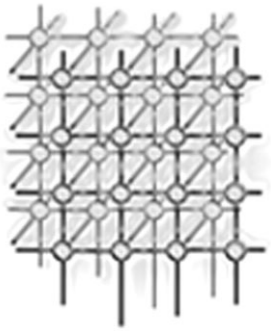
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2006; **18**:1293–1316
Published online 13 December 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.991

# GAUGE: Grid Automation and Generative Environment‡



Francisco Hernández*,†, Purushotham Bangalore, Jeff Gray, Zhijie Guan and Kevin Reilly

*Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, U.S.A.*

**SUMMARY**

**The Grid has proven to be a successful paradigm for distributed computing. However, constructing applications that exploit all the benefits that the Grid offers is still not optimal for both inexperienced and experienced users. Recent approaches to solving this problem employ a high-level abstract layer to ease the construction of applications for different Grid environments. These approaches help facilitate construction of Grid applications, but they are still tied to specific programming languages or platforms. A new approach is presented in this paper that uses concepts of domain-specific modeling (DSM) to build a high-level abstract layer. With this DSM-based abstract layer, the users are able to create Grid applications without knowledge of specific programming languages or being bound to specific Grid platforms. An additional benefit of DSM provides the capability to generate software artifacts for various Grid environments. This paper presents the Grid Automation and Generative Environment (GAUGE). The goal of GAUGE is to automate the generation of Grid applications to allow inexperienced users to exploit the Grid fully. At the same time, GAUGE provides an open framework in which experienced users can build upon and extend to tailor their applications to particular Grid environments or specific platforms. GAUGE employs domain-specific modeling techniques to accomplish this challenging task. Copyright © 2005 John Wiley & Sons, Ltd.**

KEY WORDS: Grid workflow; domain-specific modeling; Grid computing; visual languages

## 1. INTRODUCTION

The Grid has proven to be a successful paradigm for distributed computing [1,2]. It provides dependable, collaborative, and secure access to remote computational, data, and instrumentation resources. Different Grid frameworks have emerged to provide support for this model of computation (e.g., Globus [3], Legion [4], and Unicore [5]). The Globus Toolkit [3] is perhaps the most widely

*Correspondence to: Francisco Hernández, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL 35294-1170, U.S.A.
†E-mail: hernandf@cis.uab.edu
‡Project Web site http://www.uab.cis.edu/hernandf/projects/gauge.



*Received 1 June 2004*
*Revised 16 March 2005*
*Accepted 27 April 2005*

used framework to construct Grid applications. Globus provides a common middleware that considers resources as entities of a virtual organization. The middleware is formed by different components that provide services to integrate distributed resources in a Grid computing environment. Creating an application that uses Globus requires the composition of several of these components. Unfortunately, constructing Grid applications is still difficult despite all the capabilities that Globus provides [6]. The Java Commodity Grid Toolkit (Java CoG Kit) [7] was later created to assist in the development of applications using Globus services. This was a step towards simplifying the construction of applications for the Globus Toolkit. The Java CoG Kit helps a user navigate the intricacies of the Globus components more easily by introducing a new programming model for the Grid. Furthermore, the Java CoG Kit provides many utility components that enhance the functionality of Globus.

However, developing applications for the Grid remains difficult for many users. This is due in part to the complexity of the back-end systems, where even potentially inexperienced users are exposed to all the details of the underlying Grid technologies [8]. Traditionally, problem solving environments (PSE), or portals [9], have been developed to ease the construction of Grid applications for the un-savvy Grid user. PSEs provide a high-level view for specifying Grid-enabled applications and rely on middleware to connect with the Grid component resources [10]. This kind of tool expedites simple tasks (i.e. simple job submissions, and checking the status of a previously submitted job), but it lacks the flexibility to define a complex sequence of tasks. Thus, an inexperienced user is not equipped with the capabilities for exploiting the Grid without having to decipher the intricacies of the Grid technologies. In such cases, users can become overwhelmed by the numerous low-level details involved in engineering a Grid application. By raising the level of abstraction, the user only needs to know the applications and the resources they want to use in order to access the Grid.

Experienced Grid users (developers) also struggle when developing Grid applications. We observe two principal reasons for this problem: (1) current software engineering practices (e.g., reusability, modeling, and rapid prototyping) have not been fully explored for the Grid model; and (2) the Grid applications usually consist of different back-end implementations, but there is no standard for the application development process [11]. Thus, a developer has to write application-specific code for each back-end. A solution to improve this kind of development is to introduce a high-level abstraction layer for hiding the complex back-ends and provide functionality for different Grid architectures [11–13]. Yet, it can be observed that performing the abstraction at a programming-language level is still not optimal, and that the two problems enumerated above can be lessened if the abstraction is realized at the *domain level*. To work at the domain level, we use the concepts of domain-specific modeling. Domain-specific modeling (DSM) [14] enables the user to employ familiar concepts in the Grid domain to construct models of Grid applications. These models can then be synthesized (i.e. translated) into different representations (e.g., XML configuration files or source code). The benefit of working at a model level is that the models can be manipulated as first-class development artifacts, which means that work with them can be automated [15]. With this technology, a user focuses on higher levels of abstraction at the problem space and is able to avoid low-level details, such as low-level Grid middleware and their usage.

This paper presents the Grid Automation and Generative Environment (GAUGE). GAUGE aims to enable inexperienced users to take full advantage of the Grid infrastructure and at the same time assist developers in modeling and prototyping Grid applications. Because the use of Globus and the Java CoG Kit has proven to be a successful combination, GAUGE relies on this combination for interfacing with Grid infrastructure. The approach employed by GAUGE provides a high-level abstract layer for the

construction of Grid applications. This layer is composed of visual models that are constructed using concepts of domain-specific modeling. Programs that manage the application execution are generated from the corresponding visual models. Thus, users need not learn how to use the Globus Toolkit or the Java CoG Kit in order to develop Grid-enabled applications. GAUGE also provides an open framework in which developers can reuse components and tailor the automation environment to work with a particular Grid architecture.

The remainder of this paper is organized as follows. Section 2 provides an overview of GAUGE. Section 3 outlines the requirements considered when developing the automation environment. The architecture of GAUGE is introduced in Section 4. Section 5 presents the implementation of GAUGE, including the construction of the user interface layer, the domain transformation layer, the code generation layer, and an illustrative example. Section 6 presents related work, and Section 7 outlines future directions for research in this area. Finally, Section 8 gives conclusions of our work.

## 2. GAUGE TOOL

A *Grid application* is an executable software artifact that is suited to work in a Grid environment. Under current practice, a Grid application is created by composing different sub-tasks that generate an execution flow. Ideally, these sub-tasks would have a degree of Grid awareness. However, in practice, most sub-tasks are originally designed for executing on dedicated supercomputers or parallel computers with no attention to Grid computing. Thus, it is necessary to move them to a Grid environment [16]. Besides the execution flow, the (Grid) applications also require specification of the data flow needed to accomplish the execution flow. The data flow is required because it establishes pre- and post-conditions for task execution [17]. The combination of both flows defines a *workflow*. The complexity of building a workflow system for the Grid arises from the ability of each of the constituent tasks to use different resources. Thus, an in-depth understanding of the underlying environment (hardware and software) is typically needed to define a Grid workflow. The Grid application consists of the control code that executes tasks on different resources (execution flow), and also moves and copies files between those resources (data flow).

GAUGE is a tool that automates the development of *Grid applications*. The goal of GAUGE is to automate the generation of the control code that manages a particular Grid application in a manner that can function in any system that uses a Grid framework, such as the *Globus Toolkit* [3]. Current realization of GAUGE supports the following Grid tasks: job submissions, and explicit file transfers. An Open Grid Service Architecture (OGSA) [18] compliant Grid services implementation is under development.

Two types of users are considered: operators and developers. Operators are knowledgeable about the domain of a specific application's tasks, as well as the manner in which the relevant tasks are combined to form the final Grid application. However, operators typically are not familiar with the underlying Grid technologies. In order for them to fully utilize the Grid, complex programs often must be developed to manage the control of the applications. Developers, the other type of users, are assumed to be Grid savvy. A developer possesses knowledge about the intricacies of the underlying Grid technologies, but often lacks ability in handling the specifics of the different application constituents for a specific domain. Maximizing the use of the Grid infrastructure requires a synergistic coordination between operator and developer roles. One of the objectives of GAUGE is to facilitate the interaction between these two types of users. GAUGE focuses on the domain experts (operators) and helps them

