Support for Specification and Scheduling of Workflow Applications on the Grid

Yonghong Yan, Barbara M. Chapman
{*yanyh, chapman*}*@cs.uh.edu*

Department of Computer Science
University of Houston

Abstract

Large-scale applications are increasingly composed of a number of different components, which may interact in complex ways at run time. Such applications may consume substantial resources and are often loosely coupled; hence they are obvious candidates for grid computing. At the University of Houston, we have been working to develop and deploy a state-of-the-art Air Quality Forecast application in order to reliably predict atmospheric pollution in our region. This application has a complex workflow and non-trivial amounts of data must be transferred between different components during execution. In our search for gridware to support its automated scheduling and execution, we did not find a system that permitted its specification, scheduling and job launch as desired. To address this gap in functionality, we have created the dataflow language GAMDL to describe a workflow application's logic. In this presentation, we describe our application, its requirements, and the features of GAMDL as well as the metascheduling architecture being implemented to complete our support for the production of air quality forecasts for our local and state officials.

1 Introduction

Grid environments [5] are increasingly being used by domain scientists with large-scale applications, such as physics [17], climate modeling [1], or atmospheric study [18]. These applications are no longer being developed as monolithic codes, but incorporate multiple dependent modules, and entail the transfer and storage of a large amount of data. Enabling such an application on grid environments is much more complex than enabling an application that can be wrapped as a simple grid job. Several issues need to be addressed when deploying such applications, including description of application structure, integration of application execution with grid scheduling and workflow systems, and runtime coordination of application workflow, etc.

Modeling a domain application in grid environments concerns about how to describe application computational modules, data sets and module relationships so that application can be seamlessly integrated with grid middlewares. Current workflow description languages [6] model application control-flow in middleware level with terms like job or service, assuming users have knowledge of grid computing. Related workflow engines also have very limited or even no grid-level scheduling functionalities. Additional extensions are required to integrate with a grid metascheduler [12].

In this paper, we present a high-level abstract language for domain scientists to describe their applications for grid deployment and integration, Grid Application Modeling and

Description Languages (GAMDL). GAMDL describes the data-flow structures of domain complex problems, but allows the definition of control-flow within application data-flow using conditioned properties and conditioned pipes. GAMDL is intuitive and very easy to use for users without background of grid computing and is much more powerful than other languages to describe similar application entities using multiple-value properties.

GAMDL design is not purely language based, but with direct application support and support for grid middleware integration. GAMDL is driven, and used by a production application, UH AQF [11], and serve as basis for the integration between a grid metascheduler and workflow systems in GRACCE project [16]. GAMDL associates job specification within application description, addressing the issue of integrating grid application and job workflow in the level of application description. GAMDL also allows job execution history to be specified in application description, which may be utilized by grid metascheduler for resource co-allocation and execution prediction.

The papers is organized as follows. In next section, the motivating application (UH AQF) and GRACCE project are introduced, and requirements and related work for application modeling and description are studied. Section 3 defines GAMDL constructs and related schemas. Section 4 presents GAMDL module job specification language. In section 5, examples of using GAMDL are presented, including AQF example and an example of complex workflow. Finally, conclusions and future work are outlined in the last section.

2 Motivation

GAMDL is the result of the collaboration work between domain users and grid experts during the process of enabling AQF on UH campus grid in GRACCE project [2], and was designed to satisfy both the needs of domain scientists for a description method to easily present their applications and the needs of integration of grid middlewares and applications.

2.1 AQF Scientific Application and GRACCE Project

Air Quality Forecasting (AQF) application is an integrated computational model for regional and local air quality forecasts that is composed of three subsystems: the PSU/NCAR MM5 weather forecast model [4], the SMOKE emission system [13], and EPA's CMAQ chemical transport model [3]. AQF execution is a computational sequence of the three subsystems with increasing resolution and decreasing geographical boundaries. Fig 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 continental USA, the 12km provides data across the south central USA, and the 4km 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 their dependent files are transferred to the allocated resources when these files become available. To enable AQF-like applications on grid environments, GRACCE (Grid Application Coordination, Collaboration and Execution) project was proposed to develop a set of grid middlewares for grid application deployment. The vision of GRACCE is to provide domain scientists an application-specific grid environment, supporting from the management of an application and its dataset, to the automatic execution and viewing of results.

