GSFL: A Workflow Framework for Grid Services

Sriram Krishnan,[1,2] Patrick Wagstrom,[1,3] Gregor von Laszewski[1]
[1]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
[2]Indiana University, Bloomington, IN 47405
[3]Illinois Institute of Technology, Chicago, IL 60616

Abstract

The *Open Grid Services Architecture* (OGSA) is addressing the challenge of integrating services spread across distributed, heterogeneous, dynamic virtual organizations, using the concepts and technologies from both the Grid and Web service communities. The Web service community has realized that Web services can reach their full potential only if there exists a mechanism to describe the various interactions between the services and dynamically compose new services out of existing ones. This situation is true in the case of Grid services as well. In this paper, we analyze existing technologies that address workflow for Web services, and try to leverage them for Grid services, which have different needs from standard Web services. We discuss these special needs and present the *Grid Services Flow Language* (GSFL), which addresses them for Grid services within the OGSA framework.

KEY WORDS: Grid; OGSA; OGSI; Grid services; Web services; Workflow

1 Introduction

The Web services approach is rapidly gaining momentum in industry. The World Wide Web Consortium (W3C) [30] defines a Web service as a software application identified by a URI [6], whose interfaces and bindings are capable of being defined, described, and discovered by XML artifacts and which can support direct interactions with other software applications using XML-based messages via Internet-based protocols. A more general and descriptive definition can be found in [17], where a Web service is defined as a *platform and implementation independent* software component that can be

- *described* using a service description language,
- *published* to a registry of services,
- *discovered* through standard mechanisms,
- *invoked* through a declared API, usually over a network, and
- *composed* with other services.

The goal of Web services is interoperability. It follows from the definitions that a requestor can access Web services by using *standard* mechanisms. In the ideal world, any requestor can interface with any application that claims to be a Web service irrespective of the language and the environment that either of them uses. This feature makes the Web services approach appealing to modern enterprise and interorganizational computing systems.

Grid computing involves the agglomeration of diverse resources in dynamic, distributed virtual organizations [14]. *Grid technologies* are infrastructures that support the sharing and coordinated use of such diverse resources. Grid technologies [12] are currently seeing widespread adoption in the scientific computing community. Apart from the problems that are inherent in dealing with resources (i.e., algorithms and problem-solving techniques, resource management, security, instrumentation and performance analysis, network infrastructure, etc.), Grid technologies have to solve problems similar to those addressed by Web services, namely description, discovery, communication, remote invocation, and so forth. Recently, this fact has been recognized by the Grid community, resulting in the development of the Open Grid Services Architecture (OGSA) [13].

OGSA is the result of lessons learned while developing the Globus Toolkit^{TM}[16], which has become the de facto standard for Grid computing. OGSA uses the Web Services Description Language (WSDL) [9] to achieve self-describing and discoverable services. It defines a set of standard interfaces that a *Grid Service* may export and that enable features such as discovery, service lookup, lifetime management, notification, and credential management. In a similar vein to Web services technology, Grid services can realize their full potential only if there is a mechanism to dynamically compose new services out of existing ones. To create such a mechanism, we must not only describe the order in which these services and their methods execute but also present a way in which such an agglomerate can export itself as a service. In this context, we define the term *workflow* as a set of rules that define the interactions between a set of services in order to be composed into a *metaservice*.

In this paper we show the current state of workflow languages for Web services and how they relate to Grid services. We point out some shortcomings of such languages in the context of the Grid. As a solution, we propose the Grid Services Flow Language (GSFL), which we describe in detail. We present the research problems posed and the approach we take towards them.

2 Technology Survey

The field of workflow languages for Web services has seen considerable activity in the recent past. Various methods have been proposed as standards by the major Web services software providers. A thorough survey of all the existing technologies is impossible, given the high rate of change in this field; instead, we focus on some of the larger projects that are getting attention at this time.

2.1 Web Services Flow Language (WSFL)

Web Services Flow Language [19], which is under development by IBM, is one of the approaches to Web services workflow. WSFL describes the composition of Web services by using a *flow model* and a *global model*. The flow model defines a series of *activities* that represent the operations of the composite Web service, and specifies the order in which these activities execute. It defines the flow of control and data between the various activities, using *controlLinks* and *dataLinks*, respectively. In most cases, the data flow closely follows the control flow; however, WSFL is flexible enough to accommodate cases where this may not be true.

