Campus Grids Meet Applications: Modeling, Metascheduling and Integration[*] [†]

Yonghong Yan, and Barbara M. Chapman

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-06-03

May 1, 2006

Keywords: Campus Grid, Workflow Modeling, Metascheduling, Execution Plan, Resource Reservation

Abstract

Air Quality Forecasting (AQF) is a new discipline that attempts to reliably predict atmospheric pollution. An AQF application has complex workflows and in order to produce timely and reliable forecast results, each execution requires access to diverse and distributed computational and storage resources. Deploying AQF on grids is one option to satisfy such needs, but requires the related grid middleware to support automated workflow scheduling and execution on grid resources.

In this paper, we analyze the challenges in deploying an AQF application in a campus grid environment and present our current efforts to develop a general solution for grid-enabling scientific workflow applications in the GRACCE project. In GRACCE, an application's workflow is described using GAMDL, a powerful dataflow language for describing application logic. The GRACCE metascheduling architecture provides the functionalities required for co-allocating grid resources for workflow tasks, scheduling the workflows and monitoring their execution. By providing an integrated framework for modeling and metascheduling scientific workflow applications on grid resources, we make it easy to build a customized environment with end-to-end support for application grid deployment, from the management of an application and its dataset, to the automatic execution and analysis of its results.

[*]The work is performed as part of The University of Houston's Sun Microsystems Center of Excellence in Geosciences [45].
[†]This is an extended version of the paper that is accepted for publication in the Journal of Grid Computing, 2006.

Campus Grids Meet Applications: Modeling, Metascheduling and Integration[*] [†]

Yonghong Yan, and Barbara M. Chapman

Abstract

Air Quality Forecasting (AQF) is a new discipline that attempts to reliably predict atmospheric pollution. An AQF application has complex workflows and in order to produce timely and reliable forecast results, each execution requires access to diverse and distributed computational and storage resources. Deploying AQF on grids is one option to satisfy such needs, but requires the related grid middleware to support automated workflow scheduling and execution on grid resources.

In this paper, we analyze the challenges in deploying an AQF application in a campus grid environment and present our current efforts to develop a general solution for grid-enabling scientific workflow applications in the GRACCE project. In GRACCE, an application's workflow is described using GAMDL, a powerful dataflow language for describing application logic. The GRACCE metascheduling architecture provides the functionalities required for co-allocating grid resources for workflow tasks, scheduling the workflows and monitoring their execution. By providing an integrated framework for modeling and metascheduling scientific workflow applications on grid resources, we make it easy to build a customized environment with end-to-end support for application grid deployment, from the management of an application and its dataset, to the automatic execution and analysis of its results.

Index Terms

Campus Grid, Workflow Modeling, Metascheduling, Execution Plan, Resource Reservation

I. Introduction

Air Quality Forecasting (AQF) [30] is a new discipline that attempts to reliably predict atmospheric pollution. A real-world AQF application incorporates multiple, interdependent computational modules that make intensive use of numerical tools, requires high compute power for the simulation of meteorological and chemical processes, and entails the transfer, storage and analysis of a huge amount of observational and simulation data [7]. We participate in an effort to build such a service, with the goal of providing timely, reliable forecasts of air quality for the Houston-Galveston region and for several other regions in the South Central USA that have encountered problems with air quality in the recent past [4], [5]. On-going work at the University of Houston (UH) aims to create, test and deploy an AQF application as well as to establish a suitable development and deployment environment.

Grid technologies and middleware [16] provide a potential strategy for meeting the computational and storage needs of AQF executions. In grids, users with large-scale problems such as AQF application are able to exploit multiple distributed high performance computing resources to produce high quality results that cannot be achieved from single-domain resources. As grid technology becomes mature and standardized, the act of deploying applications on a grid to efficiently use its resources becomes more important than the work on basic technology and its standardization. Current middleware efforts focus on the core grid-enabling technologies and hence, application-level management and deployment-related issues often fall outside their scope. Therefore, additional efforts are required to augment grid middleware in order to realize end-to-end support for domain scientists. In general, such efforts are still in the experimental stage and related experiences are very application-specific and technology oriented.

The current efforts, including ours [3], [4], on grid application deployment often focus on packaging or wrapping of legacy application codes with services and utilities for grid execution and data transport, and offer these via a web interface to end users. The provision of automated application-specific scheduling and execution on grid resources, and thus end-to-end grid environment support to scientists, remains an unsolved challenge. This paper presents our experience of deploying an AQF application on a campus grid environment and our current efforts to develop a solution for grid-enabling AQF-like applications as part of the GRACCE project [36]. Our initial efforts

[*]The work is performed as part of The University of Houston's Sun Microsystems Center of Excellence in Geosciences [45].
[†]This is an extended version of the paper that is accepted for publication in the Journal of Grid Computing, 2006.

provided a working, but not feature-complete solution to support AQF runs on resources across our campus grid at the UH.

The GRACCE project aims to provide end users with a comprehensive application grid platform, with support for the management of the application and its dataset, as well as the automatic execution and viewing of results. In GRACCE, application coordination and collaboration (as typical in a workflow) are described using GAMDL, a powerful dataflow language to model an application's logic. The GRACCE metascheduling architecture is designed to be a layer on top of and to extend the available grid middleware to provide automated grid resource co-allocation, workflow coordination and runtime control. The architecture includes a workflow-orchestrated metascheduler with planning and reservation features, an event-driven workflow engine able to coordinate the scheduling process and job execution, and a runtime system to control workflow execution.

The organization of this paper is as follows. In Section 2, we introduce the AQF application, our initial efforts in deploying AQF on the UH Campus grid [3], and related issues. Following this, the motivations of GRACCE and the GRACCE solution are discussed and introduced in Section 3. Section 4 presents the GAMDL application model language, including its structure and features. In Section 5, the GRACCE metascheduling architecture is described in details, including the architectural subsystems and their functionalities, and core algorithms and mechanisms. In Section 6, related work is extensively studied. Finally, our work is concluded in Section 7.

II. Experiences Running AQF on a Campus Grid

The initial grid deployment of our AQF application utilized the functionalities provided by the Globus toolkit [13] and realized a working solution to support AQF runs on the resources across our campus grid [4]. In this section, we introduce the AQF application, its current deployment status and the issues that arise with the current approach.

A. AQF Introduction

Our Air Quality Forecasting (AQF[†]) application is an integrated computational model for regional and local air quality forecasts, and is composed of three subsystems: the PSU/NCAR MM5 weather forecast model [10], the SMOKE emission system [32], and EPA's CMAQ chemical transport model [8]. An AQF execution is a computational sequence of the three subsystems with increasing resolution and decreasing geographical boundaries. Figure 1 illustrates the workflow of a nested 2-day forecasting operation over a single region of interest by a three-domain computation. The 36km domain computation provides coarse forecast data over the continental USA, the 12km domain provides data across the south central USA, and the 4km domain forecasts air quality across a smaller geographic region. A full forecast in an urban area requires an additional level of refinement based upon a 1km domain. Each rectangle represents a computational module and each arrow indicates the flow of data between modules. AQF modules may execute on heterogeneous, distributed resources provided that the dependent files of each module are transferred to the allocated resources as they become available.

B. Initial Experience of AQF Deployment on Campus Grid

The UH campus grid consists of heterogeneous resources including clusters of Sun SMPs, a Beowulf cluster and an SGI visualization system with 9 TB storage, all at the UH High Performance Computing Center (HPCC) [38]; and of Beowulf clusters, Sun SMPs and several Sun workstations in several other departments. The AQF modules are installed and configured on these resources, and disk and tape space is allocated for their daily executions. Sun Grid Engine (SGE) [43] and Platform LSF [41] have been installed to manage resources within the individual administrative domains. The Globus toolkit [13] is installed on these resources to provide the utilities for grid job execution and remote file transfer. The UH HPCC serves as CA [15] for our campus grid and is responsible for granting grid accounts. To make it as easy as possible for users to interact with the services provided through the campus grid, EZ-Grid [4], a light-weight web-based portal, has been developed. It uses the Java CoG Kit [12] to provide a convenient interface to all Globus functions, including grid authentication with X.509 certificates, job specification, submission and management, file transfers, and query of grid resource information and load status.

In this setup, the AQF workflow is described in an XML file, and a Perl script controls the workflow execution using the Globus toolkit and EZ-Grid. A module in the workflow is described as a task which is mapped to a grid

[†]the acronym AQF from here on denotes our AQF application unless noted otherwise.

