INFORMATION SOCIETY TECHNOLOGIES
(IST)
PROGRAMME


# OpenMolGRID

## SPECIFICATION OF THE GRID INTERFACE FOR CLASSES OF APPLICATIONS TO SUPPORT AUTOMATED WORKFLOWS

| | |
|---|---|
| Contract Reference: | **IST-2001-37238** |
| Document identifier: | **OpenMolGRID-4-D4.2a-0109-0-4-MetaPlugin** |
| Date: | **21/11/2003** |
| Work package: | **WP 4: Grid Integration** |
| Partner | **UT, UU, NEGRI, FZJ, CGX** |
| Lead Partner | **FZJ** |
| Document status: | **Approved** |
| Classification: | **PUBLIC** |
| Deliverable identifier: | **D4.2a** |

Abstract: This document contains the specification of the UNICORE client components (MetaPlugin, Resource information provider, OpenMolGRID containers, workflow description language), UNICORE server components (application wrappers, metadata) that provide support for automated workflows in UNICORE

## Delivery Slip

| | Name | Partner | Date |
|---|---|---|---|
| **From** | B. Schuller | FZJ | 5/11/2003 |
| **Verified by** | M.Romberg | FZJ | 5/11/2003 |
| **Approved by** | G.H.F.Diercksen(TC)<br>R. Ferenczi (QE) | OMC<br>CGX | 21/11/2003<br>13/11/2003 |

## Document Log

| Issue | Date | Comment | Author |
|---|---|---|---|
| 0-0 | 14/10/2003 | initial version | M.Romberg, B.Schuller |
| 0-1 | 15/10/2003 | Structural changes | B. Schuller, M. Romberg |
| 0-2 | 05/11/2003 | | B. Schuller, M. Romberg |
| 0-3 | 17/11/2003 | | B. Schuller, M. Romberg |
| 0-4 | 21/11/2003 | | B. Schuller, M. Romberg |

## Document Change Log

| Issue | Item | Reason for Change |
|---|---|---|
| 0-2 | wording, spelling, clarity of presentation improved | internal review process |
| 0-3 | Appendix A, section 4 and figure 7 updated | internal review process |
| 0-4 | Typos corrected, figures 1 and 8 replaced by new version | Internal review process |

## Files

Files in this section relate to actual storage locations on the BSCW server located at https://hermes.chem.ut.ee/bscw/bscw.cgi. The URL below describes the location on BSCW from the root OpenMolGRID directory

| Software Products | User files / URL |
|---|---|
| Word 2000/XP | OpenMolGRID/Workpackage 4/Deliverables/ OpenMolGRID-4-D4.2a-0109-0-4-MetaPlugin |

## Project information

| | |
|---|---|
| Project acronym: | OpenMolGRID |
| Project full title: | Open Computing GRID for Molecular Science and Engineering |
| Proposal/Contract no.: | IST-2001-37238 |
| European Commission: | |
| Project Officer: | Annalisa Bogliolo |
| Address: | European Commission - DG Information Society<br>F2 - Grids for Complex Problem Solving<br>B-1049 Brussels - Belgium |
| Office: | BU31 4/79 |
| Phone: | +32 2295 81 31 |
| Fax: | +32 2299 17 49 |
| E-mail | annalisa.bogliolo@cec.eu.int |
| Project Coordinator: | Mathilde Romberg |
| Address: | Forschungszentrum Jülich GmbH<br>ZAM<br>D-52425 Jülich - Germany |
| Phone: | +49 2461 61 3703 |
| Fax: | +49 2461 61 6656 |
| E-mail | m.romberg@fz-juelich.de |

# Contents

# 1. Introduction

## 1.1. Purpose and scope

This document specifies the software architecture that is needed to add support for complex workflows to the base UNICORE system. Based on this architecture, the processes used within OpenMolGRID, such as molecular design and engineering, will be implemented.

An extension to the UNICORE client, called "MetaPlugin", is specified. It deals with workflow support and is supplemented by components for resource management. On the server side we specify the application layer (Abstract Resource Interface) needed for workflow support.

The architecture that is specified in this document is put into the general OpenMolGRID context in Deliverable D4.5a [1].

