1

Java CoG Kit Workflow

Gregor von Laszewski[1,2], Mihael Hategan[2], and Deepti Kodeboyina[1]

[1] Argonne National Laboratory, Argonne IL, 60430, USA gregor@mcs.anl.gov
[2] University of Chicago, Research Institute Suite 405, 5640 South Ellis Avenue, Chicago IL, 60637, USA

1.1 Introduction

In order to satisfy the need for sophisticated experiment and simulation management solutions for the scientific user community, various frameworks must be provided. Such frameworks include APIs, services, templates, patterns, GUIs, command line tools, and workflow systems that are specifically addressed towards the goal of assisting in the complex process of experiment and simulation management. Workflow by itself is just one of the ingredients to a successful experiment and simulation management tool.

The Java CoG Kit provides an extensive framework that helps in the creation of process management frameworks for Grid and non Grid resource environments. Hence, process management in the Java CoG Kit can be defined using a Java API providing task sets, queues, and Direct Acyclic Graphs (DAGs). An alternate solution is provided in a parallel extensible scripting language with an XML syntax (a native syntax is also simultaneously supported). Visualization and monitoring interfaces are provided for both solutions, with plans for developing more sophisticated but simple to use editors. However, in this chapter we will mostly focus on our workflow solutions. The Java CoG Kit workflow solutions are developed around an abstract, high level, asynchronous task library which integrates the common Grid tasks: job submission, file transfer, and file operations.

The chapter is structured as follows. First, we provide an overview of the Java CoG Kit and its evolution that led to an integrated approach to Grid Computing. We present the task abstractions library which is necessary for a flexible Grid workflow system. Next, we provide an overview of the different workflow solutions that are supported by the Java CoG Kit. Our main section focuses on only one of these solutions, in the form of a parallel scripting language which supports an XML syntax for easy integration with other tools, as well as a native, more human-oriented syntax. Additionally, a workflow repository of components is also presented, which allows sharing of workflows between multiple participants, and dynamic modification of workflows. We exemplify the use of the workflow system with a simple, conceptual application. We conclude the chapter with ongoing research activities.

1.1.1 Overview of the Java CoG Kit

One of the goals of the Java Commodity Grid (CoG) Kit is to allow Grid users, Grid application developers, and Grid administrators to easily use, program, and administer grids from a high-level framework. The Java CoG Kit leverages the portability and availability of a wide range of libraries associated with the Java framework, while promoting easy and rapid Grid application development. The Java CoG Kit started with the development of a client-side and partial server-side implementation of the classic Globus (Globus Toolkit 2.x) libraries under the name of "jglobus". Today jglobus includes, among others, Java implementations of the Globus Security Infrastructure (GSI) libraries, GridFTP, myProxy, and GRAM. The jglobus library is a core component of both Globus Toolkit 3 and Globus Toolkit 4.

Today, the Java CoG Kit provides rich concepts and functionality to support process management which goes beyond that of the Globus Toolkit. One of the concepts that has proven to be useful in protecting

the user from frequent changes in the standards development is the concept of abstractions and providers. Through simple abstractions we have build a layer on top of the Grid middleware that satisfies many users by giving them access to functions such as file transfer, or job submission. These functions hide much of the internal complexity present within the Grid middleware. Furthermore, it projects the ability to be able to reuse commodity protocols and services for process execution and file transfer instead of only relying on Grid protocols. In order to integrate new services all a developer has to do to define a relatively simple set of providers that follow a standard interface definition. In addition to the abstraction and provider concept the Java CoG Kit provides also user-friendly graphical tools, workflows, and support for portal developers.

Hence, the Java CoG Kit integrates a variety of concepts to address the needs posed by the development of a flexible Grid upperware toolkit as depicted in Figure 1.1. End users will be able to access the Grid through standalone applications, a desktop, or a portal. Command line tools allow users to define workflow scripts easily. Programming is achieved through services, abstractions, APIs, and workflows. Additionally, we integrate commodity tools, protocols, approaches, methodologies, while accessing the Grid through commodity technologies and Grid toolkits. Through this integrated approach the Java CoG Kit provides significant enhancements to the Globus Toolkit. Hence, the Java CoG Kit provides a much needed add on functionality to Grid developers and users while focusing on the integration of Grid patterns through the availability of a toolkit targeted for the development of Grid upperware.