In GRACCE solutions, a metascheduler was defined to integrate domain applications and grid infrastructural middlewares into such a grid environment. Domain users are

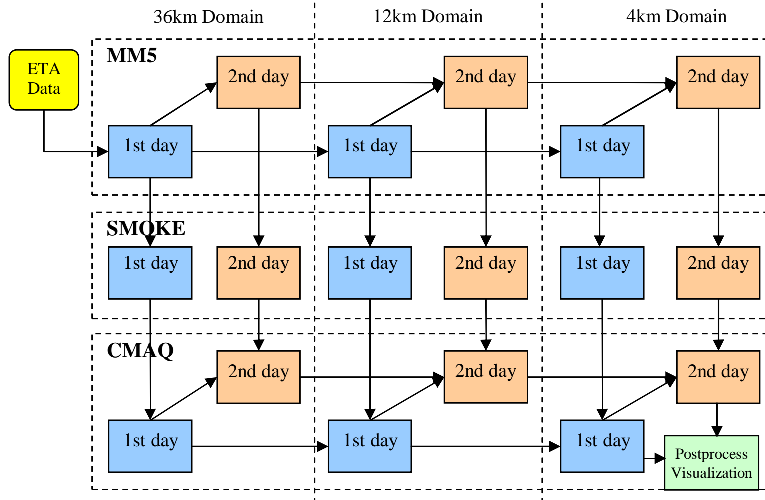


Figure 1: An AQF Application Workflow

only required to provide application descriptions and specify their resource requirements. GRACCE metascheduler is responsible to allocate grid resources for application modules, place module jobs on resources for execution and monitor them, and return the results back to users. GRACCE metascheduler architecture has three components, a metascheduler with planning and resource reservation capabilities, an event-driven workflow system and a runtime execution and monitoring system [12].

During the process of deploying AQF, we are requested by users to provide a general-purpose and easy-to-use method to describe application entities and structures. GRACCE also needs a description language and a workflow system that can be easily integrated with grid metascheduler. To find solutions for the needs of the two parties, we summarize these requirements as follows:

- Able to describe application logics, and support advanced structures, such as loops, branches, and nested modules;
- Description should not require additional translation from the original logics of an application. For example, if an application is defined with data-flow structure, description should not require users to extract the control-flow structure in order to use the description language.
- Defined languages should be easy to use for domain scientists, requiring at most introductory knowledge of grid computing.
- Description should be easily converted to job submission scripts and workflow descriptions in order to integrate with grid middlewares.
- It is better that the description supports the mapping between XML, RDBMS, and Java object for the integration with grid information and accounting systems.

2.2 GRACCE Application Modeling and Description Language

To meet the requirements of both domain users and grid integration, we defined a set of XML schema, called GAMDL for grid application modeling and description. A GAMDL

document defines application entities, including executables, data files and modules, and the dependency relationships of the defined entities. In summary, 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. As a result, there is no need to explicitly specify resource multirequests in a workflow.
- GAMDL allows similar modules to be easily described using multi-value properties. The description document is structured by using entity uid and uid references to ensure its human-readability. We shall see details and examples of these features in the next sections.

3 GAMDL Structures and Core Concepts

3.1 Multi-Value Property and Universal ID

A **Multi-value property** (mvproperty), as its name implies, 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 $dmsz = \{36k, 12k, 4k\}$, $day = \{d1, d2\}$, the sentence aqf-$mm5$-$\$\{dmsz\}$-$\$\{day\}$ represents all the six instances ($\#dmsz * \#day$) 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.

In the GAMDL description of AQF, the mvproperties are defined as follows in a file (uhaqf.mvproperties), which is specified in the *mvproperty* element of a GAMDL document:

```
md={mm5, smoke, cmaq}    # Three AQF subsystems
dmsz={36k, 12k, 4k}      # Three AQF domains
day={d1, d2}             # Two-day forecasting
vdmsz={12k, 4k}          # The visualized domain
```

The Uid (Universal ID) attribute uniquely identifies an entity and an entity can be defined in one document and referenced in other documents by its Uid. The uses of Uid in GAMDL allow users to re-use the entities that are already defined for other applications. It is also not necessary to define all entities in one document. Application entities are normally defined in several documents, and other documents may reference the defined entities using their Uids.

3.2 Module

The core concept in GAMDL for describing a workflow application is "Module". A `Module` is a workflow task whose execution accomplishes certain application goals. A module's execution consumes computational resources and input data sets and generates output data. A module may be associated with one or more jobs, each of which is able to complete the module's task. A typical case of having multiple jobs is when the module code has been compiled into several binaries for different platforms. Each of these binaries can be specified in one job. The following code fragment describes the six CMAQ modules in Figure 1 using the mvproperties we defined before.

```
<mvproperty file="uhaqf.mvproperties"/>
  ... ...
<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">
    ... ...
  </jobSpec>
</module>
```
CMAQ Module Definition

3.3 GAMDL Application and Workflow

In GAMDL, an application and its workflow are separatedly defined. The application definition provides a high-level description of the application logics from the viewpoints of an end user and an application workflow specifies an (partial) execution instance of the application. The advantage of this spearation is to allow users to specify different workflows of an application based on their needs without defining a new application each time. This is especially useful for the recurrent execution of an application in several different modes.