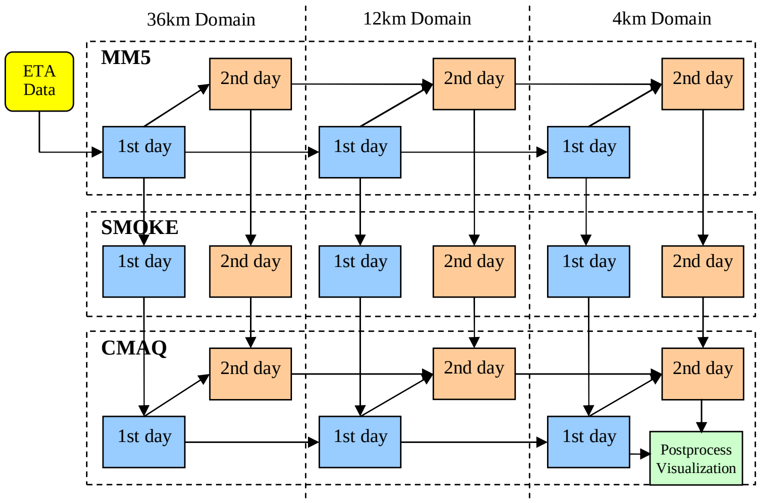

Fig. 1. AQF Application Workflow

computational job (thus module, task and job refer to the same entity in different contexts, the term *module* is used in an application context, *task* in a workflow context, and *job* in a grid context). Dependencies between modules are specified as parent-child relationships between tasks, in which parent tasks produce the data to be consumed by child tasks. For each task, details about the executable and target resources are hard-coded in the task RSL files [44]. The Perl script reads the XML file and controls the overall execution of the AQF tasks, including submitting jobs to grid resources, initiating file transfers when the data are available, and resolving task dependencies.

There are several problems with the above solution. Firstly, computational resources are pre-allocated for the AQF tasks and are assumed to be available during the task execution periods. The allocated resources specified in the task RSL files are defined by system administrators, who also reserve the resources in the local schedulers to ensure their availabilities. Obviously, this human-scheduling approach is not suitable for dynamic grid environments – resource allocation should be automated to provide best decisions based on up-to-date resource load status information. Secondly, failures in a grid resource will cause the failure of the whole AQF run unless a user intervenes. There is no back-up strategy to allocate resources for a task whose dedicated resource fails. Specifying a secondary resource in the RSL is one solution, yet normally the secondary resource is rather busy in our environment, and that task would have to wait in the local queue if submitted. Thirdly, the non-standard XML and script approach for workflow description and execution control is error-prone and places a major burden on end users and system administrators. Users are required to become adept in XML, Perl and RSL, which is a daunting additional effort when working on AQF deployment. Instead of having to get involved in all details of grid setup and resource scheduling, our users expect a complete application execution environment, from a graphical user interface to specify the application's configuration, to a viewer for execution results.

III. Motivations and GRACCE

AQF represents a large number of domain applications that would like to exploit grid-enabled resources for their computation. These applications are no longer being developed as monolithic codes, but incorporate multiple interdependent modules, and entail the transfer and storage of large amounts of data. Enabling such an application in grid environments is much more complex than enabling an application that can be wrapped as a single grid job. From our experience, the following topics must be addressed in order to provide support for automatic execution of AQF-like workflow applications, and to integrate them into a grid environment in a manner appropriate for end users:

- **Automatic scheduling and execution of the application modules:** For AQF-like applications which consist of multiple interdependent computational modules, submitting a module job should be automated, requiring no user interaction unless an errors occur. During workflow execution, actions to handle the dependencies (such as transferring intermediate files between resources) should also be automatically initiated as the files become available.

- **Resource co-allocation for workflow computation modules and files:** Such capabilities should be provided by software and should make the best decisions in a dynamic grid environment to ensure application level Quality-of-Service. In our AQF case, this means AQF execution must be completed by a certain deadline.
- **Execution monitoring, failure handling and notification:** Status and errors in workflow execution should be reported to users without a need for them to actively check.
- **Modeling and describing application logic and jobs:** A modeling language or GUI interface should be available for end users to describe application dataflow, control logic and grid jobs.
- **Portal interface:** A web portal should be available for end users to access the capabilities as described above, and to manage their applications and the application data sets on the grid.

Many grid software packages have been developed that aim to provide partial solutions to the problems listed above. In Section VI, we study related software and tools, including workflow definition languages for application modeling, co-allocation and reservation services provided by various projects, and workflow enacting engines for workflow execution control. In our studies, we noticed that most of the current efforts address the fundamental of the above specific issues, such as co-allocation, reservation, planning and workflow execution. To the best of our knowledge, there are no efforts to develop an integrated grid metascheduling system. Efforts related to grid metascheduling for complex workflow applications are also missing.

To provide full-featured support for the automatic execution of AQF-like applications in grid environments, the integration of available solutions is as important as solving the various technical issues. An integrated solution should provide end-to-end support for application execution on a grid. It should provide an extensible framework that can accommodate the diverse range of requirements imposed both by the applications and by the underlying grid systems. Such a framework should be suitable for a wide range of distributed applications, but also support the construction of customized environments.

Our Solution: GRACCE – Building An Application Grid Environment

Driven by the needs of our local AQF effort, the GRACCE (Grid Application Coordination, Collaboration and Execution) project [36] was proposed to develop a set of grid middleware for application deployment. The vision of GRACCE is to provide scientists with a framework to build an application-specific grid environment, from the management of an application and its dataset, to the automatic execution and viewing of results. In the GRACCE framework, end users are only required to provide application descriptions and resource requirements. GRACCE is responsible for allocating grid resources for tasks, placing tasks on the allocated resources for execution, monitoring them, and returning their results back to users. More specifically, GRACCE provides an integrated solution to the following issues:

- **Application modeling and description:** GRACCE provides a modeling language, GAMDL, to describe application logic and workflow. GAMDL has several advanced features that are not available in other similar solutions and is a basis for the integration of a grid metascheduler and workflow systems in GRACCE.
- **Grid metascheduling:** GRACCE defines a grid metascheduling architecture for workflow applications that addresses the issues of resource co-allocation and reservation, workflow coordination, and workflow job execution and monitoring.
- **Workflow orchestration and resource co-allocation:** In GRACCE, the resource allocation and reservation process is based upon the application workflow; for example, sibling module jobs are allocated on concurrent resources.
- **Standalone system for job submission and monitoring:** Conventionally, the functionalities of job submission and monitoring are provided by the scheduling or workflow systems. In GRACCE, they are provided by a stand-alone runtime system, which allows the runtime system to be developed and integrated without changing the scheduler and workflow systems.
- **Integration:** The GRACCE metascheduling architecture is an extensible framework that integrates solutions to various technical problems in the area of grid scheduling and workflow. The architecture is also a platform for users to integrate their domain applications into grid environments.

IV. Grid Application Modeling and Description

To deploy a domain workflow application in grid environments, we must first be able to model and describe the application workflow and resource requirements in a manner that is appropriate for both end users and for integrating with low-level grid middleware. As we outline further below in Section VI, most current workflow languages model application dataflow as a DAG (Directed Acyclic Graph), and some few of them are able to describe loops or conditional branches. These languages require users to specify which tasks are executed in parallel and which tasks must be sequentially executed. These execution relationships could be easily constructed from a dataflow DAG with software tools, instead of asking users to provide such information.

To help in workflow scheduling, a modeling language should be able to describe the job information pertaining to a workflow task, e.g. resource requirements, execution history or profile data. Current workflow languages do not have this capability and rely on other languages such as RSL for this purpose, which introduces additional complexities when specifying resource multirequests for workflow tasks.

To provide support for both application logic description and grid metascheduling, the GRACCE Application Modeling and Description Language (GAMDL) was created with the following features:

- It describes both application data dependencies and control logic (loops and conditional branches) at a high level of abstraction.
- It separates the description of application logic and execution workflow, so that support for partial workflow does not introduce additional complexities.
- It associates grid job specifications with application module descriptions to support workflow-orchestrated metascheduling. As a result, there is no need to explicitly specify resource multirequests in a workflow.
- GAMDL allows similar modules to be easily described using multiple-value properties;

In this section, we introduce the core concepts and features of GAMDL. The GAMDL descriptions for the AQF application are listed in APPENDIX A. We refer interested readers to [36] for the GAMDL specification and schema details.

A. GAMDL Core Documents

GAMDL provides two documents to describe a distributed application and its workflow, the `Application` document and the `AppRun` document. An `Application` document serves to define application entities, such as executables, data files and modules, and the dependency relationships between them. An `AppRun` document describe a workflow of an application, such as the modules that are required for the workflow, the start module(s) and the start time of the workflow. The `Application` document provides a high-level abstract description of the application from the viewpoints of end users and it should encompass all the related entities required for an application. The `AppRun` document specifies an execution of the application. The separation of the two into different documents enables the user to specify different workflows based on their needs without defining a new application each time. This is especially useful for the recurrent execution of an application.