## 1.2. Document Overview

OpenMolGRID uses the basic UNICORE infrastructure and extends it in various ways to provide support for complex workflows.

Figure 1 shows an overview of the workflow support components specified in this document.

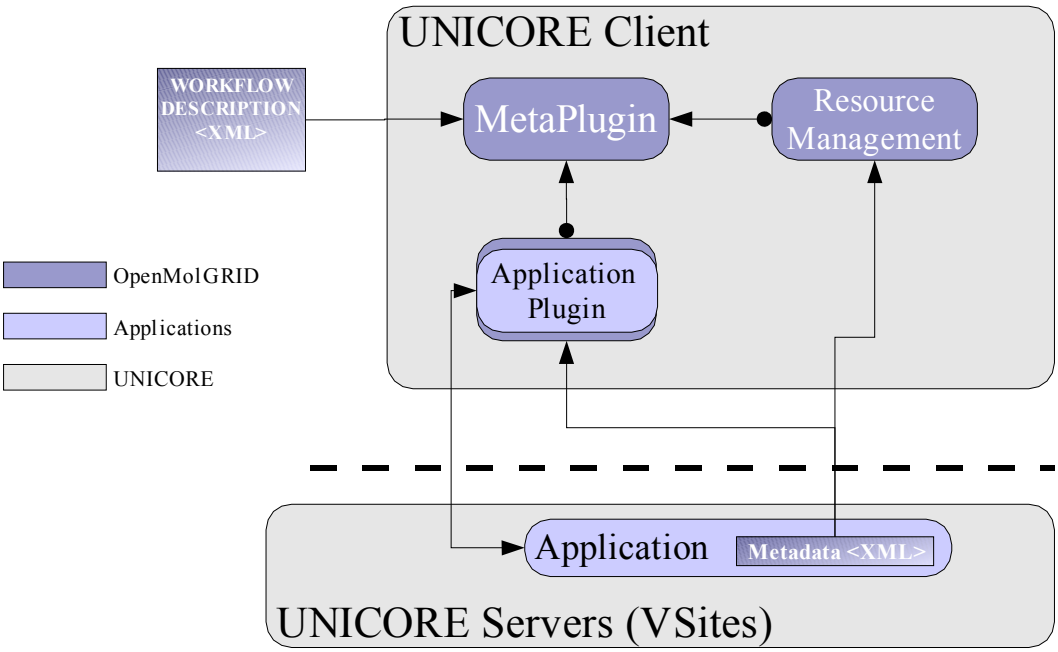


**Figure 1**: Overview of components for Workflow support

On the client side, workflow support is provided by an extension component to the base UNICORE client which is called MetaPlugin, and several supporting components.

The MetaPlugin basically provides a UNICORE *job group*, which can contain subjobs and subtasks of arbitrary complexity. However, in contrast to the standard job group, the MetaPlugin contains added functionality. This can be summarised in two main topics:

- Read workflows and build UNICORE jobs from them
- Find and allocate resources needed for the job

The workflows are specified in XML format which is read by the MetaPlugin. From this workflow a UNICORE job is build within the job preparation area of the client. A component for resource management is used to find UNICORE sites and other resources that are needed for the job. Application specific interfaces (Plugins) supporting automated workflows communicate with the MetaPlugin via a special interface.

On the server side, the software packages have to be wrapped by UNICORE applications. A metadata file describes the capabilities and specifics of the application to the client components.

### 1.3. Document Structure

The requirements that were used to design the workflow support components are given in section 2. Section 3 specifies the MetaPlugin. The interface between the MetaPlugin and the application plugins used by OpenMolGRID is specified in section 4. The server side application architecture is specified in section 5. Section 6 deals with resource management, specifiying the interfaces between the MetaPlugin and the resource management components.

# 2. Overview of Requirements

The OpenMolGRID workflow support is subject to various requirements, since it has to provide a certain functionality (functional requirements), it is embedded into the UNICORE infrastructure (non-functional requirements) and it has to offer its functionality to the user (user-interface requirements).

## 2.1. Functional requirements

Several use cases have been defined in WP2 and WP3 for their applications and workflows, which lead to the following functional requirements for the workflow support components.

