Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language*

Thomas Fahringer, Jun Qin, Stefan Hainzer
Institute of Computer Science, University of Innsbruck
Technikerstr. 21a, 6020 Innsbruck, Austria
{Thomas.Fahringer, Jun.Qin, Stefan.Hainzer}@uibk.ac.at

Abstract

Currently Grid application developers often configure available application components into a workflow of tasks that they can submit for executing on the Grid. In this paper, we present an Abstract Grid Workflow Language (AGWL) for describing Grid workflow applications at a high level of abstraction. AGWL has been designed such that the user can concentrate on specifying Grid applications without dealing with either the complexity of the Grid or any specific implementation technology (e.g. Web service). AGWL is an XML-based language which allows a programmer to define a graph of activities that refer mostly to computational tasks. Activities are connected by control and data flow links. A rich set of constructs (compound activities) is provided to simplify the specification of Grid workflow applications which includes compound activities such as `if`, `forEach` *and* `while` *loops as well as advanced compound activities including parallel sections, parallel loops and collection iterators. Moreover, AGWL supports a generic high level access mechanism to data repositories. AGWL is the main interface to the ASKALON Grid application development environment and has been applied to numerous real world applications. We describe a material science workflow that has been successfully ported to a Grid infrastructure based on an AGWL specification. Only a dozen AGWL activities are needed to describe a workflow with several hundred activity instances.*

1. Introduction

Grid computing [1] enables the virtualization of distributed and heterogeneous resources such as CPUs, storage systems, sensor networks and scientific instruments thus a unified view of resources, authentication, authorization and accounting can be represented to applications and end users. With the advent of Grid technologies, scientists and engineers are building more and more complex applications to manage and process large data sets and execute scientific experiments on distributed Grid resources. Grid workflow applications are emerging as one of the most important and challenging Grid application classes.

A Grid workflow application can be seen as a collection of activities (computational tasks) that are processed in a well-defined order to accomplish a specific goal. These activities may be executed on heterogeneous resources which are geographically distributed. There is no central ownership and control in a typical Grid environment. The computational and networking capabilities can vary significantly over time. These distributed and dynamic characteristics of the Grid impose many challenges for a Grid application development and computing environment.

In this paper, we describe AGWL, an XML-based high level Abstract Grid Workflow Language, which shields the details of the underlying Grid infrastructure and allows programmers to compose workflow applications in an intuitive way. AGWL has been carefully designed to include a set of essential constructs to simplify the specification of Grid workflow applications. In addition, programmers can specify high-level constraints and properties for activities and data flow connections which may be useful for a runtime system to optimize the workflow execution.

The paper is organized as followed. In the next section related work is described and compared against AGWL. Section 3 presents an overview of the development process of Grid workflow applications. Section 4 defines AGWL and its language constructs. A material science workflow application based on AGWL is discussed in Section 5. Finally, conclusions and future work are outlined in the last section.

*The work described in this paper is partially supported by the Austrian Grid Project, funded by the Austrian BMBWK (Federal Ministry for Education, Science and Culture) under contract GZ 4003/2-VI/4c/2004.

0-7803-9074-1/05/$20.00 ©2005 IEEE

2. Related Work

Although workflow applications have been extensively studied in areas such as business process modeling and web services [2], it is relatively new in the Grid computing area. Specifically, BPEL [2] is a low level and Web service specific language that is often referred to as a Grid assembly language.

Several efforts towards a Grid workflow specification language have been made. The Workflow Enactment Engine (WFEE) [3] is based on a workflow language xWFL. The Grid Service Flow Language (GSFL) [4] supports the specification of workflow descriptions for Grid services in the OGSA framework. However, both WFEE and GSFL miss some important control flow constructs such as branches and loops. Grid workflows in the GriPhyN [5] project are limited to acyclic graphs and no advanced constructs for the parallel execution of "components" is supported. DAGMan [6] is also limited to acyclic graph workflows.

Existing work commonly suffers by one or several of the following drawbacks: control flow limitations (e.g. no branches or loops), limited mechanism for expressing parallelism (e.g. no parallel sections or loops), restricted data flow mechanisms (e.g. limited to files), implementation specific (focus on Web services, Java classes, software components, etc.), and low level constructs (e.g. start/stop tasks, transfer data, queue task for execution, etc.) that should be part of the workflow enactment engine.

In contrast, AGWL provides an advanced and user-oriented Grid workflow language which shields the complexity of the underlying Grid infrastructure and runtime environment from the application developer. AGWL supports a reasonable and important set of control and data flow constructs to build Grid workflow applications. AGWL is also a modularized Grid workflow language which supports the declaration of sub-workflows that can be invoked by other workflows. Most existing workflow languages are based on a simple I/O system (e.g. files) whereas AGWL also enables a generic access mechanism to data repositories which is important for many applications. AGWL includes several constructs (properties and constraints) which can be used by an application developer to provide additional information to optimize the workflow execution by the underlying runtime environment.

3. System Overview

Figure 1 shows an overview of the development process of Grid workflow applications in the ASKALON Grid application development environment [7] from an abstract representation to an actual execution on a Grid infrastructure. The development process consists of three fundamental procedures: Composition, Reification and Execution.

