Experience with Adapting a *WS-BPEL* Runtime for eScience Workflows

Thilina Gunarathne, Chathura Herath, Eran Chinthaka, Suresh Marru
Pervasive Technology Institute
Indiana University
Bloomington, IN 47408
{tgunarat, cherath, echintha, smarru}@indiana.edu

ABSTRACT

Scientists believe in the concept of collective intelligence and are increasingly collaborating with their peers, sharing data and simulation techniques. These collaborations are made possible by building eScience infrastructures. eScience infrastructures build and assemble various scientific workflow and data management tools which provide rich end user functionality while abstracting the complexities of many underlying technologies. For instance, workflow systems provide a means to execute complex sequence of tasks with or without intensive user intervention and in ways that support flexible reordering and reconfiguration of the workflow. As the workflow technologies continue to emerge, the need for interoperability and standardization clamorous. The Web Services Business Process Execution Language (WS-BPEL) provides one such standard way of defining workflows. WS-BPEL specification encompasses broad range of workflow composition and description capabilities that can be applied to both abstract as well as concrete executable components.

Scientific workflows with their agile characteristics present significant challenges in embracing WS-BPEL for eScience purposes. In this paper we discuss the experiences in adopting a WS-BPEL runtime within an eScience infrastructure with reference to an early implementation of a custom eScience motivated BPEL like workflow engine. Specifically the paper focuses on replacing the early adopter research system with a widely used open source WS-BPEL runtime, Apache ODE, while retaining the interoperable design to switch to any WS-BPEL compliant workflow runtime in future. The paper discusses the challenges encountered in extending a business motivated workflow engine for scientific workflow executions. Further, the paper presents performance benchmarks for the developed system.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.12 [**Software Engineering**]: Interoperability; H.3.4 [**Information Storage and Retrieval**]: Distributed systems; D.2.13 [**Software Engineering**]: Reusable Software

GCE '09 Portland, Oregon USA

Keywords

eScience, workflows, scientific application abstractions

1. INTRODUCTION

In the era of data deluge and availability of vast computational power, scientific communities are gearing toward solving deeper and larger problems often crossing multiple domain disciplines. These interconnected science problems mandate scientists to share applications, combine multiple application logics in flexible but planned order, and orchestrate them together using workflows. Workflow systems loaded with features to construct, execute, intervene, manage and reuse of workflows are deepening their usage. Most of these workflow systems describe workflows in a custom format imposing a large interoperability challenge. A solution to this growing problem is to adopt a standard workflow description language. The Business Process Execution Language for Web Services (WS-BPEL)[3] is a broad specification covering both abstract and concrete executions of components and enables to define a broad array of workflow structures. WS-BPEL has become the de-facto standard for specifying web service based business processes and service compositions. Even though, scientific workflows with their agile characteristics present significant challenges in embracing WS-BPEL for eScience purposes. In this paper we discuss the experiences in adopting WS-BPEL in a large-scale workflow system for Linked Environments for Atmospheric Discovery (LEAD) project [10]. Further we describe our experiences in transiting from early research implementation based on BPEL4WS [4] to an open source WS-BPEL compliant engine, Apache Orchestration Director Engine (ODE)[18].

The Linked Environments for Atmospheric Discovery (LEAD) project is pioneering new approaches for integrating, modeling, and mining complex weather data and cyberinfrastructure systems to enable faster-than-real-time forecasts of mesoscale weather systems including those than can produce tornadoes and other severe weather. Funded by the National Science Foundation Large Information Technology Research program, LEAD is a multidisciplinary effort involving nine institutions and more than 100 scientists, students, and technical staff. To address the challenges of this large endeavour, LEAD has adopted a Service Oriented Architecture (SOA) in creating an integrated, scalable framework in which meteorological analysis tools, forecast models, and data repositories can operate as a dynamically adaptive, on-demand infrastructure. Unlike static environments, these dynamic systems can change configuration rapidly and automatically

in response to weather, react to decision-driven inputs from users, initiate other processes automatically, and steer remote observing technologies to optimize data collection for the problem at hand.

LEAD poses significant challenges to the workflow systems demanding support for responsive, long running, hierarchical; parametric sweep (forecast ensembles) workflows and for data streams. A single service invocation from a workflow instance can consume time in the order of few minutes to few hours. These long service invocation times mandates the use of asynchronous communication within the workflows. The long execution times prohibit the dependency on user computers for the coordination of the workflow for several reasons such as the user should be able to continue his normal work, the possibility of users' personal computer being unavailable and the possible inability of the personal computer to accept outside network connections. In the LEAD architecture the workflow engine is deployed as a server as oppose to running in the client computer.

In 2004, LEAD implemented a research workflow system based on a subset of BPEL4WS 1.1[4] specification, named Grid Process Orchestration Engine (GPEL)[24]. GPEL was specifically designed for eScience usage with LEAD like requirements of long running workflows and a decoupled client end enactment engine. The use of GPEL execution system in LEAD was tremendously successful leading to wide usage of the infrastructure in various educational and research efforts including National Collegiate Weather Forecasting Competition[8], Storm Prediction Center's Hazardous Weather Test bed and Vortex2 Tornado Tracking Experiments. GPEL provided many experiences to early adopters of the BPEL efforts.

However, graduating from a research system to a production facility and to reduce the maintenance load, LEAD had to consider upgrading the workflow engine from an internal implementation to an openly available industry standard version. The goals of the porting were, (i) fully support WS-BPEL 2.0 features; (ii) ensure portability & avoid locking in to a run time, by strictly adhering to open widely used standards as much as possible & by avoid using particular run time specific features; (iii) ensure sustainability by choosing a well supported run time with minimal custom changes; (iv) minimize changes to the other legacy components of LEAD, (v) improve the scalability & the performance.

Among many commercial implementations, an open source workflow engine, Apache ODE, stands out with its wide usage, community development and fully compliant with WS-BPEL 2.0. The Apache Software Foundation based engine with its active developer and user communities was a natural choice for the LEAD project to adopt. But still due to the adherence to goal (ii), eScience infrastructures using this system will have the freedom to adopt any WS-BPEL 2.0 compliant workflow runtime with minimal changes.