B. Application Dataflow Description

GAMDL models the dataflow of a grid application using the same concept as a DAG, and captures both the dependency relationships between modules and the intermediate files associated with these relationships. Dependency relationships are defined via either a parent-children (PCn) pattern or a child-parents (CPs) pattern. A PCn relationship has a parent module and one or more child modules, and a CPs relationship has a child module and one or more parent modules. Intermediate files in a relationship are specified as pipes. A **pipe** has a `pipeIn` and a `pipeOut` element; `pipeIn` specifies the piped output file of the parent task, and `pipeOut` specifies the piped input file of the child task. Each pipe represents only one intermediate file.

GAMDL includes conditional pipes to allow the decision about whether a dependency should be handled to be made during application execution. A **conditional pipe** associates a pipe with an *if* boolean condition which will be evaluated after the module completes execution. If it evaluates to *true*, the pipe is processed; otherwise, it is not processed. If the conditions on all pipes in a relationship are evaluated as *false*, runtime dependencies are not established and the child module will not be executed.

C. Control Logic Description

GAMDL allows the specification of control logic, such as loops or conditional branches, by using conditional pipes and variables. A **variable** is a <*name*, *value*> pair associated with an *if* condition. A new value can only be assigned to the variable if the associated *if* condition evaluates to *true*; where there is no condition, an assignment is always made. If the *value* being assigned is in the form of *value1:value2*, *value1* is assigned if the *if* condition is *true* and *value2* is assigned otherwise. In GAMDL, complex flow controls are achieved by the proper assignment of variable values and reasoning on the conditions associated with pipes and variables A module may assign values to variables before its execution (in a `preAssign` element) and/or after its execution (in a `postAssign` element). The condition associated with a variable assignment or a pipe is permitted to reference system environment variables as well as variables defined in other modules. In APPENDIX A.5, we give an example showing how a workflow with loops and conditional branches is specified using conditional pipes and variables.

D. GAMDL Module Job Specification

In GAMDL, a job description specifies the details about how an application module is to be constructed as a grid computational job, such as executable arguments and resource request information. A module consists of multiple jobs and a job can only be associated with a module. In this schema, a job workflow is constructed based on module dependencies. In an application's `AppRun` document, which specifies an execution, the included modules, their dependency relationships and the start module(s) constructs a module graph. The job(s) associated with the modules are used to construct the corresponding job workflow.

The association of job specifications with a module also eliminates the need to specify resource multirequests for workflow applications. When allocating resources for workflow tasks, a metascheduler makes resource allocation decisions based on the given module (task) dependencies. For example, the sibling modules in the workflow are allocated on resources concurrently. If RSL or the GGF Job Submission Description Language (JSDL) [40] were used for this instead, the end user would need to explicitly specify resource multirequests to achieve resource co-allocation [23], [21]. With GAMDL, the user only needs to specify the resource requests for each module job, and the metascheduler makes the co-allocation decisions based on the module workflow.

Finally, GAMDL introduces job profile specification, which allows a metascheduler to evaluate historical information on module executions to help the resource allocation process. For applications like AQF that run everyday with similar scenarios, it is very easy to predict the execution behavior of a module on resources on which the module has been executed previously. Based on these predictions, the metascheduler can make much better resource co-allocation decisions.

V. The Integrated GRACCE Metascheduling Architecture

A Grid Metascheduler is "one level of grid middleware that discovers, evaluates and co-allocates resources for grid jobs, and coordinates activities between multiple heterogeneous schedulers that operate at local or cluster level". According to this definition, a metascheduler should have two main capabilities: the "scheduling" capability that allows it to co-allocate resources for applications requiring collaboration between multiple sites, and the "meta" capability to negotiate with local schedulers to satisfy global grid requests. In the survey of related work in Section VI, we show that current efforts to develop a metascheduler [34], [42], [1], [25] emphasize the "meta" or brokering functions. Those efforts with some focus on scheduling capabilities address the specific issues of grid co-scheduling; they include DUROC and GARA for co-allocation and reservation [14], [21], SNAP for negotiation [22], and Pegasus for planning [9]. It is very hard to integrate these solutions to produce a full-featured grid metascheduler. Some workflow systems have scheduling features [24], [17], [6], [9], but they do not provide the complete co-scheduling functionality.

There are two basic strategies for scheduling tasks in workflow applications: just-in-time scheduling and look-ahead scheduling. A just-in-time scheduler allocates resources for each individual task when it is ready to start. The allocation process is a one-time resource discovery and match-making activity. Look-ahead scheduling plans the execution of all or a subset of tasks and makes allocation decisions for them in advance. For heavily-loaded environments where resources are not immediately available, such planning together with resource reservation can greatly help to reduce the job queue waiting time. But if using a just-in-time scheduler, it would be hard to find resources for a newly submitted job in such situations. So just-in-time scheduling is only suitable in lightly-loaded environments or non-prime periods. The GRACCE metascheduler design is based on look-ahead scheduling.

The GRACCE Metascheduling Architecture

GRACCE provides support for workflow-orchestrated metascheduling and defines an architecture to implement a look-ahead scheduling and execution system. The architecture addresses the issues of metascheduling, workflow coordination and workflow job execution, and integrates their solutions into a middleware platform. From this platform, end users can build an application-specific grid environment to manage a grid application in its entire life cycle.

As shown in Figure 2, the GRACCE metascheduling architecture has three subsystems, the `Metascheduler`, the `GridDAG` workflow engine, and the `EPExec` runtime system. It employs the concept of an Execution Plan for a workflow job. The **Execution Plan (EP)** contains the scheduling decisions for workflow tasks and the mechanisms to handle task dependencies. The `EP` is generated by the `Metascheduler` in the scheduling process, and is used by the `GridDAG` to coordinate task dependencies. `EPExec` submits workflow tasks to their allocated resources and manages their execution according to the `EP`.

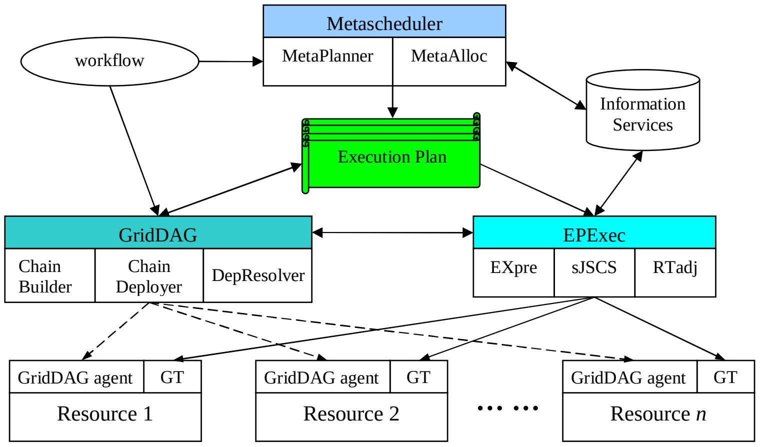

Fig. 2. The GRACCE Metascheduling Architecture

The **Metascheduler** has two components, `MetaPlanner` and `MetaAlloc`. It plans job execution and co-allocates resources for workflow tasks. `MetaPlanner` predicts the execution scenario for each task, which is about when and how the task should be launched. `MetaAlloc` discovers suitable resources, negotiates the resource provision and makes reservation with resource providers. The whole metascheduling process is based upon the job workflow; the decisions made are used to create the job `EP`. The `Metascheduler` is described in detail in Section V-A.

GridDAG is an event-driven workflow coordination system. At the scheduling stage, `GridDAG` decides how to handle dependencies and determines the event activities that are involved in the handling. These decisions are appended to the job `EP`. During job execution, `GridDAG` coordinates the execution of dependent tasks by handling and resolving task dependencies. In Section V-B, `GridDAG` components and event mechanisms are further discussed.

EPExec is a runtime execution system for workflow jobs. Given a job `EP`, `EPExec` submits task jobs to the allocated resources, and monitors and manages the execution of these tasks. `EPExec` sends events related to file availability or to the status change of task execution to `GridDAG` for the purpose of handling task dependencies. During execution, `EPExec` may adjust the `EP` according to the real execution scenario. `EPExec` is discussed in more detail in Section V-C.