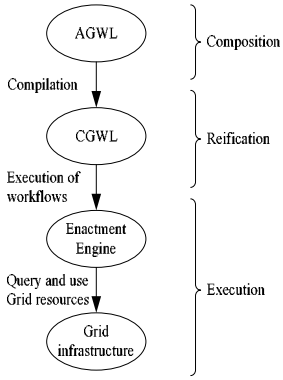


Figure 1 The development process of Grid workflow applications in ASKALON

Composition: The user composes the Grid workflow by writing an AGWL program. At this level, activities correspond mostly to computational tasks and the specification of their input and their output data. There is no notion of how the input data is actually delivered to activities and how activities are implemented, invoked and terminated. AGWL contains all the information specified by the user during the workflow composition.

Reification: A transformation system in ASKALON compiles AGWL file to a CGWL (Concrete Grid Workflow Language) representation through extracting more concrete information such as data types from an activity registry and inserting them into the workflow description. The transformation system also tries to optimize the workflow. For example, changing parts of a `sequence` construct (see Section 4.2.1) into a `parallel` construct (see Section 4.2.1) by checking the data dependencies. CGWL represents a workflow which can be scheduled and executed. In contrast to AGWL, CGWL is implementation dependent. However, CGWL is not seen by the AGWL programmer.

Execution: CGWL is interpreted by the underlying workflow runtime environment of ASKALON to construct and execute the Grid workflow application on a Grid infrastructure. More details about the execution of workflow applications can be found at [7].

4. AGWL Specification

An AGWL workflow consists of *activities*. An activity can be either an *atomic activity*, which refers to a single computational task, or a *compound activity*, which encloses some atomic activities or other compound activities that are connected by control and data flows. A workflow is a compound activity. The composition of a workflow application or a compound

activity is done by specifying all its enclosed activities as well as their control and data flow connections.

In addition, properties and constraints can be specified for activities and data flow connections to optimize the execution of workflow applications.

4.1. Atomic Activities

The definition of an atomic activity is shown in Figure 2.

```
<activity name="name" type="type">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <dataOut name="name" />*
</activity>
```

Figure 2 The atomic activity

Activity Name: The activity name serves as an identifier for the activity. Activities must be organized in an `AGWL-workflow` or a `sub-workflow` which define a scope for them. In the scope, the name of each activity is unique.

Activity Type: In AGWL, activities are described by activity types. An activity type is an abstract description of a group of activity instances (concrete implementations of computational entities) deployed in the Grid. For instance, an activity type *MatrixMult* can be used to describe a group of two activity instances which are deployed as an executable and a web service and both of them implement matrix multiplication. The activity type must be defined in an Activity Type Definition (ATD, see Section 4.5) file before defining AGWL workflows. Activity types shield the implementation details of activity instances from the AGWL programmer. Locating and invoking activity instances are done by an underlying runtime environment for AGWL.

Data-In/Data-Out Ports: The data contained in data-in or data-out ports is denoted as a data package. The data in activities can be interchanged through the exchange of data packages along the specified data flows. The number and types of the data-in/data-out ports are determined by the chosen activity type. The data-in port can be specified by (1) setting its `source` attribute to the name of a data-out port of another activity in the same workflow in the form of `activity-name/data-out-port-name`, (2) setting its `source` attribute to the name of an abstract data container *repository*, or (3) specifying an XML constant in the `value` element, if the needed data is an XML constant and no `source` is specified. We will address the *repository* in detail in Section 4.3.6. Linking the data-in ports and data-out ports of different activities through the `source` attributes of the data-in ports defines the data flows of AGWL workflow applications.

4.2. Compound Activities

In a compound activity, the specifications for its name attribute, its data-in/data-out ports and the source attributes of its data-in ports are similar to that for the identical attributes in an atomic activity. To avoid redundancy, we will not separately explain these attributes in the following sections.

There are two kinds of compound activities in AGWL: basic compound activities, which are similar to some well known constructs of high-level languages such as `if`, `for` or `while`, and advanced compound activities, which enable the expression of parallelism at a high level, such as parallel loops.

4.2.1. Basic Compound Activities

- Sequence Activity

The `sequence` compound activity (Figure 3) imposes a sequential control flow on all of its contained activities. If an AGWL programmer specifies two activities one after the other, it is supposed to be a `sequence` activity by default.

```
<sequence name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <activity>+
  <dataOut name="name" source="source" />*
</sequence>
```

Figure 3 The `sequence` activity

Data-Out Ports: For all compound activities, we need the `source` attributes for data-out ports to specify internal data flows from data-out ports of inner activities to data-out ports of the compound activities themselves.

- Parallel Activity

The `parallel` compound activity (Figure 4) indicates that all `sections`, in which some activities are contained, can be executed simultaneously.

```
<parallel name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <section>
    <activity>+
  </section>+
  <dataOut name="name" source="source" />*
</parallel>
```

Figure 4 The `parallel` activity

- Conditional Activities

There are two conditional activities in AGWL: `if` (Figure 5) and `switch` (Figure 6). Both of them enable the conditional execution of activities which depends on the condition expression of these activities.

Condition: In the `if` or `switch` activity, the condition is an XSLT [8] expression. The values of the data-in ports, which are referred in the condition, must be interpretable as XML.

```
<if name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <condition> condition </condition>
  <then>
    <activity>+
  </then>
  (<else>
    <activity>+
  </else>)?
  <dataOut name="name" source="source" />*
</if>
```

Figure 5 The `if` activity

```
<switch name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <case condition="condition"
        (break="true|false")?>
    <activity>+
  </case>+
  (<default>
    <activity>+
  </default>)?
  <dataOut name="name" source="source" />*
</switch>
```

Figure 6 The `switch` activity

Break: The optional `break` attributes in the `case` elements terminate the enclosing `switch` activity. And the control flow continues with the first activity following the switch activity. Without an explicit `break`, the control flow will sequentially pass through all subsequent `case` elements.