Further in this paper we describe the LEAD architecture and the challenges in integrating WS-BPEL and Apache ODE into an existing eScience infrastructure. This paper describes the extensions that were implemented in to the ODE enactment engine and the auxiliary scientific workflow specific WS-BPEL logic that had to be auto generated to in to the WS-BPEL documents to support the eScience requirements. Further we present a performance analysis of the scientific workflow extensions enabled ODE Engine.

Further in this paper we describe the LEAD architecture and the challenges in integrating WS-BPEL and Apache ODE into an existing eScience infrastructure. This paper describes the extensions that were implemented in to the ODE enactment engine and the auxiliary scientific workflow specific WS-BPEL logic that had to be auto generated to in to the WS-BPEL documents to support the eScience requirements. Further we present performance analysis of the workflows orchestrated by scientific workflow extensions enabled ODE Engine.

2. *LEAD* WORKFLOW ARCHITECTURE

LEAD science project was one of the early projects implementing an agile eScience infrastructure based on service oriented architecture concepts in building a science gateway. LEAD user communities have embraced the user-friendly, dynamically adaptive infrastructure for educational and research purposes. The LEAD architecture can be classified into three major layers. As shown in figure 1, the top most is the user interaction layer. The XBaya Graphical User Interface [21] serves as the user-facing interface enabling scientists to construct, execute and monitor workflow executions. The entire workflow system can be operated from a custom desktop application or can be coupled with the Open Grid Computing Environments (OGCE) based web portal interface.

Web services provide standardized interfaces that are defined, described, and discovered as xml artifacts. These services facilitate secure controlled interactions with other web services and software components, by using well known standardized message formats. The LEAD middleware layer includes a web service wrapper called GFac[16], which wraps command line scientific applications into web services. These wrapped application services can be invoked as stand alone or can be orchestrated into workflows. These transient application services catalogue all their job management and data movement activity by means of a messaging framework for provenance gathering and data catalogue services. The second layer of the infrastructure representing various middleware components also includes a workflow composition API, execution engine and monitoring capabilities.

The third layer represents various computing and data resources from local workstations to computational clouds like Amazon EC2. The GFac toolkit has built-in functionality to manage all jobs and data transfers. The toolkit accepts a request from the workflow execution system and translates the request into data movements and job submissions to local, grid or cloud computing resources.

LEAD Scientists deploy scientific applications on various compute resources and wrap them as application services, which are registered with a web service registry called XRegistry. Further the scientists use XBaya workflow GUI to browse and construct the registered services into workflows. These workflows at composition time are abstract in nature and do not have any concrete instances. At run-time, the workflow systems contacts the Dynamic Service Creator (DSC) which queries registry for any running instances and binds the abstract workflow to concrete service instances. In the case when no service instances are found, the DSC service, contacts the Generic Factory Service which on-demand creates service instances and binds them to the workflow instance. Without the late binding and abstract descriptions, all application services that are part of a workflow have to be running all the time presenting a huge monitoring, main-

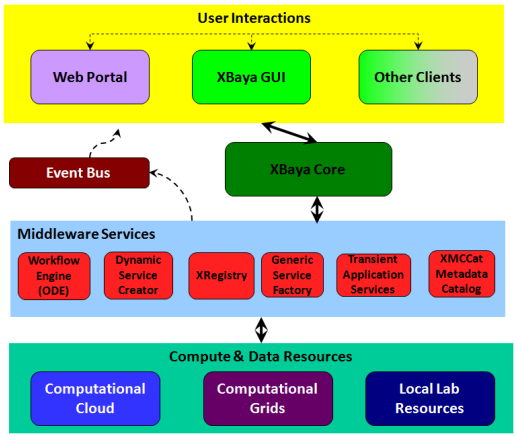

Figure 1: Lead Architecture

tenance and resource requirements. On-demand service creation and late binding enables the workflow system to support a large number of application services with minimal maintenance and optimal use of service hosting resources.

3. CHALLENGES EXTENDING *WS-BPEL* & ODE FOR E-SCIENCE

LEAD Workflows constructed with transient scientific application services were orchestrated by a BPEL like workflow engine, GPEL. To reduce the foot print, to improve the performance and to support new features , while retaining the stability and integrity of the existing LEAD infrastructure, WS-BPEL 2.0 compliant Apache ODE is swapped in as the workflow enactment engine. As discussed in section 1, the agile eScience requirements challenged WS-BPEL and ODE. To fill the void, it is not ideal to branch a community code to add custom enhancements as it puts the burden of maintenance back to the project. Also this would lock-in the project to a particular workflow engine. Effort was made to address most of the eScience requirements through auto-generation of auxiliary WS-BPEL logic, avoiding the engine lock-in, where ever possible through the XBaya workflow composer. The ODE extension based architecture, allowed us to implement the few remaining eScience requirements as pluggable modules while retaining the core engine as the community supported version. This section describes various extensions made to the Apache ODE engine and the auxiliary BPEL logic that gets generated in to the WS-BPEL workflow documents to support rest of the eScience requirements.

3.1 Propagation of workflow context information

In SOA infrastructures like LEAD, the communication is done through SOAP [13] messages. A SOAP envelope contains two major parts, the header and the body. The body element contains the application payload. An optional header element can be used to pass optional information that does not belong to the application payload. According to the SOAP specification such information may include context information required to process the message.

A common requirement in a scientific workflow infrastructure involving distributed services is the need to pass around the context information belonging to the workflow instance. Context information typically consist of unique identifiers for management & monitoring purposes, authentication information needed to access resources, and the resource identifiers needed to be used by the services. Within LEAD, this information is encapsulated as LEAD Context Header (LCH). The LCH is not consumed by the application itself, but only needed by the infrastructure hence it is encoded into the SOAP header. The LEAD context header contains following information.

- Unique identifiers, to identify the workflow tracking notifications of a given invocation. For example, experiment ID and workflow instance ID together with Service (Node) ID can relate notification messages to a specific node of a workflow instance in a particular experiment.
- End Point References (EPR) of all infrastructure middleware services (e.g. Registry, DSC, GFac) that are needed in the given workflow/service invocation.
- Workflow level configurations information, to stage output data, to perform compute resource scheduling, urgent computing parameters, etc.
- Security information, to enable authentication within the infrastructure services and to authorize with computational resources.