The life-cycle of a workflow job in the GRACCE metascheduling architecture is described briefly below:

1) Users submit a workflow job specified by a GAMDL document to GRACCE.
2) The `Metascheduler` plans the execution of the job and allocates resources for the individual tasks. It writes the decision details into the job `EP`.
3) The job `EP` is forwarded to `GridDAG` which will decide on and set up the mechanisms of dependency handling; these details are added to the job `EP`.
4) According to the job `EP`, `EPExec` submits the first task of the workflow to its allocated resource and monitors it, thus begins the execution cycle of the job.

5) During job execution, `GridDAG` handles task dependencies based on the workflow and the job `EP`. When all dependencies of a task are resolved, `GridDAG` notifies `EPExec` to submit it to its allocated resource.

A. The GRACCE Metascheduler

GRACCE's `Metascheduler` has two components, `MetaPlanner` and `MetaAlloc`. `MetaPlanner` predicts and identifies the execution window for each task, and `MetaAlloc` searches a list of candidate resources, negotiates and makes the necessary agreement with resource providers. The relative execution order of dependent tasks is maintained by the metascheduling process. For independent tasks, the `Metascheduler` also considers the possibility of executing them concurrently.

1) Execution Planning: Identify Task Execution Window: A task's execution window (`EW`) is a time frame for carrying out that task. `EWstart` is the `EW` start time, and `EWlength` denotes the `EW` length – `EWlength` is equal to the task wall-clock time plus a configurable buffer time. The `EW` of an ancestor task must finish before the start of its dependent tasks, but the `EW`'s of independent tasks can overlap.

Given a workflow job, the `Metascheduler` planning process identifies the `EW`s for each task using a breadth-first graph traversal algorithm. The algorithm starts with the allocation of resources for the first task of the workflow by `MetaAlloc`. When resources are allocated, `MetaAlloc` also identifies the task `EW`. Then, the `Metascheduler` processes the child tasks of the first task. First, `MetaAlloc` discovers a list of candidate resources for each child task and calculates the cost of dependency handling between the resource(s) for the parent task and the candidate resources for child tasks. Secondly, `MetaPlanner` predicts the task `EW` for each of the candidate resources. The `EWstart` is calculated by adding the `EWstart` and `EWlength` of the parent task as well as the time required for dependency handling. Thirdly, the task `EW` predicted for each candidate resource are processed again by `MetaAlloc`, which will allocate the best resource for the task and determine its `EW`. The `Metascheduler` then moves on to process other tasks.

Since this is a look-ahead scheduling algorithm, it requires the specification of a task's wall-clock time and input/output file sizes. If they are not supplied by users, the `MetaPlanner` predicts them based on the task execution history or profiling information, which are included in the job's GAMDL specification. For recurring jobs, such as AQF, users normally provide historical or profile data to characterize the tasks' execution on different resources.

2) Resource Co-Allocation, Negotiation and Reservation: `MetaAlloc` allocates computational resources for workflow tasks in a sequence of resource discovery, negotiation, and reservation. During resource discovery, `MetaAlloc` queries the Grid Information Services for resources that satisfy the task resource requirements and are available during its `EW`. Firstly, resources are selected by a simple match-making of each attribute of a task's specification with static resource information. The resources on which the task is able to run are further evaluated according to their runtime information. Then, the selected resources are checked for their availability during the task `EW`, and `MetaAlloc` finally identifies a list of candidate resources. In the negotiation and reservation stage, `MetaAlloc` requests reservation for the candidate resources during a tasks `EW`. If the local schedulers grant the requests, `MetaAlloc` chooses the one that can provide the earliest `EW` for the task. A reservation ID is returned to be used to later access the reservation. If no reservation could be made on any of the candidates, grace periods are added to the `EW` and `MetaAlloc` again requests reservation for other wall-clock periods within the `EW` until a reservation is made. If `MetaAlloc` cannot reserve any resource for the task, `Metascheduler` stops to work on this task and forwards the partial `EP` to `EPExec` to launch the job. During job execution, `MetaAlloc` periodically attempts to allocate resources for this task.

B. The GridDAG Workflow System

`GridDAG` is our event-driven workflow system; it is able to coordinate the scheduling and execution of the dependent tasks of a workflow job. Compared with other workflow enacting engines, `GridDAG` is a pure coordination system, without any execution or monitoring functionalities, which are provided by `EPExec` in GRACCE. This gives `GridDAG` the flexibility to integrate with various remote execution and monitoring utilities. Different coordination mechanisms can be developed in `GridDAG` without necessitating additional effort to integrate them with other GRACCE subsystems.

1) The GridDAG Eventing Mechanisms: Events are notifications of a status change of task executions or file transfers, data availabilities, or other situations defined by users, such as for resource accounting purposes. An event producer detects certain situations or a status change, generates the corresponding event messages and distributes them. An event consumer receives an event message and invokes the event handlers. The `GridDAG` event mechanism is based on the WS-Notification standard [47], so event messages are XML documents – which allows the implementations to be platform-neutral in distributed heterogeneous environments.

Four components in `GridDAG` support the eventing mechanisms: the event chain builder, chain deployer, `GridDAG` agent, and `DepResolver`. The chain builder reads the job `EP` forwarded from the `Metascheduler` and generates the event chains according to the `EP`. An event chain is an ordered sequence of events from the participating producers to consumers. The chain deployer sends subscription requests to producers. A `Subscription` represents the relationship between a consumer, producer, and related event messages. These relationships constitute the runtime event chains of a workflow job. `GridDAG` agents coordinate the runtime event activities in each grid resource. Firstly, as a producer, `GridDAG` agents detect events occurring on the host resources and send out event messages. Secondly, as a consumer, `GridDAG` agents receive event messages from other agents or `EPExec` and take actions accordingly. `DepResolver` is the overall coordinator of dependency handling and resolving. `DepResolver` keeps track of the states of task dependencies and decides whether all of the dependencies are resolved.

2) Data Dependency Handling: Using the `GridDAG` eventing mechanism to handle data dependency, file transfers can be in either destination-pull (D-P) or source-push (S-P) mode. The event sequences for these two modes are shown in Figure 3. In the D-P mode, when the `GridDAG` agent on the source resource detects that files are available (1), it sends a corresponding event to the destination `GridDAG` agents (2). The destination `GridDAG` agents fetch the files (3) and send events to `DepResolver` notifying it of file arrivals (4). For the S-P mode, when files are available (1), the source `GridDAG` agent transfers them to the destination resources (2) and sends an event to the destination `GridDAG` agents and to `DepResolver` indicating that the intermediate files have been transferred (3). We expect that the D-P mode works better when multiple destinations are waiting for the same set of data. the S-P mode is suitable for situations where and data production and movement can be pipelined.

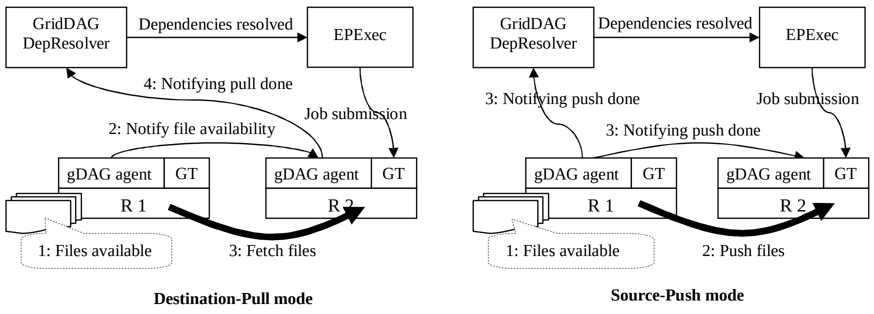

Fig. 3. Event Sequence in File Transfer

C. EPExec Runtime system

`EPExec` (`EP` Executer) is the runtime execution system for workflow jobs according to the job `EP`. `EPExec` has three components, `EXEpre`, `sJSCS`, and `RTadj`, to provide the functionalities of job submission, job monitoring, and the runtime adjustment of a job `EP`.

`EPExec` is implementation-independent of the `Metascheduler` and `GridDAG` subsystems and communicates with them via platform-neutral event messages. This ensures it to be flexible enough to integrate with various middleware packages. Different `EPExec`'s can be developed to support different methods of job submission and remote execution without requiring any changes to the `Metascheduler` and the `GridDAG`.

1) Execution Preparation: `EPExec`'s execution preparation (`EXEpre`) adds the the required information for job submission and workflow control to the job `EP`. The details depend on the grid middleware that it is developed on. Assuming that Globus GRAM is responsible for job submission and `GridDAG` for workflow coordination, its work can be summarized as follows:

- **Preparation for GRAM job submission:** `EXEpre` parses the tasks specification, generate a Globus RSL file, and specifies the locations and names of input and output files on the selected resources for the task.
- **Preparation for dependency handling:** `EXEpre` forwards the job `EP` to `GridDAG` which sets up the event chains and configures the event consumers and producers, as discussed above.