Data-Out Ports: The control flow outcome of the `if` or `switch` activity is commonly unknown at compile time. Therefore, it is not allowed to connect a data-out port of an inner activity of a conditional activity to a data-in port of an activity outside of this conditional activity. Instead, data-out ports are defined for the `if` or `switch` activity which can be connected to data-in ports of other activities outside of the `if` or `switch` activity. The `source` attribute specifies the possible internal data flow from data-out ports of internal activities to data-out ports of the enclosing conditional activity. It contains a comma separated list of data-out ports of some inner activities. This list must contain one entry for each possible branch in the `if` or `switch` activity. Especially, if the optional `else` branch in the `if` construct or the optional `default` branch in the `switch` construct is not specified, the name of data-in ports must be given to ensure that the data-out ports are set with valid data packages.

- Loop Activities

There are three loop activities in AGWL: `while` (Figure 7), `for` (Figure 8) and `forEach`. The `while` activity can be used to execute the loop body zero or more times. The `for` activity is provided to execute its body multiple times controlled by a counter. The `forEach` activity has been included in AGWL to enable the iteration over a data package collection. The loop body is executed once for each element in the data package collection. The `forEach` activity (not shown due to lack of space) is similar to the `for` activity except that in the `forEach` activity the first data-in port of must refer to a data package collection over which this activity iterates and the `loopElement` instead of the `loopCounter` is used as the loop index.

```
<while name="name">
  <dataIn name="name" (source="source")?
    (loopSource="loopSource")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <condition> condition </condition>
  <loopBody>
    <activity>+
  </loopBody>
  <dataOut name="name" source="source"/>*
</while>
```

Figure 7 The `while` activity

```
<for name="name">
  <dataIn name="name" (source="source")?
    (loopSource="loopSource")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <loopCounter name="name" from="from"
               to="to" (step="step")? />
  <loopBody>
    <activity>+
  </loopBody>
  <dataOut name="name" source="source"/>*
</for>
```

Figure 8 The `for` activity

Data-In Ports: In these loop activities, the optional `loopSource` attribute of data-in ports is used to express a cyclic data flow which is commonly linked to a data-out port of an activity inside the loop body, in the form of `activity-name/data-out-port-name`. After each execution of the loop body, the data-in ports of the loop activity receive the new values specified by the `loopSource` attributes. Such data-in ports receive the final results of the loop body after the loop terminates.

Condition: The condition (expressed as an XSLT expression) of the `while` construct controls how often the loop body is executed. This condition is evaluated before the loop body is executed. The loop is executed until this condition is evaluated to false.

Loop Counter: The `loopCounter` in the `for` activity controls how often the loop body is executed. It provides an implicit data-in port for the activities in the loop body and can be accessed by its name. The value of the `loopCounter` is initially assigned to the value specified by the variable `from` and is increased by the value of `step` until it reaches the value of `to` or larger. The values of `from, to, step` can be expressed as constants or as an XPATH [9] expression where the values of data-in ports, which must be interpretable as XML, can be used. The values of `from, to` and `step` are only evaluated once at the beginning of an invocation of the `for` activity.

Data-Out Ports: In these loop activities, the `source` attributes of the data-out ports are set to the names of the data-in ports with the `loopSource`

attribute. Such data-in ports are set to the final results after the execution of the loop activity. Activities outside of a loop activity can access the final results through the data-out ports of the loop activity after the loop terminates.

- Directed Acyclic Graph (DAG) activity

For specifying more complex control flow, AGWL includes the `dag` compound activity (Figure 9). The control flow of a `dag` activity is described as a directed acyclic graph. In order to specify the execution order of activities, a special wrapper element `dagNode` is introduced. An activity in a `dag` can be executed as soon as all of its predecessors terminated. Thus a dag activity has a potential for parallel execution of its contained activities.

Dag Node Name: Each `dagNode` must have a unique name in the scope defined by the `dag`. Each `dagNode` element contains an activity.

Dag Node Predecessor: The `predecessor` attribute of a specific dagNode *dn* is a comma separated list of names of some other `dagNode` elements which must be executed before *dn* can be executed. Those `dagNode` elements without predecessors are root elements of the `dag` activity. AGWL allows multiple roots in a `dag` activity. The root elements, as well as those `dagNode` elements whose predecessors have finished, can be executed immediately possibly in parallel with other dagNode elements.

```
<dag>
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <dagNode name="name" predecessor="names" />
    <activity>
  </dagNode>+
  <dataOut name="name" source="source" />
</dag>
```

Figure 9 The `dag` activity

4.2.2. Advanced Compound Activities

Advanced compound activities are supported by AGWL to express that multiple activities can be executed in parallel which include: `parallelFor` (Figure 10) and `parallelForEach`.

```
<parallelFor name="name">
  <dataIn name="name" (source="source")?>
    (<value> value as XML </value>)?
  </dataIn>*
  <loopCounter name="name" from="from"
               to="to" (step="step")? />
  <loopBody>
    <activity>+
  </loopBody>
  <dataOut name="name" source="source" />*
</parallelFor>
```

Figure 10 The `parallelFor` activity

The `parallelFor`/`parallelForEach` activity is similar to the `for`/`forEach` activity with the difference that in `parallelFor`/ `parallelForEach` all loop iterations can be executed simultaneously. It is assumed that the data input of any iteration is independent of data produced by other iterations of the same activity. The semantics of the attributes in parallel loop activities is identical to those in the sequential loop activities.

