition conditions and predicates that are evaluated at runtime and determine whether a link is followed or not.

4.5 Long Running Transactions / Processes

Workflow processes can occur over a long period of time requiring long running conversations and transactions between partners. Long running processes require the workflow engine to mange and correctly identify the specific workflow instance so that Web service messages can be routed correctly. To implement long running conversations and transactions, process identification tokens (identifiers) have to be embedded and maintained within Web service messages (either in the SOAP header or in the SOAP body). The tokens allow the BPEL engine to route the message to the correct workflow instance. Tokens can be created and embedded either by the invoking activity or by a responding activity. If a partner service is a regular Web service (i.e., not a partner BPEL workflow), then it must at least maintain the tokens in a message payload in order to maintain the long running transaction. For the most part, when considering regular Web services, the invoking workflow is responsible for creating and embedding the tokens. In BPEL, these tokens are termed 'correlation properties.' BPEL defines two WSDL extensions for correlation properties, the first for declaring the correlation properties in the WSDL file and specifying its type (refer to Fig. 11a), and the second for declaring its location in a WSDL message (refer to Fig. 11b). Correlation properties defined in a WSDL message are termed 'property aliases.'

```
<!-- Importing namespace related to WSA -->
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003
/03/addressing"

<!-- Declaring variable of data type EPR -->
<variable name="dynamicPartnerReference"
      element="wsa:EndpointReference"/>

<!-- Updating the variable
dynamicPartnerReference -->
……
<!-- Assigning dynamicPartnerReference to
partnerLink -->
<assign>
      <copy>
            <from variable="
dynamicPartnerReference "/>
            <to partnerLink="DummyService"/>
      </copy>
</assign>
```

Fig. 10. Dynamic binding of the partnerLink Web service in the BPEL4WS script.

```
a)
<bpws:property name="correlationData"
type="xsd:int"/>

b)
<bpws:propertyAlias
messageType="ns1:Callback"
    part="jobNumber"
    propertyName="ns1:correlationData"/>

<bpws:propertyAlias messageType="ns1:
updateVariable "
    part="jobNumber"
    propertyName="ns1:correlationData"/>
```

Fig. 11. WSDL extensions defined by the BPEL for the correlation properties.

Multiple correlation properties declared in the WSDL interface can be combined to produce a 'correlationSets' element that is defined in the BPEL script. A 'correlationSets' element defines the identification token of a long running workflow instance. Fig. 12 shows an example of a 'correlationSets' element that is defined in a BPEL script. The correlation set is based on the property aliases defined in the WSDL.

```
<correlationSets>
<correlationSet name="myCorrelationSet"
    properties="na1:correlationData"/>
</correlationSets>
```

Fig. 12. CorrelationSets in the BPEL script.

A correlation set that has been defined previously in a BPEL script can be later referenced by a long running activity. During process execution, either the invoking workflow or the partner service must create an instance of the correlation set to successfully correlate subsequent messages with the correct workflow instance. In Fig. 13, the invoke activity

initiates the correlation set which is reused by the receive activity.

```
    <invoke name=" submitJob "
partnerLink="jobProcessor"
            portType="ns1:JobProcessor"
operation="anyLRP"
            inputVariable="serviceRequest ">
        <correlations>
         <correlation initiate="yes"
set="myCorrelationSet"
       pattern="out"/>
       </correlations>
    </invoke>

    <receive name="jobResult"
partnerLink="client"
portType="client:BPELProcess"
            operation="update"
variable="updateVariable"
createInstance="no">
        <correlations>
         <correlation initiate="no"
set="myCorrelationSet"/>
       </correlations>
    </receive>
```

Fig. 13. Correlation set initiated in the invoke activity and maintained in the receive activity.

If a workflow engine successfully locates an existing workflow instance identified by the correlation set, the message can be routed to the correct instance. If an instance cannot be located, a new instance may be created if the createInstance attribute of the receive activity specifies a value of 'yes' (refer to Fig. 13). If multiple instances qualify, then a runtime exception is raised, indicating an ambiguity caused by a correlation set that is non-specific.

4.6 Management of Workflow

The BPEL specification does not provide any direct mechanism to support management and monitoring of a workflow script. This next section outlines and proposes several potential strategies for workflow management and monitoring.