The `Application` document defines the application entities, such as data files and modules, and the module dependency relationships. In our example, these AQF entities are defined in another three files and the `Application` document references the three files to include them. The `AppRun` document describe an AQF workflow, introducing the modules that are required for the workflow and identifying the start module(s) of the workflow.

```
<application name="" uid="" >
  <version build="" suffix="" major="" description="" minor="" uid=""/>
  <appExes>
    <exe uid=""> ... </exe>
      ... ...
  </appExes>
  <appDataFiles>
    <file uid=""> ... </file>
      ... ...
  </appDataFiles>
  <appModules>
    <module uid=""> ... </module>
      ... ...
  </appModules>
  <appMdRships>
    <pCnRshipSet>
      <PCnRship parentMdUidRef="">
        <childMd uidRef=""> ... </childMd>
          ... ...
      </PCnRship>
    </pCnRshipSet>
  </appMdRships>
</application>
```

GAMDL Application Document Structure

```
<appRun uid="uhaqfrun-v0" appUid="uhaqf" startTime="2005-07-16T15:23:15">
  <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>
```

An AQF Workflow Definition

4 GAMDL Specification of Application Logics

4.1 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 of *PCnRship* element has a parent module and one or more child modules, and a CPs relationship of *CPsRship* element 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. The next code fragment is part of the *appMdRships* document in the "aqfmddep.xml" file. It describes the PCn relationships between the SMOKE and CMAQ modules, and between the CMAQ modules for the first day and the second day forecasting (see Figure 1):

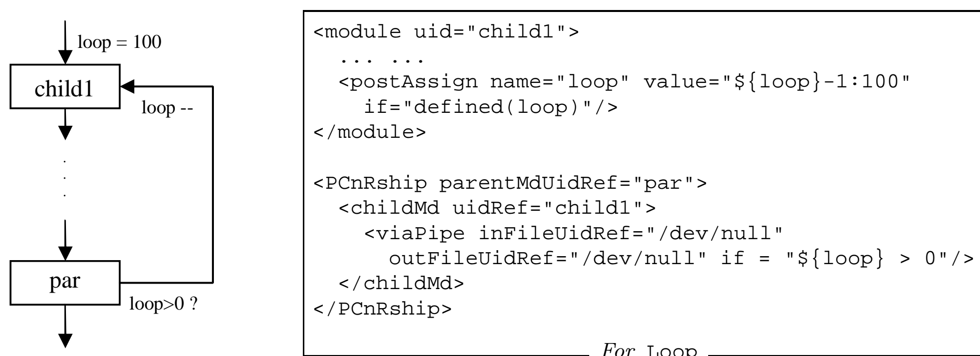
```
<appMdRships>
  <mvproperty file="uhaqf.mvproperties"/>
    ... ...
  <PCnRship parentMdUidRef="smoke-${dmsz}-${day}">
    <childMd uidRef="cmaq-${dmsz}-${day}">
      <viaPipe inFileUidRef="smoke-${dmsz}-${day}-out1"
        outFileUidRef="cmaq-${dmsz}-${day}-in1"/>
    </childMd>
  </PCnRship>

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

4.2 Control Logic Description

GAMDL allows the specification of control logic, such as loops or conditional branches, by using conditional pipes and variables. A **conditional pipe** associates a pipe with a 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 to *false*, runtime dependencies are not established and the child module will not be executed. A **GAMDL 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 GAMDL variables defined in other modules. In the following *for loop* example, the *child1* module `postAssign`s the loop index (*loop*) as 100 if the *loop* variable has not yet defined (which means this iteration is entering the loop), or *${loop} - 1* in each following iteration. In the *pCnRship* of *par* module and *child1* module, a null pipe (using */dev/null* file) is specified with *if* condition as *${loop} > 0.* In each iteration, if the condition is evaluated as "*true*", the pipe is established and the control is passed to the *child1* module.




```
<module uid="child1">
  ... ...
  <postAssign name="loop" value="${loop}-1:100"
    if="defined(loop)"/>
</module>

<PCnRship parentMdUidRef="par">
  <childMd uidRef="child1">
    <viaPipe inFileUidRef="/dev/null"
      outFileUidRef="/dev/null" if = "${loop} > 0"/>
  </childMd>
</PCnRship>
```
For Loop

In APPENDIX A, we give a more detailed example showing how a workflow with loops

and conditional branches is specified using conditional pipes and variables.

5 GAMDL Support for Resource Co-Allocation

5.1 GAMDL Job Specification

A job specification in GAMDL is splitted into three parts, the execution schedule, the execution configuration, and the resource request. **An execution schedule** includes information about *when and where* of the job executable (binary or script) to be launched, such as its start time and host name. **An execution configuration** includes information about *how* the job executable to be launched, such as its launcher (e.g. a bash shell), directory, arguments and environment variables. **The resource request** of a job includes similar information to that of an execution schedule, but such information is provided for a scheduler to generate an execution schedule. When specifying a job, users could either specify the job resource request and let the scheduler make the scheduling decision and generate the execution schedule, or could directly specify the schedule.

The execution schedule is of the *ExeScheduleType* with the following schema:

```
<xsd:complexType name="ExeScheduleType">
  <xsd:attribute name="startTime" type="xsd:dateTime" />
  <xsd:attribute name="host" type="xsd:string" />
  <xsd:attribute name="hostCPUList" type="xsd:string" />
  <xsd:attribute name="numCPU" type="xsd:integer" />
  <xsd:attribute name="memSize" type="xsd:nonNegativeInteger" />
  <xsd:attribute name="retry" type="xsd:string" />
</xsd:complexType>
```

ExeScheduleType

The execution configuration is of the *ExeConfigType* with the following schema:

```
<xsd:complexType name="ExeConfigType">
  <xsd:sequence>
    <xsd:element name="launcher" type="xsd:string" />
    <xsd:element name="launcherArgu" type="xsd:string" />
    <xsd:element name="directory" type="xsd:string" />
    <xsd:element name="arguments" type="xsd:string" />
    <xsd:element name="env" type="gamdl:EnvironmentType" />
    <xsd:element name="stdin" type="xsd:string" />
    <xsd:element name="stdout" type="xsd:string" />
    <xsd:element name="stderr" type="xsd:string" />

    <xsd:element name="indicator" type="gamdl:ScriptType" />
    <xsd:element name="preprocesser" type="gamdl:ScriptType" />
    <xsd:element name="postprocesser" type="gamdl:ScriptType" />
    <xsd:element name="cleaner" type="gamdl:ScriptType" />

    <xsd:element name="inArgu" type="gamdl:VariableType" />
    <xsd:element name="outArgu" type="gamdl:VariableType" />
    <xsd:element name="preAssign" type="gamdl:VariableType" />
    <xsd:element name="postAssign" type="gamdl:VariableType" />
  </xsd:sequence>
  <xsd:attribute name="retry" type="xsd:string" />
</xsd:complexType>
```

ExeConfigType

The Resource Request is of the *ResourceReqType* with the following schema:

```
  <xsd:complexType name="ResourceReqType">
    <xsd:attribute name="startTime" type="xsd:dateTime" />
    <xsd:attribute name="endTime" type="xsd:dateTime" />
    <xsd:attribute name="host" type="xsd:string" />
    <xsd:attribute name="hostCPUList" type="xsd:string" />
    <xsd:attribute name="maxCPU" type="xsd:integer" />
    <xsd:attribute name="minCPU" type="xsd:integer" />
    <xsd:attribute name="maxWTime" type="xsd:long" />
    <xsd:attribute name="maxCPUTime" type="xsd:long" />
    <xsd:attribute name="maxMem" type="xsd:nonNegativeInteger" />
    <xsd:attribute name="minMem" type="xsd:nonNegativeInteger" />
  </xsd:complexType>
```
ResourceReqType

5.2 Job Execution Profile

JobExeProfiles of gmjsl:JobExeProfileSetType type specifies the historical and profiling information of module executions on different grid resources. On a grid resource, a module may have been executing several times and each execution is described as an execution scenario of gmjsl:ExeScenarioType type, which is a list of consumed resources of the module execution. Scaling algorithms are used to predict an unhappened scenario based on the available scenarios.

The purpose of module jobExeProfiles is to provide grid metascheduler historical information of module executions to help resource-allocation decision making. For applications like AQF that run everyday with similar scenarios, it is very easy to predict the execution scenario of a module on the resources on which the module have been executing. Based on such predictions, metascheduler can make much better decisions of resource co-allocation for module jobs. Also statistical analysis, data normalization and scaling may also be performed on the history executions for other purposes in scheduling, such as the coordination between applications.

```
<xsd:complexType name="JobExeProfileSetType">
  <xsd:sequence>
    <xsd:element name="exeProfile"
                 type="gmjsl:ExeProfileType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ExeProfileType">
  <xsd:sequence>
    <xsd:element name="name"/>
    <xsd:element name="resourceName"/>
    <xsd:element name="resourceGlobalURI"/>
    <xsd:element name="scenarios"
                 type="gmjsl:ExeScenarioType"/>
    <xsd:element name="CPUTimeScalingAlgorithm"
                 type="gmjsl:ScalingAlgorithmType"/>
    <xsd:element name="VMemScalingAlgorithm"
                 type="gmjsl:ScalingAlgorithmType"/>
          ...
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ExeScenarioType">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:element name="numberOfCPU"/>
  <xsd:element name="hostCPUList"/>
  <xsd:element name="exeWindow"
               type="gamdl:TimeWindowType"/>
  <xsd:element name="consumedCPUTime"/>
```

```
  <xsd:element name="consumedMemory"/>
   ...
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ScalingAlgorithmType">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string"/>
  <xsd:element name="formula" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Job Execution Profile Specification