4.3. Data Flows in AGWL

The data flow describes that the data flows from data-out ports to data-in ports. To fully understand the AGWL data flow model, some aspects are examined in more detail in the following.

4.3.1. Well-defined Data-in and Data-out Ports

For each activity in AGWL it must be guaranteed that whenever the control flow reaches the activity, all the data-in ports of the activity have been assigned to well-defined values (valid data packages). When the control flow leaves, all its data-out ports must be well-defined as well. For activities with a trivial control flow (one predecessor and one successor), the definition of this constraint is straightforward. For all other activities with a more complex control flow (e.g. conditional activities, sequential loop activities, parallel loop activities) the corresponding data flow can become less trivial which will be explored in the following sections.

4.3.2. Data Flows in Conditional Activities

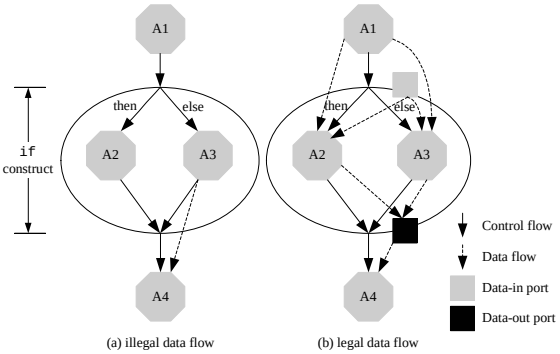


Figure 11 The data flow in conditional activities

Data flow is more complex for conditional activities. In the following we discuss the `if` activity to explain data flow issues for conditional activities. Figure 11(a) and Figure 11(b) illustrate two possible data flows of the `if` activity. Figure 11(a) shows an illegal data flow that is not allowed in AGWL. If the `then` branch is executed, then the data-in port of activity `A4` would be undefined. Therefore, the `then` branch must define the data-out ports of the `if` activity as shown in Figure 11(b). Activity `A4` is not allowed to be linked to a data-out port of an inner activity of the `if` activity. Instead, it must be linked to a data-out port of the `if` activity. In this case, it is ensured that each data-out port is well-

defined with a valid data package for each possible execution path.

4.3.3. Data Flows in Sequential Loop Activities

The data flow in sequential loop activities occurs in `while`, `for` and `forEach` activities. We explain the data flow model for the `while` activity which is similar for all other sequential loop activities. For the `while` activity, we have to describe the flow of data from one iteration to the next one, i.e. the output of one iteration serves as the input of the subsequent iteration. To model the data flow for the `while` activity, we have to consider two cases:

(1) The loop body is never executed if the loop condition is never evaluated to true.
(2) The loop body is executed multiple times which implies a cyclic data flow.

Figure 12 illustrates the data flow model for the `while` activity in AGWL. The control flow inside the loop body is not shown in order to avoid overloading this figure.

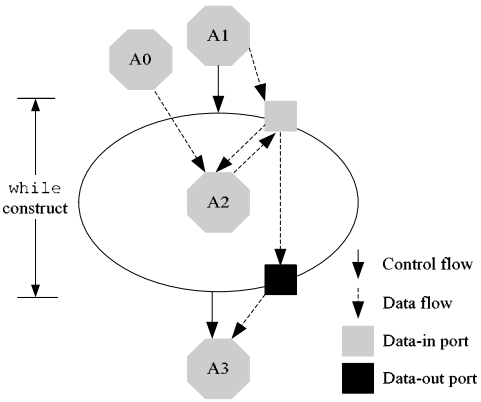


Figure 12 The data flow in sequential loop activities

(1) The data-in ports of the `while` activity are assigned initial values specified by the `source` attribute (e.g. a source activity and its data-in ports).
(2) If the loop condition is evaluated to false, the data flow continues at step (4). If true, the inner activity `A2` is executed. `A2` can obtain data from either the data-in ports of the `while` activity or from some outside activities such as `A0`.
(3) Once `A2` has been executed, data packages from its data-out ports can be transferred to the data-in ports of the surrounding `while` activity, specified by its `loopSource` attribute. Then the loop condition is evaluated again with the new data packages arrived at the data-in ports.
(4) When the loop condition is evaluated to false, the current data packages of the data-in ports are mapped to the data-out ports of the `while` activity.
(5) Activities outside the `while` activity can obtain the data packages from the data-out ports of the `while` activity only.

For each possible execution path inside the `while` activity, it is ensured that the data flow is well-defined and valid data packages are available at all data-in and data-out ports of all executed activities. In case that `A2` receives data packages from an activity outside of the `while` activity, these data packages remain constant and accessible for all loop iterations. Note that the number of possible data-out ports of the `while` activity must be smaller than or equal to the number of its data-in ports.

4.3.4. Data Flows in Parallel Loop Activities

In the following we describe the data flow mechanism for parallel loop activities exemplified by the `parallelFor` activity (Figure 13). The data flow for the `parallelForEach` activity is similar. In contrast to sequential loop activities, there is no need for a cyclic data flow. Since in general it cannot be decided at compile time how many times the loop body will be executed, a data package collection is used to hold all result data produced by all the loop iterations when the execution of the loop activity is finished.