2) EP Execution and Monitoring: `EPExec` starts the workflow execution by submitting the job corresponding to the first task to its allocated resources according to the job `EP`. `sJSCS` (simple Job Submission and Control Service) is a utility to respond to such submission requests from `EPExec`. It calls the remote execution functions, such as Globus *globus-job-submit* to submit a single-executable job. The job is submitted using its resource reservation ID; this ensures the task is launched within its `EW`. A successful submission returns a global job ID, which `EPExec` uses for job monitoring and control.

`EPExec` monitors task executions in both passive-notification (P-N) mode and active-checking (A-C) mode. In the P-N mode, `EPExec` relies on the event messages about job status change to track the job. These messages are sent by the `GridDAG` agent on the resource where the job executes. In the A-C mode, `EPExec` calls `sJSCS` to query the current state of job execution. The P-N mode alleviates `EPExec` from the frequent calling of `sJSCS`; but `EPExec` may lose track of the job if the event mechanism fails. So normally, both the P-N and A-C modes are enabled in `EPExec` for a close monitoring of the job.

3) Runtime Adjustment: `EPExec` coordinates task executions so that the executions follow the `EP`. But if a task completes after its `EW`, the `RTadj` (Runtime Adjuster) component of `EPExec` may take actions to adjust the `EP` or to make up the delay. In most situations, those tasks that depend on the late task can be started within their `EW`'s and `RTadj` does not need to adjust them. But if the late completions cause the expiration of reservations of the dependent tasks and they cannot be started in their `EW`'s, `RTadj` uses the following strategy to try to make up the delay:

First, `EPExec` submits these tasks to their allocated resources without using reservation. The jobs may be held in the resource local queues. `RTadj` then requests `Metascheduler` to discover alternative resources for these tasks. If suitable resources are discovered and allocated, `EPExec` submits copies of these tasks to these resources. During execution, `EPExec` identifies the copy that it thinks will complete first and kills the others. Thus `RTadj` does its best to make up for the lost time in past job execution and to minimize the negative impacts on the execution of later tasks.

If it seems impossible to follow the initial job `EP`, `RTadj` will consider re-scheduling the rest of the tasks. In this case, `RTadj` forwards the job sub-workflow to `Metascheduler` to reschedule. Re-scheduling may cause low resource usage or wastage because of the cancellation of prior reservations. The `Metascheduler` tries to avoid this situation by scheduling other jobs onto these reservations if possible.

VI. Related Work

Various projects attempt to solve the problems of grid application modeling and grid scheduling, and provide certain features that are also incorporated into GRACCE. In this section we survey related work in three areas, grid application modeling and workflow description, grid metascheduling, and workflow engines with scheduling features.

A. Application Modeling and Workflow Descriptions

Many efforts have aimed to support the specification of workflows for grid applications. Condors DAG [35] enables the use of a graph that depicts parent-child relationships to specify workflow in applications. Gridbus workflow [19] describes application DAGs using an XML-based language (xWFL). But both Condor DAG and Gridbus workflow do not handle advanced control logic.

The Triana [6] workflow definition language specifies component-based application workflows. Components are units of executions implemented as Java classes. Non-Java applications have to be wrapped in Java in order to use Triana.

Business Process Execution Language (BPEL) [33], which combines IBM's WSFL [46] and Microsoft's XLANG [27] standards, is an XML-based workflow definition language to describe enterprise business processes. BPEL works at web services level; additional extensions are needed to make it easy to use for end users.

Abstract Grid Workflow Language (AGWL) [28] “describes” application control flow using constructs of an imperative programming style and is able to describe various complex control logic. But users are required to specify the details of the control flow, such as stating which modules can be executed in parallel. For dataflow applications, users have to transform the dataflow into control-flow to use AGWL.

Karajan of the Globus CoG Kit [39] is a powerful workflow system that includes a workflow description language and a workflow engine. Karajan uses similar constructs as AGWL to describe task dependencies, execution relationships and control logic. To use Karajan, users must know the task execution hosts in advance, as well as the source and destination hosts for file transfers. So Karajan workflow language is more suitable for low-level workflow execution control than for high-level application modeling.

B. Resource Co-Allocation and Grid Metascheduling

There are also many systems that address specific issues of grid co-scheduling. Globus GRAM [23] and RSL [44] are the early, de-facto standards for providing solutions for secure job execution in metacomputing environments. DUROC [21] is an early effort to address the issues of resource co-allocation in the context of Globus and RSL. Globus GARA [14], Maui Silver [42] and the architecture defined in [11] introduce advanced reservation into the GRAM co-allocation architecture [31]. SNAP [22], which extends Globus’ GRAM and GARA, proposes a service negotiation protocol for grid scheduling. Pegasus [9] casts workflow job scheduling issues into the form of AI planning.

The Grid Scheduling Use Case Document from the GGF Grid Scheduling Architecture Research Group (GSA-RG) [37] collects scheduling scenarios for several different types of applications, including complex workflow applications, component-based applications, application-oriented scheduling in a knowledge grid (K-Grid), agent-based scheduling, etc. Here, the most relevant two cases, workflow scheduling and K-Grid scheduler, are outlined.

The use case for workflow scheduling aims to define the functional requirements, the process and other related issues of scheduling workflow applications in grid environments. This use case mentions that an execution schedule should be generated for workflow applications, but does not address how to generate such a schedule, which we believe is the most important step of the workflow scheduling process.

The K-Grid scheduler [2] described in the second use case is a performance-oriented resource allocation service for knowledge discovery and data mining applications. It predicts the computational and I/O cost for each allocation and makes the best-possible decisions based on this estimation. But the K-Grid scheduler does not reserve resources for applications and relies on the grid resource discovery services to find the best available resources.

The Community Scheduler Framework (CSF) [34] implements a set of grid services which provide basic capabilities for grid job submission and resource reservation. These services, developed as wrappers for some local scheduler utilities, provide a good starting point to develop a brokerage system. But CSF services only cater for single executable jobs and lack functionalities for grid co-scheduling.

Maui Silver [42] is an advanced reservation-based grid scheduler which allows a single job to be scheduled across distributed clusters. Silver relies on the local scheduler to specify and coordinate the job workflow, which limits it usage to simple workflow applications.

Nimrod/G [25] is a resource management system with a focus on computational economy and schedules tasks based on their deadlines and budgets. Nimrod/G also addresses issues of scheduling single jobs, and does not address the requirements of workflow applications.

MARS [1] proposes an on-demand scheduler which discovers and schedules the required resources for a critical-priority task to start immediately. MARS uses a forecasting strategy to predict runtime resource parameters, such as queue lengths, utilization, etc.

C. Workflow Engines With Scheduling Features

Workflow related activities have been extensively studied in [18]. Most workflow systems, such as DAGMan [35], myGrid Taverna [29], GridAnt and Karajan [39] use a very simple strategy to schedule workflow tasks, that is, they simply submit tasks when their dependencies are resolved. These workflow systems do not have scheduling and resource allocation features. So, in the following, we only outline those workflow systems that do have certain scheduling features.

The Triana [6] workflow engine is a decentralized, event-based system. In Triana, users specify the distribution policies for mapping workflow tasks onto resources, which could be parallel or pipelined. Triana assumes that resources are available for these tasks, so it does not allocate resources for them.

The scheduling algorithm of ASKALON [24], [26] and in [20] are similar to our approach to planning the workflow executions. These algorithms traverse the workflow graph or subgraph and generate the execution schedule of the workflow tasks. But they do not specify how the execution time for each workflow task is determined. As we mentioned before, such information and its accuracy are a vital part for planning and look-ahead scheduling.

In GridFlow [17], a workflow is executed according to a simulated schedule. If large delays occur in some sub-workflows, the rest or all of the workflow may be sent back to the simulation engine and rescheduled. The concept of a simulated schedule is similar to the execution plan in our architecture. But GridFlow does not address resource co-allocation and reservation issues in the simulated schedule.

Pegasus [9] constructs a job execution DAG with scheduling information from the application DAG logic. This process includes querying Globus MDS to find resources for computation and data movement, and querying a Globus replica location service to locate data replicas. Pegasus submits this DAG to Condor DAGMan for execution, and hence cannot co-exist with other local schedulers.