The global model defines how the activities of the composite Web service are mapped into the operations of the individual Web services, using what are called *plugLinks*. These *plugLinks* can be used to connect WSDL operations with similar, but *dual* signatures. For example, a notification operation of a Web service can be connected to a one-way operation of another Web service, while a solicit-response operation of a service can be connected to a request-response operation of another service. While the flow model describes the orchestration of the various activities in the workflow, the global model describes how these are implemented by the participating Web services.

WSFL identifies the services participating in the workflow by using a *locator* element, which supports the following bindings:

- *static*, where a reference to a WSDL or WSFL definition is provided
- *local*, where the service implementation is local
- *uddi*, where the service implementation is looked up using the UDDI [28] API
- *mobility*, where the service provider is referenced in a message generated by some activity
- *any*, where the service provider is not restricted by the flow model

WSFL also supports lifecycle operations for the flow model of the composite Web services. It supports operations such as *spawn*, *call* (a blocking spawn), *suspend*, *resume*, *enquire*, and *terminate*. The advantages of WSFL are its logical consistency with WSDL and the ability to define Web services that are recursively composed of other Web services.

Version 1.0 of WSFL was released by IBM in May 2001; however, not much has been written about WSFL since then. No popular implementation of WSFL is available, although a few groups which are working on one [10].

2.2 XLANG: Web Services for Business Process Design

XLANG [25], a language under development by Microsoft Corporation, is used to model business processes as autonomous agents. In WSDL the unit of action is an operation, that can be either on a stateless service (such as a stock quote) or on a stateful (subservient) service where the interaction defines the beginning and end of the process. There is a third model, however, in which business processes may be *autonomous agents*, such as a supply chain. In this chain, input and output messages occur in a defined order, deemed a *service process*. It is based on π-calculus theory for the connection and synchronizations of automatons.

XLANG defines the following set of operations as extensions to the standard WSDL operations to assist in this modeling:

- *Delays* allow a thread to stall for a specified time period or until another condition is met.
- *Raise* is a method to raise exceptions for certain actions.
- *Process control* combines actions together with conditional and iterative statements.
- *Correlation* provides a method for declaring longer running conversations.
- *Transaction support* allows definition of rollback procedures if one action in the execution fails.
- *Contracts* create agglomerate services by facilitating one-way bindings between ports.

Like many of these technologies XLANG is still evolving. Currently it lacks methods to add services dynamically to a business process and also does not have a mechanism to export the methods of these services as part of the workflow. These will be addressed in future revisions of the language. However, little has been written about XLANG since it's initial release in May 2001.

2.3 Web Services Conversation Language (WSCL)

Web Services Conversation Language [26] is a conversation language framework under development by the Hewlett-Packard Company, for modeling the sequencing of the interactions or operations of *one interface*. It fills the gap between mere interface definition languages that do not specify any choreography and more complex process or flow languages that describe complex global multi-party conversations and processes. The major elements of the WSCL specification are as follows:

- *document type descriptions*, which reference the types of documents that the service can accept or transmit, defined using XML schemas [11];
- *interactions*, which model the actions of a conversation between two participants;
- *transitions*, which specify the ordering relationships between the interactions; and
- *conversations*, which list all the interactions and transitions that make up a conversation.

The conversations are the public interfaces supported by a service. They add semantics to the WSDL for the service by also specifying the possible ordering of the operations. WSCL does not, however, address the recursive composition of Web services, which is what we aim to do for Grid services.

2.4 Other Related Work

The Web Services Choreography Interface (WSCI) [4], a language under development by Sun, Intalio, SAP, and BEA. is aimed at application-to-application integration on a tighter level than that proposed by XLANG; however, WSCI was proposed in June 2002, and is still fairly new at the time of writing this paper. The Business Process Modeling Language (BPML) [3] is a metalanguage for modeling business processes. BPML provides an abstracted execution model for collaborative and transactional business processes based on the concept of a transactional finite-state machine. DAGMan [2] is a metascheduler for Condor [1] that manages dependencies between jobs. Despite the fact that DAGMan does not deal with the workflow for Web services, the concept of using a directed acyclic graph to represent a set of programs where the inputs, outputs, and the execution are interdependent can be applied to describe the dependencies between the Web services. The XCAT Application Factories [15] address workflow issues for Grid-based components within the Common Component Architecture (CCA) [5] framework. XCAT allows components to be connected to each other dynamically, making it possible to build applications in ways not possible with the standard Web services model.

