GridPML: A Process Modeling Language and History Capture System for Grid Service Composition

Hua Ma, Susan D. Urban, Yang Xiao, Suzanne W. Dietrich

Department of Computer Science and Engineering

Arizona State University

Tempe, AZ 85287-8809

hua.ma@asu.edu s.urban@asu.edu yang.xiao@asu.edu dietrich@asu.edu

Abstract

This paper presents a process modeling language known as the GridPML for the composition of Grid Services. The GridPML is an XML-based language that supports basic control flow constructs adopted from Web Service composition languages with features for invoking Grid Services. The GridPML defines an approach to Grid Service lifecycle management that creates service instances as needed at run time. The GridPML also supports the use of process parameters, complex type variable assignment from different Grid Services, and collection type indexing. A process execution history containing the execution context is created and maintained by the GridPML execution engine, forming the basis of future research for analyzing the semantic correctness of concurrent process execution. In addition to presenting the design and implementation of the GridPML language, this paper also outlines future directions for exception handling and recovery features for the GridPML language and execution environment.

INTRODUCTION

The current trend in the development of distributed applications is to move away from tightly-coupled monolithic applications to loosely-coupled, multi-platform, service-based architectures [16] within the context of the World Wide Web. The goal of Web Services is interoperability in loosely-coupled environments, providing a standards-based realization of the service-oriented computing paradigm. Web Services are stateless and thus cannot maintain information about previous service invocations. Furthermore, the lifetime of a Web Service is bound to the lifetime of a Web Services container. As a result, a Web Service becomes available when the server hosting the Web Service is started and remains available until the server is stopped.

The Open Grid Services Infrastructure (OGSI) specification, released in 2003 by the OGSI Working Group of the Global Grid Forum, defines a set of conventions and extensions on the use of the Web Service Description Language (WSDL) and XML Schema to enable stateful Web Services. At the core of OGSI is the Grid Service [5], a Web Service that conforms to a set of conventions for purposes such as service lifetime management, inspection, and notification of service state changes. Grid Services provide the controlled management of the distributed and often long-lived state that is commonly required in distributed applications, introducing a factory/instance approach to service instance management, in addition to the normal Web Service approach to service instantiation. Instead of having one stateless service shared by all users, a central service factory is in charge of maintaining a set of service instances. A client can therefore create and destroy its own service instances.

This research is investigating the development of a process modeling language and a corresponding execution engine for processes that execute over Grid Services. The research is being conducted in the context of the *DeltaGrid* project, an extension of the IRules project [17] for the composition of distributed

components. The IRules project has developed a middle-tier, rule processing environment for the use of declarative, active rules in the integration of distributed components. The IRules processing environment was initially implemented for component integration based on the Enterprise Java Beans [9] model.

Compared with the original IRules project, the DeltaGrid focuses on building a semantically-robust execution environment by integrating distributed event, query, rule, and transaction processing capabilities with Grid Services that are capable of capturing incremental changes (i.e., deltas) in the underlying data sources. The DeltaGrid requires the use of a process scripting language that is consistent with the service-oriented requirements of Grid Services, integrating process execution with the occurrence of events and the execution of rules in the DeltaGrid. The DeltaGrid environment is also focused on capturing the execution history of processes, events, and rules for use in the recovery of failed transactions and application exceptions using the deltas captured within individual Grid Services.

The focus of this paper is on the design of the Grid Process Modeling Language (GridPML) together with its execution engine, event generation capabilities, and process history capture system. The GridPML has been designed as an XML-based language that integrates features from several existing process modeling languages for Web and Grid Services. Although a Grid Service is essentially a Web Service, Grid Services are invoked in a different manner from Web Services. A contribution of the GridPML is that it supports basic control flow constructs adopted from Web Service composition languages with features for invoking Grid Services. In addition, the GridPML defines an approach to Grid Service lifecycle management that creates service instances as needed at run time, which is different from the static lifecycle definition proposed in other Grid Service composition languages [6, 8]. The GridPML also supports the use of process parameters, complex type variable assignment from different Grid Services, and collection type indexing.