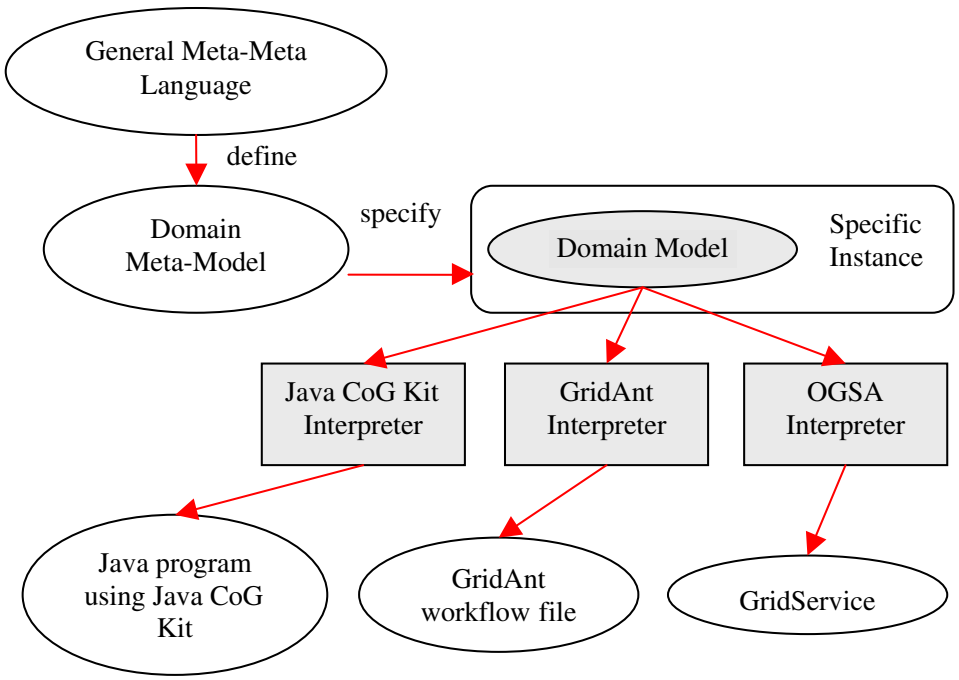



Figure 1. Domain-specific modeling overview. A meta-model is defined using a general meta-meta model language. The meta-models specify the class of models that can be constructed. Different interpreters generate different output representations from the same models.

by providing an environment in which they can graphically specify the workflow for their application and automatically generate the code that manages the execution of the workflow (execution and data flow). For the developers, this tool provides an open framework in which they can reuse components (e.g., the user interface and the domain transformation layer), and tailor the automation environment to work within a particular Grid environment.

Interaction with the system is realized through a graphical user interface that is provided by the Generic Modeling Environment (GME) [19]. GME is a domain-specific modeling environment that can be configured and adapted from meta-level specifications that describe the domain [20]. The domain-specific models are developed by first creating a meta-model that specifies the syntax and static semantics of a visual language; the dynamic semantics are introduced by an interpreter that synthesizes the models into different representations [20] as shown in Figure 1. By using a general meta-meta modeling language, in our case provided by GME, a domain meta-model is defined. The meta-model specifies a visual language that can be used to create specific instances of models. Users interact with these models and are able to synthesize code by using a model interpreter. Different output representations can be generated from the same domain models. For example, in Figure 1 a first interpreter (lower left box) generates a Java program that utilizes the Java CoG Kit

to control the execution of the workflow. A second interpreter (lower-middle box) generates a specification file for Grid Ant [21]. Finally, the third interpreter (lower-right box) generates a Grid Service specification [18]. In this manner, the level of abstraction is raised because users operate at a model level instead of a programming language level.

The Globus Domain Model (GDM) [6,22] is the visual language that configures and adapts GME to function in the Grid domain. The visual language shields the underlying complexity from the user by incorporating concepts from the Grid domain, and associates those concepts with graphical elements (GDM constructs).

In addition to the Grid domain constructs, GDM supports a set of constructs that allows the specification of workflows from simple tasks that can be defined within GAUGE. According to [17], functionality of workflow systems can be expressed by a set of recurring abstractions in the specification of workflows (*workflow patterns*). These abstractions are based on a set of routing primitives [23] that control the workflow's execution: sequence, splits, joins, and iteration. GDM supports a basic set of control routing primitives for implementing the workflow's functionality: implicit AND-Split, implicit AND-Join, explicit XOR-Split, and explicit XOR-Join. With these basic constructs, the user controls the execution flow of the workflows, and is also able to define basic workflow patterns [24] such as: sequence, parallel split, synchronization, exclusive choice, simple merge, and arbitrary cycles. This set of patterns is important because it creates a workflow language capable of executing tasks in sequence and in parallel, executing a task an arbitrary number of times, and steering the execution flow according to particular conditions encountered during the execution of the workflow. All of these capabilities are required [21] when implementing a Grid workflow. The manner in which GDM is constructed is explained in detail in Section 5.1.

The user's interaction with GDM generates graphical models of the workflows that visually express the workflow's tasks, and the execution and data flow between those tasks. GME provides an API that traverses the internal representation of the models. A model interpreter uses this API to translate the models into an application that manages the execution of the workflow. This is a typical example of software translation because a new artifact is created from a description at a different level of abstraction [25]. A key benefit of model-driven techniques is that different output representations can be generated based on the same model properties [19] (as seen on Figure 1). Currently, GAUGE is able to generate (1) a Java application that interfaces with Globus using the Java CoG Kit as well as (2) a Grid Ant specification file. GAUGE can also partially generate an initial OGSA Grid service representation that must be submitted to a Grid services environment for its execution.

Although the user interface of GAUGE is aimed at users who are not experts in the different Grid technologies, developers can also benefit by reusing GAUGE's infrastructure. In this manner, applications for different Grid environments can be generated from the same graphical models. This reduces the time spent in developing Grid architecture prototypes, allowing developers to verify that a specific Grid architecture is feasible before investing resources in developing it for production purposes.

## 3. REQUIREMENTS

Two requirements guided the design of GAUGE: (1) the simplicity in the specification of the workflows; and (2) the generality in the specifics of the workflow tasks. The challenge of the

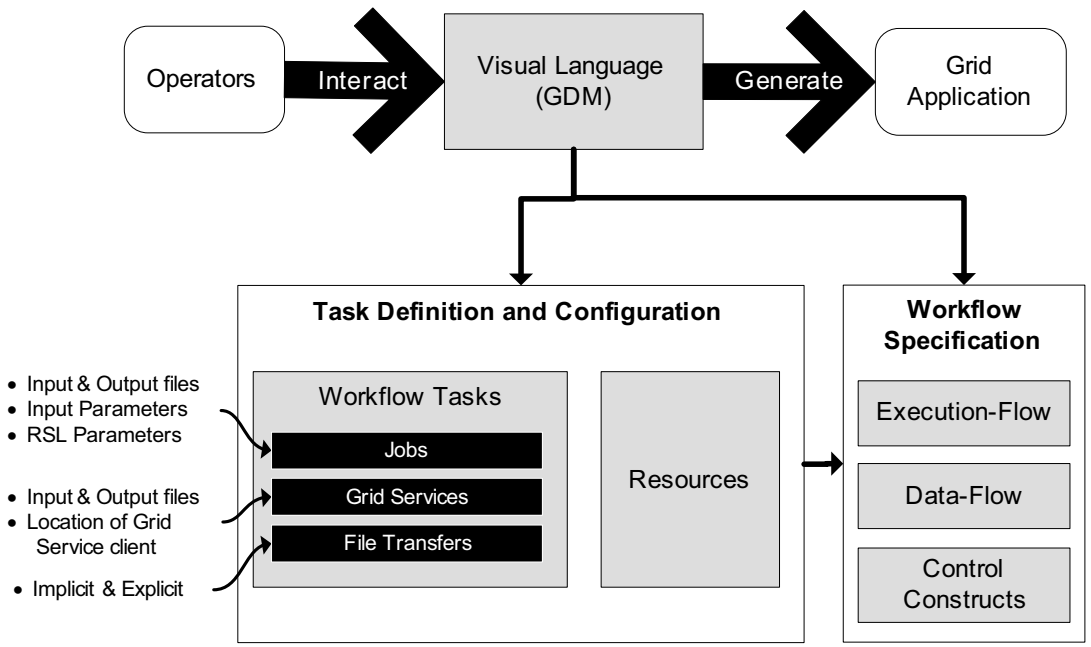



Figure 2. Function of the GAUGE modeling process. The operators interact with a visual language that provides definition and configuration of tasks, and specification of workflows, from which the code that forms the Grid application is generated.

first requirement is hiding the underlying complexity (i.e. heterogeneous and distributed back-end resources, Grid technologies) from the specification of the workflows. The solution employed is to use concepts that are familiar to the users (i.e. location of hosts, job specifications, file transfers) and associate those concepts with graphical elements. This association defines a visual language. Thus, users graphically combine these elements and generate the corresponding workflows. The second requirement permits the creation of a general tool that is not tied to a specific Grid environment. To fulfill this requirement, the workflow tasks are considered external with respect to GAUGE.