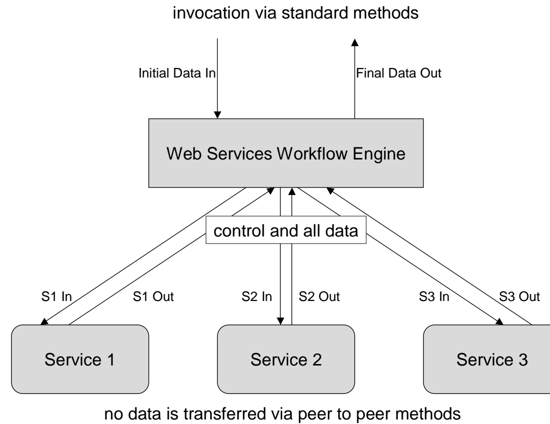

Figure 1: Web Services Workflow Model

3 Grid Workflow Requirements

As a result of the above survey and our analysis of Grid use cases, we have established a set of requirements for a workflow specification for the Grid. In this section, we describe these requirements and discuss how existing Web service technologies do not address all of them, despite providing invaluable techniques that we reuse.

Just as the Web service technologies aim to do, the Grid workflow specification should allow specific activities implemented by individual services to be exported as activities of the workflow. It should also allow the exported activities to trigger a chain of other activities. Current technologies such as WSFL address this issue effectively. Hence, we try to incorporate these features presented by WSFL into the Grid Services Flow Language. Furthermore, the activities exported in such a manner should also be described in the same manner as the service itself. In this sense, the specification should be rich enough to describe the workflow such that the WSDL for the workflow entity (henceforth referred to as a *workflow coordinator*) can be auto-generated from the specification. The workflow coordinator must be able to handle the methods that have been dynamically exported as a composition of the various activities of the workflow, in such a way that clients can access them using the same standard tools that they use to deal with the individual services. This is an important requirement for recursive composition of services.

Despite the fact that it is theoretically possible to define peer-to-peer interactions between Web services that are part of the workflow in languages such as WSFL (via *plugLinks*), it is not practically possible as solicit-response and notification operations are not fully defined in WSDL 1.1. There are multiple interpretations of these operations in the Web service community, and there is considerable debate about their removal in the forthcoming WSDL 1.2 [8] specification. As a result, and as has been pointed out by [15], existing Web services define their workflow in such a way that the workflow engine has to intermediate at each step of the application sequence, as shown in Figure 1. However, the workflow engine does not end up being a bottleneck in business-to-business communication as there may only be a moderate level of data transmission between the services. For Grid-based services, however, exchanging large amounts of data is the norm. Having a central workflow engine relay the data between the services would be a bad idea in this case. The workflow specification needs to be able to allow communication between the services, as depicted in Figure 2.

As we mentioned, OGSA adds extensions to WSDL in order to address Grid-specific needs. It addresses communication between Grid services by using *notificationSources* and *notificationSinks*, which allow services to carry out

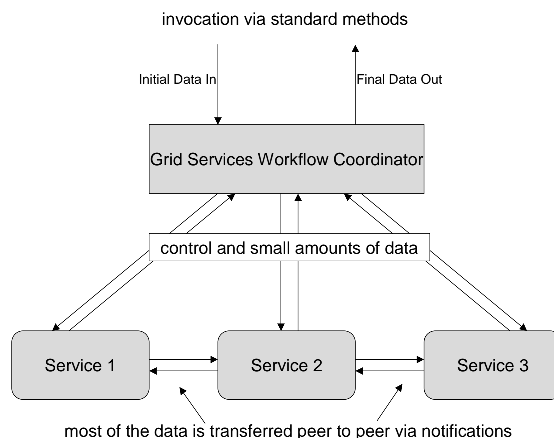

Figure 2: Grid Services Workflow Model

asynchronous delivery of messages between each other. GSFL must provide a mechanism to connect notificationSources and notificationSinks, thus obviating the need for the workflow engine to mediate at every step. Additionally, OGSA uses *registries* and *factories* for locating and creating Grid services, respectively. These must be appropriately handled by GSFL.