LEAD infrastructure mandates the propagation of LCH with all the application specific SOAP messages. This requires the workflow runtime to propagate the LCH information received in the workflow input (instance creation) message to every service invocation message sent by the workflow runtime.We implemented the LCH propagation logic using the WS-BPEL itself using a standard compliant mechanism, so that we can extend ODE to support this requirement while not changing the ODE core. This enhancement resulted in BPEL documents which are portable across different BPEL engines while achieving the needed eScience functionality.

An often-overlooked feature of WSDL [7] is its ability to bind message parts to the SOAP header blocks. Refer to section 3.7 and example 3 of WSDL 1.1 specification, for more details on defining SOAP header block elements using WSDL. All the services in the LEAD system that require the context header information, including the BPEL processes, define the "LEADContextHeader" as a message part in its' WSDL bound to the SOAP header as follows.

```
<definitions ...>
  <message name="requestMessage">
    <part name="params" element="tns:payload"/>
    <part name="leadHeader" element="lc:context"/>
  </message>
  ....
  <binding ...
    <operation name="Run">
      <input message="tns:requestMessage">
        <soap:body parts="params" use="literal"/>
        <soap:header message="tns:requestMessage"
                part="leadHeader" use="literal"/>
      </input>
    </operation>
  </binding> ...
</definitions>
```

Defining "LEADContextHeader" as a part of the input message in the WSDL of the BPEL process enables the proces instances to access that information using BPEL variables, which in turn can be copied to the subsequent outgoing service invocation messages. XBaya, the workflow composer of the LEAD project has the ability to generate the appropriate BPEL logic to copy the LEAD context header from the initial workflow input message to the outgoing messages as described above.

3.2 Asynchronous invocation

Scientific workflows typically consist of long running tasks which translate to long running service invocations in our use cases. As described in [14] it's almost impossible and resource consuming to invoke these services in a synchronous blocking manner. Further the service invocation messages get routed across on many intermediary proxy services to perform necessary tasks, before reaching the destination service. invocation message reaches the destination service.] This further necessitates the use of asynchronous non-blocking messaging for the service invocations. Unfortunately the WS-BPEL specification does not provide integrated support for WS-Addressing [12] and does not provide a standard unambiguous mechanism to specify asynchronous request/response type web service invocations.

There exist two popular mechanisms which can be used to implement asynchronous messaging for request/response type web service operations, (i) implementing as two one way messages; (ii) making the invoke operation inherently asynchronous with the use of WS-Addressing. The first method requires reply address information to be propagated to the web service using a proprietary mechanism and the services to be modified to send back the response to the specified address. Since we are adapting WS-BPEL to an existing SOA system, it's impossible to change each and every service to cater this requirement. Since almost all the LEAD service invocations are required to be performed in an asynchronous non-blocking manner, initially we implemented the second method, modifying the service invocation layer of the Apache ODE engine to perform all the invocation asynchronously. But this method does not guarantee the portability of the behaviour of the process document.

As a solution to the above issues we proposed [14] and implemented a WS-Addressing based WS-BPEL extension for asynchronous invocation of request/response type web service operations from WS-BPEL processes. We hope the community will consider it as a starting point to produce a standard WS-BPEL extension for that purpose. Currently the development LEAD system uses the inherently asynchronous non-blocking implementation, while the non-blocking BPEL extension, which is currently on testing phase, is expected to be rolled out to development and production stacks in the near future.

3.3 Notifications & monitoring

3.3.1 Assigning ID's for services

Monitoring in a traditional workflow system typically means tracking the state changes of the workflow instances from the workflow engine point of view. But in a complex system like LEAD, there are lots of apparatus working behind the covers throughout the course of a workflow instance, that are crucial to the successful completion of workflow instance. Also a single service invocation in LEAD typically results in executing a large computationally intensive job in a computational grid which involves job submission, large data transfers, job queuing, etc. Because of these reasons, much finer grained monitoring of all these steps are required to track the progress of a LEAD workflow instance.

Such fine grain monitoring necessitates unique identifiers for each service invocation. In the LEAD system XBaya workflow composer assigns unique ID's for each service node in the workflow, called node-id and workflow time-step, which are placed in the "LeadContextHeader" when invoking the respective services. Service implementations already have instrumentation in place to generate notification messages using the information in the LCH. These notifications together with unique ids such as "workflow-node-id" are used by the workflow tracking system and the XBaya to track the progress of the workflow. Similar to the LEAD header propagation, we rely on BPEL code to copy the appropriate workflow-node-id and workflow-time-step literal values to the LEAD context header of the service invocation messages.

```
<bpel:assign>
  <bpel:copy keepSrcElementName="no">
    <bpel:from>
      <bpel:literal>Forecast_Model</bpel:literal>
    </bpel:from>
    <bpel:to part="leadHeader"
                    variable="ModelInput">
      <bpel:query>
        <![CDATA[/leadcntx:workflow-node-id]]>
      </bpel:query>
    </bpel:to>
  </bpel:copy> ...
</bpel:assign>
<bpel:invoke inputVariable="ModelInput" ... />
```

3.3.2 Generation of notifications from BPEL engine

Apart from enabling the generation of notifications from the services, we also needed to emit notifications from the WS-BPEL engine itself about its state changes. The WS-BPEL specification does not recommend a standard mechanism to generate notifications as it is out of the scope of the objectives of the specification. One possible options is to use WS-BPEL to emit notifications through the addition of extra <invoke> activities, but this makes the process document to be extra long and the notification not to represent the actual state change. Also this makes adds a direct dependency to the workflow tracking sub-system (Eg: delays, failures) from the process instance, while ideally it should to be an orthogonal layer on top of the process runtime.

Almost all the WS-BPEL runtimes provide plug-in mechanisms for the notification generation. We developed a runtime-specific custom event listener based on the LEAD workflow tracking library [23], for Apache-ODE to generate the required notifications for the LEAD system. This event listener relies on the information we received on the "LEAD-ContextHeader" to publish the notifications. While this mechanism is not portable, it's always possible for us to develop notification-handlers to the other WS-BPEL run-times without much effort.