The function of the GAUGE modeling process is presented in Figure 2. Operators interact with the visual language in two ways: (1) defining and configuring the workflow tasks; and (2) specifying the execution flow that generates the workflow. Defining and configuring tasks involves specifying the resources (hardware) that are used for the tasks. Communication between tasks is conducted via files, and each job and Grid service definition contains a list of the required input/output files. Jobs require basic input parameters (e.g., executable, directory, standard output), but the complete specification of the Resource Specification Language (RSL) [26] is also permitted. In addition to the list of input and output files, Grid services require the path to the client that controls the execution of the Grid service. File transfers can be implicit or explicit, but the operator can only define the explicit file transfers. The execution flow defines the workflow by specifying the order of the tasks. The data flow is based

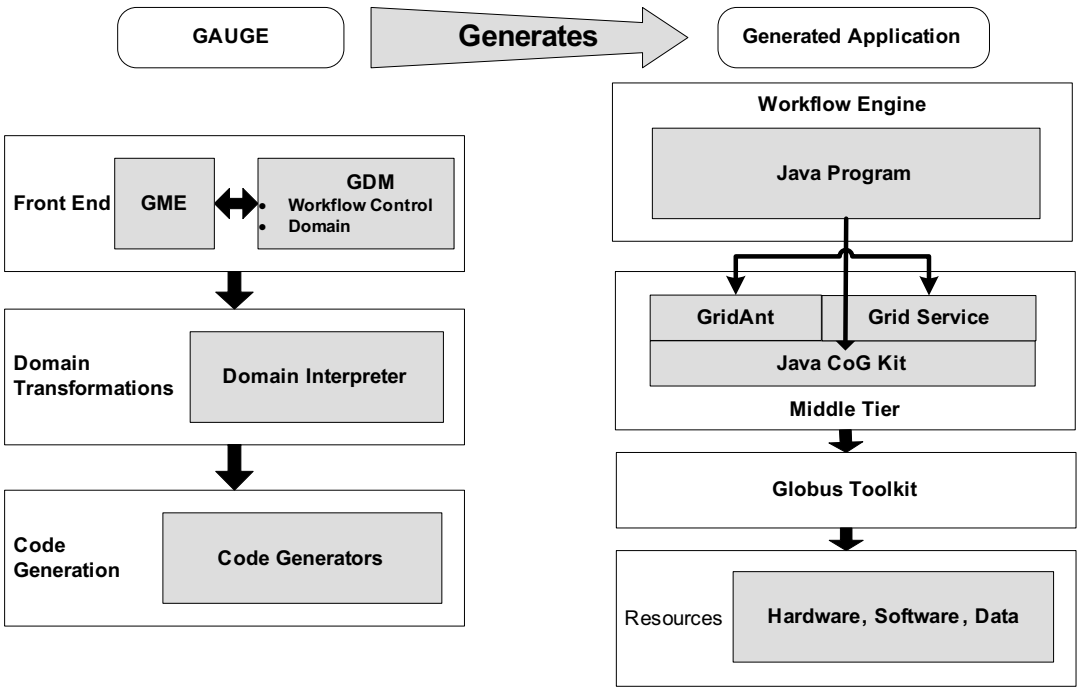



Figure 3. Architecture of the modeling environment. The left-hand side of the figure presents the architecture for GAUGE. The right-hand side presents the organization of the programs generated by GAUGE.

on the execution flow, because it is guided by the list of input and output files that are defined for each workflow task. The data flow is implemented by moving files between tasks, which require adding implicit file transfers to the execution flow. Control flow constructs (XOR-split, AND-split, XOR-join, AND-join) are also required when specifying a workflow. With these basic constructs, the user is able to execute tasks in sequence and in parallel, execute a task an arbitrary number of times, and steer the execution flow according to particular conditions encountered during the execution of the workflow. After the workflow specifications are completed, the user is able to generate the code that manages the application's execution.

## 4. ARCHITECTURE

The left-hand side of Figure 3 shows the architecture of GAUGE. The major components consist of a front-end, a domain transformation tier, and a code-generation tier. The front-end is realized by the GME [19], which is configured by the visual language GDM. GDM uses concepts of workflow control mixed with concepts of the Grid domain. The middle tier (middle-left-hand side of the diagram)

is comprised of the domain transformations layer. This layer provides a Java API that wraps the API provided by GME to access the GDM models. The lower-left-hand side of the diagram shows the code generation layer. This layer generates code for a particular Grid environment; in our case it generates Java code that manages the execution of the workflows defined in GDM. A developer can extend GAUGE by replacing the code generation layer and by interacting with the middle layer generated code suited for a different Grid environment, or the developer can generate code written in another programming language.

The structures of the generated applications currently supported are shown on the right-hand side of Figure 3. The generated application consists of a Java program that controls and manages the correct execution of the workflows. This application performs the function of what is typically known as the workflow engine [21]. A GridAnt [21] specification file can also be generated. Work is underway to generate corresponding Grid services (conforming to the OGSA [18] specification) requests from the generated Java programs. Both types of applications interact with the back-end systems through the Java CoG Kit [7], and the Globus Toolkit [3]. Currently, we are also working on generating a Python [27] script that interacts with the Globus Toolkit using PyGlobus [28].

## 5. IMPLEMENTATION

In this section, the implementation of GAUGE is described. For the sake of clarity, the system is split into three components: user interface, domain transformation, and code generation.

### 5.1. User interface

The user interface presents an environment in which the users can graphically manage the workflows of their Grid applications. The front-end hides the intricacies of the underlying Grid environment from the end-user. Therefore, it is not required to be proficient in Grid technologies to write complex Grid applications. The graphical interface is provided by the GME, which is configured using the visual language GDM [6,22]. This combination defines the interaction of the user with the system (Section 5.1.1). Aside from the Grid domain constructs, the visual language also employs control constructs (Section 5.1.2) that are generally used to generate workflow patterns; this facilitates control of the execution of the workflow.

#### *5.1.1. GDM*

GDM is a visual language that was constructed using concepts of domain-specific modeling. Domain-specific modeling has been useful in automating different kinds of applications in which the environment is dynamic, including embedded systems [29], automotive manufacturing [30], and complex QoS applications [31].

GDM was written using the general paradigm provided by GME [19]. This general paradigm consists of meta-language elements specified in the form of a UML class diagram [32]. The domain-specific concepts are expressed as stereotypes of specific classes. A design engineer is able to define the type of models that can be constructed by specifying the relationships between the stereotypes. The three basic classes supported by GME are atom, model, and connection. Atoms are classes of

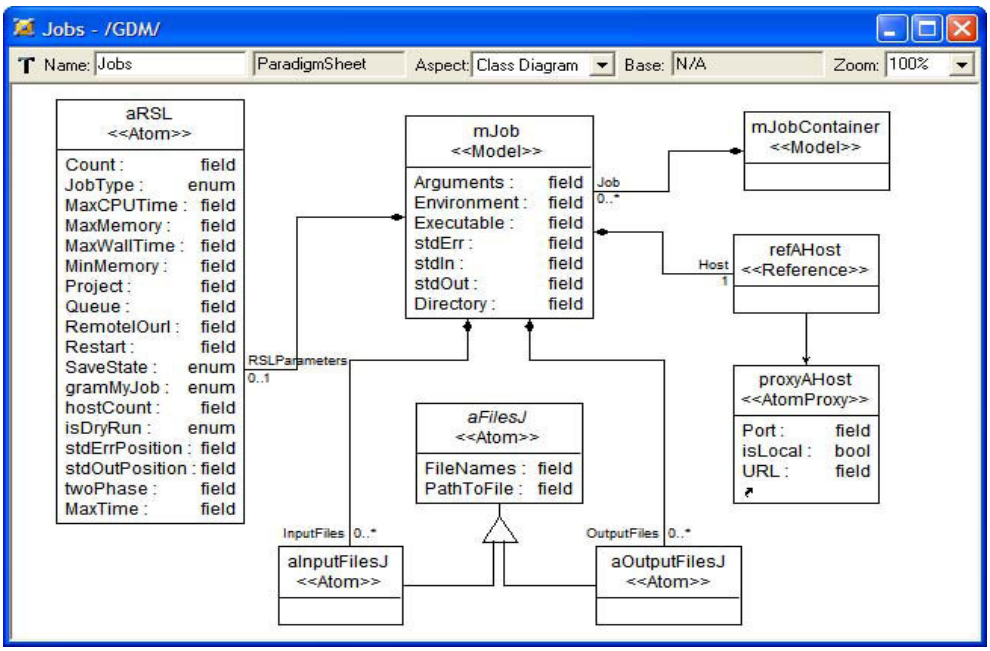



Figure 4. GDM language specification for job tasks. This figure presents the manner in which the relation of the different (GME) classes specify a construct in GDM.

objects that do not contain other objects. Models are container classes, and connections associate models and atoms.

Figure 4 illustrates the use of these basic classes that generate the job specification aspect of GDM. To fully define a job submission (see Section 3), the user should specify the machine to which the job will be submitted, input parameters and the lists of input and output files. RSL parameters are optional. The central concept in the figure is `mJob`, which represents a job submission that is defined as a model to contain the rest of the specifications. The attributes that `mJob` contains are also specified in the class diagram. These attributes represent the input parameters to be specified when defining the job tasks. `mJobContainer` represents a container for job specifications that allows the user to group jobs according to the specific preferences. `refAHost`, defined below `mJobContainer`, represents the machine for the job. Notice that `refAHost` is declared as a reference class. This means that, when declaring a job in GDM, a user must first define the host to use and then declare a reference to it. Any changes in the original host will be reflected in all its references. Because `refAHost` is defined in other parts of the meta-model, `refAHost` is associated with a Proxy class that points to the original host definition. The line between `mJob` and `refAHost` indicates that `mJob` contains `refAHost`. The cardinality of this association indicates that there must be exactly one host defined for each job.