The <pick> activity. BPEL provides the <pick> activity, through which the workflow can be made to wait for the occurrence of selected events (manually selected if necessary). Events can be message events, handled with the <onMessage> activity, or chronologic alarm events, which are handled with the <onAlarm> activity. For each event, there is an activity or a set of activities that should be performed. For example, a <pick> activity may specify several <onMessage> elements and several <onAlarm> elements. The <onMessage> activity waits for the messages from the partner services or for the user interaction (user involvement in the workflow is vendor specific). In summary, the <pick> activity is only a limited management facility.

Wrap engine-specific classes with Web Services Invocation Framework (WSIF) [38]. Different workflow engines provide their own proprietary management capabilities and user interaction functionalities. These 'engine-specific' management capabilities are generally not portable. For example, the Oracle BPEL Process is a freely available implementation of the BPEL specification and provides its own API to interact with the workflow engine at runtime. To achieve portability, custom and engine-specific classes can be wrapped and exposed through WSIF, the Web service Invocation Framework. WSIF exposes Plain Old Java Objects (POJO), Enterprise Java Beans (EJB) and Java Connectors (JCA) as Web services (and provides accompanying WSDL interfaces). In the case of Oracle BPEL Process, these classes are;

C:\OraBPELPM_1\integration\orabpel\system\classes.

These classes can be exposed through the use of WSIF, packaged together as jar files, and placed in the common directory of a Web application container. Persistence of data during different stages of workflow is often achieved by exposing EJB's with WSIF.

Use checkpoints. In addition to wrapping engine-specific classes with WSIF, it is also possible to (partly) increase portability by extending engine-specific management facilities with standard checkpoints in the workflow.

A workflow monitoring web service. It is relatively easy to describe the state of a workflow in XML, especially since the BPEL specification is defined in XML. Consequently, it is easy to create a workflow monitoring Web service that can be used by workflow clients to monitor or track the execution of a workflow in a portable way. If a workflow state is available in XML format then it can be exposed as global variables that can be queried by the monitoring service. It is also possible to retrieve a workflow snapshot at run time in XML. This can be transformed as required using XSLT into user acceptable formats. However, it must be stated that extracting an XML snapshot of the current state of the executing workflow requires some understanding of the underlying workflow engine. Development of a custom monitoring service can be especially important when using workflows to coordinate large parameter sweeps. This is because it is very important to be able to monitor a large number of workflow instances and, at minimum, to be able to stop them when execution is not proceeding correctly. If workflow monitoring can be integrated with existing monitoring capabilities, it will help to give a more complete picture of workflow execution progress at any given moment [24].

A workflow history web service. A monitoring service can be enhanced by a workflow execution history service, which can be very helpful when attempting to determine what previously occurred to a workflow instance.

5. Limitations of BPEL4WS

The following bullets (1 to 5) discuss the limitations of the BPEL4WS specification. Different workflow engine vendors have attempted to address these limitations by augmenting their products with additional features. These features are implemented either through specification extension, or by providing custom APIs. Luckily, most of the workflow engines are either open source or free to use, allowing the workflow designer to select an appropriate product based on their requirements.

Reusability of primitive and structured activities in BPEL is limited. For example, it is not possible to re-execute an activity that is defined earlier in the script by referring to it later. In this scenario, the activity needs to be re-defined where necessary, which makes the workflow script overly complicated and unnecessarily large. The ActiveBPEL™ Designer [30] facilitates reuse of activities through the use of

'BPELets.' This custom feature allows the designer to create their own BPEL compositions (activities) that are likely to be reused later in the process or in other projects. A list of 'BPELets' can be added to the BPEL Tools palette. This allows the workflow designer to drag a BPELet onto a diagramming canvas. BPEL Designer automatically generates and adds the underlying BPEL code to the workflow. Despite ease of use for the designer, there is still repetition of code as new code fragments are added for each BPELet.

It is not possible to trigger an <onMessage> or <onAlarm> event within the workflow for the purposes of dynamically modifying the workflow at runtime. This shortcoming can be addressed to some extent by throwing an exception and invoking a compensation handler. Alternatively, a link can be used to notify an activity when another activity discovers a condition of interest.

