A Repository Service for Grid Workflow Components

Gregor von Laszewski[1,2,•] and Deepti Kodeboyina[1]

[1]Argonne National Laboratory, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60440

[2]University of Chicago, Computation Institute, Research Institutes Building #402, 5640 South Ellis Avenue, Chicago, IL 60637-1433

[•]Corresponding author: gregor@mcs.anl.gov

`http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-workflow-repository.pdf`

Abstract

As part of the Java CoG Kit we have defined a sophisticated workflow framework. This workflow framework projects an integrated approach towards executing tasks in Grid and non-Grid environments. One of the services needed is a convenient service to store, retrieve, and modify workflow components defined by the community similar to systems such as the comprehensive perl archive network. The availability of such a service will not only allow the definition of components useful for the greater grid community, but it will also be possible that it can be reused to support dynamically changing workflows managed by collaborative groups. In this paper, we present a simple extensible framework to design, build, and deploy a workflow repository service. This repository is intended to be used in ad-hoc Grids or in community Grids.

1 Introduction

Grids have become a valuable asset to many projects for conducting complex scientific discovery or business processes. In [10] we have introduced an extensible workflow framework that can easily interact with infrastructures using Grid and also commodity execution frameworks. The question arises how can we encourage and foster the reuse of components that are shared among a user group? In many projects, it is common to use repositories to share information or components with a community. As our dynamic workflow system introduces a module concept with namespaces and includes it is possible to utilize this feature as part of a comprehensive repository framework for Grids. In this paper, we will discuss the architecture of such a framework and what it will take to implement it. The paper is structured as follows. First, we present a short overview of some prominent projects and products for repositories. Next, we analyze our specific requirements based on the utilization within Grid frameworks. We have augmented them with a number of use cases. We, then present an architecture that addresses these requirements. Finally, we conclude the paper and point to future activities that we will conduct.

2 Repositories

While building a repository for our workflow components, we reviewed a number of existing solutions and related research. The use of repository-like systems has evolved from a simple directory where files are stored and managed by a single user to todays peer-to-peer file-sharing methods (see Figure 1).

Selected Repository trends	**Selected Grid Repository trends**
File duplication	Information Service
⇓	⇓
File sharing	Grid CVS
⇓	⇓
Object codes	Machine code repository
⇓	⇓
Centralised repositories	File Replica Service
⇓	⇓
Distributed repositories	Workflow Repository Service
⇓	
P2P File sharing	

Figure 1. Several trends that influence the development of repositories and lead to a variety of repository supporting technologies in Grid and non-Grid environments.

We observe that the development of source code and object code libraries motivated their management in repositories to manage them for or within a community. On the object code level, we see simple rules for archiving, building, and deployment in order to encourage reuse by other code developers. Issues that are of concern are different object and source code that must be managed for different architectures. An example for such a system is the well-known *ar* command in UNIX. The development of shared-code repositories with version control has enabled the development of code that is maintained by a group of people. Examples for such systems with central repositories are CVS [3] and Subversion [9]. In contrast, systems such as Bitkeeper [1] and git [4] allow us to distribute the code management on different servers.

Another important aspect is the dynamic inclusion of code during runtime. Besides object libraries, systems such as the comprehensive Perl archive network (CPAN) [2] and jEdit [7] enhance this concept by being able to download source code or precompiled objects from a repository and integrate them in running applications.

For Grids, modified versions to cvs were at one point available to be able to authenticate against CVS with Grid credentials. A more elaborate framework was developed as part of the Cactus code as it enhanced the simple Grid repository with so called "thorns" that represent reusable components within the Cactus framework [8]. Based on an idea by the first author, the Entrada [**?**] system used a modified version of jedit to do component integration into a GUI for Grids. A later version of Entrada replaced the logic of dynamic component integration with its own implementation. Both systems are no longer maintained.

Additional commodity tools that are typically associated with code development are listed in [5].

3 Abstraction for a Repository Model

All of the existing tools and systems define a component repository model [11] that addresses a number of management issues ranging from managing just an object repository to managing the deployment of components from a repository. Each of these issues require the definition of a model that describes the behavior of the repository framework with regards to issues identified to be important for a particular use case. We have identified and distinguishe between the following models.