`aRSL` permits the specification of all the RSL parameters for the job. This information is optional, as can be seen in the cardinality of the association (0..1). Finally, `mJob` also contains the list of input and output files. As typical in UML models, the triangle indicates that both kinds of lists inherit attributes and connections from a basic `aFilesJ` entity.

In addition to assisting in the construction of the visual language, GME provides an environment in which the configurations and the definitions of resources, tasks, and workflows can be persistent and shared between users. All these features make GME an ideal choice for the front-end of GAUGE.

### *5.1.2. Control constructs*

van der Aalst *et al.* [17] introduce the concept of workflow patterns to identify the functionality of workflow systems. From our observation, patterns are abstractions that keep recurring in the specification of workflows. These abstractions are based on a set of routing primitives that control the workflow's execution. GAUGE supports a set of basic control constructs that, in turn, allow the specification of workflow patterns.

GAUGE supports four constructs to manage the execution workflow. These constructs provide basic functionality: sequence, splits (XOR and AND), joins (XOR and AND), and iteration. Complex workflows can be specified through composition of these primitive functions. However, these constructs are not able to represent certain types of concurrency (e.g. deferred choice or unordered sequences [24]). In those cases, using a Petri net [33] is a more natural approach as can be seen in [34], where we developed a meta-model for a Petri net that serves as the front-end of a workflow system.

Figure 5(a) presents the manner in which the sequential construct is specified in GAUGE. The workflow example consists of a file transfer (FT), a job submission (A), and a Grid service execution (B). Each one of these three tasks requires the previous one to be complete before they can start processing. Notice the manner in which the data flow is conducted: B requires a file from A, labeled '`preproc`'. By labeling the connection, an implicit file transfer is conducted between the output files of A and the input files of B. However, there is no named connection between FT and A. The reason is that FT is an explicit file transfer, so the user is explicitly moving the input files of A. Although not shown in the figure, a task may require both types of transfers to collect its input files. This is also supported by GAUGE.

AND-Splits and AND-Joins are declared implicitly within the workflow tasks and do not require a specific construct. In AND-Splits, the sequential execution is broken into many parallel branches. The AND-Join must wait until all the branches have finished before it can begin its processing. The manner in which these constructs are enacted in GAUGE is depicted in Figure 5(b). Job A splits the execution into Job B and Job C. Grid service D waits for the completion of B and C to begin its processing. The connections named *all* (between A–C, B–D, and C–D) indicate that all the output files from the source (A, B, C) need to be transferred to the destination (C, B, D). The star in the connection between A and B indicates that there are several files to be transferred between A and B, but not all the output files from A need to be transferred.

The XOR-Split is the conditional construct supported in GAUGE. The function of this construct is to select one of several branches based on a decision. The implementation of this construct is system dependent. In some systems there is a trigger that indicates which branch to choose [24]. The challenge in implementing this construct in GAUGE is that task executions are considered external to GAUGE. Because of this, GAUGE does not have knowledge about the data involved in the execution of the task.

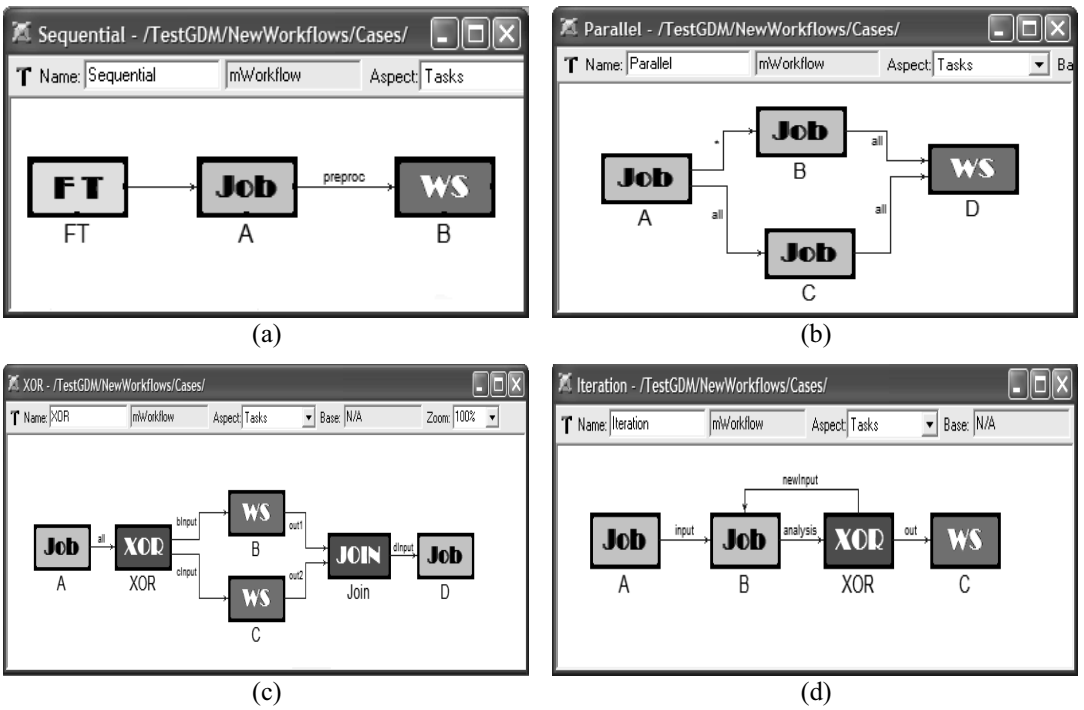



Figure 5. Control constructs supported by GAUGE. (a) Sequential execution of tasks. (b) AND-Split and Join, both B and C are executed after A. D waits for B and C to start its execution. (c) XOR-Split and Join. After completion of A, either B or C is executed (but not both). (d) B can be executed many times until the condition in the XOR routes the flow to C.

The approach taken is to ask the user to analyze the data and trigger the correct path to the workflow engine. The user does this by writing an analyzer program whose output file indicates the correct branch. However, in the case when the selection of the path depends on which path finishes its execution first, the user need only use the XOR construct without writing an analyzer program. When one of the paths finishes executing, it signals the JOIN to continue executing the workflow and forget about the other paths. Figure 5(c) shows how to use the XOR constructs in GAUGE. After the execution of Job A, the task inside the XOR construct decides if the next task to execute is Grid service B or C. After the task inside the XOR makes the decision, it signals its choice to the XOR construct which in turn starts the execution of either B or C. Finally, the XOR indicates the branch taken to the Join construct. In this manner, the Join construct knows which of the branches was taken and can start the execution of D when either B or C finishes executing.

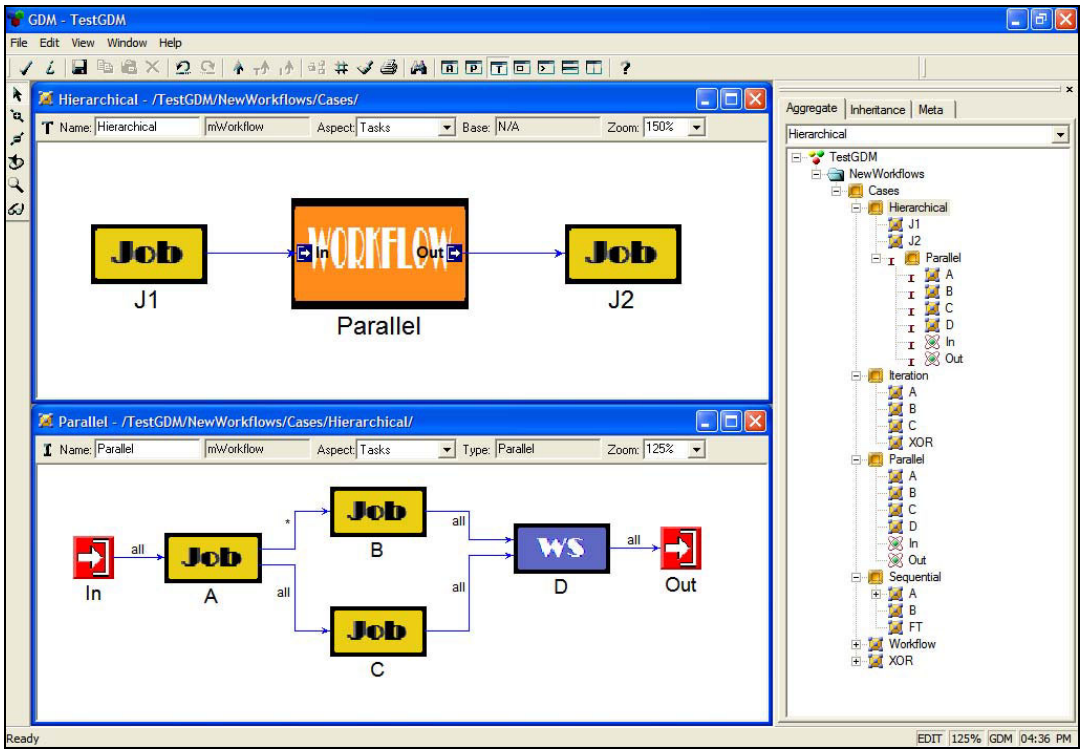



Figure 6. Inner workflow. Parallel is an inner workflow that is embedded into a sequential workflow.