The workflow support component has to

- Offer predefined workflow(s) with major tasks and dependencies being visible
- Allow modification of the workflow by the user
- Insert tasks for data conversion (i.e. input preparation appropriate for generation of 3D structures) wherever the application cannot take care of it by itself
- Look for matches between output file input file specifications of two sequential applications in the workflow and insert appropriate data conversion applications if necessary
- Allow for user intervention at predetermined positions in the workflow
- Distribute tasks to multiple Vsites
- Estimate resource requirements for single tasks (don't ask the user)
- Select application resource
- Select target site(s)
- Insert transfer tasks where necessary (i.e. when different Vsites for subsequent steps in the workflow have been selected)
- Store intermediate results of the major tasks

## 2.2. Non-functional Requirements

The OpenMolGRID client software is based on the UNICOREpro client release 1.0.7.

The plugins used as application specific interfaces have to be useable by themselves, i.e. outside of a workflow. This means that the workflow support in application plugins has to be an addition to the usual plugin interface, not a replacement for it.

## 2.3. User Interface Requirements

The user interface requirements have emerged as the result of intensive discussions on workflows during a technical meeting in April 2003:

- The job preparation menu provides an "Add OpenMolGRID Workflow" option which offers several predefined workflows, i.e. "Descriptor Calculation" or "3D Molfile and Mopac file generation" for loading
- The job tree is displayed on the left hand side
- The expanded job tree with dependencies is displayed on the right hand side of the job preparation window
- Automatically generated tasks are not displayed (wherever possible)
- User may edit the job tree
- User has to fill in additional input for major tasks
- The interface guides the user in finding the places where input is necessary
- Job monitoring displays status of all major tasks
- User may check for intermediate results of predetermined job steps, edit them, and release the subsequent step(s) in the workflow

These requirements as a whole are taken as foundation for the specification of the MetaPlugin.

# 3. Specification of the MetaPlugin and Supporting Software

The OpenMolGRID MetaPlugin is capable of dealing with arbitrary workflows. The workflows are formulated in an XML language, which is specified in Appendix A. The MetaPlugin reads such a workflow e.g. from a file and generates the UNICORE job tree from it. The following Figure 2 gives an overview of the roles and functionalities of the MetaPlugin.