3.4 Instance Creation

In the earlier LEAD architecture (GPEL), the workflow instance creation step was separated from the actual workflow execution [24]. Executing a workflow requires explicit calling of a specific createInstance() method of the workflow engine to create a workflow instance, prior to sending the first workflow input message. This initial createInstance() call creates an unique-workflow-instance-id, an unique Endpoint reference (EPR) and a unique WSDL for the workflow instance making workflow instances first class citizens of the infrastructure. This extra step and the unique ids are used by the LEAD system to setup the resources and auxiliary services needed for the running of the workflow instance. The returned WSDL and EPR are used to send the workflow input message to the workflow instance, starting the actual execution of the instance.

In WS-BPEL, the workflow instances are created implicitly when messages are received to <receive> activities marked as "createInstance=true" and does not provide the luxury of explicit workflow instance creation. In WS-BPEL all the workflow instances share the EPR's and the WSDL's of the parent workflow. In the current LEAD system the generation of the workflow runtime independent unique workflow instance id and the preparation of resources are done prior to invoking the workflow with the input message. The generated workflow instance id is embedded in the workflow context header of the workflow input message.

LEAD workflow instance id is different from the Apache ODE internal process instance id, which is needed in case a user wants to use the ODE specific management API to manage or monitor the workflow instance. We include the ODE process instance id and the LEAD workflow instance id in the first "workflow instance created" notification generated by the workflow run time, allowing the user to create a correlation between the two ids.

3.5 Variable initializing

WS-BPEL requires complex variables to be initialized before using them. This also applies when copying values in to a variable, as the structure of the variable value needs to be present before copying in to it, except when you copy in the whole value of the variable. Initialization of variables can be done either inline or through an explicit <assign> activity. Some WS-BPEL implementations support automatic initialization of certain variable types. But this cannot be done accurately in some of the cases especially when there can be ambiguities resulting due to "optional" elements. Apache ODE does not support automatic initialization of variables. We also felt that it is safer to have the variables initialized rather than depending on an ambiguous feature of some WS-BPEL engines. XBaya workflow composer of the LEAD system, using its domain knowledge adds explicit variable initialization <assign> activities before usage of the corresponding variables inside the workflow.

3.6 Deployment

XBaya Infrastructure is designed to support multiple workflow engines. Authors of the workflows will compose workflows inside XBaya and they should be able to deploy these workflows in to different workflow runtimes. But WS-BPEL and associated standards do not define a standard way of packaging and deploying workflows in to a workflow enactment engine. Not only different engines support different workflow languages, but also each workflow engine has its own way of packaging and deploying workflows.

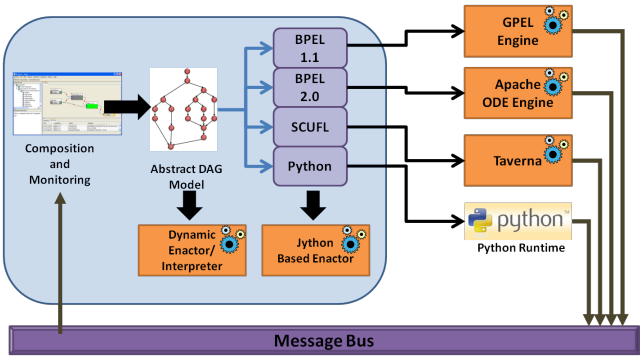

Figure 2: Xbaya and the LEAD workflow sub system

This was a major challenge to surpass to empower supporting multiple workflow enactment engines. A decoupled workflow proxy service was created to handle the complexities in non-standard deployment to workflow engines. XBaya, having authored the workflows, contains enough information to create deployment descriptors and workflow description in it. It will hand over this information to our workflow proxy service, which in turn will properly package these information and deploy to a given workflow engine.

For the time being ODE specific deployment descriptor generation and BPEL generation happens inside XBaya itself. XBaya will send this information to workflow proxy service. But in the future, XBaya will only send workflow engine independent information model to the workflow proxy service. Workflow proxy service will then generate the descriptors and workflow document, create the deployment bundle and deploy in to its workflow engine. This will make XBaya fully workflow language independent and ease the addition of new workflow engines.

3.7 XBaya Workflow Client

XBaya workflow tool is the main point of interaction for the scientist with the workflow system and it provides a high level SOA based programming model to interact with the service layer of the workflow system. This scientific workflow-programming model has been recognized as the accepted standard across different scientific disciplines as the preferred programming model for scientific computing.

The XBaya workflow system facilitates three modes of operations with respect to the different stages of workflow execution, (i) Workflow composition ; (ii) Workflow orchestration ; (iii) Workflow monitoring. Besides interacting with these different phases of workflow life cycle, XBaya also manages authentication and authorization of workflow users and it provides comprehensive security infrastructure based on Grid Security Infrastructure while facilitating user authorizations as well as user groups.

XBaya provides a high level workflow description language which is referred to as the Abstract DAG model, which is independent of conventional workflow execution languages. This allows the composition of the workflow to be completely decoupled from the execution as well as workflow to be transformed to different workflow execution languages easily. The different workflow enactment environments do have their merits and demerits, and depending on the domain science the optimal workflow enactment environment should be chosen to capitalize on the merits. As discussed in this pa-