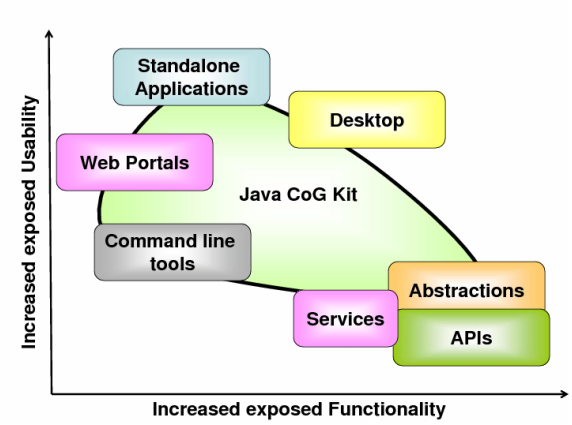

Fig. 1.1. An Integrated Approach.

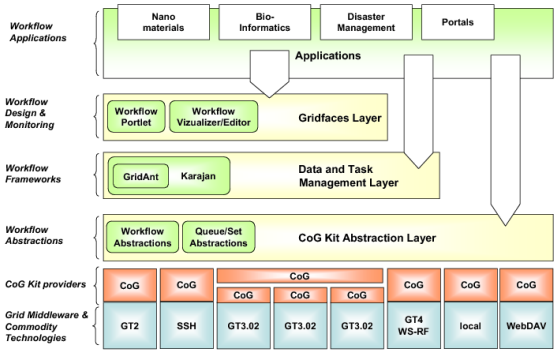

Fig. 1.2. A layered Architecture.

1.1.2 Abstractions and Providers

The Architecture of the Java CoG Kit is derived from a layered module concept that allows easier maintenance and bridges the gap between applications and the Grid middleware 1.2. It allows for the easy integration of enhancements developed by the community. One of the strengths of the Toolkit is that it is based on the abstractions and providers model.

We have identified a number of useful basic and advanced abstractions that help in the development of Grid applications. These abstractions include job executions, file transfers, workflow abstractions, and 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 its functionality.

We introduced the concept of Grid providers that allow a variety of Grid middleware to be integrated into the Java CoG Kit.The Abstractions allow the developer to choose at runtime to which Grid middleware services tasks related to job submission and file transfer are submitted. This capability is enabled through customized dynamic class loading thus facilitating late binding against an existing production Grid.

1.1.3 Workflow Concepts in the Java CoG Kit

The origin of the adaptive workflow framework is based on the work described in [2], which defines a dynamically adapting scheduling algorithm that chooses optimized algorithms based on performances measurements to identify a set of resources that fulfill a given high-level function such as a matrix multiplication. The task of conducting the matrix multiplication is specified at the time the workflow is specified. However, its instantiation and the appropriate selection of which algorithm to choose are conducted during runtime. Other important evolutionary steps are the development of GECCO [**?**], that included dynamic fault detections and a workflow debugger; and GridAnt [**?**] that uses the commodity tool ant to manage grid tasks.

Today, the Java CoG Kit contains a number of workflow concepts that have been shaped by our long collaborations with experimental scientists [4, 3]. We evolved several concepts as part of the Java CoG Kit workflow framework. These concepts include (a) abstractions for queuing systems and workflow graphs with simple dependencies as found in [2]; (b) event-based notifications and monitoring as found in [2, 7, 6, 11]; (c) elementary fault tolerant features; (d) a simple execution pattern [7], now termed *cog pattern*; (e) hierarchical graphs [7]; (f) structured control flow with loops and conditions [8]; (g) visual interfaces to augment the workflow execution [2]; (h) an adaptive workflow framework; and (i) workflow component repositories.

1.1.4 Lessons learned from GridAnt

The Apache Ant project presents certain characteristics that seem to make it suitable as a workflow language and engine. Features such as native dependency structured build targets make it easy to write declarative dependency based workflows.

Ant is designed around the concept of targets and tasks. Targets exist only as top-level entities. Dependencies between targets are specified using target names as handles. Targets in Ant are executed sequentially, without the possibility to exploit the parallelism inherent in a dependency-based specification. Targets are composed of tasks which are generally executed sequentially. Parallel execution of tasks can be achieved with the `<parallel>` task. It executes its nested tasks in parallel, synchronously. Globally scoped immutable properties can be used to provide value abstractions. Conditional execution is achieved at the target level based on property values. Iterations are not possible in Ant.

GridAnt [5] is an extension to the Apache Ant build system, which adds the following features:

- The `<gridExecute>` and `<gridTransfer>` tasks, allowing job submission and file transfers using the Java CoG Kit abstractions API.
- A `<gridAuthenticate>` task which launches a GSI proxy certificate initialization window.
- A generic progress viewer which can visualize target dependencies and track the state of the execution of each target (Figure 1.3).
- Partial iteration capabilities. Full support for iterations featuring iteration variables was impossible due to the immutable nature of Ant properties, an aspect which was deeply ingrained into multiple areas of the Ant engine.