A unique aspect of the GridPML execution engine is that it has been designed to communicate with the DeltaGrid event handler and process history capture system. Events can be generated before and after a Grid Service invocation. This feature has been designed to allow the DeltaGrid environment to integrate process execution with event and rule processing capabilities. The execution engine also creates and maintains a process execution history containing the execution context. The context includes process identifiers, variables shared by the process and invocations, and method events. The execution history provides the information that will be used in the broader scope of the DeltaGrid project to ensure the semantic correctness of concurrent process execution.

The rest of this paper is organized as follows. After outlining related work, the paper presents the GridPML language design using a process example from an ONLINE SHOPPING application. The process history capture system that has been integrated into the GridPML execution engine is then described in the context of our future directions for exception handling and process recovery in the DeltaGrid project. The paper concludes with a summary and discussion of the future research.

RELATED WORK

BPEL4WS [13] and BPML [14] are XML-based process modeling languages that provide flexible control flow for business processes over Web Services, including sequential/parallel execution, conditional branching and looping. These languages can respond to events through an event handler and provide basic exception handling through compensation and fault handler specification. BPEL4WS has several execution engines such as ActiveBPEL [1], BPWS4J [4], and Oracle BPEL Process Manager [11]. However both languages only work for Web Service composition due to the interface difference between Web Services and Grid Services.

The Grid Service Flow Language (GSFL) [8] and the Service Workflow Language (SWFL) [6] have been proposed for Grid Service composition. GSFL models peer-to-peer service interaction among Grid Services to avoid the bottleneck caused by centralized execution management. GSFL only supports sequential execution and lacks flexible control flow features found in BPEL4WS and BPML. The SWFL focuses on providing Java-oriented conditional and loop constructs to convert a job description to executable Java code.

Compared with BPEL4WS, BPML, and SWFL, the GridPML is also a service-oriented composition language defined in XML, consisting of six language constructs for workflow modeling. The GridPML supports Grid Service features such as the Grid Service Handle (GSH)/Grid Service Reference (GSR) and service lifecycle management. Unlike existing composition language, the GridPML also supports process parameters as well as variable assignment for complex types from different services. Another unique aspect of the GridPML is that it supports event and history generation features for integrating the execution engine with a distributed rule processor and with an exception handling and failure recovery mechanism.

THE GridPML LANGUAGE DESIGN

This section presents the GridPML language features. The first subsection describes the structure of a GridPML script. The second subsection provides specific examples of GridPML using an ONLINE SHOPPING example.

Language Structure

The structure of a GridPML script is shown in Figure 1 using the syntax in [13]. As indicated, there are four syntactic groups in a process definition: processParams, serviceProvider, variables, and activity.

```
<process name= "ncname" targetNamespace= "uri" >
   <processParams>?
      <param name= "ncname" type= "enumeration">+
   </processParams>
   <serviceProvider name= "ncname" type= "ncname">+
      <locator type= "enumeration" handle= "uri"/>
   </serviceProvider>
   <variables>?
      <variable name= "ncname" messageType= "qname"? type= "qname"? />+
   </variables>
   <activity>+
</process>
```

Figure 1: GridPML Structure

A process script contains zero or one processParams element, inside which input and output parameters are defined. The actual parameters are defined in the nested param element, which refers to the variables defined in the variables section of the process definition. The param name should be one of the variables defined in the variable element and the value of the type attribute should be a string enumeration type that can be used to specify that the parameter is either an input or output parameter.

All Grid Services that are part of a GridPML process are defined in the list of serviceProvider elements. A service provider is identified by a unique name. The serviceProvider definition also contains a type attribute, which can be specified as persistent or factory to indicate the Grid Service invocation mechanism. When a service provider is defined as a factory, a service instance will be created by the service factory and operations will be invoked on this instance. On the other hand, when a service provider is defined as persistent, Grid Service operations will be invoked on an existing service instance.

Service providers can be located using the locator nested element. The locator element has two attributes: type and handle. The type attribute can be specified as either persistent or factory, indicating the Grid Service invocation mechanism as described above. The handle defines the GSH of a Grid Service. If the service is defined as factory, the value of the handle should be specified to the service factory GSH. If a service is defined as persistent, an instance GSH should be assigned to the handle attribute.