6 Workflow Specification and Resource Multirequest

A workflow of an application is specified by the *appRun* document, which is of *AppRunType*. See the next code fragement for the *AppRunType* schema.

```
<xsd:complexType name="AppRunType">
  <xsd:sequence>
    <xsd:element name="inArgu" type="gamdl:VariableType" />
    <xsd:element name="outArgu" type="gamdl:VariableType" />

    <xsd:element name="description" type="xsd:string" />
    <xsd:element name="dftMdJobIndex" type="xsd:integer" />
    <xsd:element name="dftMdJobResrcReq" type="gamdl:ResourceReqType" />
    <xsd:element name="dftMdJobExeConfig" type="gamdl:ExeConfigType" />
    <xsd:element name="dftMdJobExeSchedule" type="gamdl:ExeScheduleType" />
    <xsd:element name="mdRun" type="gamdl:MdInAppRunType" />
    <xsd:element name="startMd" type="gamdl:UidRefSetType" />

    <xsd:element name="preAssign" type="gamdl:VariableType" />
    <xsd:element name="postAssign" type="gamdl:VariableType" />
  </xsd:sequence>
  <xsd:attribute name="uid" type="gamdl:UidType" />
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="description" type="xsd:string" />

  <xsd:attribute name="appUid" type="gamdl:UidType" />
  <xsd:attribute name="appFile" type="xsd:anyURI" />
  <xsd:attribute name="startTime" type="xsd:dateTime" />
  <xsd:attribute name="endTime" type="xsd:dateTime" />
  <xsd:attribute name="reoccuring" type="xsd:boolean" />
</xsd:complexType>

<xsd:complexType name="MdInAppRunType">
  <xsd:sequence>
    <xsd:element name="jobResrcReq" type="gamdl:ResourceReqType" />
    <xsd:element name="jobExeConfig" type="gamdl:ExeConfigType" />
    <xsd:element name="jobExeSchedule" type="gamdl:ExeScheduleType" />
  </xsd:sequence>
  <xsd:attribute name="mdUidRef" type="gamdl:UidType" />
  <xsd:attribute name="jobIndex" type="xsd:integer" />
</xsd:complexType>
```

AppRunType

7 Related Works

Regarding these requirements, there have been a number of efforts to define a language or method. Condor DAG (Directed Acyclic Graph) [15] allows the description of parent-child

relationship of jobs, thus provides the basics for building application workflows. Yet Condor DAG only works with Condor managed resources and DAG cannot describe complex workflows such as loop or branches. Business Process Execution Language (BPEL) [14] is an XML-based workflow definition language to describe enterprise business processes. BPEL is in low-level web service level; additional extension and wrapping development are needed to make it easy of use by grid application owners. XScufl is a specific workflow definition language for Taverna project [23], but XScufl is too fine grained for describing scientific applications. Abstract Grid Workflow Language (AGWL) [10] “describes” application control flow using constructs of imperative programming style. For data-flow applications, users have to translate the dataflow into control-flow to use AGWL. Semantics web [9] standards, Resource Description Framework (RDF) [21] and Web Ontology Language (OWL) [20], aim to provide another structuring and description framework that allows data to be integrated in a much larger-scale than what current HTML-framework provides. But the general-purpose semantic web standards are very abstract for specific fields and no available vocabularies are defined for grid applications and workflows.

7.1 DAGMan

The Directed Acyclic Graph Manager (DAGMan) is a workflow scheduler for Condor jobs. DAGMan uses DAG as the data structure to represent job dependencies. Each job is a node in the graph and the edges identify their dependencies.

Condor supports different types of executables by specifying the execution environment (“universe”) in job description files. The *universe* provided by Condor includes Standard, Vanilla, PVM, MPI, Globus, Java, Scheduler.

7.1.1 DAG Description

```
# Filename: diamond.dag
#
# Job specification in the format of ‘‘Job <jobName> <JobCondorScript>’’
Job  A  A.condor
Job  B  B.condor
Job  C  C.condor
Job  D  D.condor
# the preprocessing and postprocessing scripts for jobs
Script PRE  A top_pre.csh
Script PRE  B mid_pre.perl  $JOB
Script POST B mid_post.perl $JOB $RETURN
Script PRE  C mid_pre.perl  $JOB
Script POST C mid_post.perl $JOB $RETURN
Script PRE  D bot_pre.csh
# dependency relationships
PARENT A CHILD B C
PARENT B C CHILD D
# Number of retries if a job fails
Retry  C 3
```

So in terms of expressivity, DAGMan is script based and supports various types of applications if users wrap their application details into scripts. Loops and conditional branches are prohibited in DAGMan, because it would lead to deadlock. DAGMan support partial workflow execution in the workflow description files. If a job is marked as DONE, then this job is not scheduled.