VII. Conclusions

AQF is a typical application that requires multiple computational resources collaboratively available to produce air quality forecasting results in a timely fashion. To satisfy such requirement in grids, many application specific issues need to be addressed in order to provide a production-quality environment. This paper presents our solutions to create such an environment for domain scientists to enable their applications on grid. Driven by our AQF application and based on our experiences of AQF deployment on the UH campus grid, the ongoing GRACCE project aims to provide an end-to-end solution for automatic application execution on grid environments. Using the GRACCE middleware, domain scientists are only required to specify the application logic and resource requirements; GRACCE is responsible for allocating grid resources for the application, for launching the application, and for the delivery of the results back to the users.

Currently, there are two major efforts in GRACCE: GAMDL and the GRACCE metascheduler. GAMDL provides an intuitive means to model an application for grid deployment. It describes application dataflows and allows control logic to be directly expressed, without additional translation from the dataflow. GAMDL associates job specifications with workflow descriptions so that there is no need to specify application-level resource multirequests. GAMDL also allows the specification of job execution history and profiling data, which are used by the `Metascheduler` to predict workflow execution behavior.

Our metascheduling efforts in GRACCE study the grid metascheduling issues for workflow jobs, define the term “Grid Metascheduler” and a metascheduling architecture for workflow applications. The architecture provides and integrates solutions to various grid scheduling related issues. The three subsystems in the architecture, the `Metascheduler`, the `GridDAG` workflow engine, and the `EPExec` runtime system constitute an extensible platform for the integration of grid middleware and applications. Grid middleware solutions can be easily interfaced with one of the subsystems without changing the other subsystems; and from this platform, end users without any in-depth grid knowledge are able to easily deploy their applications on the grid.

Acknowledgments

We wish to thank Dr. Daewon Byun for supporting our efforts to explore AQF, and Dr. Jiwen He for his help in understanding MM5 and its deployment. We would also like to thank the HPCC team for their help in building a testbed to facilitate this initiative. Finally, we thank our teammate Babu Sundaram, and Andre Merzky from Vrije Universiteit for their comments on earlier versions of this paper.

References

[1] A. Bose, B. Wickman, and C. Wood, *MARS: A Metascheduler for Distributed Resources in Campus Grids*, Proceedings of Fifth IEEE/ACM International Workshop on Grid Computing, 2004

[2] A. Pugliese, and D. Talia, *Application-oriented scheduling in the Knowledge Grid: a model and architecture*, International Conference on Computational Science and its Applications (ICCSA), 2004. LNCS 3044, pp. 55-65, Springer-Verlag, Berlin, Germany, 2004

[3] B. M. Chapman, P. Raghunath, B. Sundaram, and Y. Yan, *Air Quality Prediction in a Production Quality Grid Environment*, Engineering the Grid: Status and Perspective, edited by J. Dongarra, et al, Spring 2005

[4] B. M. Chapman, H. Donepudi, J. He, Y. Li, P. Raghunath, B. Sundaram and Y.Yan, *Grid Environment with Web-Based Portal Access for Air Quality Modeling*, Parallel and Distributed Scientific and Engineering Computing, Practice and Experience, 2003

[5] B. M. Chapman, Y. Li and B. Sundaram, and J. He, *Computational Environment for Air Quality Modeling in Texas*, Use of High Performance Computing in Meteorology, World Scientific Publishing, 2003

[6] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor and I. Wang, *Programming Scientific and Distributed Workflow with Triana Services*, Special Issue of Concurrency and Computation: Practice and Experience, 2005

[7] D. W. Byun, J. Pleim, R. Tang, and A. Bourgeois, *Meteorology-Chemistry Interface Processor (MCIP) for Models-3 Community Multiscale Air Quality (CMAQ) Modeling System*, Washington, DC, U.S. Environmental Protection Agency, Office of Research and Development, 1999.

[8] D. W. Byun, and K. Schere, *EPA's Third Generation Air Quality Modeling System: Description of the Models-3 Community Multiscale Air Quality (CMAQ) Model*, Journal of Mech. Review, 2004

[9] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny, *Pegasus: Mapping Scientific Workflows onto the Grid*, Across Grids Conference 2004, Nicosia, Cyprus

[10] G. Grell, J. Dudhia, and D. Stauffer, *A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5)*, NCAR Tech Notes, TN-398+STR

[11] G. Mateescu, *Quality of Service on the Grid Via Metascheduling with Resource Co-Scheduling and Co-Reservation*, International Journal of High Performance Computing Applications, Vol. 17, No. 3, 209-218, 2003

[12] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, *A Java Commodity Grid Kit*, Concurrency and Computation: Practice and Experience, vol. 13, no. 8-9, pp. 643-662, 2001

[13] I. Foster and C. Kesselman, *Globus: A metacomputing infrastructure toolkit*, International Journal of Supercomputer Applications, Summer 1997

[14] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation.* Intl Workshop on Quality of Service, 1999

[15] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, *A Security Architecture for Computational Grids*, ACM Conference on Computers and Security, 83-91, 1998

[16] I. Foster, C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, International Journal of High Performance Computing Applications, 15 (3). 200-222. 2001

[17] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd, *GridFlow: Workflow Management for Grid Computing*, In Proceedings of 3rd International Symposium on Cluster Computing and the Grid, at Tokyo, Japan, May 12-15, 2003

[18] J. Yu, and R. Buyya, *A Taxonomy of Workflow Management Systems for Grid Computing*, Technical Report, GRIDS-TR-2005-1, University of Melbourne, Australia, March, 2005

[19] J. Yu, and R. Buyya, *A Novel Architecture for Realizing Grid Workflow using Tuple Spaces*, Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Pittsburgh, 2004

[20] J. Yu, R. Buyya and C. K. Tham, *QoS-based Scheduling of Workflow Applications on Service Grids*, Technical Report, GRIDS-TR-2005-8, University of Melbourne, Australia, June 9, 2005

[21] K. Czajkowski, I. Foster, and C. Kesselman, *Resource Co-Allocation in Computational Grids*, Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8), pp. 219-228, 1999

[22] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke, *SNAP: A Protocol for negotiating service level agreements and coordinating resource management in distributed systems*, Lecture Notes in Computer Science, 2537:153-183, 2002

[23] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, *A Resource Management Architecture for Metacomputing Systems*, Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998

[24] M. Wieczorek, R. Prodan and T. Fahringer, *Scheduling of Scientific Workflows in the ASKALON Grid Environment*, ACM SIGMOD Record Journal, 2005

[25] R. Buyya, D. Abramson, and J. Giddy, *Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid*, The 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), May 2000

[26] R. Prodan and T. Fahringer, *Dynamic Scheduling of Scientific Workflow Applications on the Grid using a Modular Optimisation Tool: A Case Study*, 20th Symposion of Applied Computing, 2005

[27] S. Thatte, *XLANG-Web Services for Business Process Design*, Microsoft Corporation, 2001

[28] T. Fahringer, J. Qin and S. Hainzer, *Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language*, Proceedings of Cluster Computing and Grid, 2005

[29] T. Oinn, et al, *Taverna: Lessons in creating a workflow environment for the life sciences*, Concurrency and Computation: Practice and Experience Grid Workflow Special Issue, 09, 2002

[30] W. F. Dabberdt, M. A. Carroll, D. Baumgardner, G. Carmichael, and R. Cohen *Meteorological research needs for improved air quality forecasting*, The 11th Prospectus Development Team of the U.S. Weather Research Program, 2004

[31] W. Smith, I. Foster, and V. Taylor, *Scheduling with Advanced Reservations.* Proceedings of the IPDPS Conference, May 2000

[32] Z. Adelman and M. Houyoux, *Processing the National Emissions Inventory 96 (NEI96) version 3.11 with SMOKE*, The Emission Inventory Conference: One Atmosphere, One Inventory, Many Challenges, 1-3 May, Denver, CO, U.S. Environmental Protection Agency, 2001

[33] BPEL4WS: Business Process Execution Language for Web Services v1.0,
`http://www.106.ibm.com/developerworks/webservices/library/wsbpel`

[34] Community Scheduler Framework,
`http://www.platform.com/products/Globus`

[35] DAGMan (Directed Acyclic Graph Manager),
`http://www.cs.wisc.edu/condor/dagman`
[36] GRACCE: Grid Application Coordination, Collaboration and Execution,
`http://www.cs.uh.edu/~yanyh/gracce`
[37] Grid Scheduling Architecture Research Group,
`https://forge.gridforum.org/projects/gsa-rg`
[38] High Performance Computing Center, University of Houston,
`http://www.hpcc.uh.edu/`
[39] Java CoG Kit Karajan/Gridant Workflow Guide,
`http://www.cogkit.org/release/4_0_a1/manual/workflow.pdf`
[40] Job Submission Description Language, GGF,
`https://forge.gridforum.org/projects/jsdl-wg`
[41] Load Sharing Facility, Resource Management and Job Scheduling System, `http://www.platform.com/products/HPC/`
[42] Maui Moab Grid Scheduler (Silver),
`http://www.clusterresources.com/products/mgs`
[43] Sun Grid Engine, Sun Microsystems,
`http://gridengine.sunsource.net/`
[44] The Globus Resource Specification Language RSL v1.0,
`http://www-fp.globus.org/gram/rsl_spec1.html`
[45] The University of Houston's Sun Microsystems Center of Excellence in Geosciences, `http://www.suncoe.uh.edu`
[46] Web Services Flow Language (WSFL),
`http:///www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`
[47] Web Services Notification,
`http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn`

APPENDIX A: GAMDL Examples

In this appendix, the core GAMDL description of the AQF application is listed in A.1 - A.3. A.1 reproduces the contents of the AQF `application` document, which includes four files to describe the AQF entities. A.2 gives the contents of the "*aqfexe.xml*" file which specifies the AQF modules. A.3 lists the file "*aqfmddep.xml*" that specifies AQF module relationships as `PCnRship` elements. In A.4, we give an example of an AQF workflow in an `AppRun` document. A.5 provides the contents of a workflow example that has control logic.