Product model. It describes the products that are included in the repository and their relationship with each other. A product may consist of multiple objects. An example for a product model would be to organize products by topics and to assign the objects to the appropriate topics.

Object model. describes what kind of objects are included in a product. This may include documentation including requirements, specifications, designs, manuals, models, but also the source code, test cases, and examples or a combination thereof.

Object composition model. It describes the relationships and dependencies between the objects. This composition model may go beyond the product model and may also describe the set of allowed compositions without specifying a product.

Version model. It describes how items are to be versioned.

Distribution Model. It describes in which way the repository is distributed. Typical distribution models are centralized and decentralized. Along with the access model, more complex distributions can be described as for example that introduced by the peer-to-peer community.

Access model. It describes how, what, which, and when users are allowed to access and modify the objects stored in the repository.

Operation Model. It describes how the repository is administered and how backups are being conducted.

Deployment Model. It describes how the repository is deployed in a production environment. Ease of deployment is of special importance to groups that like to set up their own repositories.

In the following sections, we will specify models for a Grid-based workflow component repository that focuses on our Karajan workflow engine. Before we do so, we will first review the architecture of the Java CoG Kit as it significantly influences our repository architecture.

4 The Java CoG Kit Architecture

In order to support our vision of integration workflows management tools into the Java CoG Kit, we have identified a number of higher level abstractions including Grid tasks, transfers, jobs, queues, hierarchical graphs, schedulers, and workflows, and control flows, which make the development of Grid programs easier [10]. However, in contrast to other Grid efforts we have provided a mechanism in our workflow management framework that allows the integration of a variety of Grid and commodity middleware in an easy-to-comprehend framework based on the concepts of protocol independent abstractions, providers, and bindings. These are discussed below.

Providers. We have introduced the concept of Grid providers that allow different Grid middleware to be used as a part of an instantiation of the Grid abstractions. Hence the programmer does not have to worry about the particularities of the Grid middleware. Through dynamic class loading, we have the ability to do late binding against an existing production Grid. This includes the implementation of the Grid (task) abstractions, version binding against existing Grid Toolkits, and resource binding.

Abstractions. We have identified a number of useful abstractions that help in the development of elementary Grid applications. These abstractions include job executions, file transfers, workflow abstractions, job queues and can be used by higher level abstractions for rapid prototyping. As the Java CoG Kit is extensible users can include their own abstractions and enhance the functionality of the Java CoG Kit.

Bindings. Through these concepts, the Java CoG Kit protects your development investments by protecting you from changes to the Grid middleware.

Based on these elementary concepts, we designed a layered architecture that allows the gradual enhancement of workflow capabilities within our application (see Figure 2).

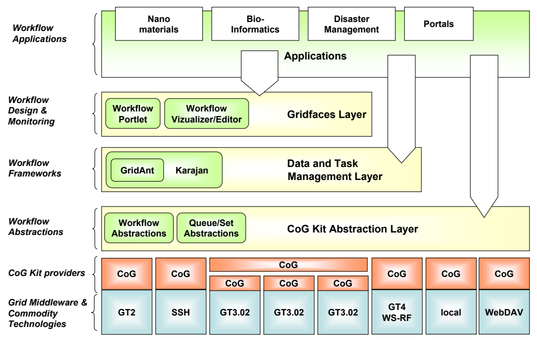

Figure 2. The layered approach of the Java CoG Kit provides mechanisms for incrementally enhancing workflow management components.

On the bottom of the architecture, we have the typical Grid middleware. Above it lies our Java CoG Kit abstraction layer that focuses on job submission file transfer and authentication. With the help of CoG providers, we can now access a number of different Grid middleware.

The simplest form of workflow abstractions that we support are embedded in the definition of a task. This may include a file transfer, a job submission, an authentication or any other task that has to be done. Our tasks are defined to have a status that can be queried. Based on this elementary definition, we define task queues and sets.

On the next higher level, we define APIs, tools, and services that help in the coordination of such tasks. It is handled by a workflow engine that we have derived from GridAnt. However, the scalability of GridAnt was limited. Hence, we designed a complete new workflow engine with many more advanced language features. This workflow engine is also called *Java CoG Kit workflow Karajan engine*.