The BPEL specification does not explicitly support user interactions, which results in different levels of support for different workflow engines. As a result, vendor-specific code dependencies are often introduced. Support for user interaction varies in functionality and syntax. For example, Fig. 14 shows the injection of user interaction code in the 'IBM WebSphere* Application Server Process Choreographer,' which has extended the 'invoke' activity [19].

```
<bpel:invoke partnerLink="null"
     portType="abc:maintenancePT"
    operation="performMaintenance"
    inputVariable="ClientInformation"
    outputVariable="statusAndCost">
   <wpc:staff>
      <wpc:potentialOwner>
         <staff:membersOfRole
role="fieldWorker"/
      </wpc:potentialOwner>
      <wpc:reader>
         <staff:membersOfRole
role="manager"/>
      </wpc:reader>
   </wpc:staff>
</bpel:invoke>
```

Fig. 14. Human interaction support in the IBM WebSphere* Application Server Process Choreographer via the null partnerLink and staff extension.

```
xmlns:task="http://services.oracle.com/bpel/t
ask"
<partnerLink name="customTaskManager"

partnerLinkType="task:TaskManager"
                partnerRole="TaskManager"
                myRole="TaskManagerRequester"
/>
```

Fig. 15. Human interaction support in the Oracle BPEL Server via the TaskManager.

Similarly the Oracle BPEL Process Manager supports user interaction through the 'Task Manager Service.' The Task Manager is a built-in service that enables the developer to include manual tasks in BPEL processes. Task Manager is an asynchronous service and provides two interfaces for interaction. Fig. 15 demonstrates the use of the Task Manager Service. A partner link called 'customTaskManager' is added which represents the Task Manager Service. The location of the Oracle BPEL Task Manager WSDL interface is: http://localhost:9700/orabpel/default/TaskManager/TaskManager?wsdl. The remaining steps are similar to invoking operations on any Web service.

Recently a new project BPEL4People has been launched with the objective of standardizing the explicit inclusion of manual tasks in BPEL processes. The most important extensions introduced in BPEL4People are 'people activities' and 'people links'. A people activity is used to define user interactions; in other words, tasks that a user has to perform. For each people activity, the BPEL server must create work items and distribute them to users eligible to execute them. People activities can have input and output variables and can specify deadlines.

Often, there are situations when a developer would want to inject Java code directly into a workflow script, either to implement simple actions or to re-use code that already models an activity. To do this, the developer has two standard options:

- Wrap local objects as local Web services and invoke the services from the workflow script. This approach can affect the performance of the overall workflow if a large number of Java objects are deployed as local Web services, especially if they are referenced frequently.
- Use Web Services Invocation Framework (WSIF) [32] to invoke local Java objects using the SOAP protocol. This approach requires learning the WSIF API and configuration with the workflow engine. Oracle BPEL Process Manager has built in support for WSIF.

The non standard option is to rely on proprietary solutions. The ActiveBPEL Engine [29] supports 'Custom Invoke Handlers' to invoke local or custom objects. Oracle BPEL Process Manager extends the BPEL specification through the use of the <bpelx:exec> activity. The <bpelx:exec> allows the user to embed Java code in the workflow script, similar to Java code in Java Server Pages (JSP). IBM and BEA also floated the concept of injecting Java code in the workflow script in the form of a new specification called 'BPEL for Java' (BPELJ) [5]. The BPELJ specification was not widely accepted by the Web service community and its support is limited to their own orchestration engines.

The BPEL4WS specification does not address Web service message level security. For example, a partner service may require authentication credentials such as signed Security Assertion Markup Language (SAML) assertions [9] to be added to the SOAP message header. Consequently, different orchestration engines have extended the original BPEL specification to allow manipulation of the SOAP header. Oracle BPEL Process Manager has two extensions to either compose or consume SOAP headers in a workflow script ('inputHeaderVariable' and 'outputHeaderVariable'). An alternative approach would be to use custom invoke handlers using APIs of choice to embed security credentials into the SOAP message.

5. Conclusion

Scientific workflows are similar to business processes in many respects, but their constantly changing and potentially long-running nature makes it difficult to anticipate all contingencies at design time. This produces a need for agile

monitoring and steering capabilities for scientific workflows. BPEL4WS is designed specifically to orchestrate Web services into workflows from the business perspective, but there is growing interest in the use of a Web services for scientific parallel and distributed computing [12]. This interest is fuelled by the availability of Web services that are tailored to Grid Computing in compliance with Web Services Resource Framework (WSRF) [39]. The emergence of BPEL4WS as the de-facto industry standard for Web service orchestration is significant for scientific domains. In house developed solutions are appealing in the short term as they may be well tailored and tuned to address specific scientific problems. However, their cross platform availability and acceptance across multiple institutes is limited. Relying on standards attracts a broader audience enabling greater contributions from all quarters of research. We conclude that BPEL4WS is currently the best candidate for orchestrating scientific and Grid services. The limitations of BPEL4WS can be overcome by supporting standard pattern-based technologies such as WSIF, J2EE and WS-* specifications. The use of WSIF especially facilitates portability and plug-in of required code into a workflow such as engine-specific support for workflow monitoring and management. Development of the WSIF based workflow plug-in is the natural and standard way to add missing components in the BPEL workflow. In most cases, careful application of BPEL4WS with a well-anticipated and investigated orchestration model can improve a scientific workflow. A desktop BPEL4WS execution engine gives a researcher control on the runtime configuration, monitoring and steering of a workflow with flexible execution, enhanced user interaction, and intelligent response to dynamic situations. A portal-based solution may be more appropriate for making pre-defined workflows available to a large number of end users.