The use of the Ant engine posed the following problems which limited the possibility of implementing complex workflows:

- Inability to concurrently execute targets
- Lack of full iteration support
- Difficulties in expressing conditional execution
- Scalability limitations in parallelism, due to the extensive use of native threads, leading to memory exhaustion when running workflows describing large numbers of parallel tasks
- Difficulties in the ability to consistently capture the state of the execution of a particular workflow, leading to an inability to add checkpointing and resuming abilities to Ant.
- The authors of Ant favored verbose specifications and made abstractions, parameterized execution, or self-contained extensibility difficult.

All of these disadvantages motivated us to develop a more streamlined and powerful workflow framework.

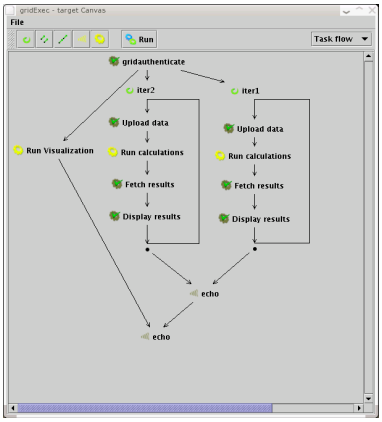

Fig. 1.3. GridAnt Viewer

1.2 The Java CoG Kit Karajan Workflow Framework

Karajan was designed and implemented when it became apparent that the shortcoming of GridAnt could not be avoided without a complete re-design of the engine. It provides a cleaner and clearer separation between its composing parts that have been specifically designed to address the shortcomings of our previous systems. The name "Karajan originates from the name of a famous conductor of the Berlin symphony orchestra.

1.2.1 Architecture

The architecture of the Java CoG Kit Karajan framework is displayed in Figure 1.4. It contains the workflow engine that interfaces with high level components namely a visualization component that provides a visual representation of the workflow structure and allows monitoring of the execution, a checkpointing subsystem that allows the checkpointing of the current state of the workflow, and a workflow service that allows the execution of workflows in behalf of a user. A number of convenience libraries enable the workflow engine to access specific functionalities such as a task library to enable access to Grid services, a forms library to enable the dynamic creation of forms as part of workflow tasks, a Java library to extend the workflow language with elements based on Java classes, and a core library that includes convenience abstractions used within the workflow engine.

The language structure specification is designed carfully, so it can be syntactically formulated in two ways. One possibility is to use an XML based syntax that has its origin from Gridant but is significantly enhanced with features such as control structures. The other way is based on the desire to have more simplified syntax for scripting that includes such features as the replacement of XML begin and tags witch simple brackets. This syntax is significantly shorter than the XML syntax and provides the script designer with a rapid prototyping mechanism. The languages can be transformed into each other.

The workflow execution engine employs a lightweight threading in order to support large scale workflows efficiently.

The philosophy of Karajan is based on the definition of hierarchical workflow components. However instead of just supporting Direct Acyclic Graphs (DAG) a much more powerful internal implementation is provided that is also reflected within the language structure. Hence we provide primitives for generic sequential and parallel execution, sequential and parallel iterations, conditional execution and functional abstraction. At the same time we provide support for common data types such as lists and maps that are specifically targeted to support parameter studies.

The Grid interface is enabled with the help of the Java CoG Kit Abstractions API that we introduced earlier. Through the use of the provider concept which provides a mechanism to interact with tasks by defining specific task handlers for different Grid middleware and services, the decision of how a particular