At the next level, we define Gridfaces that are visual abstractions shared amongst stand-alone applications or portals. With the help of Gridfaces, it will become easy to develop visuals for either portals or stand-alone applications. The value of the Java CoG Kit workflow solution lies in its simplicity and its ability to be integrated in a solution that allows us to expose workflow to a variety of users. As indicated in Figure 3, we are prototyping a system that provides an API based on abstractions and the integration of services. Command line interfaces, web portals, and a Grid Desktop that expose the workflow functionality in a convenient user interface are also under development. The integration of these tools is possible through an integrated but modular architecture as depicted in Figure 4. Our workflow system contains at its heart the workflow engine that is augmented by a variety of tools, our workflow specification languages, and programing frameworks to reuse workflows and the engine. It is important to note that the workflow engine is itself an abstraction and could for example be replaced by a specialized version suitable and customized for a particular community.

In the rest of the paper, we will focus on the architecture of our workflow repository service.

5 Repository Requirements

In our repository architecture design we would like to address the following requirements. The repository must be easy to maintain and setup. It should support single or multiple users. We should be able to replicate the repository. The database backend should be replicable.

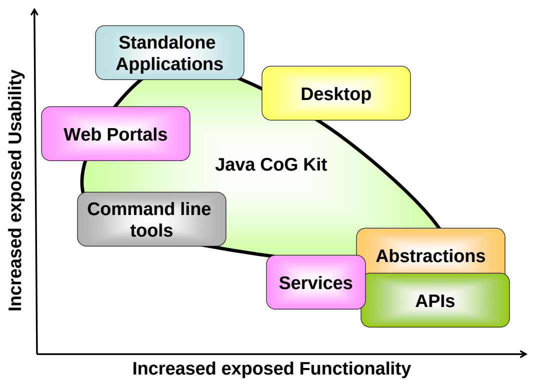

Figure 3. The Java CoG Kit integrates several mechanisms that together build a powerful workflow management system for Grids.

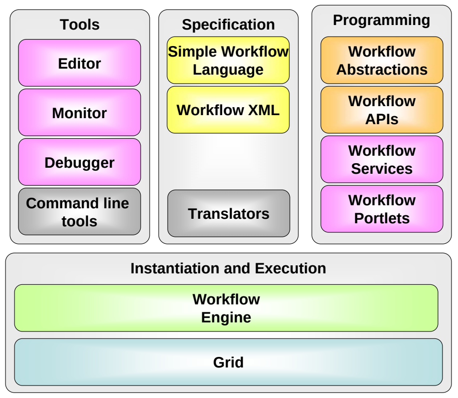

Figure 4. Components of the Java CoG Kit workflow framework

An easy programming interface must be available that is exposed through a Java interface [6] as well as through Java CoG Kit workflow elements [10]. A Web services or Grid services schema will be easily derivable from this interface.

In addition to these elementary requirements, we have identified a number ofmore sophisticated requirements and project the in simple use cases. These requirements influence our architecture and its implementation significantly. We depict the use cases in Figure 5. Here the symbols **U** represent users, **R** represents a repository, and **P** represents a provider that uses commodity or Grid technologies as backend. Relationships between the components are indicated with an arrow. The use cases are numbered from i-xi).

i) A user maintains his own repository.

ii) Several users share a repository.

iii) A User maintains a repository that can be replicated to a remote resource.

iv) Several users maintain a shared repository, but the repository is actually distributed.

v) A repository has one provider.

vi) A repository has multiple providers.

vii) A distributed repository has different providers for the distributed parts of the repository when the repository is replicated.

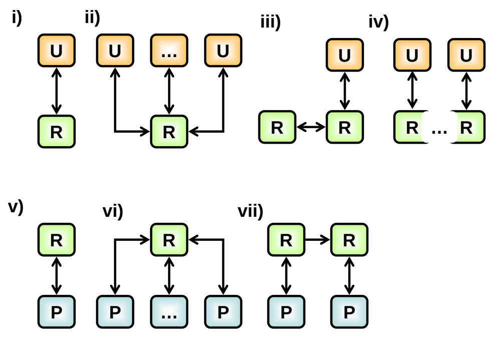

Figure 5. Use cases of the repository integration into the infrastructure.