Another capability of XOR is that it allows the specification of iterations within a workflow. For example, in Figure 5(d), after Job B has completed its execution, the XOR task analyzes the output and decides whether the next task to execute is Grid service C or Job B. In the case that B needs to be repeated, the XOR also provides it with a new input file (‘`newInput`’).

The last of the basic control constructs supported by GAUGE is inner workflows, which allow the user to reuse any previously defined workflows. Inner workflows are the basis for constructing hierarchical workflows. Figure 6 presents the way that inner workflows work in GAUGE. *Parallel* is defined as a normal workflow and is later used inside the sequential workflow *Hierarchical*. Any changes to Parallel are reflected in each workflow that uses Parallel as an inner workflow. This capability is based on the use of prototypes and clones, which is a feature of GME [20].

With these basic constructs, GAUGE is able to generate the following workflow patterns [24]: Sequence, Parallel Split, Synchronization, Exclusive Choice, Simple Merge, and Arbitrary Cycles. These patterns are important because they provide the capabilities required to implement a Grid workflow management system.

### 5.2. Domain transformation

As seen in Section 5.1.1, instances of models are specified by using the visual language GDM. The domain transformation layer is responsible for parsing the models created with GDM, and converting the low-level representation of the models (meta-meta language) into a higher-level representation that can be utilized by the model interpreters to understand the models. In our previous work [6], once the workflows are specified, a model interpreter traverses the internal representation of the models and generates the control code that manages the workflow execution. However, if more than one output representation is needed (e.g., Java and Python) this code is redundant and can be refactored into a different layer. Accordingly, the purpose of this layer is the facilitation of the development of the model interpreters.

In order to facilitate the interpretation of the models, GME provides an API that can be used to traverse the internal representation of the models. The GAUGE intermediate layer is implemented by providing a set of Java classes that hide the low-level API provided by GME to access the GDM models. This implementation eases the development of software artifacts for different Grid environments. Thus, a developer can reuse the graphical models and build code generators for specific Grid environments.

Figure 7 presents the UML class diagram of the GAUGE intermediate layer. This intermediate representation is presented at a higher level of abstraction than the one provided by GDM. Instead of considering models, atoms, and connections (as when the GME API is used), a developer accessing these objects would work with jobs, Grid services, and file transfers as presented by the GAUGE intermediate layer.

The manner in which the information from a GDM job is extracted can be seen in Figure 8. This figure shows part of three methods that perform this extraction (please refer to Figure 4 to see how the GDM job is defined). Given a GDM '*mJob*' model specifying a job, '*loadModel*' (lines 1–18) extracts and converts the GDM elements that are defined inside the job model: (1) the name of the model (line 5); (2) the RSL parameters that further specify the execution details of the job (lines 8–9); (3) the host to run the job (lines 12–13), and the various input and output files required for the job to execute correctly (line 16). Given a GDM '*aRSL*' atom specifying the RSL parameters for the job, '*extractRSL*' (lines 20–29) extracts the RSL parameters. The code in lines 22–27 is repeated for all the RSL parameters. Finally, given a GDM '*refAHost*' reference to previously defined host, '*extractHost*' (lines 31–43) looks for the definition of that host in the collection of previously defined hosts (line 38). Similar code is used to extract the information of the input and output files.

### 5.3. Code generation

The final layer of GAUGE generates code from the visual models. Using the intermediate layer, GAUGE accesses the models and generates the corresponding code that controls the execution of the Grid application. As presented in Section 2, GAUGE can generate a Java-based representation using the Java CoG Kit to interface with the Globus Toolkit, and a Grid Ant [21] specification file. Currently, we are working on generating two more output representations: a Python [27] script that interacts with the Globus Toolkit using PyGlobus [28], and a Grid service representation. It is worth noting that all of these output representations are based on the same Grid domain models.

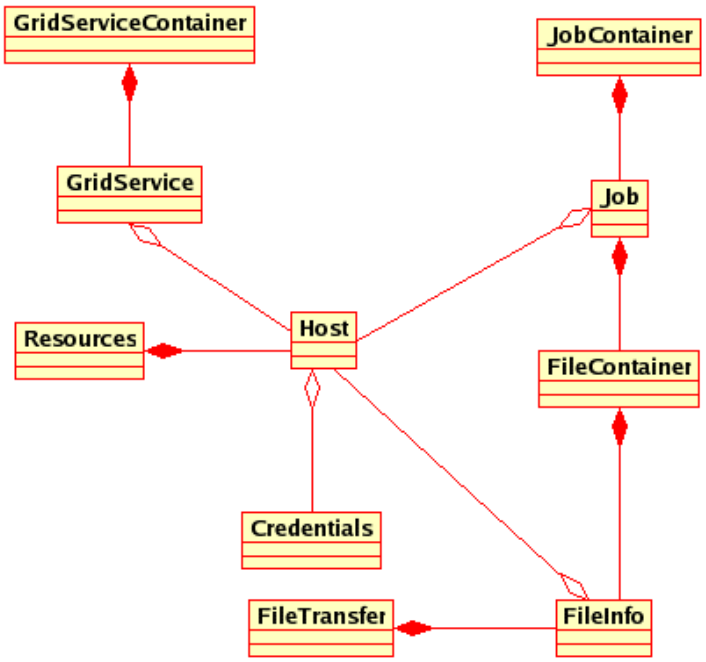



Figure 7. UML class diagram of the GAUGE intermediate layer.

Thus, any other implementation that a developer wishes to construct would be transparent to the end-user as long as the domain-model language (GDM) is maintained.

However, this layer (the code generation layer) is not as tightly coupled to the previous layers as the intermediate layer is to the user interface. Changes to GDM will entail changes to the intermediate layer, but not necessarily involve changes to some of the model interpreters. For example, if a new job parameter is needed for a new output representation, this change need only be made to GDM and the intermediate representation. The interpreter for this new output representation will use this new parameter, but the new parameter is ignored by the previous interpreters. In the same manner, some information may be redundant, or even unnecessary, for some output representations but needed for others. For example, the information about the hosts is not needed if the output representation is for a Grid framework that automatically decides where to run each job.

Furthermore, the quality of the output representations is improved because the model interpreters can be developed by Grid experts. Errors are also minimized at earlier stages because the interpreters are developed and tested independently, and are optimized for a particular Grid framework.

In the Java applications that are currently supported by GAUGE, data dependence is defined by the sequence of tasks (e.g., if B is defined after A, B waits until A finishes to start execution).

```
1  public void loadModel(JBuilderModel jobModel,
2                        Resources hostsDefined) throws Exception
3  {
4          ......
5          jobName = jobModel.getName(); /* name of the gme job */
6
7          /* 1) get the RSL parameters */
8          Vector rslParameters = jobModel.getAtoms(Constants.ROLE_JOB_RSL);
9          ExtractRSL((JBuilderAtom)rslParameters.elementAt(0));
10
11         /* 2) get the hosts */
12         Vector gmeHost = jobModel.getReferences(Constants.ROLE_JOB_HOST);
13         extractHost((JBuilderReference)gmeHost.elementAt(0), hostsDefined);
14
15         /* 3) finally extract the information about the input/output files */
16         extractFiles(jobModel);
17         ......
18 }
19
20 private void extractRSL(JBuilderAtom gmeRSL)
21 {
22         rslDirectory = gmeRSL.getIAtom().getStrAttrByName(
23                        Constants.ATTR_JOB_RSL_DIRECTORY);
24         rslExecutable = gmeRSL.getIAtom().getStrAttrByName(
25                        Constants.ATTR_JOB_RSL_EXECUTABLE);
26         rslArguments = gmeRSL.getIAtom().getStrAttrByName(
27                        Constants.ATTR_JOB_RSL_ARGUMENTS);
28 ......
29 }
30
31 private void extractHost(JBuilderReference host, Resources hosts)
32 {
33         /* get the gme name of the host */
34         String hostName = host.getReferred().getName();
35         ......
36         /* find the host using the collection previously defined */
37         try {
38             hostForJob = hosts.findHost(hostName);
39         } catch (Exception e) {
40             /* in case the host can't be found, throw an exception */
41             Debug.printMessage(" The host " + hostName + " hasn't been defined.\n");
42         }
43 }
```

Figure 8. Code to convert a GDM job to the intermediate representation. `loadModel` performs the whole conversion, `extractRSL` shows how the RSL parameters are converted and `extractHost` shows how the information from the hosts (resources) is extracted.