Variables in the GridPML provide the means for holding messages that constitute the state of a process. The messages held are often those that have been received from a Grid Service or are being sent to a Grid Service. Variables also contain the internal state of the process. Currently, only global variable declarations are supported. A variable may be a GWSDL message type or an XML Schema simple type. The name of a variable should be unique within a process. The attribute type refers to an XML Schema simple type, while the attribute messageType refers to a GWSDL complex type. Variables associated with message types can be specified as input or output variables for invoke activities.

Since the focus of the first version of the GridPML is on development of the execution engine and interfacing the execution to the DeltaGrid event and history capture components, the initial design of the GridPML has adopted a subset of the basic activities and control flow statements from BPEL4WS as shown in Table 1. The activities supported by the GridPML are assign, invoke, flow, and sequence activities. The control flow constructs include switch and while activities.

ONLINE SHOPPING Checkout Example

Figure 2 presents the architecture of an ONLINE SHOPPING application for illustrating the features of the GridPML.

The application includes four Grid Services: a shopping service, a creditCard service, an inventory service, and a shipping service. The shopping service maintains contact and order information for each customer. The shipping service provides contact with several shipping contractors, such as DHL, FedEx, and UPS. For each customer order, the shopping service will negotiate with the shipping providers to choose an appropriate shipping service based on the customer order information. The creditCard service is the service that bills the customer credit card account for each order. The inventory service provides an interface for updating the merchandise stock when customer orders are placed.

Table 1: Activities in the GridPML

Activity	Description
assign	Changes the value of a property.
invoke	Performs or invokes an operation involving. the exchange of input and output messages.
flow	Executes activities in parallel.
sequence	Executes activities in sequential order.
switch	Executes activities from one of multiple sets, based on a Boolean value.
while	Executes activities zero or more times based on the truth value of a condition.

Figure 3 presents a UML activity diagram for an order checkout process, defining activities that occur when a customer submits an order request. As shown in Figure 3, the process first communicates with the shopping service to retrieve relevant customer information (such as the login name and selected credit card number) and a list of product information from the order (item number and quantity ordered for all items on the order). The process then invokes the creditCard service to validate the card and charge the appropriate account. If the card is not valid or the account is not in good standing, the process notifies the customer. Otherwise, the inventory is updated with the details of the order, a shipping request is sent through the shipping service, and the customer is notified of the successful completion of the order request.

The GridPML script begins with the declaration of services associated with execution of the script as shown in Figure 4. The GridPML follows language syntax similar to GSFL for service definition. The serviceProvider defines the services that are invoked by the script. As shown in Figure 4, four services are defined for the shopping, shipping, inventory, and creditCard services.