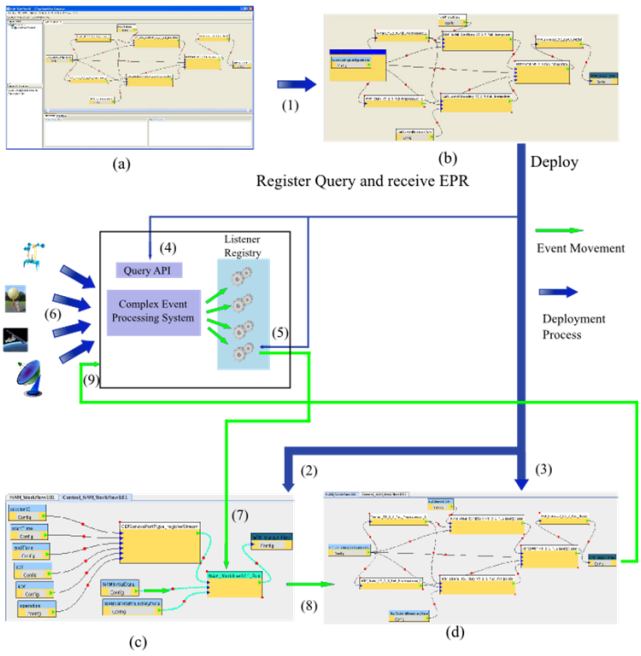

Figure 3: Stream mining system

per, the Apache ODE workflow engine is well equipped to handle long running workflows in a scalable manner where as XBaya Dynamic enactor would provide dynamic user interaction during workflow execution thus providing better steering of the workflow. Figure 2 illustrates the architecture of the XBaya workflow tool and how the interaction with the different workflow engines would take place and how the Abstract DAG model may get compiled into each execution environment as necessary.

4. NEW USE CASES

In this section we explore some of the new use cases that were made possible in the LEAD system by our adaptation of WS-BPEL compliant run time.

4.1 Stream mining

In this framework we identify the need for the scientific experiments to be able to interact with event streams because most scientific sensors like weather radars, telescopes, etc tend to produce events of periodic nature which can be abstracted to data streams. But scientific workflows are static data input processes and are not very good at manipulating stream. The integration of the Streams in to the workflow system was handled by introducing new workflow semantics to the workflow editor and by providing a stream processing system capable of manipulating streams efficiently.

The model would be to provide two workflows, a control workflow that would manage the stream and an Actual workflow the user has deployed.The control workflow consists of new workflow semantic referred to as a receive-invoke-loop, which is basically a <receive> activity waiting for an incoming event and an <invoke> activity that invokes a service with the received event and this continues in a loop. Idea is to let the data stream be channeled to the receive-invoke-loop so for each event in the data stream it would invoke a service and in this case the service would be the workflow service of the original workflow deployed by the user.

Having this in a single receive-invoke-loop does not give too much flexibility but the workflow semantics introduced in this allows the scientist to program data streams similar to a programming model used for programming workflows. Difference would be data dependency edges in the workflow graphs may now represent data stream rather than single data event. Further this would allow stream filters to be incorporated in the workflow that would allow much flexibility for scientists.

4.2 <for-each> for parametric studies

Many scientific applications involve parametric studies exploring a parameter space to identify or optimize a solution. Most popular way to perform a parametric study is by iterating through the parametric space for a certain variable, while keeping the other parameters constant. The ability to efficiently perform parametric studies can be identified as a core requirement of a scientific workflow system.

WS-BPEL <for-each> activity allows iterative execution of a WS-BPEL activity either sequentially or in parallel, making it a natural choice for implementing parametric study workflows. The iterable activity can be a structured activity, such as a <sequence> activity, acting as a place holder for other activities, giving <for-each> the ability to execute complex WS-BPEL logic in parallel. This capability of the new infrastructure is being currently explored for the parametric study use cases in the LEAD system as well as for the chemical informatics domain.

4.3 Recovery of failed workflow instances

There have been studies of the applicability of WS-BPEL fault handling to implement fault tolerance for the scientific workflows. But WS-BPEL exception handling and compensation is mainly aimed at undoing the effects of the failed workflow and to perform successful clean-up, which are important for the business workflows. As mentioned by Wassermann[28], the above mentioned backward recovery focus of BPEL fault handling complicates the forward error recovery required by the scientific workflows.

LEAD service invocations are prone to many types of failures as the best effort compute resources are susceptible to hardware, power or cooling troubles. More over, various abstractions build on the compute resources (depicted as middleware in Figure 1) hides lot of heterogeneity and complexity and also struggle battling enforcing standards. All of these factors leave the eScience infrastructure tackling the reliability issues of multiple layer underneath. End users interact with the eScience infrastructures and expect a stable platform to do science, burdening the implementation of robust fault mitigation strategies. The LEAD infrastructure employs multi-level error recovery mechanisms to handle the expected failures of service invocations like unavailability of compute resources, failures of execution and data transfers. This multi-level fault tolerance takes away the burden from the workflow run time to handle most of the failures, detailed description of handling failures at the computational resource interactions are detailed in [17]. However, eScience infrastructure failures, though at a lesser magnitude contributes to over all downtime and the workflow system should act on these failures and is discussed below in more detail. But in a nutshell, failures are handled in multiple layers making it feasible to avoid the use of WS-BPEL

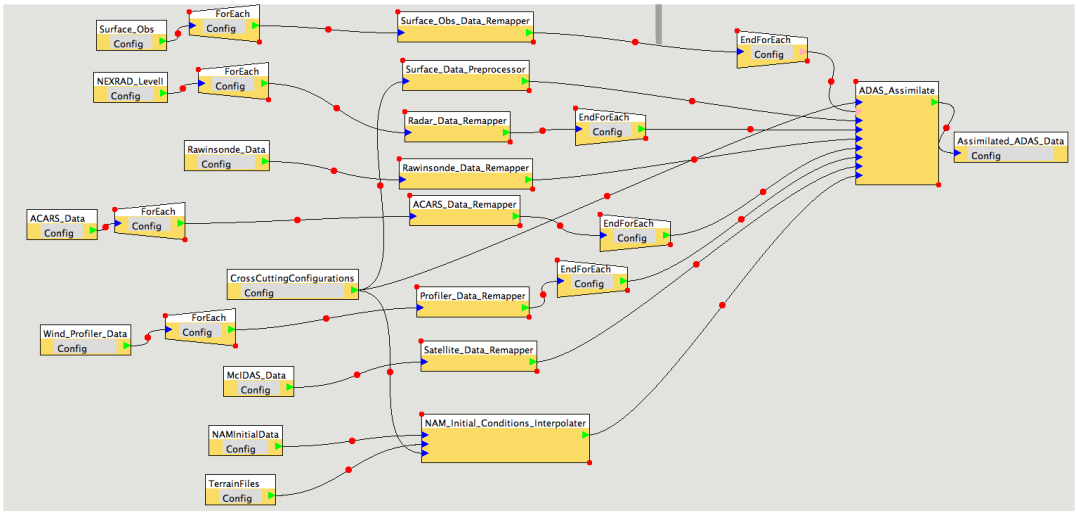

Figure 4: LEAD Data Assimilation using <for-each> activities

exception handling or WS-BPEL compensation in our system.