It is conceivable that certain Grid services in the workflow will not be executing while others are. One reason may be the fact that the services that need to execute earlier run for weeks; another reason may be or that the service that executes later needs data from the former to bootstrap itself. The Grid workflow specification should be able to handle these particular needs. Additionally, it should also be able to handle instantiation of the individual Grid services on a per method or a per workflow instance basis. If the Grid services are instantiated on a per workflow instance basis, certain activities exported by the workflow may not function because a certain Grid service may have run to completion, or may not have been instantiated yet. In such a case, certain ordering has to be imposed on the exported activities, such as the one proposed by WSCL.

In the following section, we describe the Grid Services Flow Language and how we address the requirements specified.

4 GSFL Overview

The Grid Services Flow Language is an XML-based language that allows the specification of workflow descriptions for Grid services in the OGSA framework. It has been defined by using XML schemas. A simplified architecture is shown in Figure 3. It has the following important features, which we expand in the following subsections.

- *Service Providers*, which are the list of services taking part in the workflow;
- *Activity Model*, which describes the list of important activities in the workflow;
- *Composition Model*, which describes the interactions between the individual services; and
- *Lifecycle Model*, which describes the lifecycle for the various activities and the services that are part of the workflow.

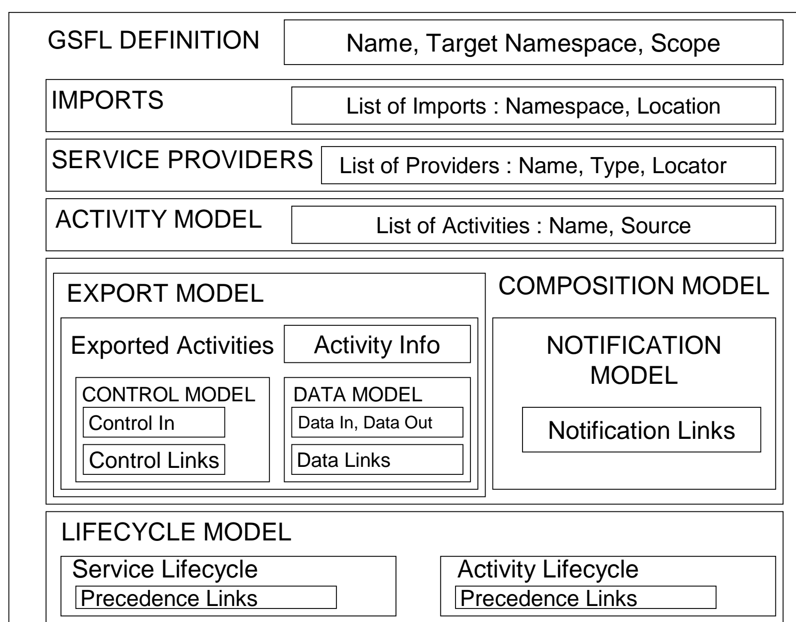

Figure 3: Architecture of the GSFL

4.1 Service Providers

All the services that are part of the workflow have to be specified in the list of *serviceProviders*. The service providers are identified throughout the GSFL document by a unique *name*, which is specified as part of the definition. The definition also contains the *type* of the service provider, which is the type of the Grid service, as specified by its WSDL specification. Service providers can be located by using the *locator* element, which allows looking up service providers in a number of ways. Services can be located *statically*, via a static URL that points to an already running service. They can also be started up by using *factories*, the handle to which is available in the GSFL document. Services can also be looked up using *registries*.

4.2 Activity Model

The *activityModel* lists all the operations belonging to the individual service providers, that play a role in the workflow. It contains a list of activities each of which has a *name* for identification purposes and a *source*, which is a reference to an operation in a Web service defined by an *endPointType* element. The endPointType element contains the names of the operation, portType, portName, and serviceName for a particular operation.

4.3 Composition Model

The *compositionModel* describes how the different Grid services are composed to form a new Grid service. It describes the control and data flow between various operations of the services, and also the direct communication between them in a peer-to-peer fashion. It consists of an export model and a notification model.

4.3.1 Export Model

The *exportModel* contains the list of activities that must be exported as operations of the workflow process. Any client can invoke these operations on the workflow instance by using standard mechanisms. Since the workflow instance can also be viewed as a standard Grid service, it can be used recursively as part of another workflow process. For each activity exported, the control and data flow are described by the *controlModel* and the *dataModel*, respectively.

The *controlModel* describes the chain of activities that are invoked when the exported activity is invoked by a client. Each controlModel element contains an attribute *controlIn* that references the first activity to be executed when the exported activity is invoked. Each controlModel also contains a series of *controlLinks*, which is a precedence list of all activities that need to be successively invoked as part of this exported activity.