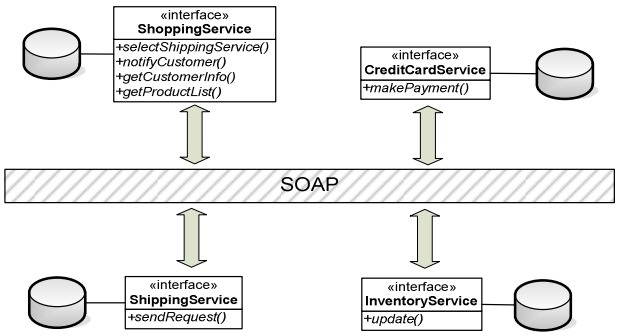

Figure 2: Online Shopping Application Architecture

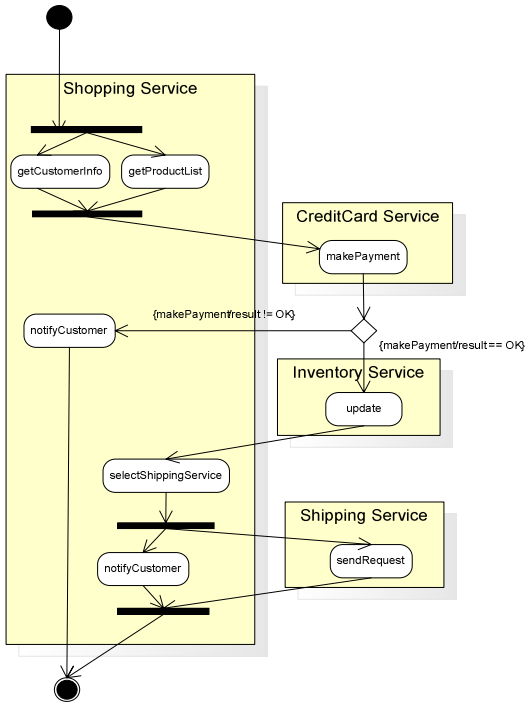

Figure 3: Activity Diagram of the Checkout Process

```
<serviceProvider name="shopping" type="shoppingServiceType">
      <locator type="factory"
        handle="http://localhost:8080/ogsa/services/irules/service/shopping/Sho
        ppingFactoryService"/>
</serviceProvider>
<serviceProvider name="inventory" type="inventoryServiceType">
      <locator type="factory" handle="... ..."/>
</serviceProvider>
<serviceProvider name="shipping" type="shippingServiceType">
      <locator type="factory" handle="... ..."/>
</serviceProvider>
<serviceProvider name="creditCard" type="creditCardServiceType">
      <locator type="factory" handle="... ..."/>
</serviceProvider>
```

Figure 4: Service Provider Definitions

The parameters associated with a script are also defined as shown in Figure 5. Complex type variables are used

for input and output parameters, demonstrating the user-defined type handling capability of the GridPML. For the process in Figure 3, the orderId and loginName variables are defined as process input parameters. The input parameters are then defined as variables, together with the additional message variables that are used to communicate with each Grid Service. The type of a variable can be specified either as an XML Schema simple type by using the type attribute or a message type that has been defined in the service GWSDL file by using the messageType attribute.

```
<processParams>
      <param name="orderId" type="input"/>
      <param name="loginName" type="input"/>
</processParams>
<variables>
      <!--input -->
      <variable name="orderId" type="xsd:string"/>
      <variable name="loginName" type="xsd:string"/>
      <!--shopping variables -->
      <variable name="getCustomerInfoInput"
        messageType="sho:getCustomerInfo"/>
      ……
      <!--creditcard variables -->
      <variable name="makePaymentInput"
        messageType="cc:makePayment"/>
      ……
      <!--inventory variables -->
      <variable name="updateInventoryInput" messageType="inv:update"/>
      <!--shipping variables -->
      <variable name="sendRequestInput" messageType="shi:sendRequest"/>
</variables>
```

Figure 5: Parameter and Variable Definitions

Before execution of the concurrent invoke activities within the scope of a flow activity, several assign activities are executed to establish the input parameters, such as loginName and orderId, as shown in Figure 6. The assign activity follows the syntax of BPEL4WS. An advantage of the GridPML, however, is that it supports the assignment of complex variables that have the same type structure, even if they are defined as different message types as a result of different Grid Service namespaces. The only requirement is that the complex type variables have the same nested type structure. XML sequential collections (i.e., an element with a maxOccurs attribute) are included in the support for assignment of complex variables. The type checking procedure performs a test between two complex types based on the concept of deep equality from object-oriented database systems [7]. Only the type values of the leaf elements of each complex type are considered during the type checking process.

After input parameters values are assigned, Figure 7 illustrates that the process spawns two concurrent threads to get customer contact information and order information from the shopping service by invoking the getCustomerInfo and getProductList methods, respectively. Before invoking an operation on a service, the execution engine will determine whether the service instance exists or not. If no instance exists, the execution engine will create a new service instance and maintain the instance in a pool so that the same instance can be shared by other invocations.

```
<assign name="assign1">
   <copy>
      <from variable="loginName"/>
      <to variable="getCustomerInfoInput" part="loginName"/>
   </copy>
   <copy>
      <from variable="orderId"/>
      <to variable="getProductListInput" part="orderId"/>
   </copy>
……
</assign>
```

Figure 6: Variable Assignment Statements

```
<flow name="flow1">
        <invoke name="getCustomerInfo" serviceName="shopping"
           portType="sho:ShoppingPortType"
           operation="getCustomerInfo"
           inputVariable="getCustomerInfoInput"
           outputVariable="getCustomerInfoOutput"/>
        <invoke name="getProductList" serviceName="shopping"
           portType="sho:ShoppingPortType"
           operation="getProductList"
           inputVariable="getProductListInput"
           outputVariable="getProductListOutput"/>
</flow>
```

Figure 7: Flow Activity Definition

An abbreviated example of the switch activity is shown in Figure 8. After getting the customer and order information, the process continues the execution of charging the customer by invoking the makePayment method on the creditCard Grid Service (not shown in Figure 8). If the invocation is successful (makePaymentOutput/result == OK), the inventory is updated and the products are shipped to the customer. Otherwise, the customer is notified about the failure to process the order. In addition to the switch activity, Figure 8 also demonstrates the use of the sequence activity (for enclosing multiple activity statements), as well as the assignment of complex types (getProductListOutput and updateInventoryList are XML sequential collections with the same structure).