Applicability: DAGMan does not support automatic intermediate data movement, so users have to specify data movement transfer through reprocessing and post-processing

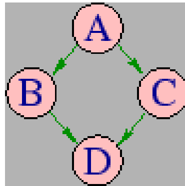


Figure 2: Diamond Workflow

commands associated with processing job. DAGMan mainly support workflows that have predefined behavior before execution and cannot handle such situation that the workflow can change over runtime. Also DAGMan is restricted within Condor managed clusters.

7.2 The Chimera Virtual Data System [?, ?, ?] and Pegasus in GridPhyN

The Chimera Virtual Data System (VDS) of GriPhyN [17], is a set of tools for data-processing workflow management, including expressing, executing, and tracking the results of workflows.

7.2.1 Workflow Description

VDS provides a location-independent, high-level "Virtual Data Language" - VDL [?] to specify application data and data processing modules. A set of application programs are described as transformations and the executions of transformations are described as derivations. Derivations produce or consume data files, which are described as data objects. A data object is always a logical file and a separate replica catalog or replica location service is used to map a logical file to its physical locations. The dependencies between derivations in terms of data objects consisitues the application workflow, which can be modeled as DAGs.

```
TR t1( output a2, input a1,
       none env="100000",
       none pa="500" ) {
   app vanilla = "/usr/bin/app3";
   arg parg = "-p "${none:pa};
   arg farg = "-f "${input:a1};
   arg xarg = "-x -y ";
   arg stdout = ${output:a2};
   profile env.MAXMEM = ${none:env};
}
```

```
DV t1(
   a2=@{output:run1.exp15.T1932.summary},
   a1=@{input:run1.exp15.T1932.raw},
   env="20000", pa="600" );

export MAXMEM=20000
/usr/bin/app3 -p 600 \
   -f run1.exp15.T1932.raw -x -y \
   > run1.exp15.T1932.summary
```

In the above example, This definition reads as follows. The first line assigns the transformation a name (t1) for use by derivation definitions, and declares that t1 reads one input file (formal parameter name a1) and produces one output file (formal parameter name a2). The parameters declared in the TR header line are transformation arguments and can only be file names or textual arguments.

The APP statement specifies (potentially as an LFN) the executable that implements the execution. The first three ARG statements describe how the command line arguments to app3 (as opposed to the transformation arguments to t1) are constructed. Each ARG statement comprises a name (here, parg, farg, and xarg) followed by a default value, which may refer to transformation arguments (e.g., a1) to be replaced at invocation time by their value. The special argument stdout (the fourth ARG statement in the example) is used to specify a filename into which the standard output of an application would be redirected.

Argument strings are concatenated in the order in which they appear in the TR statement to form the command line. The reason for introducing argument names is that these names can be used within DV statements to override the default argument values specified by the TR statement. Finally, the PROFILE statement specifies a default value for a Unix environment variable (MAXMEM) to be added to the environment for the execution of app3.

A DV statement defines a derivation. When the VDL interpreter processes such a statement, it records a transformation invocation within the virtual data catalog. A DV statement supplies LFNs for the formal filename parameters declared in the transformation and thus specifies the actual logical files read and produced by that invocation. For example, the following statement records an invocation of transformation t1 defined above.

The string immediately after the DV keyword names the transformation invoked by the derivation. In contrast to transformations, derivations need not be named explicitly via VDL statements. They can be located in the catalog by searching for them via the logical filenames named in their IN and OUT declarations as well as by other attributes, as discussed below. Actual parameters in a derivation and formal parameters in a transformation are associated by name. For example, the statements above result in parameter a1 of t1 receiving the value run1.exp15.T1932.raw and a2 the value run1.exp15.T1932.summary.

In Chimera, VDL definitions are stored in the Virtual Data Catalog - VDC - that provides for the tracking of the provenance of all files derived by an application. Chimera VDS contains the recipe to produce a given logical file, and on request produces an abstract workflow which will produce the file. The workflow is constructed by SQL-querying the Chimera VDL and the workflow is in the form of a DAG of program execution steps.

Pegasus [?] is a workflow manger in GriPhyN [17] that takes the abstract workflow and maps it to the available grid resources for execution. The mapping process first queries the Globus Replica Location Service (RLS) to find the location of the input files, as specified by their logical filenames in the DAG. The RLS returns a list of physical locations for the files. Based on the file locations and its destinations, workflow nodes (transfer jobs) are created to handle data movements. For data files that needs to be generated, Pegasus queries Globus MDS to find resources for computations. It finally produces a concrete workflow (CW), and submits the CW to Condor-G/DAGMan for execution.