The *dataModel* describes the flow of data that occurs when an exported activity is invoked. This flow may not necessarily mirror the flow of control between the various activities. Each dataModel element contains an attribute *dataInTo* (shown as "Data In" in Figure 3) which signifies the activity that will receive the data provided as input to the exported activity. A *dataOutFrom* attribute (shown as "Data Out" in the figure) designates the activity from which the data is returned to the caller.

The GSFL document provides enough information in the dataModel and the controlModel to not only dynamically generate the WSDL for the activities exported but also support the invocation of such exported activities dynamically, as we will explain in Section 5.

4.3.2 Notification Model

The *notificationModel* solves the problem of the workflow engine mediating at every step of an activity. As mentioned before, OGSA services communicate with each other using notificationSources and notificationSinks. The notificationModel provides a mechanism to link such sources to sinks and vice versa, along with a particular *topic*, using *notificationLinks*. The services can now communicate large amounts of data among each other, without having the need to go through the workflow engine. Users can still use the control and data models to communicate control messages and small amounts of data between each other, but the use of notification messages is the recommended form of communication for large amounts of data.

The composition model is illustrated by a simple example shown in Figure 4. Two services, A and B, constitute the workflow. The notification model consists of a single notification link A → B, which connects the notification source of service A to the notification sink of service B. The export model consists of a couple of activities that are exported, one of which is shown in detail in the figure. One of the exported activities is implemented by operations P and R of service A, and operation Q of service B. The control model of the exported activity consists of control links $P \rightarrow Q$ and $Q \rightarrow R$. Operation P of service A serves as the controlIn for the exported activity. The data model consists of a single data link $P \rightarrow Q$. This may possibly be because operation R may not need any data to be invoked. This is an example where the data links need not necessarily be the same as the control links. Operation P serves as the dataInTo, while operation Q serves as the dataOutFrom for the exported activity. Thus, invocation of the exported activity will trigger the set of operations described above, following the flow of control and data described by the different types of links.

4.4 Lifecycle Model

The *lifecycleModel* addresses the order in which the services and the activities execute. The *serviceLifecycleModel* contains a list of precedence links describing the order in which the services execute. Hence, all services need not be instantiated at startup but can be started once the preceding services have stopped executing.

The lifecycleModel uses the *scope* attribute for the workflow, which can be one of *session* or *application*. Session scope means that no state will be maintained between calls to the workflow engine. All calls to the engine are legal. Services are instantiated for each call by using the serviceLifecycleModel, and these services will be alive when calls to them are made.

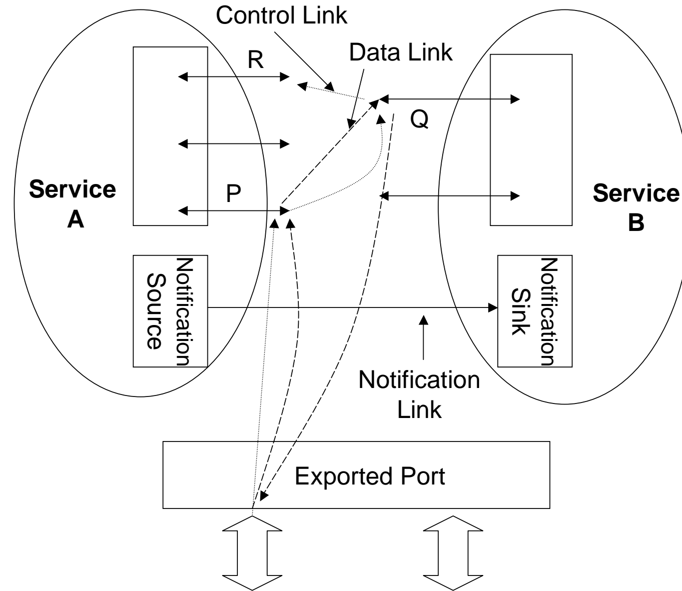

Figure 4: The Composition Model in action

Application scope means that state will be maintained in the workflow engine between calls. Services are instantiated only once per instance of the workflow, by using the serviceLifecycleModel. Hence, not all calls to the workflow engine may be valid, because the services that implement these activities may not be alive. Hence, we add an *activityLifecycleModel*, which describes the order in which they can be invoked. In other words, some activities can be invoked only if certain other activities have already been successfully invoked; for example a *checkout* operation in an online shopping system may be invoked only after one or more *buy* operations. Following the activityLifecycleModel will ensure that all the services will be alive when calls to them are made in the proper order.