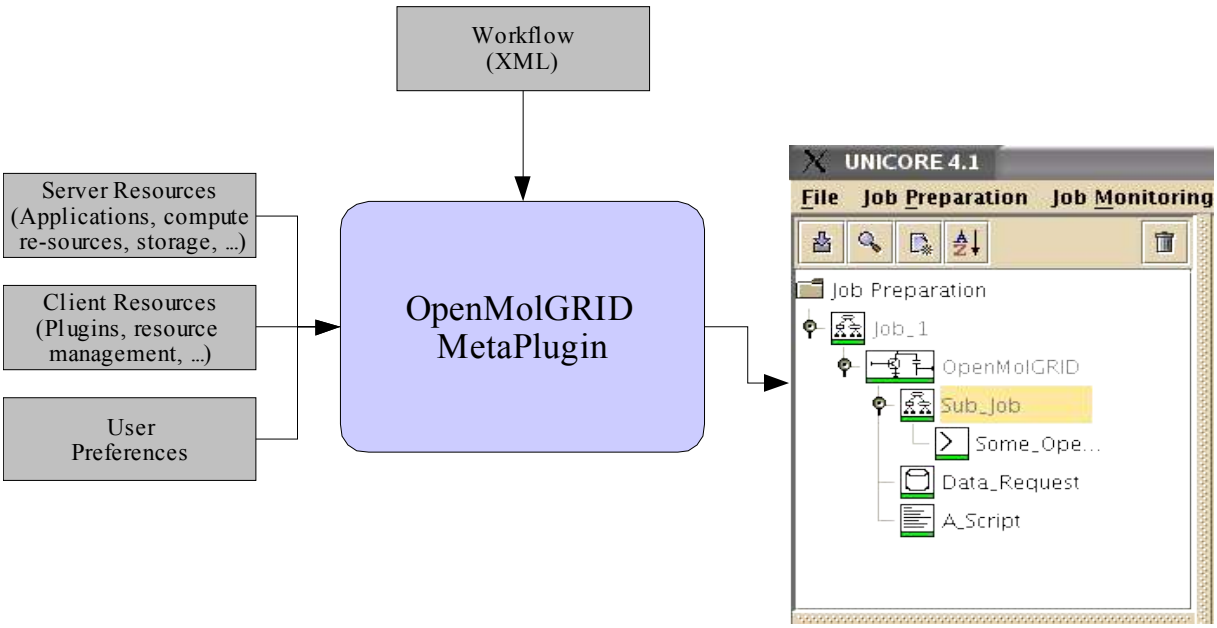


**Figure 2:** Functionality of the MetaPlugin

The following can be regarded as input to the MetaPlugin:

- Workflow description (in XML format)
- User preferences (defaults)
- Server resources (applications, compute resources, storage, databases, ...)
- Client resources (available plugins, resource management components, ...)

From this, the MetaPlugin generates a UNICORE job in the job preparation area (JPA) of the UNICORE client.

## 3.1. Workflows

Workflows consist of tasks, groups and dependencies, all of which have their UNICORE job preparation area (JPA) counterparts. The workflows in OpenMolGRID will be specified in an XML language, which is given in Appendix A: XML document style for workflow description. The present section defines the concepts used in the rest of this chapter.

### 3.1.1. Tasks

The workflows used in OpenMolGRID are based on the *task* concept. A task is a single step of a more complex whole, characterized by well defined input and output.

Tasks are supported client-side either by built-in client functionality (for example shell scripts, which is supported by the built-in ScriptTask) or by plugins.

Built-in UNICORE tasks relevant to OpenMolGRID are

- Script: shell scripts
- Hold: hold a job forever (i.e. until user manually initiates the job continuation)

Within OpenMolGRID, a lot of tasks will be defined, such as Database requests, decriptor calculation steps, 2D to 3D conversion of molecular structures, etc. The *name* of the task is used to match server side and client side resources in order to execute it, as will be explained in more detail in sections 3.3 and 3.4.

### 3.1.2. Groups

Groups are containers for tasks and other groups. UNICORE supports the following groups

- sub job: a container for subtasks and subgroups
- DoN: iterate contents *N* times
- DoRepeat: repeat until condition matches
- IfThenElse: decision making, with *Then* and *Else* subgroups

### 3.1.3. Dependencies

Dependencies indicate which job components have to wait for other job components to finish before they can be executed.

## 3.2. MetaPlugin: overview

The MetaPlugin will be realised as a UNICORE task plugin itself, and follow the plugin interface described in reference [2]. However, the MetaPlugin needs to provide a *jobgroup*, and not only a single task.
UNICORE task plugins consist of three main classes,

- The main plugin class
- The JPAPanel providing a GUI to the contents of one container in the Job Preparation area
- The container class that encapsulates the UNICORE AJO funtionality, and is used to store the current state of the GUI.

The hierarchy of the main classes of the MetaPlugin is shown in Figure 3.