GAMDL uses **Multiple-value properties** (mvproperties) to specify similar application entities. An mvproperty is a property that may have multiple values. It is defined as $mvpName = \{v0, v1, ..., vn\}$, and is referenced by $\$mvpName$. $\#mvpName$ returns the number of values defined. A reference to $mvpName$ replicates the referencing sentence $\#mvpName$ times; in each replica, the reference is replaced with a distinct one of its values. For example, if we define $dmsz = \{36K, 12K, 4K\}$, $day = \{1d, 2d\}$, the sentence $aqf - mm5 - \$\{dmsz\} - \$\{day\}$ can represent all 6 instances ($\#md * \#dmsz * \#day$) of the AQF MM5 modules in Figure 1. In an XML document, the replication of an mvproperty reference is per-element based. When the GAMDL parser encounters a reference to an mvproperty, it replicates the nearest outer element that contains the reference. This element is called the containing element of the mvproperty reference. The processor does not recursively process the same references in the child element of the containing element, instead, it instantiates all references to the mvproperty in a replicate element with the same value. We refer readers to [36] for the details of parsing mvproperty in XML.

The mvproperty element in each of the GAMDL document includes a file (uhaqf.mvproperties) to specify the mvproperties used; it defines four mvproperties as follows:

```
md={mm5, smoke, cmaq}
dmsz={36K, 12K, 4K}
day={d1, d2}
vdmsz={12k, 4k}
```

A.1: The *application* Document

```
<application uid="uhaqf" name="UH-AQF-2005" ... >
  <mvproperty file="uhaqf.mvproperties"/>
  <description>UH AQF project in 2005 using GRACCE</description>
  <version name="UH-AQF-2005" major="2005" minor="06" build="03"/>

  <reference href="aqffiles.xml"/>
  <reference href="aqfexe.xml"/>
  <reference href="aqfmd.xml"/>
  <reference href="aqfmddep.xml"/>
</application>
```

A.2: The *appModules* Document

```
<appModules>    <mvproperty file="uhaqf.mvproperties"/>
  <module uid="mm5-36k" name="MM5 36k for three days">
    <inputFiles>
      <ref uid="mm5-36k-in1"/>
      <ref uid="mm5-36k-in2"/>
      <ref uid="mm5-36k-in3"/></inputFiles>
    <outputFiles>
      <ref uid="mm5-36k-out1"/>
      <ref uid="mm5-36k-out2"/>
      <ref uid="mm5-36k-out3"/></outputFiles>
    <jobSpec name="mm5-36k job spec">
      <launcher>/bin/mm5-36k.sh</launcher>
      <directory>/aqf/mm5/</directory>
      <arguments>quiet</arguments>
      <stdin>/dev/null</stdin>
      <stdout>/aqf/log/mm5-36k/stdout.log</stdout>
      <stderr>/aqf/log/mm5-36k/stderr.log</stderr>
      <indicator name="mm5-36kindicator.sh" />
      <preprocesser name="premm5-36k.sh" />
      <postprocesser name="postmm5-36k.sh" />
      <cleaner name="mm5-36kcleaner.sh" />
      <jobType>mpi</jobType>
      <sysArch uid="Linux-x86" uname_m="i686" uname_s="Linux"/>
      <maxNumberOfCPU>24</maxNumberOfCPU>
      <minNumberOfCPU>8</minNumberOfCPU>
    </jobSpec></module>

  <module uid="mm5-${vdmsz}">
    <inputFiles>
      <ref uid="mm5-${vdmsz}-in1"/>
      <ref uid="mm5-${vdmsz}-in2"/>
      <ref uid="mm5-${vdmsz}-in3"/></inputFiles>
    <outputFiles>
      <ref uid="mm5-${vdmsz}-out1"/>
      <ref uid="mm5-${vdmsz}-out2"/>
      <ref uid="mm5-${vdmsz}-out3"/></outputFiles>
    <jobSpec name="mm5-${vdmsz} job spec">
      <launcher>/bin/ mm5-${vdmsz}.sh</launcher>
      <directory>/aqf/mm5/</directory>
      <indicator name="mm5-${vdmsz}indicator.sh" />
      <preprocesser name="premm5-${vdmsz}.sh" />
      <postprocesser name="postmm5-${vdmsz}.sh" />
      <cleaner name="mm5-${vdmsz}cleaner.sh" />
      <jobType>mpi</jobType>
      <maxNumberOfCPU>24</maxNumberOfCPU>
      <minNumberOfCPU>4</minNumberOfCPU>
    </jobSpec></module>

  <module uid="smoke-${dmsz}-${day}" >
    <inputFiles><ref uid="smoke-${dmsz}-${day}-in1"/></inputFiles>
    <outputFiles><ref uid="smoke-${dmsz}-${day}-out1"/></outputFiles>
    <jobSpec name="smoke-${dmsz}-${day} job spec">
      <launcher>/bin/smoke-${dmsz}-${day}.sh</launcher>
      <directory>/aqf/smoke/</directory>
      <arguments>quiet</arguments>
      <stdout>/aqf/log/smoke-${dmsz}-${day}/stdout.log</stdout>
      <stderr>/aqf/log/smoke-${dmsz}-${day}/stderr.log</stderr>
      <jobType>single</jobType>
      <sysArch uid="Linux-x86" uname_m="i686" uname_s="Linux"/>
    </jobSpec></module>

  <module uid="cmaq-${dmsz}-${day}">
    <inputFiles>
      <ref uid="cmaq-${dmsz}-${day}-in1"/>
      <ref uid="cmaq-${dmsz}-${day}-in2"/></inputFiles>
    <outputFiles>
      <ref uid="cmaq-${dmsz}-${day}-out1"/>
      <ref uid="cmaq-${dmsz}-${day}-out2"/></outputFiles>
    <jobSpec name="cmaq-${dmsz}-${day} job spec">
      <launcher>/bin/ cmaq-${dmsz}-${day}.sh</launcher>
      <directory>/aqf/cmaq/</directory>
      <stdout>/aqf/log/cmaq-${dmsz}-${day}/stdout.log</stdout>
      <stderr>/aqf/log/cmaq-${dmsz}-${day}/stderr.log</stderr>
      <jobType>mpi</jobType>
      <sysArch uid="Linux-x86" uname_m="i686" uname_s="Linux"/>
    </jobSpec></module>

  <module uid="postv-${vdmsz}-${day}" >
```

```
    <inputFiles><ref uid="postv-${vdmsz}-${day}-in1"/></inputFiles>
    <outputFiles><ref uid="postv-${vdmsz}-${day}-out1"/></outputFiles>
    <jobSpec name="postv-${vdmsz}-${day} job spec">
      <launcher>/bin/ postv-${vdmsz}-${day}.sh</launcher>
      <exeUidRef>uhaqf-postv</exeUidRef>
      <arguments>quiet</arguments>
      <stdout>/aqf/log/postv-${vdmsz}-${day}/stdout.log</stdout>
      <stderr>/aqf/log/postv-${vdmsz}-${day}/stderr.log</stderr>
      <jobType>mpi</jobType>
      <sysArch uid="Linux-x86" uname_m="i686" uname_s="Linux" />
    </jobSpec></module>
</appModules>
```