We believe this design addressess the requirements for workflow for Grid services. We now discuss some of the issues involved in the implementation of a GSFL engine.

5 Implementation Details

As a basis for the prototypical development of GSFL we have selected the OGSI Technology Preview [20]. This is a Java-based implementation of the current Grid Services Specification [27] using Apache Tomcat [21] and Apache Axis [22]. Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet [24] and JavaServer Pages [23] technologies, while Axis is an implementation of the Simple Object Access Protocol (SOAP) [7] submission to W3C. In the following subsections, we describe a few important parts of the GSFL implementation.

5.1 GSFL Parsing

We use Castor [29] for parsing the GSFL schema and generating Java bindings for the same. Castor is an open source data binding framework for Java, and it can be used to generate the source for Java classes representing an XML schema. It also provides methods to unmarshal XML documents conforming to a particular schema into corresponding Java objects, and vice versa. Using Castor, we autogenerate the Java bindings for our XML schema, thus obviating

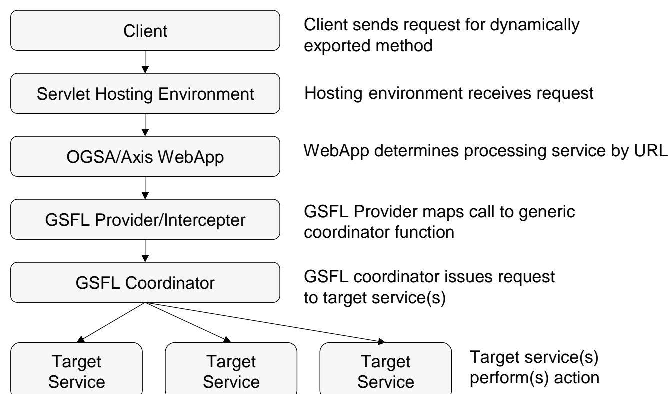

Figure 5: GSFL Control Flow

the need for us to write code to do the same. This significantly reduced our code development time.

Although Castor maps the GSFL schema into appropriate Java classes, it has no understanding of the semantics associated with the elements of the schema; for example Castor has no idea that the *Import* type in the GSFL schema has enough information (namespace & location) to import another GSFL document. Castor will only map a type such as *Import* to a Java class having the fields - namespace & location, with the appropriate getter and setter methods. Hence, we had to add a wrapper around the Castor generated classes so that such situations can be handled. This strategy was much easier than writing all of the above code ourselves.

5.2 WSDL Auto-generation

Apart from using Castor for the Java bindings for the GSFL document, we use the WSDL4J [18] for dealing with the WSDL documents of the individual services. WSDL4J is a toolkit that allows the creation, representation, and manipulation of WSDL documents describing services.

As we described in Section 4, the GSFL document contains enough information about the activities that are exported. Since the WSDL for the services participating in the workflow is available, the *Types* for the input and output messages, and the *Messages* themselves can be easily found out for these activities. Information about the rest of the WSDL, such as *Operations*, *Ports*, etc. can be easily derived from the GSFL itself. The key in the autogeneration of the WSDL for the workflow was to represent enough information the GSFL schema.

5.3 GSFL Coordinator

The heart of a GSFL workflow is the GSFLCoordinator service, which creates virtual ports and services that map to processes internal to the workflow. These ports are virtual: they do not physically exist on the GSFLCoordinator. However, a client can invoke methods on these ports, as they get mapped to a set of other calls with respect to the GSFLCoordinator.

A diagram representing the usual flow of control is shown in Figure 5. A client wishing to execute a GSFL workflow first begins an instance of the GSFL coordinator via the standard OGSA factory methods. The coordinator instance is then sent the GSFL that represents the workflow. Upon receipt of the GSFL document, the coordinator dynamically

generates a WSDL document with all of the newly exported operations included. This WSDL can then be used by clients wishing to execute the workflow and the operations that are dynamically exported. When a client calls an operation that has been exported by the GSFL coordinator, a request is sent via the servlet container to the OGSA webapp (Web Application). The OGSA webapp uses the mappings in the `server-config.wsdd` to send the call the GSFLProvider/Intercepter, which has been implemented as an extension to the standard OGSA RPCURIProvider class, which is responsible for dispatching incoming requests to the correct service instance based on the URL of the request. If the standard provider had been used, this invocation would have failed, since the operation does not physically exist on the GSFL Coordinator. The custom provider intercepts this call, however, and sends it to a generic marshaler function in the coordinator. Based on the information provided in the GSFL document, the coordinator then processes the request and maps the operation to a set of calls, which are, in fact, implemented by the set of services participating in the workflow.