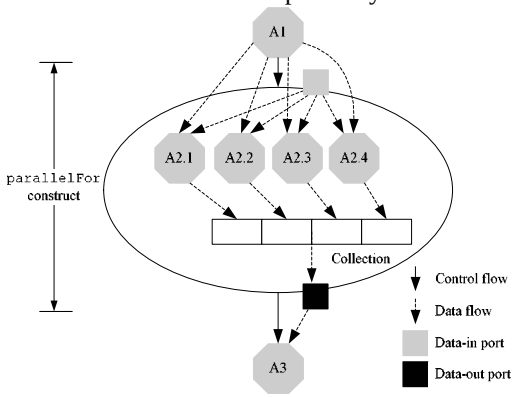


Figure 13 The data flow in parallel loop activities

Before any iteration of the `parallelFor` activity is executed, the number of iterations is evaluated. Then for each iteration an instance of the loop body (activity `A2` in the Figure 13) is invoked concurrently. Each instance (`A2.1`, `A2.2`, `A2.3` or `A2.4`) receives a unique iteration index from the loop counter of the `parallelFor` activity. In the case of `parallelForEach`, each instance would receive an element of the first data-in port of the `parallelForEach` activity which is a data package collection (over which the `parallelForEach` activity iterates). Every loop instance may also receive some other input data from either the data-in ports of the `parallelFor` activity or some outside activities such as `A1`. Each instance may also have its own data-

in ports. At the end of its execution, each instance of the loop body writes its results into the data package collection specified by the data-out ports of the `parallelFor` activity. This data package collection can be accessed through the data-out ports of the `parallelFor` activity by subsequent activities such as A3.

4.3.5. Data Flows in the DAG Activity

Figure 14 illustrates a possible data flow in a `dag` activity. The control flow inside the `dag` activity is not shown to avoid overloading the figure. A3, A5 are the root elements of the `dag` activity. In the `dag` activity, the user can define an arbitrary complex acyclic data flow. It is assumed that activities associated with the data source are executed before those activities related to the data destination in accordance with the defined control flow.

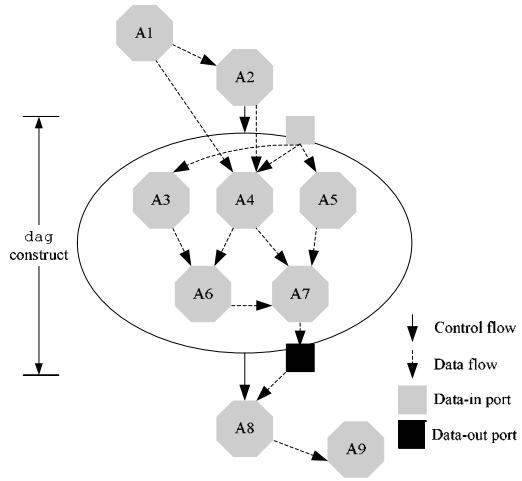


Figure 14 The data flow in the `dag` activity

4.3.6. Advanced Data Flows

In addition to the data flow among activities, AGWL supports the data flow between activities and special entities called *repositories*, which are abstractions for data containers. They are used to model, for instance, saving intermediate results or querying data resources without knowing any details about how repositories are actually implemented, e.g. file servers, databases, etc. A repository has a unique name in the context of a workflow application. The retrieval/insertion of information is made by specifying the `source`/`saveto` attribute with the repository name in data-in/data-out ports. The following example illustrates how an activity can store its data-out port (named `out`) into a repository (named `rep`).

```
<dataOut name="out" saveto="rep">
```

4.4. Properties and Constraints

In AGWL, properties and constraints can be defined by the user to provide additional information for a workflow runtime environment to optimize and steer the execution of workflow applications. Properties provide hints about the behaviour of activities, e.g. the expected size of the input data, the estimated computational complexity, etc. Constraints should be complied by the underlying workflow runtime environment, e. g. to minimize execution time, to provide as much memory as possible, to run on the specific host architecture, etc. The user can define properties and constraints elements for activities and for data-in and data-out ports (Figure 15).

```
<activity name="name" type="type">
  <dataIn name="name" (source="source")?>
    <properties>...</properties>
    <constraints>...</constraints>
  </dataIn>*
  <properties>...</properties>
  <constraints>...</constraints>
  <dataOut name="name">
    <properties>...</properties>
    <constraints>...</constraints>
  </dataOut>*
<activity>
```

Figure 15 Properties and constraints

4.5. The Structure of an AGWL file

The complete structure of an AGWL file (Figure 16) consists of importing activity type definitions, importing workflows, describing workflows and invoking sub-workflows. We address each part in the following sections.

```
 1 <agwl-workflow>
 2   <importATD url="url" name="name">
 3     (<alternativeUrl> url </alternativeUrl>)*
 4   </importATD>*
 5   <importWD url="url" name="name">
 6     (<alternativeUrl> url </alternativeUrl>)*
 7   </importWD>*
 8   <subWorkflow name="name">
 9     <dataIn name="name" />*
10     <body>
11       <activity>+
12     </body>
13     <dataOut name="name" source="source" />*
14   </subWorkflow>*
15   <activity>*
16 </agwl-workflow>
```

Figure 16 An AGWL file

4.5.1. Importing Activity Type Definitions

Before using an activity type in AGWL workflows, it is required to import the definition of the activity type which is always organized in an ATD file. The ATD files may be created and published by anyone providing implementations of activities which can be reused for Grid workflow construction. Line 2-4 in Figure 16 illustrates how to import activity types by using the `importATD` element.