LEAD infrastructure uses Hasthi management framework [20] for services and systems management. Hasthi monitors the infrastructure and performs corrective actions in the event of an infrastructure failure. These corrective actions, among others, include recovering the non-functional infrastructure components and resurrecting the workflow instances that have failed due to the infrastructure failure once the system is back in a healthy state. This resurrection is done by replaying the workflow input message for the failed workflow instance, while maintaining the unique ids that identify the workflow instance inside the LEAD system. Workflow engine treats the replayed input message as yet another workflow invocation and creates a new workflow instance. This action is transparent to the rest of the LEAD system. This has been made possible by the stateless nature of LEAD workflows (with respect to the workflow engine) and by the usage of workflow engine independent ids to identify the workflows. At the same time we are pursuing another direction, where we will be using an Apache ODE management API to resurrect the workflow instance from the failed point onwards as an workflow runtime specific alternative.

5. PERFORMANCE & RELIABILITY

In this section we will be to analysing the performance and scalability of Apache ODE engine together with the scientific workflow extensions. Our testing scenarios are focused on the requirements of the scientific workflow use cases. The objective of the testing is to evaluate the workflow system with regards to the LEAD performance and scalability requirements.

We used the following two workflows, a simple service invoke and a for-each workflow for our analysis. The simple service invoke workflow (figure 5a) receives an input message, performs an asynchronous web service invocation and replies back to the client. The for-each workflow (figure 5b) receives an input message, using a <for-each>activity iterate an <sequence> activity block 'n' times and replies back to the client. The sequence activity block of the <for-each> performs an asynchronous external request-response

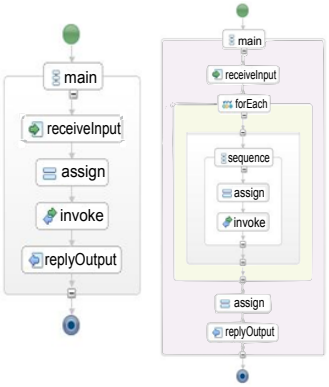

Figure 5: (a) Simple invoke workflow (b) For-each workflow

type service invocation. Apache Axis2 NIO based service running in a different LAN was used as the external service. First we benchmarked the external service from within its cluster as well as from the server which hosts the Apache ODE engine, and made sure that the service will be able to withstand the load from the workflow engine without getting saturated. The service was able to serve more than 4000 requests per second when 100 to 250 parallel clients are used to benchmark the service within the cluster using Apache Bench. The service was able to server more than 2000 requests per second when soapUI was used for benchmarking from the workflow engine host, making it evident that the service will not be a bottleneck for the performance tests.

The workflow engine and the environment were configured similar to the run time configuration of the LEAD system. MySql database based persistence storage for Apache ODE and asynchronous communication for external web service operation invocation were used during all the tests. Hence these results should not be considered as a benchmark for pure Apache ODE WS-BPEL run time, as the performance will be much greater when synchronous communication and non-persistent in memory execution will be used. The freely available SOAPUI[25] was used as the client for the workflows. We flushed the ODE database and restarted Apache Tomcat servlet container before executing each test.

Table 1: System Characteristics

Apache ODE	1.1.1
Apache Tomcat	5.5.28
Sun JDK	1.5 r20
Ubuntu Linux	9.4 (Linux kernel : 2.6.5.28-15)
MySql	5.1.31
Memory	2.5 GB
Processor	Intel Pentium 4 3.2 Ghz * 2
Network	1000 Mbps

LEAD systems most demanding requirements include weather camp and other tutorials, where large number of workflow invocations will be submitted simultaneously by different users and the data stream mining use case, where workflow invocations will be made as an when the events are received. In both the above scenarios, the workflow engine should be able sustain a good TPS rate not becoming a bottleneck and not hindering the external applications. Mean transactions per second (TPS) is calculated using the formula ((1000/Average request time) * no. of threads).

5.1 Throughput for simple service invocation

In this experiment we measured the performance with and without the workflow status notifications. Burst mode load test strategy, where a certain number of threads (40) will keep on invoking the workflow for a given number of invocations, is used for this test.

It's possible to enable, disable or limit the notifications for individual workflows using a property in the deployment descriptor. Following notifications are emitted from the workflow in this experiment setup. (i) workflow initialized - for each workflow instance (ii) workflow terminated - for each workflow instance (iii) invoking service - for each service invocation (iv) result received - for each successful service invocation. (v) fault received - for each external service invocation failure.

Table 2 and figure 6 present the results of this experiment. Looking at the results we can conclude that the workflow engine can sustain its performance in the face of many concurrent requests in both the cases. The ability to complete 1000 workflow instances, involving an actual asynchronous invocation of an external service, under a minute (under 2.5 minutes with notifications) is way above the current requirements of our system. The performance lag of the workflow with notifications can be explained based on the fact that it sends 4 real time notifications, which itself are one way web service invocations, per each workflow invocation, bringing the total to 4000 notifications in the 1000 invocation test run.