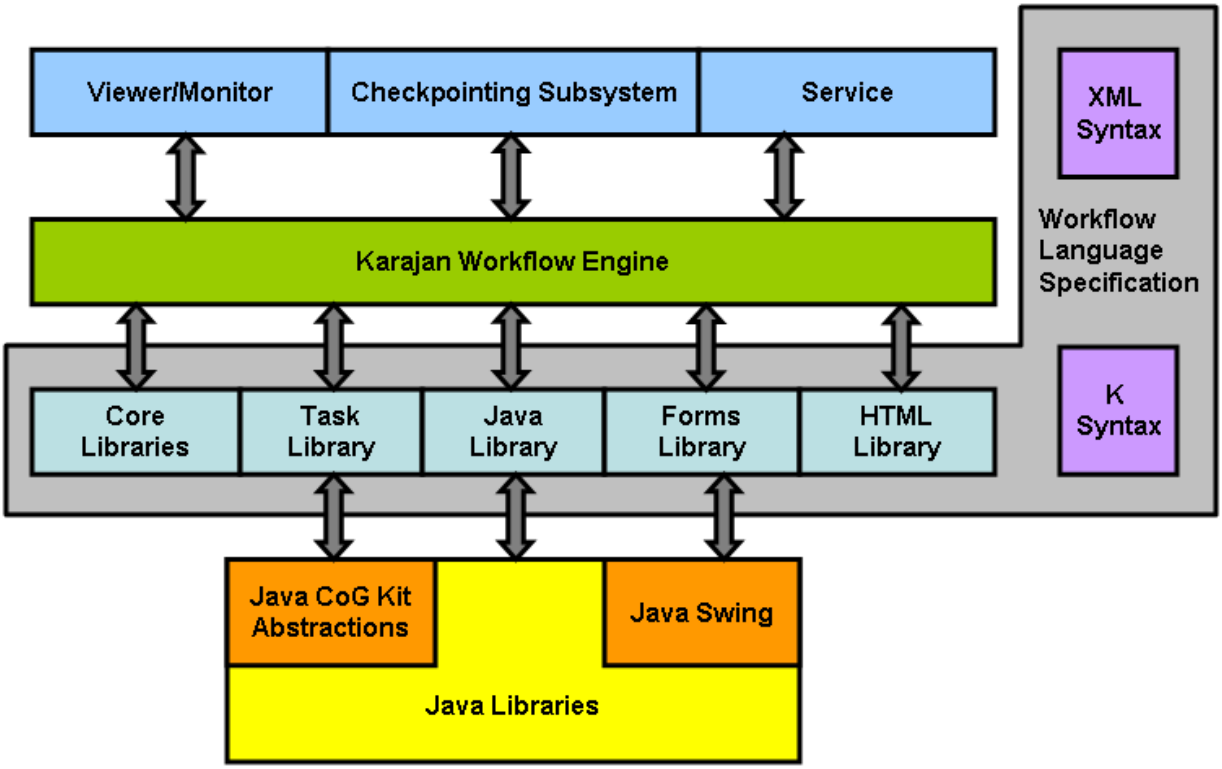

Fig. 1.4. The components of the Java CoG Kit karajan module build a sophisticated Workflow system.

task is to be executed can be deferred until the task is mapped onto a specific resource during runtime. This makes it possible to focus on the definition of tasks while deferring the actual instantiation and mapping of the component once onto a resource during runtime. The actual mapping can be performed through the use of a simple scheduler example that is included within the Karajan framework. This example also demonstrates that it will be easy to integrate user defined scheduling algorithms and make the Karajan workflow framework an ideal candidate for enhancements through the user community.

Based on this flexibility, the Karajan workflows can provide inter-operability between multiple grids. Considering that one of the fundamental problems of the Grid is that through deployment variations we can not assume that the same Grid middleware version is available everywhere. However, with the Java CoG Kit and the Karajan workflow we can formulate workflows despite the fact that the underlying resources use different versions of Grid services and standards. Consequently, interoperability is considered an elementary necessary feature of Grid workflows.

Karajan provides user directed and global fault tolerance. Through user directed fault tolerance, special library elements can be employed to ignore faults, restart faulting blocks, trap faults and provide individual actions, or specify dynamically scoped error handlers. At the global level timed or program-directed checkpointing can be used.

One of the important differences to other workflow frameworks is that Karajan can be extended both through parameterized user-defined workflow elements (functions) and/or by implementing new workflow elements in Java.

1.2.2 Language Design

The Karajan language is a declarative style language, but with strict evaluation.

Variables can be used in Karajan, but the scoping rules restrict the possibility of concurrent destructive updates. Each execution element, which is similar to a function, has its own scope, both for its arguments and its body. Variables defined in parent scopes can be read if they fall within the same lexical scope, but not written to. Attempting to write to a variable will create a new binding for that variable which will shadow a variable with the same name in any parent scope. Furthermore, parallel elements will create separate scopes for each concurrent thread.

Iteration elements can be used to execute a given set of elements sequentially or in parallel for a given set of items. Iterations can be used for both actions and data. From a data centered perspective, iterations are equivalent to multiple issues of the same parameterized data. Unrolling iterations manually while consecutively replacing the iteration variable with the respective values produces the same result as using iterations. The direct consequence is that Karajan elements support multiple return values by default.

Karajan supports parameterized abstractions, similar to function definitions. However, Karajan provides extended functionality in terms of concurrency for workflow element definitions. Besides strict evaluation in which all arguments are evaluated before the element is invoked, it is possible to define workflow elements which evaluate the arguments in parallel with the body of the definition. If an argument that is needed by the body thread is not yet evaluated, the body thread suspends execution waiting for the argument thread to evaluate the particular argument. The parallel element evaluation can achieve a result similar to the use of generators in other languages. Nonetheless, generators require special semantics to be used in the definition of generators, while the Karajan parallel element allows any other function to be used as a generator (in part due to the multiple return values which are natural in Karajan).