URL: The `url` attribute specifies the location of the ATD file. Due to the dynamic nature of the Grid infrastructure, some Grid sites may be unavailable or unreachable unexpectedly. For this reason, we provide alternative URLs for the ATD files.

ATD Name: The `name` attribute defines an identifier which is used to avoid naming conflicts. It is not unlikely that the same activity type name (e.g. `MatrixMult`) occurs in different ATD files. In order to distinguish them, we associate different ATD files (e.g. `d1.atd`, `d2.atd`) with unique identifiers (e.g. `d1`, `d2`) in different `importATD` elements. When referring to an activity type, it must be specified with the unique identifier, followed by a colon and the activity type name.

```
<agwl-workflow>
  <importATD url="http://.../d1.atd" name="d1" />
  <importATD url="http://.../d2.atd" name="d2" />
  <activity name="a1" type="d1:MatrixMult">
    <dataIn name="mat1" source="rep1" />
    <dataIn name="mat2" source="rep2" />
    <dataOut name="out" />
  </activity>
  <activity name="a2" type="d2:MatrixMult">
    <dataIn name="matr" source="a1/out" />
    <dataIn name="mat3" source="rep3" />
    <dataOut name="resMatrix" />
  </activity>
</agwl-workflow>
```

Figure 17 Importing Activity Type Definitions

In the example shown in Figure 17, the two matrices `mat1` and `mat2` in the repositories `rep1` and `rep2` are multiplied based on the activity type defined in `d1.atd`. The result `matr` is then multiplied with the matrix `mat3` in the repository `rep3` according to the activity type defined in `d2.atd`.

4.5.2. Importing Workflows

In order to modularize and reuse workflows, we provide the `importWD` element (Line 5-7 in Figure 16) to import a workflow in another workflow. A sub-workflow can be invoked in another workflow only after the enclosing workflow is imported in that workflow. For efficiency reasons, the import elements always refer to compiled AGWL files, that is, CGWL representations. The `name` attribute and the `alternativeUrl` element have the same semantic as that of the `importATD` element described in Section 4.5.1.

4.5.3. Invoking Sub-Workflows

Lines 8-14 in Figure 16 define a sub-workflow. Sub-workflows are similar to procedures in other high-level languages. They are used to modularize, encapsulate and reuse a code region. Each `subWorkflow` has a unique name and well-defined data-in and data-out ports in an AGWL source file. Inside a sub-workflow, only data flow among inner activities is allowed. Sub-workflows can be invoked from different locations. To invoke a sub-workflow in an imported workflow, the name defined in the `importWD` element, followed by a colon and the name of the sub-workflow, must be specified. To invoke a sub-workflow which is defined in the same AGWL file, only the name of the sub-workflow must be indicated (Figure 18).

```
<agwl-workflow>
  <importWD url="http://.../wf.cgwl" name="anotherWF" />

  <subWorkflow name="innerSubWF">
    <dataIn name="in1"...>...</dataIn>
    <dataIn name="in2"...>...</dataIn>
    <body>
      <activity name="innerAct" type="type">
        <dataIn name="par1" source="innerSubWF/in1" />
        <dataIn name="par2" source="innerSubWF/in2" />
        <dataOut name="res" />
      </activity>
    </body>
    <dataOut name="out" source="innerAct/res" />
  </subWorkflow>

  <activity name="a1" type="innerSubWF">
    <dataIn name="in1" source="inmat1" />
    <dataIn name="in2" source="inmat2" />
    <dataOut name="out" />
  </activity>
  <activity name="a2" type="anotherWF:subWF">
    <dataIn name="in1" source="inmat1" />
    <dataIn name="in2" source="inmat2" />
    <dataOut name="out" />
  </activity>

</agwl-workflow>
```

Figure 18 Invoking sub-workflows

5. Modeling a Real World Material Science Workflow with AGWL

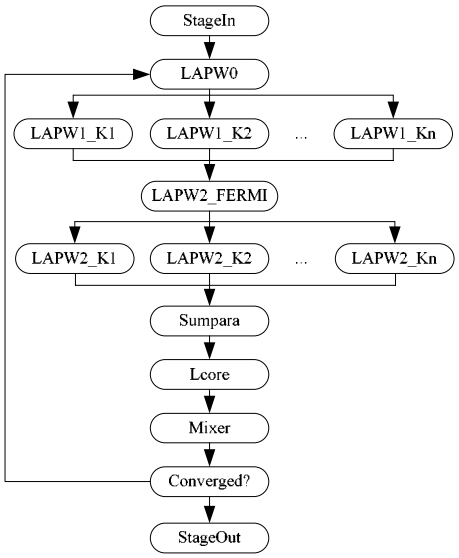


Figure 19 The WIEN2k workflow

WIEN2k [10] is a program package for performing electronic structure calculations of solids using density functional theory. The programs which compose the WIEN2k package are typically organized in a workflow illustrated in Figure 19. The `LAPW1` and `LAPW2` tasks can be executed in parallel with a number of *k-points* (atoms). A final task applied to several output files tests whether the problem convergence criterion is fulfilled (i.e. test whether the first number of the result of the `Mixer` task is "1"). The number of iterations for the convergence loop is unknown at compile time. Note that with only about one dozen AGWL activities we can describe a WIEN2k workflow