5.2 <for-each> Workflow Performance & Scalability

<for-each> is a WS-BPEL activity that iteratively executes its content for a given number of iterations. <for-each> activity supports performing these iterations sequentially or in parallel. We chose a <for-each> activity based workflow for our second experiment, as it provides a configurable (number of iterations) platform to measure the sequential as well as the parallel scalability and the performance of the workflow engine. An invocation strategy where a thread waits for a small amount of time between the subsequent invocations was used in this experiment. We

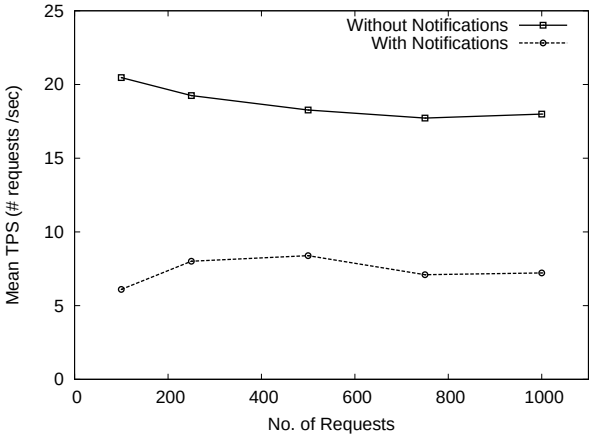

Figure 6: Simple-service-invoke workflow performance

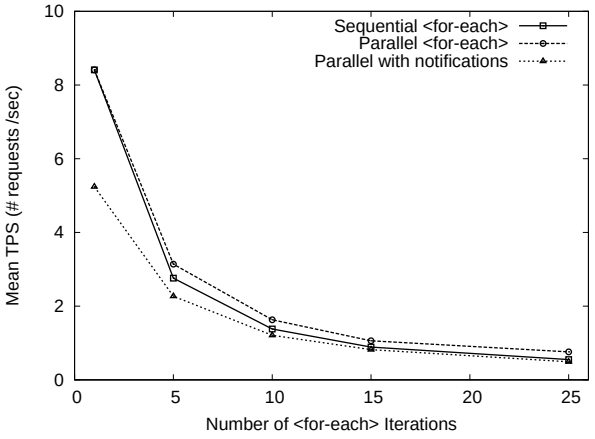

Figure 7: For-Each workflow performance and Scalability

measured 100 invocations of this workflow using 10 threads with a delay proportional to the complexity of the workflow, for varying number of <for-each> iterations. We tested the following 3 scenarios in this experiment. (i) <for-each> executing the iterations sequentially without notifications (ii) <for-each> executing the iterations in parallel without the notifications (iii) <for-each> executing the iterations in parallel with notifications.

The results are presented in table 3 and in figure 7. According to the results both sequential as well as the parallel workflows scale well. One observation that might look surprising is the performance similarity of the sequential and the parallel workflows. This should be happening due to the very fast turnaround time of the external service, which overshadows the benefits of parallelism over sequential processing. A quick test, where we used a service operation with a 10s delay for the <invoke> activity, proved this reasoning. Average turnaround time for a workflow invocation with 5 iterations in the parallel <for-each> scenario is 12990 ms (0.39 TPS) while the average turnaround time is 51517 ms (.097 TPS) for the same scenario using a sequential <for-each>, giving more than a factor of 4 speed-up in the parallel workflow. This test was done similar to the above <for-each> experiment, but with only 5 threads invoking the workflow

Table 2: Simple invoke workflow

No. of Requests	Simple Invoke		Simple Invoke with Notifications	
	Avg. Request (ms)	Mean TPS	Avg. Request (ms)	Mean TPS
100	1954	20.47	5835	6.1
250	2078	19.25	4828	8.01
500	2188	18.27	4687	8.39
750	2256	17.72	5558.5	7.1
1000	2222	17.99	5481	7.22

Table 3: For-each workflow performance and scalability

Iterations	Sequential for-each		Parallel for-each		Parallel for-each with notifications			Delay
	Avg Time per Req(ms)	Mean TPS (#/sec)	Avg Time per Req(ms)	Mean TPS (#/sec)	Avg Time per Req(ms)	Mean TPS (#/sec)	Notifications per Workflow	Used (ms)
1	1072	8.41	1072	8.41	1905	5.24	4	100
5	3064	2.76	3175	3.14	4397	2.27	12	500
10	6159	1.38	6120	1.63	8223.7	1.21	22	1000
15	9514	0.89	9367	1.06	12093	0.82	32	1500
25	15505	0.55	10424	0.76	20195	0.49	52	2500

with a delay of 15 seconds between the subsequent invocations. At the same time the parallel performance being slightly higher than the sequential even with the very fast service proves that the parallel overhead is minimal in this system. Workflow parallelism plays a very important role in the eScience workflows, where the service invocations are long running. Even making a couple of invocations in parallel makes a huge different in those scenarios.

As you can notice the overhead of the notifications system is noticeably low during the for-each experiment than in the simple-invoke experiment. This is due to the fact that the notification system [23] uses a single thread per workflow instance. When the workflow instance is much more complex and time consuming, which is the common case in eScience workflows, the overhead created by the single threaded notification system gets diminished. Still, the workflow tracking library developers are currently working on a new architecture to improve the performance.

6. RELATED WORKS

There have been efforts to evaluate the viability of use of BPEL in scientific workflow systems and [1] provides evaluation of WS-BPEL as a workflow modeling language with respect to few abstract requirements that it had identified such as fault tolerance, roll-back and recovery, user interaction and monitoring and dynamic adaptation.

Most scientific workflow systems are designed to solve problems in certain domain specific environments and because of that, the computational model presented by these workflow systems exhibit features that are specific to the particular scientific domains. Many scientific workflow systems provide workflow languages with syntax and semantics defined to address the domain issues. Scientific workflow systems like Taverna[15], Kepler[2], Triana[26], Pegasus[9] and others[28] are examples of such workflow systems. On the other hand WS-BPEL being a standard specification has standardized language syntax and semantics and thus can be developed independent of complexities associated with science domains. In almost all case workflow systems that use WS-BPEL compliant workflow execution systems tend to use third party implementations of the WS-BPEL specification in most cases using open source products. Apache ODE and Active BPEL are such WS-BPEL engines and numerous commercial vendors have published their own implementation of the WS-BPEL as well. GPEL[24] workflow engine is a rare case where a science gateway has developed their own implementation of the WS-BPEL specification with minor extensions to incorporate domain specific requirements by having minor extension to the execution semantics. The work presented in this paper can be distinguished from the related work so far because, it address the pure WS-BPEL approach to scientific workflow and how the WS-BPEL semantics can be effectively used in scientific workflow context. The use of standardized workflow execution languages also forces the workflow systems to decouple the workflow execution from the workflow composition thus allowing the workflow systems to be more interoperable.