Data flow equivalence is provided between sequential elements and their parallel counterparts. Values are returned in lexical order, regardless of the order in which they are evaluated. It is however also possible to use versions of the parallel primitives which return values in the exact order in which they are evaluated.

Due to their nature and structure, the parallel composition elements in Karajan provide and promote locality with respect to concurrency. Combined with the recursive structure of the language this allows for concurrent threads to be expressed declaratively. Concurrency thus becomes an aspect of the program rather than a separate entity.

A number of other helpful concurrent constructs are also available. Futures can be used to represent the result of a computation that is not yet available. Syntactically identical to variables, futures will cause the execution of the thread that attempts to access their value to be suspended until the value is available. Channels can be used as future lists. Similar to futures, values in channels can be accessed as they become available, otherwise causing the accessing thread to be suspended.

1.2.3 Execution Engine

The execution engine supports lightweight threading, which provides concurrency scalability, with less impact on resources than native threads. The engine does not excel in terms of performance. Nonetheless, the overall impact on Grid workflows is little, since the limitations are caused mostly by the security libraries, most of the overhead CPU time being spent during authentication, data encryption/decryption, and signature verification in the Grid libraries.

The lightweight threading engine, given the same resources, allows somewhere in the range of 2 orders of magnitude more concurrent threads when compared with the use of native threading. The engine uses a mix of cooperative and preemptive multi-threading. By default, a small number of worker threads are started and used to execute workflows. When the existing worker threads become blocked executing lengthy operations, new threads are progressively added up to a certain limit, in order to keep the latency low. The cooperative threading relies on the asyncrhoneous implementation of time-consuming operations, in particular the Grid tasks.

The language specification provides a recursive definition of a set of transformations on data and actions (side-effects) - workflow elements. The execution of a Karajan workflow consists of the execution of a root element which receives an initial data environment. The root element in turn executes all its sub-elements, which will recursively execute their sub-elements and so on. Elements do not have an internal state. The

state is maintained in the data environment which is continuously moved around and transformed. Elements can create private spaces in the data environment, which are maintained for as long as an element and its sub-elements complete execution. Parallel execution is accompanied by the creation of new environments. Since the new environments only share data that is private to parent elements, the concurrent execution of elements cannot cause concurrent destructive writing into the environment. This ensures the consistency of low-level program data when concurrent execution is involved.

The execution engine also allows the state to be checkpointed, either automatically at pre-configured time intervals, or manually at specific points in the workflow. The state of a workflow consists of all the currently executing workflow elements and the data environment on which they are executing. This information alone is sufficient to restore the execution at a later time, if severe failures occur (loss of power, hardware failure, etc.)

1.2.4 Task Library

In Karajan the task library provides the main means of interfacing with Grid middleware. The task library is built on top of the Java CoG Kit Abstractions API. As the scope of the Abstractions API was discussed earlier, we will focus on the functionality provided by the task library.

The binding of tasks to resources and protocols can be done either explicitly or delegated to a scheduler. Sometimes it is necessary to separate tasks which can be executed on any resources that provide certain capabilities from tasks which must be executed on specific resources. Therefore a mix of explicit and scheduled tasks can also be employed. For example, a workflow may involve fetching data from a given resource, process it as quickly as possible on all available computation resources, then move the resulting data on another pre-defined resource. Using purely abstract or purely concrete workflows may prevent the ability to express such a workflow.

Schedulers in the task library can be configured with a set of accepted protocols and a set of resources (although schedulers that dynamically fetch resource information are also possible, but not currently implemented). Tasks that are then not explicitly bound to a specific resource or protocol are forwarded to the scheduler, which assigns them resources and protocols based on policies. An unbound (abstract) task is composed from a type ($type$) and a specification ($spec$): $T_u = (type, spec)$. The type describes the type of task: execution, file transfer, information query[3], or file operation. A bound task is a task associated with a resource (r) and a protocol (p): $T_b = (type, spec, r, p)$. Resources can support zero or more protocols for each service type: $r = \{s | s = (type, p)\}$. Assuming that only one service of the same type and protocol exists for every resource, then the pair $(type, r, p)$ uniquely identifies a service for a resource. Consequently, given a bound task $(type, spec, r, p)$ and a resource r defined above, the task is unambiguously defined. The duty of the task scheduler is to maintain load information for resources and produce bound tasks from unbound tasks: $S : R, (type, spec) \rightarrow (type, spec, r, p)$, where R is the set of all resources available to the scheduler.