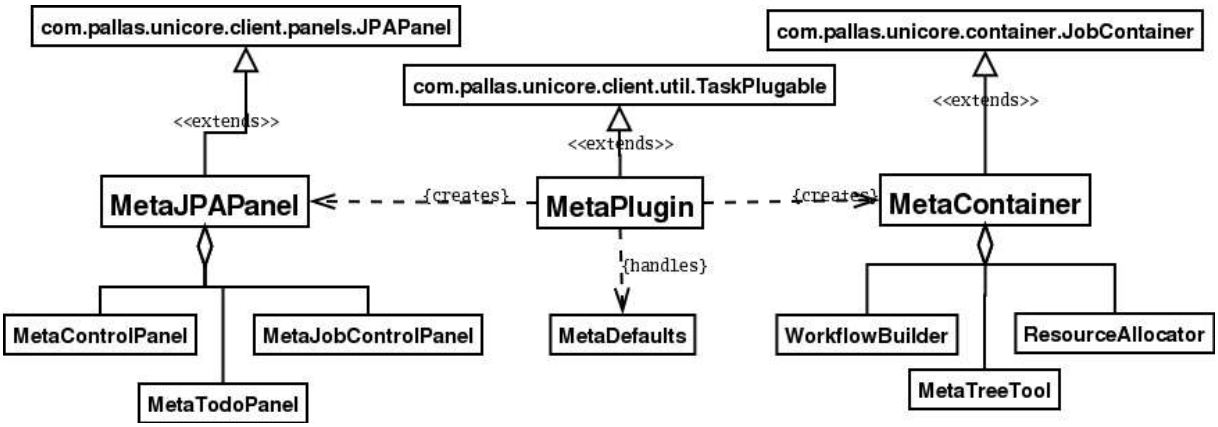


**Figure 3:** Hierarchy of the main MetaPlugin classes

The MetaPlugin is highly modularised, the various components will be specified in the rest of this chapter.

### 3.3. WorkflowBuilder

The WorkflowBuilder expects a workflow description in XML format as input, as specified in Appendix A: XML document style for workflow description. From this, it generates a job tree in the job preparation area.

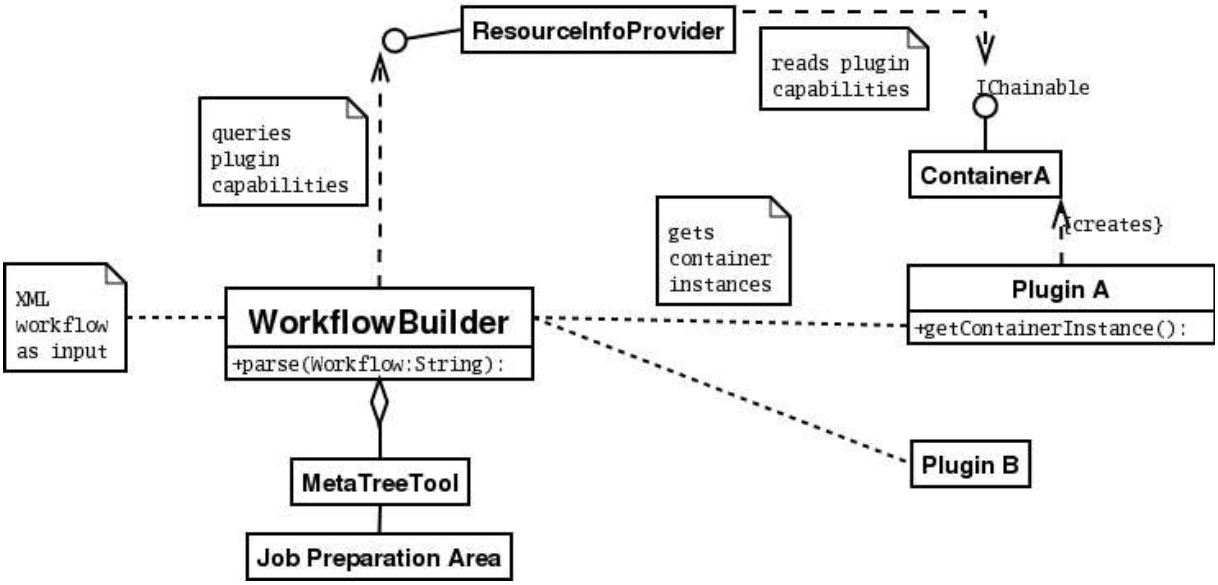


**Figure 4:** Functionality of the WorkflowBuilder

Figure 4 shows the basic functionality of the WorkflowBuilder.

- It provides a method `parse()` that takes a workflow in XML format, parses it, and builds a UNICORE job in the job preparation area of the UNICORE client
- It communicates with a ResourceInfoProvider, that can be queried about the capabilities (i.e. supported tasks) of plugins. The interface to the ResourceInfoProvider will be specified in section 6.1
- It gets container instances from the appropriate plugins, using the method `getContainerInstance()`
- It modifies the job preparation area (the JPATree) of the UNICORE client, e.g. by inserting containers. The JPATree modification is done via a utility class MetaTreeTool, since most of the functions are needed in the ResourceAllocator as well

### 3.4. ResourceAllocator

This component of the MetaPlugin deals with resource allocation for the job. In the simplest case this means finding out where (i.e. on which Vsites) the application resources needed for executing the job can be found. Then, the Vsite for each subjob and application for each task are set. Depending on the resources available, a restructuring of the job tree might become necessary.