As is obvious certain combinations can occur between the use cases listed in i-iv and v-vii. An example would be the combination of i and vii. Let us demonstrate a use case that requires the combination of both models. Consider a user that maintains a repository on his local client machine, a laptop. The repository is stored in memory. In case he wants to travel from point A to point B, he may be forced to shutdown the machine. To do so he needs to be able to replicate the repository normally stored in memory into a checkpointed state. A simple way to create such a state is to store the contents in a file save the file to a location that can be retrieved at a later time. Next time the user is working, he refreshes his local repository form the file and continues with his work.

6 Architecture

The architecture of our repository follows that of the Java CoG Kit layered architecture. This proven concept allows us to gradually enhance the functionality of the repository framework, while at the same time project reusable interfaces that can be adapted to a variety of implementations and backends.

Figure 6 depicts the architecture of our repository framework. We identify several layers and proceed from the bottom up. On the bottom of the architecture is the infrastructure layer that contains the Grid and services as well as frameworks that can be used to implement the backend on which the repository may be implemented. Possible backend systems of interest are WS-RF based services, gridftp, regular file systems, http, webdav, cvs, subversion, and naturally a database such as SQL. Other systems such as Grid RFT and db are naturally also options. Providers build the next layer in our architecture. For each of the backends, we can develop a provider that bridges between the infrastructure backend to our higher level abstractions. The abstraction layer defines a number of interfaces that simplify the development of Java code accessing the repository. Naturally, it will be simple to derive service descriptions from these interfaces as to provide web service or Grid service based protocols. Our abstractions have the advantage that the developer does not necessarily have to know anything about XML in order to develop programs for the repository. We have shown that this concept is quite effective in other efforts conducted by us [6].

The workflow engine that we designed is part of the execution layer. It makes use of the interfaces to the repository. In order to support the specification of workflow using the repository, we have added several elements to our workflow specification language that make the reuse of the repository directly within a workflow simple. The application layer consists of applications that may use the specifications that are executed by the workflow engine.

Within the repository abstractions we define interfaces for creating the repository schema, joining repositories, browsing and searching the contents, updating the repository, check pointing, and so on.

Figure 7 shows the pseudo code for connecting and interacting with the repository. As expected, we have provided methods for the connection, the retrieval of components, the upload of components, and the search of components. The components that are part of the repository are represented through additional interfaces. Figure 8 depicts the pseudo code of the interface to a component. It is important to note that we have attributes that are defined as name value attributes. The interface pseudo-code for creating a component repository schema defining the attributes is shown in Figure 9. It allows us to maintain an arbitrary number of string-based fields that are contained within the repository.

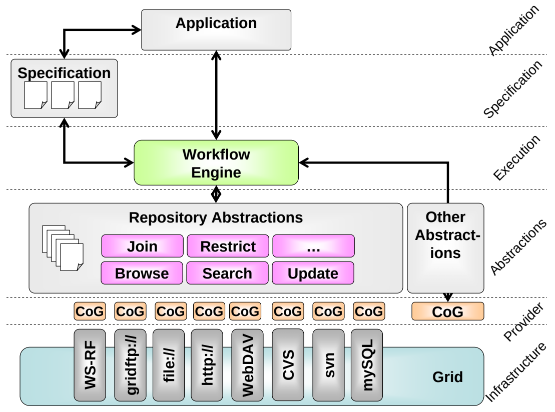

Figure 6. The architectural design of the Workflow repository follows the layered approach introduced by the Java CoG Kit.

Based on these simple interfaces it is possible to integrate a number of providers through implementations into our framework. This variability is important as it protect us from changes to the backend repository components due to version upgrades and allows the potential integration of not only Grid services but also of commodity services, such as bit torrent and other well accepted non Grid solutions.

7 Workflow Repository Model

In the previous section we have identified a general architecture for Grid repositories. Furthermore, we introduced in Figure 9 how to manage attributes that are to be stored as parts of components. In this section, we apply this architecture to the definition of a model and implementation of a workflow repository.

As we have chosen an extensible architecture, we will use a very simple repository model to support our workflow. This will allow us to quickly proceed and to expose minimal functionality to our user community. At a later stage, we will revisit our repository model and enhance this where appropriate.