Acknowledgements

The authors acknowledge funding for this work from the EPSRC WOSE project: Workflow Optimization for Services in e-Science and the BBSRC/ DTI e-HTPX project: An e-Science Resources for High Throughput Protein Crystallography.

References

[1] R Allan, R Keegan and D Meredith, e-HTPX – HPC, Grid and Web-Portal Technologies in High Throughput Protein Crystallography, All Hand Meeting, Nottingham, 2004, pp. 187-193.

[2] T Andrews,F Curbera, 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, (BPEL4WS), May 2003, ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[3] C Argyris and D Schön, Organizational Learning: A Theory of Action Perspective Revista Espanola De Investigaciones Sociologicas, Issue 77/78, 1973, pp. 345-350.

[4] P Barthelmess and J Wainer, “Workflow systems: A few definitions and a few suggestions”, Conference on Organizational Computing Systems, Milipitas, CA, 1995, pp. 138-147.

[5] M Blow, Y Goland, M Kloppmann, F Leymann, G Pfau, D Roller and Ml Rowley BPELJ: BPEL for Java technology, March 2004, http://www-128.ibm.com/developerworks/library/specification/ws-bpelj/

[6] G Bolcer, Flexible and Customizable Workflow Execution on the WWW, Ph.D. Thesis, University of California, Irvine. September, 1998.

[7] D Box, E Christensen, F Curbera, D Ferguson, Jeffrey Frey, M Hadley, C Kaler, D Langworthy, F Leymann, B Lovering, S Lucco, S Millet, N Mukhi, MNottingham, D Orchard, J Shewchuk, E Sindambiwe, T Storey, S Weerawarana and S Winkler, Web services Addressing (WS-Addressing), August 2004, http://www.w3.org/Submission/ws-addressing/

[8] L F Cabrera, G Copeland, M Feingold, R W Freund, H T Freund, J Johnson, S Joyce, C Kaler, J Klein, D Langworthy, M Little, A Nadalin, E Newcomer, D Orchard,I Robinson, J Shewchuk and Tony Storey, Web Services Coordination (WS-Coordination) 1.0, August 2005, ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf

[9] S Cantor, J Kemp, R Philpott and E Maler. Security Assertion Markup Language (SAML) V2.0, 15 March 2005, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security#samlv20

[10] E Christensen, F Curbera, G Meredith and S Weerawarana, Web services Description Language (WSDL) 1.1, March 2001, http://www.w3.org/TR/wsdl

[11] W Cox, F Cabrera, G Copeland, T Freund, J Klein, T Storey and S Thatte, Web Services Transaction (WS-Transaction) 1.0, January 2004, http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html

[12] G Cugola, E D Nitto, A Fuggetta and C Ghezzi, A framework for formalizing inconsistencies and deviations in human-centered systems. ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 5, No. 3, 1996, pp. 191-230.

[13] W Emmerich, B Butchart, L Chen and B Wassermann, Grid service orchestration using the business process execution language (BPEL). UCL-CS Research Note RN/05/07.

[14] S Fitzgerald and M A Hutchins, How workflow automation improves employee exit management. Intel Corporation, Internal Report, January 2005, http://www.intel.com/it/digital-office/reducing-departure-costs.pdf

[15] Y Han, A Sheth, C Bussler, A taxonomy of adaptive workflow management. Computer Supported Cooperative Work (CSCW), Seattle, November 1998.

[16] G E Kaiser, S E Dossick, W Jiang, J J Yang and S X Ye, WWW-based collaboration environments with distributed tool service. World Wide Web Journal, Baltzer Science Publishers, Vol. 1, No. 1, 1998, pp. 3-25.

[17] P J Kammer, G Bolcer, R N. Taylor and M Bergnam, Techniques for supporting dynamic and adaptive workflow. Computer Supported Cooperative Work (CSCW), Vol. 9, 2000, pp. 269-292.