Additionally, there might exist the need to group several tasks on the same resource. A mechanism exist in the task library to request the scheduler to supply a virtual resource which can be used to partially bind tasks for the purpose of indicating that they must be scheduled on the same resource. The virtual resource allocations can be nested if grouping on more than one resource is needed at the same time. As an example, suppose that a job must be executed somewhere, and two resulting files must be transferred on another, unique, machine. Without grouping, there would be no guarantee that the transfers would have the source files on the same machine as the one the job is executed on, nor that the transfers would have the same machine as destination.

The task library, through the Java CoG Kit Abstractions API uses an asynchronous task execution mechanism, which minimizes the number of resources (threads) created by the execution of tasks. Combined with the lightweight threading engine of Karajan, it allows for higher scalability than synchronous, native thread based implementations.

[3] Information queries are not implemented at this time

1.2.5 The Service

The Karajan service is developed to accomplish the task of writing distributed Karajan workflows. It is still in development.

The service works together with a *remote* library element in order to provide a mechanism through which parts of Karajan programs can be detached from the current interpreter and sent for execution to the service, while preserving (most) of the semantics of the language.

Built around a flexible communication layer, the service allows configuration of the mode in which remote invocations are handled. It is possible to configure, on a host or domain basis, whether persistent connections, callback connections, or polling is to be used. Such configuration is intended to allow control between performance and resource usage, and not the least, the ability to use the service from behind a firewall. The current implementation is built on top of a GSI/SSL transport mechanism, allowing GSI authentication and data privacy/encryption.

Two major modes of operation are supported:

Personal. In personal mode the service is bound to a unique GSI identity. Once authenticated, a user has unrestricted access to all Karajan libraries.

Shared. The shared mode is designed for multiple users. Authorization is done using Globus gridmap files (a simple form of an access control list). Tight security restrictions are placed on various aspect of the workflow. Only authorized data types are permitted, and certain library functions are not available, in order to prevent the possibility of privilege escalation. In shared mode, a special local provider can be used enabling grid-mapped job submission and possibly other operations.

The service allows the use of both remote and local libraries and abstractions. The use of local libraries enables a workflow to re-use libraries that are not part of the service distribution, while the use of remote libraries may allow system-customized interfaces to local resources. With remote libraries system administrators can implement system-dependent functionality and expose a common interface, allowing workflows to be written in a configuration independent fashion.

1.2.6 Examples

In Figure 1.5 we present a simple workflow which concurrently executes two jobs and transfers their output to the client machine. It makes use of the scheduling capabilities of Karajan. The `<parallel>` element executes its sub-elements (in this case the two `<allocateHost>` elements) in parallel and waits for their completion. The `<allocateHost>` element allows the grouping of tasks on a single host, represented by variables `one` and `two`, respectively. It executes its sub-elements in sequential order. The `<task:execute>` and `<task:transfer>` elements interface with the CoG Kit Abstraction library in order to provide job submission and file transfer capabilities. The duty of finding the appropriate services for submitting the execution and transfer requests is left to the scheduler.

The scheduler and resource definition file is presented in Figure 1.6. In this particular case, the resources used are composed of two hosts, each with an execution and a file transfer service.

1.2.7 Repository

The Workflow component repository [9] is a service used to store, retrieve, and search for components that can be integrated into a Java CoG Kit workflow. The repository service promotes reusability of components that can either be maintained by an individual researcher, or by a shared community of peers with similar interests.

The aim in designing a workflow repository was to dynamically include workflow components or provide the ability to modify the components while a workflow is in progress. Remote access to the repository is also an important consideration in order to utilize the components of the workflow system in a collaborative environment by providing remote workflow component storage and access to distributed group members.

```
<karajan>
    <import file="cogkit.xml"/>
    <import file="scheduler.xml"/>
    <parallel>
         <allocateHost name="one">
            <task:execute  executable="/bin/example"
                           stdout="example1.out" host="{one}"/>
            <task:transfer srchost="{one}" srcfile="example1.out"
                           desthost="localhost"/>
         </allocateHost>
         <allocateHost name="two">
             <task:execute  executable="/bin/example"
                            stdout="example1.out" host="{two}"/>
             <task:transfer srchost="{two}" srcfile="example1.out"
                            desthost="localhost"/>
         </allocateHost>
    </parallel>
</karajan>
```

Fig. 1.5. A simple workflow that uses a simple scheduler defined in Fig. 1.6

```
<karajan>
    <scheduler type="default">
        <resources>
            <host name="host1.example.org">
                <service type="execution" provider="gt2"
                         uri="host1.example.org"/>
                <service type="file-transfer" provider="gsiftp"
                         uri="host1.example.org"/>
            </host>
            <host name="host2.example.org">
                <service type="execution" provider="gt2"
                         uri="host2.example.org"/>
                <service type="file-transfer" provider="gsiftp"
                         uri="host2.example.org"/>
            </host>
        </resources>
    </scheduler>
</karajan>
```