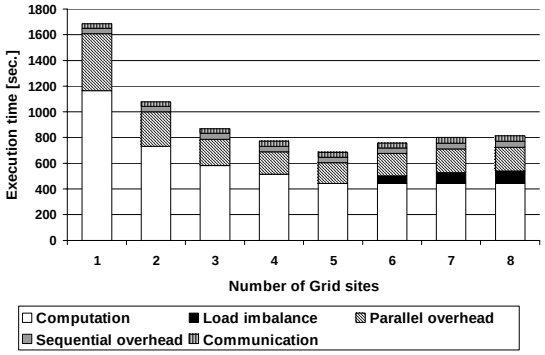
with several hundred activity instances (one for each *k-point* in both parallel loops).

```
<agwl>
  <importATD url=".../wien2k/wien2k.atd" name="w2kAtd"/>
  <workflow name="wfWien2k">
    <dataIn name="fileIn0" source="a repository"/>
    <!-- some other "dataIn"s for workflow "wfWien2k" -->
    <body>
      <activity name="Stagein" type="w2kAtd:Stagein">
        <!-- some "dataIn"s for activity "Stagein" -->
        <dataOut name="numOfProc" />
      </activity>

      <while name="whileConv">
        <!-- some "dataIn"s for while loop "whileConv" -->
        <dataIn name="val" loopSource="actConv/outVal">
          <value>true</value>
        </dataIn>
        <condition>val='true'</condition>
        <loopBody>
          <activity name="actLAPW0" type="w2kAtd:LAPW0">
            <!-- some "dataIn"s/"dataOut"s -->
            <dataOut name="outFileVsp" />
          </activity>

          <parallelFor name="pfLAPW1">
            <dataIn name="fVsp" source="actLAPW0/outFileVsp"/>
            <!-- some "dataIn"s -->
            <loopCounter name="i" from="0"
                         to="Stagein/numOfProc" step="1" />
            <loopBody>
              <activity name="actLAPW1" type="w2kAtd:LAPW1">
                <!-- some "dataIn"s/"dataOut"s -->
              </activity>
            </loopBody>
            <!-- some "dataOut"s for parallelFor "pfLAPW1" -->
          </parallelFor>

          <activity name="actLAPW2_FERMI"> ... </activity>

          <parallelFor name="pfLAPW2"> ... </parallelFor>

          <!-- some activities: Sumpara, ..., Mixer -->
          <activity name="Converged" type="w2kAtd:Converged">
            <!-- some other "dataIn"s and "dataOut"s -->
            <dataOut name="outVal"/>
          </activity>
        </loopBody>
        <dataOut name="outVal" source="whileConv/val"/>
      </while>

    </body>
    <dataOut name="outVal" source="whileConv/outVal" />
  </workflow>
</agwl>
```

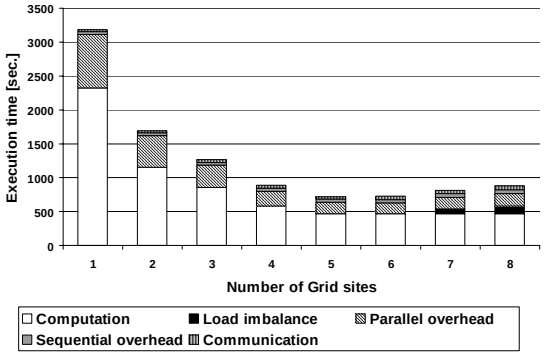
Figure 20 An excerpt of the WIEN2k AGWL representation

We developed an AGWL representation for the workflow and describe a representative excerpt of it in Figure 20. Firstly, the ATD file `wien2k.atd` is imported, in which the definition of activities like `LAPW0`, `LAPW1`, etc. are included. Next, the activity `StageIn` is invoked to prepare for the execution of the while loop `whileConv`. In this while loop, the activity `actLAPW1`, the `parallelFor` loop `pfLAPW1`, the activity `actLAPW2_FERMI`, the `parallelFor` loop `pfLAPW2` and the activities `Sumpara`, `LCore` and `Mixer` are invoked sequentially. The value of `val` to exit the loop can be changed after each loop by the data-out port of the activity `Converged`, which is refered by the `loopSource`. In the `parallelFor` loop `pfLAPW1` and `pfLAPW2`, the activities are executed in parallel. Finally, the `outVal` of the workflow `wfWien2k` is returned as the result.

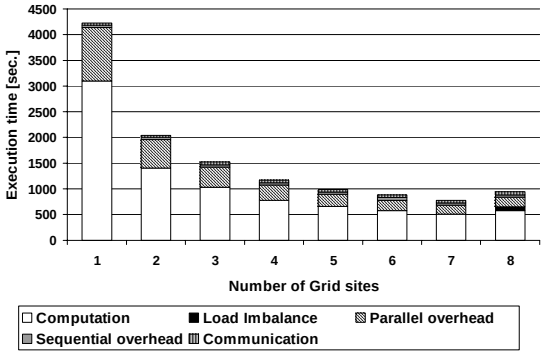
The AGWL representation of WIEN2k is compiled into a CGWL representation and executed in the ASKALON Grid environment [7], which is currently developed by the Distributed and Parallel Systems Group at the University of Innsbruck. ASKALON supports the performance-oriented execution of workflows specified in AGWL by providing a rich set