The remainder of the script uses code similar to that in

Figures 6-8 to select an appropriate shipping provider and to execute concurrent tasks that notify the customer of the order status and send the shipping request to the shipping provider. Although not illustrated in this example, the GridPML also supports a while activity as well as an indexing capability for complex types with repeating values (to simulate arrays, for example).

```
<switch name="switch">
  <case condition="${makePaymentOutput/result} == OK">
    <sequence name="nested seq1">
      <assign name="assign4">
          <copy>
                <from variable="getProductListOutput" part="products"/>
                <to variable="updateInventoryInput" part="productInfo"/>
          </copy>
      </assign>
      <invoke name="updateInventory" serviceName="inventory"
          portType="inv:InventoryPortType" operation="update"
          inputVariable="updateInventoryInput"/>
      <assign name="assign5">
      ... ...
      </assign>
      <invoke name="selectShipper" serviceName="shopping"
        portType="sho:shoppingPortType" operation="selectShippingService"
        inputVariable="selectShippingServiceInput"
        outputVariable="selectShippingServiceOutput"/>
      ... ...
    </sequence>
  </case>
  <otherwise>
      <!-- notify customer unsuccessful status -->
      ... ...
  </otherwise>
</switch>
```

Figure 8: Switch Activity Definition

THE GridPML EXECUTION ENGINE

The execution engine consists of four components as shown in Figure 9: the GridPML processor, a language parser, an execution history manager, and an event handler. The GridPML parser converts an XML document to a Java representation using an XML Java binding process [2]. An XML Schema that is used to define GridPML language syntax is used for document validation.

After the parsing process, the GridPML processor initializes process parameters, service provider information, and variable information and begins executing activities defined in the input script, such as invoking Grid Services. For each GridPML activity, a wrapper has been developed that implements the semantics of the activity. The service invocation and variable manipulation are implemented by Java reflection. Type checking for XML Schema complex types is also implemented in the GridPML processor.

Events are generated by the execution engine and processed by the event handler during the execution. The execution history is generated by the history manager and maintained in a relational database using the ObJectRelationalBridge [10] from Apache to achieve an object-to-relational mapping. An execution log capturing the execution status is also created.

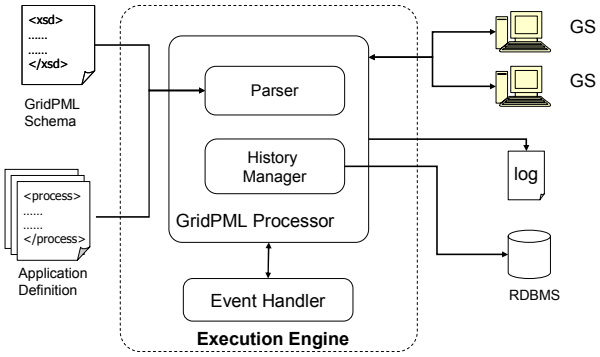

Figure 9: Architecture of the GridPML Execution Engine

PROCESS HISTORY CAPTURE SYSTEM

The purpose of the process history capture system for the GridPML is to provide a logging mechanism for concurrently-executing distributed processes. The information captured in the process history provides the basis for 1) analysis of failure recovery dependencies among concurrently-executing processes, and 2) debugging and testing of distributed process execution in a multi-process execution environment with relaxed isolation. This section presents the design of the process history capture system. The first subsection describes the process metadata and instance data captured by the history capture system. The second subsection describes our on-going work with the development of Delta-Enabled Grid Services (DEGS) and the integration of DEGS with the process history system to support a recovery system for global processes over Grid Services.