The provider has provisions to differentiate between calls to operations in dynamically exported ports and the static ones. One of the static operations of interest is *generateWSDL*, which autogenerates the WSDL for the coordinator, inclusive of the dynamic ports, as described in Section 5.2.

6 GSFL Example

An example of a possible GSFL document for a filtered search service can be seen in Figure 6. The GSFL document shown contains the composition model shown in Figure 4, along with a few additions. It consists of three service providers: A, B, C. The activities that are part of the workflow (P, Q, R, S, T) have been listed in the activity model. The export model consists of two exported activities: PQR and ST. The composition of PQR has been explained in Section 4.3. Activity ST is composed of two other activities: S and T. It consists of a control link $S \rightarrow T$ and a data link $S \rightarrow T$. In this case, the control and data flows are identical. Since the scope of the workflow is “session”, the lifecycle model of the workflow consists of only the service lifecycle. The figure shows that services A and B have to be active at the same time, while C can start after A and B are done.

Despite the fact that the figure shows a toy example, it demonstrates the various constructs of the language, and the ease with which a workflow can be defined using individual services.

7 Conclusions

In this paper, we have described the Grid Services Flow Language (GSFL), a workflow framework for Grid-based services. We surveyed existing technology that addresses workflow for Web services and investigated their applicability to Grid services. Existing Web services technologies provide a number of desirable features that we can reuse; however, certain requirements such as peer-to-peer service interaction and complicated lifecycle management for the services were found lacking. We have designed GSFL such that it addresses these requirements, while still being able to integrate the features provided by existing Web service technologies.

Broad areas of application and enhancement in the Grid Services Flow Language remain. This is still a work in progress, and the language will continue to evolve depending upon the requirements of the Grid community. Ideas for future features include the ability to handle and process exceptions, similar to those found in XLANG; automatic integration with a graphical workflow editor; and enhanced ordering of tasks in the workflow, possibly with constructs such as loops and switch statements.

References

[1] Condor : High Throughput Computing. http://www.cs.wisc.edu/condor/, 2002.

[2] DAGMan (Directed Acyclic Graph Manager). http://www.cs.wisc.edu/condor/dagman/, 2002.

```
<definitions name="Sample" scope="session" targetNamespace="http://www.mcs.anl.gov/gsfl">

<!-- List of Service Providers -->
<serviceProvider name="A" type="AServiceType">
  <locator type="factory" handle="http://localhost:8080/ogsa/services/AServiceFactory" />
</serviceProvider>

<serviceProvider name="B" type="BServiceType">
  <locator type="factory" handle="http://localhost:8080/ogsa/services/BServiceFactory" />
</serviceProvider>

<serviceProvider name="C" type="CServiceType">
  <locator type="factory" handle="http://localhost:8080/ogsa/services/CServiceFactory" />
</serviceProvider>

<!-- List of activities -->
<activityModel>
        <activity name="P">
                 <source serviceName="A" portName="aPort" operation="P" />
        </activity>
        <activity name="Q">
                 <source serviceName="B" portName="bPort" operation="Q" />
        </activity>
        <activity name="R">
                 <source serviceName="A" portName="aPort" operation="R" />
        </activity>
        <activity name="S">
                 <source serviceName="A" portName="aPort" operation="S" />
        </activity>
        <activity name="T">
                 <source serviceName="C" portName="cPort" operation="T" />
        </activity>
</activityModel>

<!-- The composition model -->
<compositionModel>
  <exportModel>
    <exportedActivity>
      <exportedActivityInfo name="PQR" portType="exportPortType" />
      <controlModel controlIn="P">
                 <controlLink label="link0" source="P" target="Q" />
                 <controlLink label="link1" source="Q" target="R" />
      </controlModel>
      <dataModel dataInTo="P" dataOutFrom="Q">
                 <dataLink label="link0" source="P" sink="Q" />
      </dataModel>
    </exportedActivity>
    <exportedActivity>
      <exportedActivityInfo name="ST" portType="exportPortType" />
      <controlModel controlIn="S">
                 <controlLink label="link0" source="S" target="T" />
      </controlModel>
      <dataModel dataInTo="S" dataOutFrom="T">
                 <dataLink label="link0" source="S" sink="T" />
      </dataModel>
    </exportedActivity>
  </exportModel>

  <notificationModel>
    <notificationLink label="link0" source="A" sink="B" topic="AB"/>
  </notificationModel>
</compositionModel>

<!-- Lifecycle for the services -->
<lifecycleModel>
  <serviceLifecycleModel>
    <precedenceLink label="link0">
      <parent>
        <element serviceName="A" />
        <element serviceName="B" />
      </parent>
      <child>
        <element serviceName="C" />
      </child>
    </precedenceLink>
  </serviceLifecycleModel>
</lifecycleModel>
</definitions>
```