Our product model is defined by the storage of components that relate to the Java CoG Kit workflow specifications. Each component in the repository refers to a component that can be included within the workflow through an enhanced include element.

```
package org.globus.cog.repository;

interface repository {

  setProvider(String type,
              String hostname,
              String port)
    // Sets the appropriate provider for the
    // component repository.

  connect ()
    // Connects to the repository.

  disconnect()
    // Disconnects from the repository.

  boolean isConnected ()
    // Returns the status of the connection.

  add (component name)
    // add the component to the repository.

  remove(String name)
    // Removes the named component.

  component search(String query)
    // returns the component that matches the
    // search expression.

  component get (String name)
      // Gets the Component Object from
      // the repository.

  load(File filename)
    // loads all components from a file with
    // the filname. The contents of the file
    // is an XML representation of the
    // components.

  save(String filename)
    // Saves the contents of the repository to
    // a named file.
}
```

Figure 7. The pseudo code of the interface that describes the repository

```
package org.globus.cog.repository;

interface component {

  String toString()
    // Converts the component to a string.

  String toXML()
    // Converts the component to an XML
    // string.

  String getMetadata(String name)
    // returns all the attributes in an XML
    // less uString.

  setAttribute(String attribute,
               String value)
    // Sets attribut to value.

  Enumeration getAttributes()
    // returns an enumeration of all defined
    // Attributes
}
```

Figure 8. The pseudocode of the interface to access a component

```
package org.globus.cog.repository;

interface componentAttributes {

  add (String name)
    // adds a named attribute and sets
    // its value to null

  add (String name, String description)
    // adds a named attribute and ist
    // description

  String get (String name)
    // returns the value of the named attribute

  set (String name, String value)
    // sets the value of a named attribute
    // by default it is null

  remove (String name)
    // removes the named attribute

  Enumeration list()
    // returns a list of all its attribute names.
}
```

Figure 9. The pseudo code of the interface that describes the repository

Our object model defines the workflow components. The components include a number of attributes and can be defined through a simple XML file. Without changes, these files can be included in other workflows. The attributes that are part of our object schema are defined as follows.

Name specifies the name of the component to call it in a workflow.

Short description is a one line description that simplifies the creation of a list function for the components included in the repository.

Description is an extensive description about what the component does and how it is used. It is essentially the manual page.

License specifies the license under which the workflow component is distributed. If left empty the Globus Toolkit license is used

Author List the list of authors and contributors or is a pointer to a reference describing a team.

Code specifies the source code of the component.

Signature is a digital signature that can be used to identify if the downloaded component is the same.[1]

Version is the version number.

Date entered specifies the date when the component was entered.

Date modified specifies the date when the component was last modified.

Deprecated by in case the component is deprecated.

Language specifies if the component is written in xml.

Type specifies the type of the component such as documentation, source code.

Keywords specifies a number of keywords this component is associated with.

Our object composition model is based on simple include statements that can be placed in the Java CoG Kit workflow specification. The include statement fetches the component from the repository and includes the contents at runtime into the location. At this time, only the current version is stored in our repository. In the future we plan to work on more extensive version models in collaboration with a user and group based access model. The distribution model is at this time only local or centralized. In the local case, the repository is stored on the local hard drive. In the distributed case, the repository is stored remotely and can be shared by authorized users. As we like to have an easy deployment and operations model we will use initially only a local file provider and a mySQL database. The later is easy to install, and actually often already installed on default operating systems.

[1]This field is added by default und is assumed to be under restricted write access.

8 Karajan language extension

In order for the repository to be used by our user community, we have developed a number of predefined elements that access the repository and are stored in the file *repository.xml*. Once this file is included in a workflow, we can use the appropriate repository functions. Within this file we define the *namespace cog:repository:*. According to the interfaces defined in earlier sections, we define corresponding elements with the appropriate names. For example, the following code snippet represents the definition of the get component as depicted in Figure 10. We left of the details of the implementation and just indicated them with ... .

```
<element name="cog:repository:get"
         arguments="name"
         types="java.lang.String">
   ...
</element>
```

Figure 10. The pseudo code for defining an element in Karajan

Instead of listing each of the definitions we list a simple example to demonstrate the ease of use in a workflow specification. We assume that we have defined a workflow element called ”task:gaussian” that we store in the repository that is located on the host repository.mcs.anl.gov through the port 4711. In our script we get the blast routine and execute it. The script realizing this is listed in Figure 11.