Process Metadata and Instance Data

Figure 10 presents the metadata and runtime instance information for a process in the DeltaGrid. The metadata reveals that 1) a delta-enabled Grid Service (DEGS) can provide multiple Operations packaged in different PortTypes, 2) a GridPML Process may invoke multiple Operations on different Grid Services (or invoke another GridPML process), and 3) a Process or an Operation can have an input and/or an output parameter defined through a unique Message type. Message type details are extracted

by querying the GWSDL file of a Grid Service.

At runtime, the execution engine will generate and store process execution information on a per instance basis, including process context, variable values, operations invoked, and events generated. The runtime portion of Figure 10 shows the execution information categorized in four classes: Process, Invoke, Variable, and Event. A Process instance has a pId to uniquely identify itself, a pName to relate back to Process metadata, a start time (sTime), an end time (eTime), and status of execution. A Process instance can invoke multiple operations through invoke activities in a GridPML script. Each Invoke activity has a unique identifier (inId), name, a start time (sTime), an end time (eTime), and an execution status. An invoke activity instance also has a portType and an operationName that relate back to the Operation and PortType classes in the metadata. Each Process instance can relate to multiple invocations, but each Invoke activity only belongs to one Process. Therefore, a composition relationship exists between the Process class and the Invoke class. A Process and an Invoke activity can contain two variables (one as input and one as output). A Variable instance has a unique identifier (vId), a value (stored as a Java object), and a type (input or output). The message type associated with a Variable instance can be found through the metadata associated with the corresponding Invoke instance.

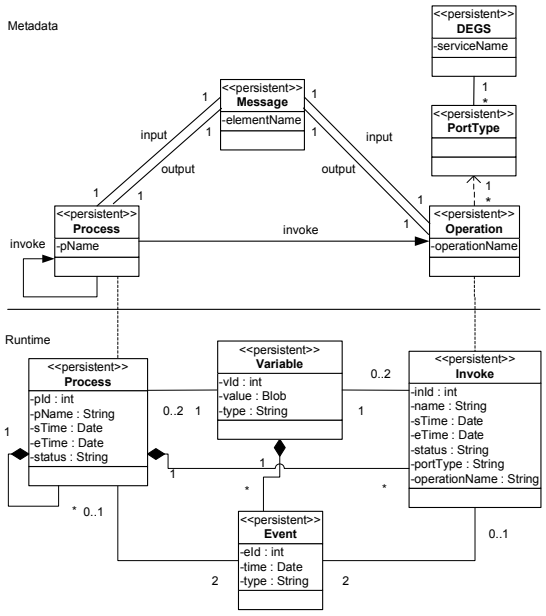

Figure 10: Process Metadata and Runtime Data

Events are generated before and after the execution of a GridPML process or invoke activity. The Event class records event history, including a unique event identifier (eId), event generation time, and event type (either before or after). Execution events can trigger rules that are used in the integration or recovery process.

Delta-Enabled Grid Services (DEGS)

A DEGS is a Grid Service that has been enhanced with an interface that provides access to incremental data changes associated with service execution in the context of global processes. Our current work is focused on 1) the design and implementation of DEGS, and 2) the integration of DEGS with the process history capture system to provide a semantically-robust execution environment for distributed processes over DEGS. The design of DEGS is based on our past research with capturing object deltas [15] and defining delta abstractions for the incremental, run-time debugging of active rule execution [18, 3]. Deltas capture incremental state changes. Delta abstractions define views over the object deltas, providing different granularity levels for examining the state changes that have occurred during the execution of active rules and update transactions. In [18, 3], we outlined the potential for using object deltas and delta abstractions as a process recovery mechanism.

We are currently designing DEGS to capture deltas using capabilities provided by most commercial database systems (such as triggers and/or the Oracle Streams capability for capturing database updates as events [12]). The process history capture system will extend the delta abstraction concept from [3] to organize deltas from DEGSs to form a complete process execution history, including not only the execution status of each activity, but also the incremental data changes recorded at the local sites. The process history will show the details of how concurrently-executing processes modify shared data in a multi-user, multi-process environment, and can be used to analyze how data modification from one process might affect other processes. Thus in a concurrent process execution environment with relaxed isolation, the deltas in the process history can provide a means for the analysis of failure recovery dependencies among concurrently executing processes and provide assistance in the recovery of dependent processes [19].