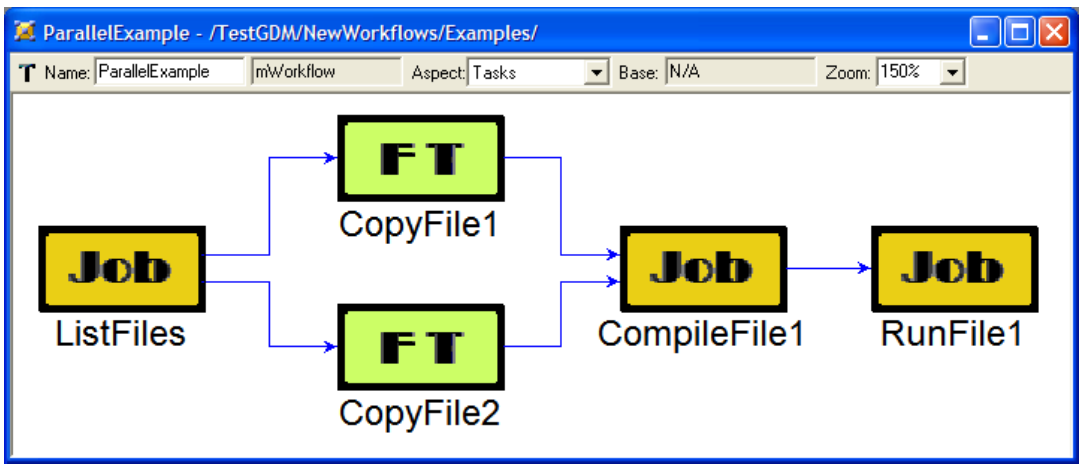



Figure 9. Definition of the workflow. The jobs are executed sequentially while the file transfers are executed concurrently.

In the case of parallel constructs, data dependence is defined at the origin of the split. When the parallel paths merge, the AND-Join construct depends on the completion of all the parallel branches (synchronization). Furthermore, when the execution flow splits (AND-Split or XOR-Split), the merging task is blocked until the parallel tasks finish their execution. At present, the generated applications communicate directly with the Java CoG Kit. This causes scalability problems due to the generation of specific code for each workflow task. A solution to this problem is currently under investigation and consists of developing a reusable workflow engine that interacts with a Grid resource broker [35] that automatically selects the resources for executing the jobs. In this case the model interpreter will generate the appropriate configurations from the graphical models.

The infrastructure presented in this section allows developers to augment and extend GAUGE by providing different code generators to tailor the generated code to a particular Grid environment or platform. At the same time it provides an easy to use automation and generative environment in which an inexperienced user can exploit the Grid without familiarity of the intricacies of the underlying technologies.

### 5.4. Illustrative example

In this section we present an example showing the interaction with GAUGE, i.e. how users specify the workflows as well as the details of the sub-tasks. The example, as can be seen in Figure 9, presents an application that first lists the files located in a directory of a remote machine. Then two files are copied in parallel, one is a Java file and the other is the input for the Java program. The Java file is compiled in

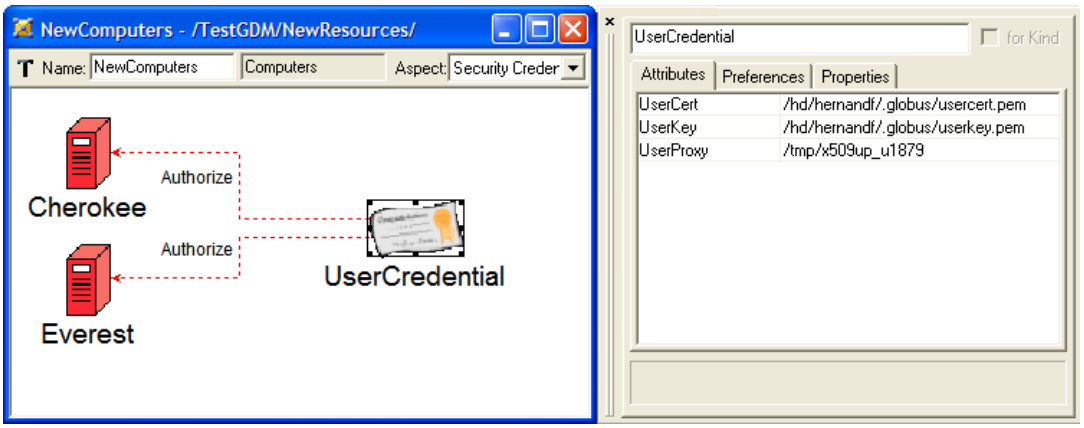



Figure 10. Resource and credentials specification. This figure shows how the information about the user credentials is specified. `UserCredential` authenticates both `Cherokee` and `Everest`. The information relevant to define a host is its URL and the port that will be used for gridFTP.

the remote machine and finally the Java program is executed. Even though this workflow uses simple commands, it serves the purpose of illustrating the use of GAUGE (for more examples the reader is referred to [6,22]).

The first step to specify a workflow is to define the resources that will be used. Figure 10 illustrates the manner in which the resources are defined. In this case two resources '`Cherokee`' and '`Everest`' are defined. The information relevant to define a host is its URL and the gridFTP port (not shown in the figure). In order to use the hosts, a user credential needs to be provided. In this example, '`UserCredential`' authorizes the use of both resources. The right-hand side of Figure 10 shows that the information relevant to authorize a resource is the user certificate, user key, and user proxy.

Figure 11 presents the way in which a job is specified as a workflow sub-task. The left-hand side of the figure indicates that the job will be run on '`Everest`' while the right-hand side shows that the executable is Java with '`Program in.txt`' as its argument.

Finally, to define a file transfer users need only specify the location of the endpoints as seen on Figure 12. The location of the endpoints is given by the combination of the host, directory, and name of the file.

In order to illustrate that different output representations can be generated from the same model, we present a Java code output representation that uses the Java CoG Kit (Figure 13), as well as a GridAnt representation (Figure 14). Comparing Figure 11 with Figure 13, it can be observed that the information specified in the visual model is used to create an RSL string (lines 4–8, Figure 13) that is used to submit the job to '`Everest`' (lines 13–16). Finally, Figure 14(a) shows the code that is generated for a job sub-task by the GridAnt model interpreter and Figure 14(b) shows the XML configuration that this model interpreter generates for the file transfer `CopyFile1`.

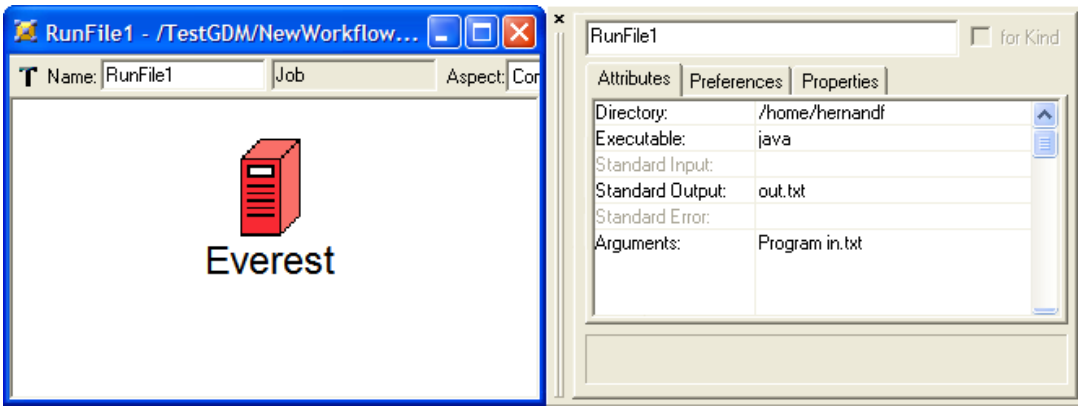



Figure 11. Job specification. This figure shows the information necessary to run the job '`RunFile1`'. The host selected for this job is `Everest`. The right-hand side of the figure shows that the executable name is '`java`' and the argument is '`Program in.txt`'.