Figure 6: Sample GSFL Document

[3] Assaf Arkin. Business Process Modelling Language. http://www.bpmi.org/bmpi-downloads/BPML-SPEC-1.0.zip, June 2002.

[4] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web Services Choreography Interface. http://wwws.sun.com/software/xml/developers/wsci/index.html, June 2002. Version 1.0.

[5] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Parallel Computing. In *Proceedings of High Performance Distributed Computing*, pages 115–124, Redondo Beach, California, 1999. CCA Forum, .

[6] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt, August 1998.

[7] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP, May 2000.

[8] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. Web Services Description Language Version 1.2. http://www.w3.org/TR/2002/WD-wsdl12-20020709/, July 2002. W3C Working Draft 9.

[9] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language. http://www.w3.org/TR/wsdl, March 2001. Revision 1.1 - March 15, 2002.

[10] Distributed Systems Department Pervasive Collaborative Computing Environment Project (PCCE), LBL. PCCE Quarterly Reports. http://www-itg.lbl.gov/Collaboratories/quarterly-reports.html, April 2002.

[11] David C. Fallside. XML Schema Part 0: Primer. http://www.w3.org/TR/xmlschema-0/, May 2001.

[12] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, July 1998.

[13] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/research/papers/ogsa.pdf, January 2002.

[14] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 15(3), 2002.

[15] Dennis Gannon, Rachana Ananthakrishnan, Sriram Krishnan, Madhusudhan Govindaraju, Lavanya Ramakrishnan, and Aleksander Slominski. Grid Web Services and Application Factories. http://www.extreme.indiana.edu/xcat/AppFactory.pdf, June 2002.

[16] The Globus Project. http://www.globus.org/, June 2002.

[17] Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, and Ryo Neyama. *Building Web Services With Java*. SAMS, 2002.

[18] IBM. The Web Services Description Language for Java Toolkit (WSDL4J). http://www-124.ibm.com/developerworks/projects/wsdl4j, July 2002.

[19] Frank Leymann. Web Services Flow Language. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.

[20] OGSI Technology Preview Release. http://www.globus.org/ogsa/releases/TechPreview/, June 2002.

[21] Apache Jakarta Project. Apache Tomcat. http://jakarta.apache.org/tomcat/, June 2002.

[22] Apache XML Project. Apache Axis. http://xml.apache.org/axis/, June 2002.

[23] Sun Microsystems Inc. Java Server Pages. http://java.sun.com/products/jsp, 2002.

[24] Sun Microsystems Inc. Java Servlet Technology. http://java.sun.com/products/servlet/index.html, 2002.

[25] Satish Thatte. XLANG: Web services for Business Process Design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[26] The Hewlett-Packard Company. Web Services Conversation Language (WSCL) 1.0. http://www.w3.org/TR/wscl10/, March 2002.

[27] Steven Tuecke, Karl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, and Carl Kesselman. Grid Service Specification (Draft 2). http://www.gridforum.org/ogsi-wg/drafts/GS_Spec_draft02_2002-06-13.pdf, June 2002.

[28] UDDI Technical White Paper. http://www-3.ibm.com/services/uddi/pubs/Iru_UDDI_Technical_White_Paper.pdf, September 2000. Universal Description, Discovery and Integration is for discovering web services.

[29] Keith Visco and Assaf Arkin. Castor. http://castor.exolab.org/, 2002.

[30] World Wide Web Consortium. http://www.w3.org/, June 2002.