Trident workflow system[6] is a commercial workflow system developed by Microsoft cooperation to facilitate scientific workflows and provides application of the trident workflow in oceanography science gateway[5]. Trident workflow system uses Windows Workflow Foundation [22] as its workflow engine which supports WS-BPEL specification but it is converted to internal representation of the workflow engine before execution.

Apart from the experience that was presented about the LEAD workflow system in the earlier section, the Open Middleware Infrastructure Institute (OMII) presents a workflow management system that uses the approach presented in earlier and utilizes a third party open source WS-BPEL implementation Active BPEL as its workflow engine and builds a scientific workflow management system [27]. This workflow system OMII-BPEL utilizes the infrastructure services to address the issues associated with the grid computing environment [11] while keeping its workflow execution semantics clean.

7. EXPERIENCE

Scientific workflow extended ODE, the deployment proxy service and XBaya has been successfully deployed on the development LEAD infrastructure for about one year, where there have been thousands of workflow deployments and redeployments. Also currently there are several other eScience projects evaluating the ODE-WS-BPEL based workflow suite.

One of the goals when adapting WS-BPEL to the LEAD system was to minimize the amount of LEAD specific changes to the workflow engine, Apache ODE. In order to achieve that, we architected most of the LEAD requirements using standard WS-BPEL logic, with the exception of notifications and asynchronous invocation support for request/response web services. Scientific workflow extensions for Apache ODE as well as WS-BPEL enabled XBaya workflow composer, which has the ability to generate WS-BPEL documents containing the WS-BPEL logic for scientific workflow extensions, are available to the users through the Open Grid Computing Environments (OGCE) project [19]. The LEAD-ODE project in OGCE repository has an Apache Maven2 build which will inject the scientific workflow extensions to the latest supported version of the Apache ODE workflow engine. Work is currently underway[14] to implement the asynchronous invocation component as a standard WS-BPEL extension, which would allow us to contribute it back to the Apache ODE community.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge the foundations laid by Aleksander Slominski and Satoshi Shirasuna and mentoring by Prof. Dennis Gannon and Prof. Beth Plale without which the current work would have not been possible. Furthermore the authors would like to thank Srinath Perara, and Lavanya Ramakrishnan for their direct and indirect contributions. The development of the infrastructure is funded through LEAD project supported by National Science Foundation (NSF) Cooperative Agreements ATM-0331594, ATM-0331591, ATM-0331574, ATM-0331480, ATM-0331579, ATM-0331586, ATM-0331587, and ATM-0331578. The workflow infrastructure is being packaged, tested and supported through NSF Award number 0721656, SDCI NMI Improvement: Open Grid Computing Environments Software for Science Gateways. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

9. REFERENCES

[1] A. Akram, D. Meredith, and R. Allan. Evaluation of bpel to scientific workflows. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 269–274, 2006.

[2] I. Altintas et al. Kepler: An extensible system for design and execution of scientific workflows. In *16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, 2004.

[3] A. Alves et al. Web services business process execution language, version 2.0, 2007.

[4] T. Andrews et al. Business process execution language for web services, version 1.1, May 2003.

[5] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, K. Grochow, and E. Lazowska. Trident: Scientific workflow workbench for oceanography. In *IEEE Congress on Services-Part I*, pages 465–466.

[6] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan. The Trident Scientific Workflow Workbench. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 317–318. IEEE Computer Society Washington, DC, USA, 2008.

[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. WSDL: Web services description language 1.1. http://www.w3.org/TR/wsdl, 2001.

[8] R. Clark et al. The LEAD-WxChallenge pilot project: enabling the community. *24th Conference on IIPS*, 2008.

[9] E. Deelman et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[10] K. Droegemeier et al. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. *20th Conf. on Interactive Information Processing Systems for Meteorology, Oceanography, and Hydrology*, 2004.

[11] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. Price. Grid service orchestration using the business process execution language (BPEL). *Journal of Grid Computing*, 3(3):283–304, 2005.

[12] M. Gudgin et al. Web services addressing, 1.0. http://www.w3.org/TR/ws-addr-core/, May 2006.

[13] M. Gudgin et al. Soap version 1.2. http://www.w3.org/TR/soap12/, 2007.

[14] T. Gunarathne et al. Ws-bpel asynchronous invocation of request/response type web service operations. Technical report, Indiana University School of Informatics & Computing, Bloomington, IN, USA, 10 2009.

[15] D. Hull et al. Taverna: a tool for building and running workflows of services. *Nucleic acids research*, 34(Web Server issue):W729, 2006.

[16] G. Kandaswamy, L. Fang, Y. Huang, S. Shirasuna, S. Marru, and D. Gannon. Building web services for scientific grid applications. *IBM Journal of Research and Development*, 50(2/3):249–260, 2006.

[17] G. Kandaswamy, A. Mandal, and D. Reed. Fault Tolerance and Recovery of Scientific Workflows on Computational Grids. *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 777–782, 2008.

[18] Apache ode project. http://ode.apache.org/.

[19] Open grid computing environments : Portal and gateway toolkit. http://www.ogce.org.

[20] S. Perera, S. Marru, T. Gunarathne, D. Gannon, and B. Plale. Application of management frameworks to manage workflow-based systems: A case study on a large scale e-science project. *Web Services, IEEE International Conference on*, 0:519–526, 2009.

[21] S. Satoshi. *A Dynamic Scientific Workflow System for the Web Services Architecture*. PhD thesis, Indiana University, Bloomington, 2007.

[22] K. Scribner. Microsoft® windows® workflow foundation step by step. 2007.

[23] Y. L. Simmhan, B. Plale, and D. Gannon. A framework for collecting provenance in data-centric scientific workflows. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 427–436, Washington, DC, USA, 2006.

[24] A. Slominski. Adapting BPEL to Scientific Workflows. *Workflows for e-Science*, pages 212–230, 2006.

[25] soapui : Web service testing tool , version 3.0.1. http://www.soapui.org.

[26] I. Taylor et al. The triana workflow environment: Architecture and applications. *Workflows for e-Science*, pages 320–339, 2007.

[27] B. Wassermann and W. Emmerich. Reliable scientific service compositions. *LECTURE NOTES IN COMPUTER SCIENCE*, 4652:14, 2007.

[28] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *Sigmod Record*, 34(3):44, 2005.