Chimera provides a method to define an abstract view of an application, including applicaiton programs, data sets and procedures to produces the required data sets. Users do not need to explicitly specify the workflow structure of the applications. Yet Chimera does not specify how to describe complex workflow structure such as loops and conditional branches. Those structures are widely used in some applications whose execution behavior and module dependency relationships can only be decided upon exections. So Chimera is best suitable for data processing applications with predefined behaviors and data flows.

no job specification, such as resource request info for scheduling purpose

7.3 Taverna

Taverna [?, ?] of myGrid project [?] is workflow environments for life-science application services. Taverna provides a GUI interface, Taverna workbench as the main user interface to the access Taverna workflow management system, such as the construction and editing of service workflow, loading and saving these in Taverna XScufl languages, and invocation and enactment of a workflow through Freefluo with Taverna extensions, etc.

mainly web service workflow

7.3.1 Taverna Scufl data model for workflow description

The Scufl language is essentially a dataflow centric language, defining a graph of data interactions between different services. The components of a Scufl workflow are:

- A set of inputs and outputs that are points for the data for the workflow.
- A set of processors each of which represents a logical service: an individual step within a workflow. A processor includes a set of input ports and a set of output ports. From the user's perspective the behaviour of a processor is to receive data on its input ports, (process the data internally) and to produce data on its output ports.

  Processors: A processor can be regarded as a function of some set of input data to a set of output data, where each function may have side effects on the execution environment that are not encapsulated within the input / output specification. Processors therefore contain ports, which are named uniquely within the scope of the processor, are defined as either input or output and may have a type assigned to them in some type scheme, but this is not currently defined within the Scufl language. Processors have a set of named input ports, a set of named output ports, a name within the scufl space, and a current execution status (initializing, waiting, running, or completed).

- A set of data links that link data sources to data destinations. The data sources can be inputs or processor output ports, and data destinations can be outputs or processor input ports.

  Data links: A data link represents the consumption of some processor output by an input of some other processor. In fact, there is nothing in the language to prevent a processor consuming one of its own outputs, although this may be rejected during the translation to some other format due to the implicit problems with cyclic workflows. Data links have a source processor and output port name, a sink processor and an input port name and an optional name within Scufl space.

- A set of coordination links that enable running order dependencies to be expressed where direct dataflow is not required by providing additional constraints on the behaviour of the linked processors.

  Concurrency constraint: Although the data link specifications are enough to ensure correct execution ordering, since we allow processors to have side effects on their environment it is often required to explicitly create constraints on the ordering of execution of different processors. Specifically, it is possible to create a gate constraint that must be satisfied before a processor can effect a particular state change; for example, processor one is only allowed to shift state from waiting to running when processor two has status 'completed'. Constraints have a processor controlled by the constraint, a state change blocked in that processor, a gate condition, and an optional name within scufl space. Concurrency constraints are particularly useful in dealing with stateful interaction with services as shown below.

The triangles at the top of Fig 3 are workflow inputs, the triangles at the bottom are workflow outputs and the green ovals are Web Service operations. The solid lines represent the data flows with the text annotations showing the data types.

The service interactions to enable a Scufl dataflow is via the data links between processors. Taverna developed a set of Processor plug-ins that handle the data flow on data links, for example, A WSDL Scufl processor implemented by a single Web Service operation

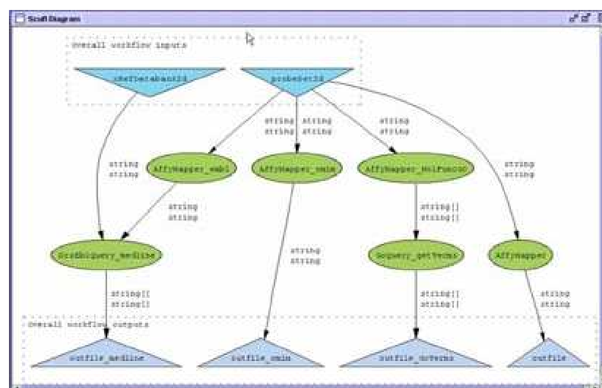


Figure 3: Diamond Workflow

described in a WSDL. The fields of the Web Service operation request message correspond to the input ports and the fields of the return message to the output ports; A local Java function processor, where services are provided by directly through a Java implementation with parameters as input ports and results as output ports; A SeqHound Processor that manages a Representational State Transfer (REST) style interface, where all information required for the service invocation is encoded in a single HTTP GET or POST request.

So mainly, data movement are encoded as message or method paramter passing, which restrict it to process small-size of datas. For large data files, some specific processor should be defined to handle file transfer.

Freefluo is the workflow enactment engine of Taverna. Freefluo accepts workflow definitions and input data from Taverna and co-ordinates the scheduling of workflow services. Often, in bioinformatics analyses, and therefore in taverna workflows, the output data of one service is the input data of the next. Freefluo can schedule these events to follow one another. Alternatively, if two services have no bearing on one another, Freefluo can schedule them to be enacted at the same time, reducing the time taken to run the whole workflow.