9 Schema representation

To enable a mechanism to describe components easily and to move them between our different layers in the architecture, we have defined a simple XML schema. In Figure 13 we show an example of a workflow component written by the Java CoG Kit group. It includes the attributes that we heave identified to be useful for our initial prototype of the repository. As the schema is extensible we can include customised tags as necessary.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
   <xs:schema targetNamespace="http://cogkit.org/.../workflow/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:simpleType name="stringType">
        <xs:restriction base="xs:string"/>
    </xs:simpleType>

    <xs:complexType name="deprecatedType">
      <simpleContent>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="version" type="xs:string" use="required"/>
      </simpleContent>
    </xs:complexType>

    <!-- definition of simple elements -->
    <xs:element name="shortDescription" type="xs:string"/>
    <xs:element name="description"      type="xs:string"/>
    <xs:element name="dateCreated"      type="xs:date"/>
    <xs:element name="dateModified"     type="xs:string"/>
    <xs:element name="language"         type="xs:string"/>
    <xs:element name="deprecates"       type="xs:string"/>
    <xs:element name="signature"        type="xs:string"/>

    <!-- definition of complex elements -->
    <xs:element name="metadata">
      <xs:complexType><!-- definition of attributes -->
        <xs:attribute name="name"       type="xs:string"  use="required"/>
        <xs:attribute name="author"     type="xs:string"  use="required"/>
        <xs:attribute name="version"    type="xs:string"  use="required"/>
          <xs:sequence>
          <xs:element ref="shortDescription"/>
             <xs:element ref="description"  minOccurs="0"/>
             <xs:element ref="dateCreated"/>
            <xs:element ref="dateModified" minOccurs="0"/>
             <xs:element ref="language"/>
             <xs:element ref="deprecates"   minOccurs="0" type="deprecatedType" />
             <xs:element ref="signature"    minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
     </xs:element>
    <xs:element name="source" type="stringType">
    </xs:element>
  </xs:schema>
```

Figure 12. Pseudocode of the general XML schema for the component description.

10 Collaborative and Dynamic Repository Use Case

With the availability of a repository service a number of collaboratory use cases can be supported. We focus on two use cases. First, the repository is centrally maintained by a group of dedicated administers and export components that are contributed by the community. Second, a repository that is restricted to a particular user group. Both repositories will have a number of overlapping requirements in regards to the organization and maintenance of the repository. However, the access to the repository is restricted. This can be achieved by using the Java CoG Kit abstraction mechanism for authentication and choosing an appropriate provider such as a Grid security enables WS-RF service. While controlling the access mechanism through a Grid solution, we can provide secure access to a user group. In the first case, we simply restrict the users that have access to the repository to the set of administrators.

What can we now achieve with such a repository? One of the issues of producing solutions to a complex scientific or business process is the software engineer-

```
<?xml version="1.0"?>
  <xmlns:k="http://cogkit.org/.../workflow/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://cogkit.org/.../workflow/workflow.xsd">