[18] N Kavantzas, D Burdett, G Ritzinger, T Fletcher and Y Lafon, Web services Choreography Description Language Version 1.0, December 2004,

http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/
[19]R. Khalaf, A. Keller, and F. Leymann, Business processes for Web services: Principles and applications. IBM Systems Journal, Vol. 45, No 2, 2006.
[20]F Leymann, Web services Flow Language (WSFL) Version 1.0, http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
[21]G J Nutt, The evolution towards flexible workflow systems. Distributed System Engineering. Vol. 3, 1996, pp. 276-294.
[22]C Peltz, Web services orchestration and choreography IEEE Computer, Vol. 36, No. 10, 2003, pp. 46-52
[23]H Saastamoinen, On the Handling of Exceptions in Information Systems, PhD thesis, University of Jyväskylä, 1995.
[24]A Slomiski , On using BPEL extensibility to implement OGSI and WSRF Grid workflows. Concurrency and Computation: Practice and Experience, Vol. 18, No. 10, 2005, pp. 1229 - 1241.
[25]L A Suchman., Office procedure as practical action: models of work and system design. ACM Transactions on Information Systems, Vol. 1, No. 4, October 1983, pp. 320 -328.
[26]S Thatte, XLANG, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
[27]W J Tolone, Introspect: a Meta-Level Specification Framework for Dynamic Evolvable Collaboration Support, Ph.D. Thesis. University of Illinois at Urbana-Champaign, 1996.
[28]P Wagstrom, S Krishnan, and G v Laszewski, GSFL: An OGSA Workflow Framework, http://www-unix.globus.org/cog/papers/gsfl-paper.pdf , 2002
[29]ActiveBPEL Engine (2.0), http://www.activebpel.org/
[30]ActiveBPEL™ Designer, http://www.active-endpoints.com/products/activebpeldes/index.html
[31]e-HTPX - A Project for High Throughput Protein Crystallography, http://www.e-htpx.ac.uk.
[32]ESRF – The European Synchrotron Radiation Facility, http://www.esrf.fr/.
[33]OPPF – The Oxford Protein Production Facility, http://www.oppf.ox.ac.uk
[34]Science Factory, Integrated Biological Computing Environment, http://www.science-factory.com/products/workflow.html
[35]SRS – Synchrotron Radiation Source, Daresbury Laboratory, http://www.srs.ac.uk/srs/
[36]The Workflow Management Coalition, http://www.wfmc.org/
[37]Web services Choreography Description Language Version 1.0, http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/
[38]Web service Invocation Framework (WSIF), http://ws.apache.org/wsif/
[39]Web Services Resource Framework (WSRF), 1.2, http://www.globus.org/wsrf/
[40]YSBL – York Structural Biology Laboratory, http://www.ysbl.york.ac.uk/

Unclear reference source info: #15

Author Bios

A Akram is currently doing research in the CCLRC e-Science Centre, Daresbury Laboratory. He is actively involved in evaluation of emerging Web services standards and composition languages. Asif is a Chemical Engineer, graduated from the University of Punjab, Pakistan. He has worked in the petro-chemical industry and has taught in different universities before moving to research field.

Dr D Meredith is member of the UK Grid Operations Support Centre for the National Grid Service and Grid Technology Group at CCLRC Daresbury Laboratory. David's main research areas include distributed computing and enterprise systems, with special application to the physical sciences. David's background is as a researcher in computational modeling of petroleum systems and Lithosphere dynamics.

Dr R Allan is head of Grid Technology Group at CCLRC Daresbury Laboratory, with an interest in distributed service development and deployment and user interfaces for research, mainly in the natural sciences. Rob's background is as a researcher in theoretical atomic physics doing large scale computational modeling. He has also developed numerical algorithms and parallel computing and Grid tools for the UK Collaborative Computational Projects and HEC communities, e.g., through CCP2, CCP6, HPCI and UKHEC initiatives. He has skills in FORTRAN, Perl and C/C++. Rob has a practical knowledge of Grid and Web services middleware and helped to set up CCLRC's e-Science Centre and the UK's Grid Support Centre and has acted as PI or co-PI in numerous high-profile research council and JISC grants. He has written several technical evaluations, including the DTI-funded Globus evaluation report (December 2001) and the JISC Sakai Evaluation report (December 2004). Rob is currently acting as e-Research representative on the Joint Framework Working Group (JISC UK and DEST Australia) and is chair of the Operations Board of the NW-GRID and chair of the Service Delivery Board of National Centre for e-Social Science.