Fig. 1.6. A scheduler and resource definition example that is reused in Fig. 1.5

The repository sub-system is still in the first stage of development and does only provide indispensable features. It enables persistence for workflows that are executed by storing them either at a local, embedded repository, or a remote repository based on the users preference. The components within the repository have metadata associated with them. Versioning and timestamps of a workflow component can be used to distinguish between the components modified over time. Independent user groups may create and maintain their own repositories which contain components with related information. However, when these groups pool their resources with groups from other domains of science, categories or namespaces are used for distinction. Provenance information for components will in future guide the selection of components.

The Java CoG Kit Karajan workflow framework allows the dynamic inclusion of workflow components through the use of an include statement. The include statement fetches the component from the repository and evaluates the contents at runtime. Components include a number of attributes and are defined through a simple XML specification. These attributes are name, short description, description, license, author, code,

signature, version, date entered, and date modified. Additionally it is possible to add user defined attributes. The predefined attributes allow provenance of the component information.

The repository architecture and design follows that of the abstraction and provider model promoted within the Java CoG Kit. Hence it is possible to Use a variety of data stores to host such a repository by developing different providers. Sharing of the repository can be enabled by starting up a network server. In this mode, the repository can operate as centralized component share system for a group of users.

We chose to implement a provider for relational databases, specifically based on Apache DERBY[1]. A provider based on a relational data store has the advantage of well defined transaction management, regular backup of data, built-in server for remote access, user management, and the possibility of replication of the component data in a distributed database. It is foreseeable that providers for other implementations include Object-oriented databases and XML databases could be developed. One may ask why is it not enough to provide one solution? The answer is based on the ease of deployment of such a service in a wide variety of user communities. Setting up and maintaining a relational database or a WebDAV server is typically more difficult than simply providing a flat file system. As our tool serves also as middleware we want to give Grid architects the freedom to choose or develop a provider that plugs into the overall architectural design of their Grid application. It also makes it possible to address scalability issues.

The repository provides the obvious functionality to its users: loading, saving, listing, searching and retrieving them in the form of an XML file along with the metadata. Other functions involve defintion of new attributes (metadata) for the components, removal of attributes for the components, listing current attributes and user management for the repository. Besides the access provided to the repository API a command line tool exists to interface with the repository in a straight forward fashion through a UNIX like command.

The example shown in FigureF:workflow-example shows how to integrate a predefined workflow component called "gaussian" into a karajan workflow. Here, the program "gaussian" which is used to compute thermochemical values in the chemistry domain is internally called using the element here to invoke the command on a remote compute server. To do so, an input file is transferred to the remote server and executed and the output is copied back to the local system.

There are a number of predefined elements that can be used to access the repository via the workflow. These constitute the base repository library and are stored in "repository.xml". Once this file is included in a workflow, we can use the appropriate repository functions that have been defined in the library which in turn call the repository API using Java elements provided by Karajan. One such is "repository:get" that retrieves a component from the repository located at "dblocation". The "provider" is a local embedded database that is created and maintained using ApacheDerby and the component is stored in the file as indiacated by "filename".

1.3 Workflow support for Experiment Management

The Java CoG Kit group has also prototyped a tool that introduces a framework for experiment management which simplifies the user's interaction with Grid environments. We have developed a service that allows the individual scientist to manage a large number of tasks as typically found in experiment management. Our service includes the ability to conduct application state notifications. Similar to the definition of standard output and standard error, we have defined standard status that allows us to conduct application status notifications. We have tested our tool with a large number of long running experiments, and shown its usability [12].

1.4 Conclusion

In this Chapter we introduced a subset of frameworks that provide experiment management support within the Java CoG Kit. We have focused explicitly on workflow solutions and motivated the Karajan workflow

```
<karajan>
    <include file="cogkit.xml"/>
    <include file="repository.xml"/>
    <include>
        <repository:get component="gaussianChem"
provider="local:derby"
                        dblocation="/home/Admin/repositoryext"/>
    </include>
    <task:transfer srcfile="H2O_B3SP.gjf"
                   destfile="H2O_B3SP.gjf"
                   desthost="hot.mcs.anl.gov">
    <gaussian inputFile="H2O_B3SP.gjf"
              nodes=2
              checkpoint file="H2O.chk"
              host="hot.mcs.anl.gov">
    <task:transfer srcfile="H2O_B3SP.log" srchost="hot.mcs.anl.gov"
                   destfile="gaussian.out">
</karajan>
```