    <k:metadata name="job-submission-form.xml" author="The CoG Kit Group" version="1.0">
      <k:shortDescription> Job-Submission </k:shortDescription>
      <k:description>
         Submission of a job to a Globus Service with input provided
         interactively using dynamically generated dialogs.
      </k:description>
      <k:dateCreated>  18-10-2004 </k:dateCreated>
      <k:dateModified> 10-05-2005 </k:dateModified>
      <k:language> gridant </k:language>
      <k:deprecates name="job-submission-form.xml" version="0.9"/>
      <k:signature>  UrXLDLBIta6skoV5/A8Q38GEw44...  </k:signature>
    </k:metadata>
    <k:source>
      <project>
        <include file="cogkit.xml"/>
        ...
        <while>
          <set name="formData">
          <form:form title="test" id="IDForm" waitOn="IDSubmit, IDQuit">
          ...
        </while>
      </project>
    </k:source>
</xml>
```

Figure 13. Sample XML document confirming to the schema for the component description.

ing process to develop components that can be reused by peers. Having a repository allows us to publish such components easily. Not only can we store them, but due to the ability of the workflow system to dynamically load them it is possible to design dynamically changing workflows dependent on the state a Grid is in. Assume we have a Grid in which we like to conduct a parameter study, but we are more interested in precise results around certain parameters than others. In case enough resources would be available, we can formulate a workflow that goes through the list of parameters one by one sorted by priority. We apply the same algorithm (formulated as workflow) on each of the parameters studies. However when the resources become oversubscribed, it may be advantageous to decrease the accuracy of the calculations for parameters that we are not that interested in. Such a calculation could be included in the workflow prior to the workflow being started. However, experience shows that observations derived during such experiments may lead to the change of certain boundary conditions. This could be as simple as *my collaborator has provided me with a faster algorithm to solve a study of one parameter*. As a users parameter study may run for month at a time it would be inconvenient to stop the ongoing study. Instead, one could ingest the new algorithm in the running system and the workflow would dynamically adapt to this change.

11 Conclusion

We have presented a general architecture for Grid repositories. The design is based on the layered architecture following practices used by the Java CoG Kit. We can use such a repository as part of our Java CoG Kit workflow system in order to enable a dynamically changing workflow. Since our workflow is interpreted, the execution of components is conducted during runtime. The existence of such a repository is useful in order to support the shared development of components or to distribute components within a community. Replication of contents from a centralized to a local repository can easily be achieved by our predefined commands. Next steps will include setting up a component repository for a trusted community to allow the integration of components by the community.

```
<project>

  <include name="cog.xml">
  <include name="cog-repository.xml">

  <!-- set the provider to a mySQL server -->
  <cog:repository:setProvider
       type="mySQL"
       host="repository.mcs.anl.gov"
       port="4711"/>;

  <!-- connect to the server -->
  <cog:repository:connect/>

  <!-- get task gaussian from repository -->
  <cog:repository:get name="task:gaussian"/>

  <!-- disconnect from the server -->
  <cog:repository:disconnect/>

  <!-- now call gaussian -->

  <task:gaussian file="parameterfile.dat"/>

</project>
```

Figure 11. The workflow specification for a simple use of the repository

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38. DARPA, DOE, and NSF support Globus Project research and development. The Java CoG Kit is supported by DOE SciDAC and NSF Alliance. We like to thank Deepti Kodeboyina for her help in improving the presentation of this paper.

References

[1] Bitkeeper. Web Page. Available from: http://www.bitkeeper.com/.

[2] Comprehensive Perl Archive Network. Web Page. Available from: http://www.ptan.org.

[3] CVS - Concurrent Version System. Web Page. Available from: http://www.gnu.org/software/cvs/.

[4] git - Global Information Tracker. Web Page. Available from: http://www.kernel.org/pub/software/scm/cogito/README.

[5] CodeOrganizer.com: Software Tools for Code Development. Web page. Available from: http://www.codeorganizer.com/index.htm.

[6] Java Commodity Grid (CoG) Kit. Web Page. Available from: http://www.cogkit.org.

[7] jEdit Programmers Text Editor. Web Page. Available from: http://www.jedit.org/.

[8] Michael Russell, Gabrielle Allen, Ian Foster, Ed Seidel, Jason Novotny, John Shalf, Gregor von Laszewski, and Greg Daues. The Astrophysics Simulation Collaboratory: A Science Portal Enabling Community Software Development. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, pages 207–215, San Francisco, CA, 7-9 August 2001. Available from: http://www.mcs.anl.gov/˜gregor/papers/astro-hpdc10.pdf.

[9] Subversion - Version Control System. Web Page. Available from: http://subversion.tigris.org/.

[10] Gregor von Laszewski and Mike Hategan. Grid Workflow - An Integrated Approach. In *To be published*, Argonne National Laboratory, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60440, 2005. Available from: http://www.mcs.anl.gov/˜gregor/papers/vonLaszewski-workflow-draft.pdf.

[11] Wikipedia: Component Repository Management. Web page. Available from: http://en.wikipedia.org/wiki/Component_repository_management.