The enactor core is used in the context of a particular language and service run-time environment. A workflow language parser is used to convert a textual workflow specification, e.g. a Scufl document, into the internal object representation of the enactor core. An invocation framework is then added to allow the enactor to actually invoke services in the run-time environment and deal with the specific data types passed between the services invoked, e.g. WSDL calls and XML message parts.

7.4 Triana [?]

Triana a graphical environment to construct and execute workflow programs and to use them, with a minimum of effort and no programming. Using Triana, you simply assemble your program from a set of building-blocks that you drag into a work-space window and connect up using your mouse. With a click of the mouse the program will perform whatever operations you want. You can tell Triana to execute your program just once or continuously, as long as data is available to it.

7.5 Karajan

7.6 Kepler

7.7 Asklon

8 Conclusion

This study work provides a detail analysis of the internal architectures in D-RMSs by decomposing a D-RMS into three modules: Job management subsystem, Physical resource management system, and scheduling and queuing subsystem. These three modules, together with the architecture in organizing them in each of our surveyed system are studied in detail. Since these systems are rather complex, to present them in a relatively detail and compare them require much more work than what we have done and this will be our future direction.

9 Conclusion and Future Work

In this paper, we presented GRACCE Application Modeling and Description Language (GAMDL), a high level abstract language for domain users to describe a grid-ready application. GAMDL, motivated by a production application, describes application data-flow structure so that users do not need to "program" the application control-flow. GAMDL is also the basis for the integration between a grid metascheduler and workflow systems in GRACCE project. GAMDL specifies a module job within application description, thus job workflow could be easily constructed. GAMDL job specifications provide a rich set of information to help grid metascheduler make resource-allocation decisions. Other features of GAMDL include the use of mvproperty to easily describe similar entities, and the ability to describe complex control-flow using conditioned properties and conditioned pipes.

We are currently using GAMDL for other applications and recommend users who have grid-ready applications to try it. Revising are possible based on users' comments and our metascheduler development progress in GRACCE. GAMDL is defined as RDF-friendly [19], which means that a GAMDL document can be easily converted to a RDF document. We are investigating related RDF and OWL concepts to make GAMDL more expressive and powerful. An option to define GAMDL using RDF and OWL is being evaluated.

References

[1] B. Allcock, I. Foster, V. Nefedova, A. Chervenak, E. Deelman, C. Kesselman, A. Sim, A. Shoshani, B. Drach, and D. Williams, *High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies,* Proceedings of the ACM/IEEE SC2001 Conference, 2001

[2] 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, H. Zima, A. Hoisie, L. Yang and B.D. Martino, Spring 2005

[3] 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

[4] G. Grell, J. Dudhia, and D. Stauffer, *A Description of the Fifth-Generation Penn State/NCAR Mesoscale Model (MM5) NCAR/TN-398+STR,* NCAR Tech Notes http://www.mmm.ucar.edu/mm5/

[5] 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.

[6] J. Yu, and R. Buyya, *A Taxonomy of Workflow Management Systems for Grid Computing,* Technical Report, GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, March 10, 2005

[7] 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.

[8] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, *A Resource Management Architecture for Metacomputing Systems,* Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998.

[9] T. Berners-Lee, J. Hendler, and O. Lassila, *The Semantic Web,* Scientific American, May 2001

[10] 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 (CCGrid 2005)

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

[12] Y. Yan, B.M. Chapman, and B. Sundaram, *Air Quality Forecasting on Campus Grid,* Workshop on Grid Applications: from Early Adopters to Mainstream Users, GGF14, Chicago, IL 2005, accepted.

[13] 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.

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

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

[16] Grid Application Coordination, Collaboration and Execution, http://www.cs.uh.edu/ yanyh/gracce

[17] Grid Physics Network, http://www.griphyn.org

[18] Linked Environments for Atmospheric Discovery, http://lead.ou.edu

[19] Make Your XML RDF-Friendly, http://www.xml.com/pub/a/2002/10/30/rdf-friendly.html

[20] OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features

[21] Resource Description Framework (RDF), http://www.w3.org/RDF

[22] The Globus Resource Specification Language RSL, http://www-fp.globus.org/gram/rsl_spec1.html

[23] The Taverna Project, http://taverna.sourceforge.net

APPENDIX A: A GAMDL Example for A 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*, respectively. The loop count is 100.

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 the *F2* pipe in *md4-md2*

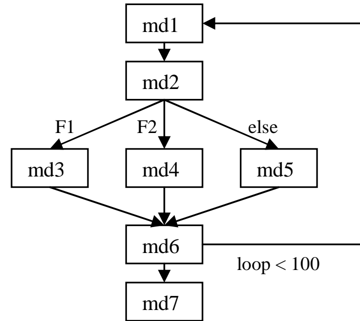


Figure 4: A Workflow with Loop and Conditional Branches

`CPsRship` and the else-pipe in *md5-md2* CPsRship are similar to the *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 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>
```

GAMDL Application Document Structure