A.3: The *appMdRships* Document

```
<appMdRships>
  <pCnRshipSet>   <mvproperty file="uhaqf.mvproperties"/>
    <PCnRship parentMdUidRef="mm5-36k">
      <childMd uidRef="mm5-12k">
        <viaPipe inFileUidRef="mm5-36k-out1" outFileUidRef="mm5-12k-in1"/>
        <viaPipe inFileUidRef="mm5-36k-out2" outFileUidRef="mm5-12k-in2"/>
        <viaPipe inFileUidRef="mm5-36k-out3" outFileUidRef="mm5-12k-in3"/>
      </childMd></PCnRship>

    <PCnRship parentMdUidRef="mm5-12k">
      <childMd uidRef="mm5-4k">
        <viaPipe inFileUidRef="mm5-12k-out1" outFileUidRef="mm5-4k-in1"/>
        <viaPipe inFileUidRef="mm5-12k-out2" outFileUidRef="mm5-4k-in2"/>
        <viaPipe inFileUidRef="mm5-12k-out3" outFileUidRef="mm5-4k-in3"/>
      </childMd></PCnRship>

    <PCnRship parentMdUidRef="mm5-${dmsz}">
      <childMd uidRef="smoke-${dmsz}-d1">
        <viaPipe inFileUidRef="mm5-${dmsz}-out1"
          outFileUidRef="smoke-${dmsz}-d1-in1"/></childMd>
      <childMd uidRef="smoke-${dmsz}-d2">
        <viaPipe inFileUidRef="mm5-${dmsz}-out2"
          outFileUidRef="smoke-${dmsz}-d2-in1"/></childMd>
      <childMd uidRef="smoke-${dmsz}-d3">
        <viaPipe inFileUidRef="mm5-${dmsz}-out3"
        outFileUidRef="smoke-${dmsz}-d2-in1"/></childMd></PCnRship>

    <PCnRship parentMdUidRef="smoke-${dmsz}-${day}">
      <childMd uidRef="cmaq-${dmsz}-${day}">
        <viaPipe inFileUidRef="smoke-${dmsz}-${day}-out1"
        outFileUidRef="cmaq-${dmsz}-${day}-in1"/></childMd></PCnRship>

    <PCnRship parentMdUidRef="cmaq-${dmsz}-d1">
      <childMd uidRef="cmaq-${dmsz}-d2">
        <viaPipe inFileUidRef="cmaq-${dmsz}-d1-out1"
          outFileUidRef="cmaq-${dmsz}-d2-in1"/></childMd></PCnRship>

    <PCnRship parentMdUidRef="cmaq-${dmsz}-d2">
      <childMd uidRef="cmaq-${dmsz}-d3">
        <viaPipe inFileUidRef="cmaq-${dmsz}-d2-out1"
          outFileUidRef="cmaq-${dmsz}-d3-in1"/></childMd></PCnRship>

    <PCnRship parentMdUidRef="cmaq-36k-${day}">
      <childMd uidRef="cmaq-12k-${day}">
        <viaPipe inFileUidRef="cmaq-36k-${day}-out2"
          outFileUidRef="cmaq-12k-${day}-in2"/></childMd>/PCnRship>

    <PCnRship parentMdUidRef="cmaq-12k-${day}">
      <childMd uidRef="cmaq-4k-${day}">
        <viaPipe inFileUidRef="cmaq-12k-${day}-out2"
          outFileUidRef="cmaq-4k-${day}-in2"/></childMd>/PCnRship>

    <PCnRship parentMdUidRef="cmaq-${vdmsz}-${day}">
      <childMd uidRef="postv-${vdmsz}-$day">
        <viaPipe inFileUidRef="cmaq-${vdmsz}-${day}-out1"
          outFileUidRef="postv-${vdmsz}-${day}-in1"/></childMd></PCnRship>
  </pCnRshipSet>
</appMdRships>
```

A.4: An *appRun* Document for uhaqf Application

```
<appRun uid="uhaqf-run" appUid="uhaqf" startTime="2005-04-16T15:23:15">
<mvproperty file="uhaqf.mvproperties"/>
  <modules>
    <ref uid="eta-download"/>
    <ref uid="mm5-${dmsz}"/>
    <ref uid="smoke-${dmsz}-${day}"/>
    <ref uid="cmaq-${dmsz}-${day}"/>
    <ref uid="postv-${vdmsz}-${day}"/></modules>
  <startMd><ref uid="eta-download"/></startMd>
</appRun>
```

A.5: An Example of Workflow with Loops and Conditional Branches

In the workflow of Figure 4, the module *md2* generates different output files (*F1*, *F2* or others) in different loops and these files are processed by module *md3, md4* or *md5*. The loop count is 100.

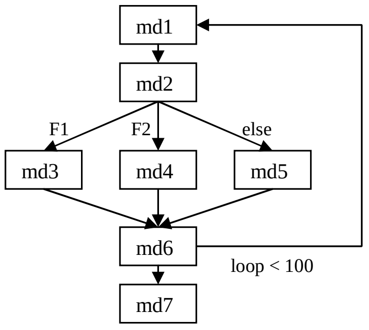

Fig. 4. A Workflow with Loop and Conditional Branches

In the GAMDL description shown in the following, module *md1* `postAssign`s a *loop* variable, whose initial value is *100* and stride is *-1*. The module *md2* `postAssign`s two variables, *F1recent* and *F2recent*. *F1recent* is set to *true* if file *F1* is generated by *md2* in the last execution, otherwise *F1recent* is set to *false*; *F2recent* is handled similarly with respect to file *F2*. The pipe condition for *md3-md2* `CPsRship` is set to "*pipe(F1)* && ${*F1recent*}", which is evaluated to *true* if *F1* is generated in the last execution and is available for piping in. The *if* conditions for *F2* pipe in *md4-md2* `CPsRship` and the else-pipe in *md5-md2* CPsRship are similar to *F1* pipe. Loop control is specified in *md1-md6* `CPsRship` of *md1* and *md6* using a null pipe with condition "${*loop*}*<100* ".

In this example, GAMDL uses some condition functions, such as *generated(F1)*, in a condition string. **A condition function** is a regular function (binary or script) that returns a boolean value and should not make any modification to its externals. In the following specification, the *pipe(fileName)* function checks whether a file can be piped in or not; The *generated(fileName)* function checks whether the module execution generates the specified file; the *defined(variableName)* function checks whether a variable is defined or not.

```
<application name="LoopCon Example" uid="loopcon" >
  <appModules>
    <module uid="md1">
      <postAssign name="loop" value="${loop}-1:100" if="defined(loop)"/>
    </module>
    <module uid="md2">
      <outputFiles>
        <ref uid="F1"/>
        <ref uid="F2"/>
        <ref uid="Fx"/></outputFiles>
      <postAssign name="F1recent" value="true:false" if="generated(F1)"/>
      <postAssign name="F2recent" value="true:false" if="generated(F2)"/>
    </module>
    <module uid="md3"><inputFiles><ref uid="F1"/></inputFiles></module>
    <module uid="md4"><inputFiles><ref uid="F2"/></inputFiles></module>
    ...
  </appModules>

  <appMdRships>
    <cPsRshipSet>
      <CPsRship childMdUidRef="md2">
        <parentMd uidRef="md1">
          <viaPipe> ... </viaPipe></parentMd></CPsRship>
      <CPsRship childMdUidRef="md3">           <!--md3-md2 CPsRship -->
```

```
        <parentMd uidRef="md2">
          <viaPipe if="pipe(F1) && ${F1recent}"
            inFileUidRef="F1" outFileUidRef="F1"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md4">            <!--md4-md2 CPsRship -->
        <parentMd uidRef="md2">
          <viaPipe if="pipe(F2) && ${F2recent}"
            inFileUidRef="F2" outFileUidRef="F2"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md5">            <!--md5-md2 CPsRship -->
        <parentMd uidRef="md2">
          <viaPipe if="!{F1recent} && !{F2recent}"
            inFileUidRef="Fx" outFileUidRef="Fx"/></parentMd></CPsRship>

      <mvproperty name="md345">
        <value>md3</value>
        <value>md4</value>
        <value>md5</value></mvproperty>

      <CPsRship childMdUidRef="md6">
        <parentMd uidRef="${md345}">
          <viaPipe if="" inFileUidRef="${md345}-out"
            outFileUidRef="${md345}-out"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md1">            <!--md1-md6 CPsRship -->
        <parentMd uidRef="md6">
          <viaPipe if=" ${loop} < 100" inFileUidRef="/dev/null"
            outFileUidRef="/dev/null"/></parentMd></CPsRship>

      <CPsRship childMdUidRef="md7">
        <parentMd uidRef="md6"><viaPipe ... /></parentMd></CPsRship>
    </cPsRshipSet>
  </appMdRships>
</application>
```