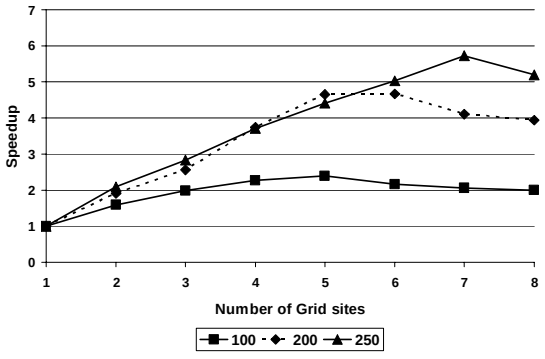
(a) 100 parallel *k-points*



(b) 200 parallel *k-points*



(c) 250 parallel *k-points*



(d) Speedup

Figure 21 Performance results of WIEN2k

of services, which includes *resource broker*, *resource monitor*, *information service*, *workflow executor*, *(meta-) scheduler*, *performance prediction*, and *performance analysis services*. All of these services are developed on top of a low-level Grid infrastructure implemented by the Globus toolkit, which provides a uniform platform for secure job submission, file transfer, discovery, and resource monitoring. ASKALON is deployed on the Austrian Grid infrastructure that aggregates several Grid sites across the country of Austria. We tested the performance of the WIEN2k Grid workflow respectively on 1, 2, 3, 4, 5, 6, 7 and 8 Grid sites. A large problem case (called *atype*) with three different problem sizes identified by the number of parallel *k-points* (100, 200 and 250) has been selected. Gescher, a local cluster used by the Wien2k scientists for their experiments, is used as the reference measurement for the single site execution. Figure 21 shows the performance results which demonstrate that by migrating from the local Gescher cluster to a distributed Grid environment, good performance results are achieved. The speedup improves with larger problem sizes indicated by the parallel *k-points* (see Figure 21(d)). The improvement comes from the parallel execution of *k-points* on multiple Grid sites that significantly decreases the computation time. The parallel overhead decreases by increasing the number of Grid sites with constant number of *k-points* because fewer tasks are scheduled to a single Grid site. The sequential overhead remains relatively constant, but its ratio to the overall execution time is smaller for large problem sizes. The communication overhead of this application is negligible since we schedule `LAPW1` and `LAPW2` to Grid sites with a single NFS file system. The communication overhead becomes more significant with increasing number of Grid sites hosting different file systems. Typically for a scalability study, the workflow performance deteriorates beyond a certain machine size (e.g., 6 for 100 or 200 *k-points*, and 8 for 250 *k-points*). This is due to sites containing slower processors that are part of the Grid infrastructure, which causes load imbalance of the workflow application.

6. Conclusions and Future Work

In this paper, we presented our work on the Abstract Grid Workflow Language (AGWL), which is a novel XML-based language for the specification of Grid workflow applications at a high level of abstraction. AGWL allows the user to concentrate on describing Grid workflows without dealing with implementation specific or low level details of the underlying Grid infrastructure. It is a powerful and user-oriented workflow language that has been tailored for performance-oriented Grid workflow applications. AGWL provides advanced workflow constructs to facilitate the parallel execution of workflows, to retrieve and store data from and to data repositories, and to modularize and reuse workflows. Properties and constraints can be specified to optimize and steer workflow execution by the underlying Grid middleware.

We have integrated AGWL in the ASKALON Grid application development and computing environment. AGWL has been extensively used for the specification of Grid workflow applications in the field of material science, river modeling, astrophysics, and finance modeling. We have also developed a UML-based graphical interface [11] for AGWL which supports the visual composition of workflows.

References

[1] I. Foster and C. Kesselman (editors), *The Grid: Blueprint for a New Computing Infrastructure*, Second Edition, Morgan Kaufmann Publishers, USA, 2004.

[2] T. Andrews, et al., Business Process Execution Language for Web Services Specification, version 1.1. IBM, Microsoft, BEA, SAP and Siebel Systems, May 5, 2003.

[3] Jia Yu and Rajkumar Buyya, A Novel Architecture for Realizing Grid Workflow using Tuple Spaces, *Proceeding of Fifth IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, PA, November 2004.

[4] Sriram Krishnan, Patrick Wagstrom and Gregor von Laszewski. GSFL: A Workflow Framework for Grid Services (Draft), Argonne National Laboratory, July 2002.

[5] P. Avery, I. Foster, GriPhyN Annual Report for 2003-2004, Technical report 2004-70, August 2004.

[6] J. Frey. Condor DAGMan: Handling Inter-Job Dependencies, 2002.

[7] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Junior, and Hong-Linh Truong. ASKALON: A Tool Set for Cluster and Grid Computing. Concurrency and Computation: Practice and Experience. John Wiley and Sons, 17/2-4, 2005. http://dps.uibk.ac.at/askalon.

[8] XSL Transformations (XSLT) version 1.0, http://www.w3.org/TR/xslt.

[9] XML Path Language (XPath) version 1.0, http://www.w3.org/TR/xpath.

[10] P. Blaha, K. Schwarz, G. Madsen, D. Kvasnicka, and J. Luitz. WIEN2k: An Augmented Plane Wave plus Local Orbitals Program for Calculating Crystal Properties. Institute of Physical and Theoretical Chemistry, Vienna University of Technology, 2001.

[11] S. Pllana, T. Fahringer, J. Testori, S. Benkner, and I. Brandic. Towards an UML Based Graphical Representation of Grid Work Applications. In The 2nd European Across Grids Conference, Nicosia, Cyprus, January 2004. Springer-Verlag.