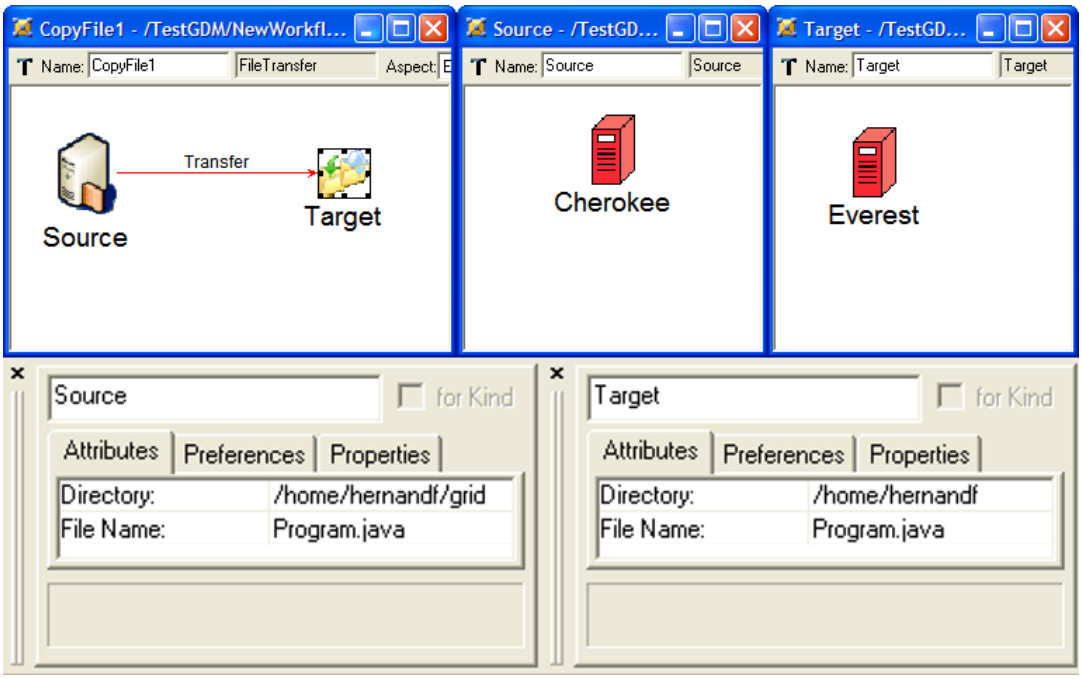



Figure 12. File transfer specification. This figure shows that defining file transfers consists of specifying the location of the files at the endpoints. The source file (Program.java) is located in '`Cherokee`' and the directory is '/home/hernandf/grid'. This file is copied to `Everest` in the directory '/home/hernandf' and it is given the same name.

```
1  ......
2  byte[] RunFile1Proxy = getByteArray("/tmp/x509up_u1879");
3
4  GlobusRSL RunFile1RSL = new GlobusRSL(); // create the rsl string
5  RunFile1RSL.setArg("Program.java");
6  RunFile1RSL.setEnvironmentVariables("(PATH=/usr/bin)");
7  RunFile1RSL.setDir("/home/hernandf");
8  RunFile1RSL.setExec("javac");
9
10 try {
11 GRAMJob RunFile1GRAM = new GRAMJob();
12 // submit the job
13 String RunFile1ID = RunFile1GRAM.submitJob(
14                  "everest00.cis.uab.edu",
15                  RunFile1Proxy,
16                  RunFile1RSL.toRSL());
17 // wait for its completion
18 String RunFile1Cond = RunFile1GRAM.checkStatusOfJob(
19                                  RunFile1ID, CompileFile1Proxy);
20 ......
```

Figure 13. `RunFile1` Java generated code. Code generated by the model interpreter for the `RunFile1` job.

```
1  <grid-execute
2   name="RunFile1"
3   server="everest00.cis.uab.edu"
4   executable="java"
5   output="out.txt"
6   directory="/home/hernandf"
7   arguments="Program in.txt"
8   environment="PATH=/usr/bin"
9  / >
```

(a)

```
1  <grid-copy
2   name="CopyFile1"
3   from="gsiftp://Cherokee.cis.uab.edu//home
                        /hernandf/grid/Program.java"
4   to="gsiftp://everest00.cis.uab.edu//home
                        /hernandf/Program.java"
5  / >
```

(b)

Figure 14. GridAnt output representation. (a) Job execution, the code that the model interpreter generates for the '`RunFile1`' job. (b) File transfer, the code that is generated for the file transfer '`CopyFile1`'.

## 6. RELATED WORK

The foundation of GAUGE is to abstract the Grid environment into a high-level layer such that the essence of the workflow is not bound to a specific Grid environment. This high-level layer presents the Grid as a set of familiar concepts so that a user can easily specify Grid applications, i.e. workflows. The specification of tasks and workflows is performed by interacting with GME, which is configured by the GDM visual language. In addition to the Grid constructs, GDM also includes workflow specific constructs (e.g., Splits and Joins). In the next subsections we present work on the areas that are relevant to this project.

### 6.1. Grid abstraction

Although there are several studies related to this topic (e.g., [12,13]), the work by Amin *et al.* [11] presents a similar work to ours. Amin proposes a technology and architecture-independent abstraction layer to provide interoperability across multiple Grid implementations, resulting in the Open Grid Computing Environment (OGCE). The main function of OGCE is to serve as a technology-independent, open, and extensible framework for client-side Grid development. The abstractions provided by OGCE are comparable to those introduced by GDM in this paper. For example, the task concept presented in [11] contains notions similar to those involved in the GDM job specification (Figure 4). However, the main difference between the studies is in the level of abstraction. In this paper the abstraction layer is realized at a domain-model level, but in [11] the abstraction layer is at a programming language level (Java). There are several advantages of working at the model level instead of the programming language level.

1. Because the essence of a problem is captured, models are much more amenable to analysis than the lower-level code representation (where the accidental complexities of programming language syntax and semantics often get in the way of higher-level analysis of the problem).
2. Because the representation is more abstract, it is often the case that a change can be made to a model that would have required many more adaptations at the code level. For those concerns that are crosscutting in nature, a simple change to a model can affect literally hundreds of places in the corresponding source code [25]. Thus, models better facilitate the rapid exploration of design decisions and changes to system configuration.
3. By focusing on the problem at the modeling level, rather than the specific technology used for a solution, the developer and user can protect and isolate themselves from technology obsolescence. That is, the models are not tied to any one solution and can easily adapt to the next generation's hot new technology.

### 6.2. Workflow systems

The idea of composing applications from reusable components is not new. For example, WebFlow [36] introduces a platform-independent system that dynamically composes new applications from reusable components by clicking and dragging icons. The job model of UNICORE [5] uses a set of directed acyclic graphs, and also permits the use of conditional and iterative execution of job groups or tasks. DAGMan [37] also maps a direct acyclic graph specification onto a physical environment. The Symphony framework [38] uses a graphical user interface for rapid collaborative development of Grid applications following a data flow paradigm. Triana [39] also offers a visual programming model for the dynamic composition of predefined software components.

Other works propose languages to specify Grid workflows. For example, Grid Workflow [40] focuses on proposing a standard for the sequence of complex high-performance computational tasks within a Grid. GridAnt [21] uses an XML-based language to specify client-side workflows. GridAnt is also able to submit the executions of tasks or file transfers by using a workflow engine based on the Apache ANT tool [41]. Another set of tools use artificial intelligence to handle the automatic creation of workflows [42,43].

The workflow based on GAUGE has been influenced by these projects, as well as van der Aalst's studies on workflows [23] and workflow patterns [17,24]. However, GAUGE is not tied to specific programming languages or platforms and it is able to generate software artifacts for various Grid environments.

## 7. FUTURE WORK

The GAUGE project is in its initial phase. The current implementation of GAUGE can handle only a limited number of tasks in the workflow due to the Java output representation. Now that the visual language and the basic three-tier infrastructure is in place, we plan to extend the functionality in four different areas: (1) improve the user interaction with the front-end; (2) integrate GAUGE with GIS [44] services; (3) write generators for different output representations (e.g., PyGlobus); and (4) improve the scalability of the generated applications by utilizing a reusable workflow engine that interacts with a Grid resource broker [35] which automatically selects the resources for executing the jobs. In this case the model interpreter will generate the appropriate configurations from the graphical models. In addition to this work in progress, future directions that will be considered involve the following aspects.

1. Improve GDM in three ways: (1) extend the language to allow a complete mapping of the Grid, which will allow a better integration with different Grid environments; (2) introduce more workflow control constructs (e.g., OR-Split and OR-Join), which would permit the implementation of more workflow patterns making GDM a more robust language; and (3) mapping the system to the new OGCE [11] specification.
2. Improve the design of the conditional constructs. The solution that is currently under investigation is the use of AI techniques to execute these kinds of decisions automatically. In particular, the use of rule-based systems is being considered. Several tools exist that help to construct rule-based systems. However, because of the particular constraints imposed by the environment (GME), the tool that is being analyzed for this purpose is the Java Expert System Shell (JESS) [45]. JESS's easy integration with programs written in the Java language is the key reason for choosing this tool.

## 8. CONCLUSIONS

This paper presents the architecture and implementation of the Grid Automation and Generative Environment (GAUGE), which enables inexperienced users to take full advantage of the Grid infrastructure. The architecture of GAUGE provides a high-level view for the construction of Grid applications using the Globus Toolkit that shields the intricacies and accidental complexities of the Grid environment.

GAUGE helps the inexperienced Grid users by providing an environment in which they can graphically specify the workflow for their application and automatically generate the code that manages the execution of the workflow (execution and data flow). GAUGE also helps developers by providing an open framework in which they can reuse components and tailor the automation environment to work with different Grid environments.

The novelty of the manner in which GAUGE was designed is that it builds an abstract layer that hides the complexities of the Grid environments. This abstract layer is represented with the visual language GDM. Users manipulate visual models that represent Grid concepts, instead of having to write code for a specific infrastructure and in a specific programming language. GDM was created using domain-specific modeling techniques. The benefits of using domain-specific modeling techniques are as follows.

1. Domain modeling removes the accidental complexities of creating workflows in a Grid by focusing on higher-levels of abstraction at the problem space rather than the solution space, such as specific low-level Grid services and their usage.
2. When exploring various workflow scenarios, modeling tools and their interpreters facilitate the more rapid ability to change the workflow details. That is, it is easier to manipulate and change domain models rather than the associated code. This is an important benefit when rapid prototyping of Grid applications is needed.
3. Model-driven techniques possess the ability to generate multiple artifacts from the same model. Thus, with the same domain knowledge different output representations can be generated, which is essential when the Grid application consists of different back-end implementations.
4. The quality of the output representations is improved since the model interpreters can be developed by Grid experts. Errors are also minimized at earlier stages because the interpreters are developed and tested independently and are optimized for a particular Grid framework.
5. The changeability and maintainability of the system is improved for both operators and developers. Operators need only to change their model and the low-level output representations change accordingly. The benefit for developers is that the three-layer architecture of GAUGE facilitates the isolation of changes to a specific layer.