Fig. 1.7. An example that demonstrates the inclusion of an element called *gaussian* that is defined in a workflow repository.

framework. The framework can be used to specify workflows through a sophisticated XML scripting language as well as an equivalent more user friendly language that we termed K. In contrast to other systems we not only support hierarchical workflows based on DAGs but have also the ability to use control structures such as if, while, and parallel in order to express easy concurrency. The language itself is extensible through defining elements and through simple data structures allows the easy specification of parameter studies. The workflows can be visualized through our simple visualization engine which also allows to monitor state changes of the workflow in real time. The workflows can actually be modified during runtime thorugh two mechanisms. First, through the definition of elements that can be deposited in a workflow repository which gets called during runtime. Second through the specification of schedulers that allows the dynamic association of resources to tasks. The execution of the workflows can either be conducted through the instantiation of a workflow on the users client, or it can be executed in behalf of the user on a service. This service will in future also allow a more distributed execution model with loosely coupled networks of workflow engines. Hence, the execution of independent workflows acting as agents for users will be one of our focus areas. Furthermore we will extend our workflow vizualizer to integrate components stored in the repository to enable a dynamically extensible workflow composition tool. We have however put great emphasize on the fact that the workflow can also be started form the command line and we will provide in future enhancements. At present we have demonstrated with our tool that we can successfully start hundred thousands of jobs due to our scalability oriented threading mechanisms part of the Karajan core engine. We will be using this engine to do modeling of the management in urban water distribution systems [10].

References

1. Apache Derby. Web Page. Available from: `http://db.apache.org/derby/`.
2. Gregor von Laszewski. An Interactive Parallel Programming Environment Applied in Atmospheric Science. In G.-R. Hoffman and N. Kreitz, editors, *Making Its Mark, Proceedings of the 6th Workshop on the Use of Parallel Processors in Meteorology*, pages 311–325, Reading, UK, 2-6 December 1996. European Centre for Medium Weather Forecast, World Scientific. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--ecwmf-interactive.pdf`.
3. Gregor von Laszewski. Java CoG Kit Workflow Concepts. *accepted for publication in Journal of Grid Computing*, 2006. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-workflow-taylor-anl.pdf`.

4. Gregor von Laszewski. The Grid-Idea and Its Evolution. *Information Technology.*, accepted for publication. Argonne National Laboratory, Argonne, IL 60439, U.S.A. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-grid-idea.pdf`.
5. Gregor von Laszewski, Kaizar Amin, Shawn Hampton, and Sandeep Nijsure. GridAnt – White Paper. Technical report, Argonne National Laboratory, 31 July 2002. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-gridant.pdf`.
6. Gregor von Laszewski, Steve Fitzgerald, Ian Foster, Carl Kesselman, Warren Smith, and Steve Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, pages 365–375, Portland, OR, 5-8 August 1997. Available from: `http://www.mcs.anl.gov/~gregor/papers/fitzgerald--hpdc97-mds.pdf`.
7. Gregor von Laszewski, Ian Foster, Jarek Gawor, Warren Smith, and Steve Tuecke. CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In *ACM Java Grande 2000 Conference*, pages 97–106, San Francisco, CA, 3-5 June 2000. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--cog-final.pdf`.
8. Gregor von Laszewski and Mike Hategan. Grid Workflow - An Integrated Approach. In *Technical Report.*, 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`.
9. Gregor von Laszewski and Deepti Kodeboyina. A Repository Service for Grid Workflow Components. In *International Conference on Autonomic and Autonomous Systems International Conference on Networking and Services*. IEEE, 23-28 October 2005. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-workflow-repository.pdf`.
10. Gregor von Laszewski, Kumar Mahinthakumar, Ranji Ranjithan, Downey Brill, Jim Uber, Ken Harrison, Sarat Sreepathi, and Emily Zechman. An Adaptive Cyberinfrastructure for Threat Management in Urban Water Distribution Systems. Technical report, Argonne National Laboratory, January 2006. To be submitted. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-water-iccs.pdf`.
11. Gregor von Laszewski, Mei-Hui Su, Joseph A. Insley, Ian Foster, John Bresnahan, Carl Kesselman, Marcus Thiebaux, Mark L. Rivers, Steve Wang, Brian Tieman, and Ian McNulty. Real-Time Analysis, Visualization, and Steering of Microtomography Experiments at Photon Sources. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, 22-24 March 1999. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--siamCmt99.pdf`.
12. Gregor von Laszewski, Tan Trieu, Phillip Zimny, and David Angulo. The Java CoG Kit Experiment Manager. Technical report, Argonne National Laboratory, June 2005. Available from: `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski-exp.pdf`.