We are also extending the GridPML to support the specification of compensation, contingency, atomic

transactions, and user-defined application exceptions. The process history capture system will be integrated with a rule processing system to handle process-level exceptions. The process designer will be able to 1) query the process execution history to decide how to recover a process based on the current execution status, and 2) analyze the process history and event log together with user-specified application information to derive the failure recovery dependencies among concurrently executing processes.

SUMMARY AND CURRENT STATUS

This paper has presented the design of the GridPML, an XML-based process modeling language for distributed processes that execute over Grid Services. We have implemented an initial version of the GridPML execution engine and history capture system. Our current work is focused on the design and implementation of DEGS, extensions to the GridPML language and execution engine for the specification of recovery and exception handling directives, and integrating the use of DEGS with the process history capture system. We are also developing algorithms for analyzing the integrated DEGS/process history capture system to identify process dependencies in the recovery process and to use the deltas associated with DEGS to support rollback capabilities for distributed processes.

References

[1] ActiveBPEL: The Open Source BPEL Engine. http://www.activebpel.org/index.html

[2] Apache XMLBeans http://xmlbeans.apache.org/,2005.

[3] Ben Abdellatif, T., *An Architecture for Active Database Systems Supporting Static and Dynamic Analysis of Active Rules Through Evolving Database States*, Ph.D. Diss., Arizona State Univ., Dept. of Comp. Sci. and Eng., Fall 1999.

[4] BPWS4J: The IBM Business Process Execution Language for Web Services Java Run, 2003.

[5] Foster, I., The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. Journal of Supercomputer Applications*, 2001.

[6] Huang,Y and Walker, D. , Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid, in *Proc.of the Int. Conf. on Computational Science,* 2003.

[7] Khoshafian S. N., Copeland G. P., Object identity, *Proc. on Object-oriented Programming Systems, Languages and Applications*, p.406-416, 1986.

[8] Krishnan, S., Wagstrom, P., Laszewski, G., GSFL: A Workflow Framework for Grid Services, Argonne National Lab., Preprint ANL/MCS-P980-0802, 2002.

[9] Monson-Haefel, R., Burke, B., and Labourey, S., Enterprise JavaBeans: O'Reilly, 2004.

[10] Object Relational Bridge-OJB 2005, http://db. apache.org /ojb/.

[11] Oracle BPEL Process Manager. http://www. oracle.com/technology/products/ias/bpel/index.html.

[12] Oracle Streams Concepts and Administration 10g Release 1 (10.1) Oracle Database Documentation Library http://www.oracle.com/pls/db10g/portal.portal_demo3?selected=4

[13] Business Process Execution Language for Web Services Version 1.1, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel /, 2003.

[14] Business Process Modeling Language, http://www. bpmi.org/specifications.esp, 2002.

[15] Sundermeir, A., Ben Abdellatif, T., Dietrich, S. W., and Urban, S. D., Object Deltas in an Active Database Development Environment, *in Proc. Of the Deductive, Object-Oriented Database Workshop*, pp.211 -229, 1997.

[16] Tan, Y., Business Service Grid: Manage Web Services and Grid Services with Service Domain technology, http://www-128.ibm.com/developerworks/ibm/library/gr-servicegrid/ 2003.

[17] Urban, S.D., Dietrich, S.W., Na,Y., Jin,Y., Sundermier, A., and Saxena, A., The IRules Project: Using Active Rules for the Integration of Distributed Software Components. *Proc. of the 9th IFIP 2.6 Working Conf. on Database Semantics: Semantic Issues in E-Commerce System*, 2001.

[18] Urban, S. D., Ben Abdellatif T., Dietrich S. W., and Sundermier, A.., Delta Abstractions: A Technique for Managing Database States in Active Rule Processing, *IEEE Trans. on Knowledge and Data Eng.*, pp. 597-612, 2003.

[19] Xiao, Y., *A Process History Capture System and Failure Recovery Framework for Integration of Delta-Enabled Grid Services*, Ph.D. Proposal, Arizona State Univ., Dept. of Comp. Sci. and Eng., Fall 2004.