The potential impact of this research and initial implementation is the reduction of the development time involved in generating applications for the Grid. Through GAUGE, users are not required to learn how to use specific Grid technologies to develop Grid-enabled applications. Rather, they construct graphical models that are at a more appropriate level of abstraction for describing the essence of the problem for a specific domain.

GAUGE's use of domain-specific modeling also addresses challenges faced by developers in improving the use of modern software engineering practices for the Grid programming model, and developing applications that interface with different Grid environments. The benefit offered by this work is an abstract high-level Grid model that can serve potentially as a basis for an extended model to completely map the Grid and facilitate the interaction between different Grid technologies and models that are not tied to specific programming languages or Grid frameworks.

## REFERENCES

1. Foster I *et al*. The grid2003 production Grid: Principles and practice. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, Honolulu, HI, June 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 236–245.
2. Pearlman L, Kesselman C, Gullapalli S, Spencer B, Futrelle J, Ricker K, Foster I, Hubbard P, Severance C. Distributed hybrid earthquake engineering experiments: Experiences with a ground-shaking Grid application. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, Honolulu, HI, June 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 14–23.
3. Foster I, Kesselman C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications* 1997; **11**(2):115–128. Available at: http://www.globus.org/.

4. Legion worldwide virtual computer. http://legion.virginia.edu [27 May 2005].
5. Unicore Web site. http://unicore.sourceforge.net/ [27 May 2005].
6. Hernández F, Bangalore P, Gray J, Reilly K. A graphical modeling environment for the generation of workflows for the globus toolkit. *Component Models and Systems for Grid Applications. Proceedings of the Workshop on Component Models and Systems for Grid Applications*, Saint Malo, France, 26 June 2004, Getov V, Kielmann T (eds.). Springer: Berlin, 2005; 79–96.
7. von Laszeweski G, Foster I, Gawor J, Lane P. A Java Commodity Grid kit. *Concurrency and Computation: Practice and Experience* 2001; **13**(8–9):643–662. Available at: http://www.cogkit.org/.
8. Haupt T, Bangalore P, Henley G. Mississippi computational Web portal. *Concurrency and Computation: Practice and Experience* 2002; **14**(13–15):1275–1287.
9. Fox G, Gannon D, Thomas M. (eds.). Special Issue: Grid Computing Environments. *Concurrency and Computation: Practice and Experience* 2002; **14**(13–15):1035–1593.
10. Fox G, Gannon D, Thomas M. A summary of Grid computing environments. *Grid Computing: Making the Global Infrastructure a Reality*, Berman F, Fox G, Hey T (eds.). Wiley: New York, 2003; 543–554.
11. Amin K, Hategan M, von Laszewski G, Zaluzec N. Abstracting the Grid. *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04)*, La Coruña, Spain, February 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 250–257.
12. Jhoney A, Kumar S, Kuchhal M, Venkatakrishnan S. Grid application framework for java (gaf4j). *Technical Report 0.9.6*, IBM Software Labs, Bangalore, India, February 2003.
13. Wolski R, Brevik J, Obertelli G, Spring N, Su A. Writing programs that run everywhere on the computational Grid. *IEEE Transactions on Parallel and Distributed Systems* 2001; **12**(10):1066–1080.
14. Gray J, Bapty T, Neema S, Tuck J. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM* 2001; **44**(10):87–93.
15. Greenfield J, Short K, Cook S, Kent S, Crupi J. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley: New York, 2004.
16. Kacsuk P, Goyeneche A, Delaitre T, Kiss T, Farkas Z, Boczko T. High-level Grid application environment to use legacy codes as OGSA Grid services. *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, PA, November 2004. ACM Press: New York, 2004.
17. van der Aalst W, ter Hofstede AH, Kiepuszewski B, Barros AP. Advanced workflow patterns. *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS2000)* (*Lecture Notes in Computer Science*, vol. 1901), Etzion O, Scheuermann P (eds.). Springer: Berlin, 2000; 18–29.
18. Foster I, Kesselman C, Nick J, Tuecke S. The physiology of the Grid: An Open Grid Services Architecture for distributed systems integration. *Technical Report*, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002; Available at: http://www.globus.org/alliance/publications/papers/ogsa.pdf [27 May 2005].
19. Lédeczi Á, Bakay Á, Maróti M, Völgyesi P, Nordstrom G, Sprinkle J, Karsai G. Composing domain-specific design environments. *IEEE Computer* 2001; **34**(11):44–51.
20. Karsai G, Maróti M, Lédeczi A, Gray J, Sztipanovits J. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology (Special Issue on Computer Automated Multi-Paradigm Modeling)* 2004; **12**(2):263–278.
21. von Laszewski G, Amin K, Hategan M, Zaluzec N, Hampton S, Rossi A. Gridant: A client-controllable Grid workflow system. *Proceedings of 37th Hawaii International Conference on System Science*, Big Island, HI, January 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 210–219.
22. Hernández F. Domain-specific models and the globus toolkit. *Technical Report UABCIS-TR-2004-0504-1*, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL, May 2004.
23. van der Aalst W, van Hee K. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. MIT Press: Cambridge, MA, 2002.
24. van der Aalst W, ter Hofstede AH, Kiepuszewski B, Barros AP. Workflow patterns. *Distributed and Parallel Databases* 2003; **14**(1):5–51.
25. Gray J *et al.* Model-driven program transformation of a large avionics framework. *Generative Programming and Component Engineering (GPCE 2004)* (*Lecture Notes in Computer Science*, vol. 3286), Karsai G, Visser E (eds.). Springer: Vancouver, BC, 2004; 361–378.
26. The globus resource specification language rsl v1.0. http://www-fp.globus.org/gram/rsl_spec1.html [27 May 2005].
27. van Rossum G. *The Python Language Reference Manual*. Network Theory Ltd, 2003.
28. Python globus (pyglobus). http://dsd.lbl.gov/gtg/projects/pyGlobus [27 May 2005].
29. Neema S, Bapty T, Gray J, Gokhale A. Generators for synthesis of qos adaptation in distributed real-time embedded systems. *Proceedings of ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)* (*Lecture Notes in Computer Science*, vol. 2487), Batory D, Consel C, Taha W (eds.). Springer: Berlin, 2002; 236–251.


30. Long E, Misra A, Sztipanovits J. Increasing productivity at Saturn. *IEEE Computer* 1998; **31**(8):35–43.
31. Bapty T, Neema S, Gray J. Model-integrated computing for composition of complex qos applications using the generic modeling environment (gme). *OMG Workshop on Real-Time and Embedded Distributed Object Computing*, Washington, DC, July 2002.
32. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Addison-Wesley: Reading, MA, 1998.
33. Peterson J. Petri nets. *ACM Computing Surveys* 1977; **9**(3):223–252.
34. Guan Z, Hernández F, Bangalore P, Gray J, Skjellum A. Grid-Flow: A Grid-enabled scientific workflow system with a Petri-net-based interface. *Concurrency and Computation: Practice and Experience* 2006; [this issue].
35. Afgan E, Velusamy V, Bangalore P. Grid resource broker with application profiling and benchmarking. *Proceedings of the European Grid Conference*, Amsterdam, Netherlands, February 2005. Springer: Berlin, 2005.
36. Akarsu E, Fox G, Furmanski W, Haupt T. WebFlow—high level programming environment and visual authoring toolkit for high performance distributed computing. *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Orlando, FL, November 1998. ACM Press: New York, 1998; 1–7.
37. Dagman (directed acyclic graph manager). http://www.cs.wisc.edu/condor/dagman/ [27 May 2005].
38. Lorch M, Kafura D. Symphony—a Java-based composition and manipulation framework for computational Grids. *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, May 2002. ACM Press: New York, 2002; 136–143.
39. Triana workflow. http://www.trianacode.org [27 May 2005].
40. Bivens H. Grid workflow. Grid Computing Environments Working Group Document. 2001. http://dps.uibk.ac.at/uploads/101/draft-bivens-grid-workflow.pdf [27 May 2005].
41. The apache ant project. http://ant.apache.org/ [27 May 2005].
42. Deelman E *et al.* Mapping abstract complex workflows onto Grid environments. *Journal of Grid Computing* 2003; **1**(1):25–39.
43. Yeo B, Hung T, Khoo B. An agent-based Grid flow management framework for the problem solving environment (PSE). *Scientific Work-flow Management Mini Symposium, GlobusWorld*, San Francisco, CA, February 2004.
44. Czajkowski K, Fitzgerald S, Foster I, Kesselman C. Grid information services for distributed resource sharing. *Proceedings of the 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, August 2001. IEEE Press: San Francisco, CA, 2001; 81–194.
45. Friedman-Hill E. *Jess in Action: Java Rule-Based Systems*. Manning Publications: Greenwich, CT, 2003.