To accomplish this, the MetaPlugin needs the resource information of all Vsites that are available to the user. These resources include the application resources with their metadata.

As a result of running the resource allocation process on a job, the job should be ready for immediate submission. All the dependencies should be correctly set, input and output files of subsequent job steps should match. However, there is some information that might be needed from the user, such as datasource to be accessed, or special parameters and for some applications. Therefore, it is not possible to have the ResourceAllocator generate a job that is immediately submittable.

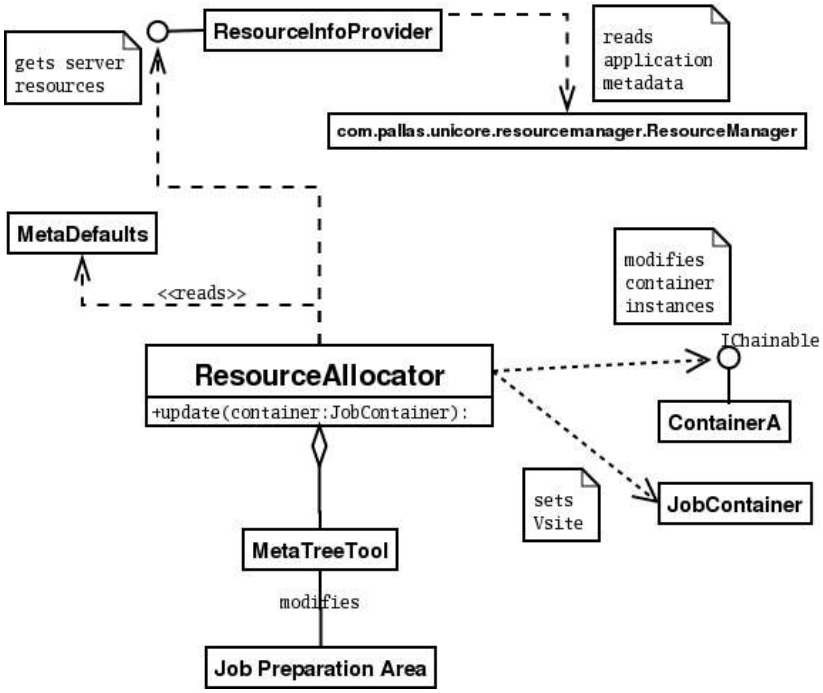


**Figure 5:** Functionality of the ResourceAllocator

Figure 5 shows the basic functionality of the ResourceAllocator. It provides a method `update (JobContainer job)` that takes a JobContainer as input. This method

- Finds one or more sites with the resources to execute the given (sub)job. For this it communicates with a ResourceInfoProvider using the interface specified in section 6.1.
- Matches input and output of the tasks contained in the JobContainer, based on the dependencies stored in the container. For this matching it uses an interface called IChainable, provided by OpenMolGRID task container classes, which is specified in section 4.1
- Sets the Application that tasks use, again making use of the IChainable interface
- Depending on the server resources found, it might become necessary to regroup tasks into subjobs, insert transfer tasks, etc. Therefore, the ResourceAllocator needs to modify the Job Preparation tree. For this it uses the MetaTreeTool utility class
- User preferences stored in the MetaDefaults class are taken into account. This can be used as a "conflict resolution" strategy, for example in case multiple possible Sites offer the same application, or multiple applications offer the same task

### 3.5. MetaContainer

The container class used for the MetaPlugin subclasses the JobContainer class, since it contains the full OpenMolGRID job in the form of the tasks and groups.

The functionality of the MetaContainer is provided in two methods.

- `parseWorkflow(workflow: String)` takes a workflow description in XML format as input, parses it and builds the corresponding UNICORE JPA tree
- `update()` updates the job based on available resources. This method corresponds to calling `ResourceAllocator.update()`, which has been described in section 3.4.

Figure 6 shows the MetaContainer class hierarchy.

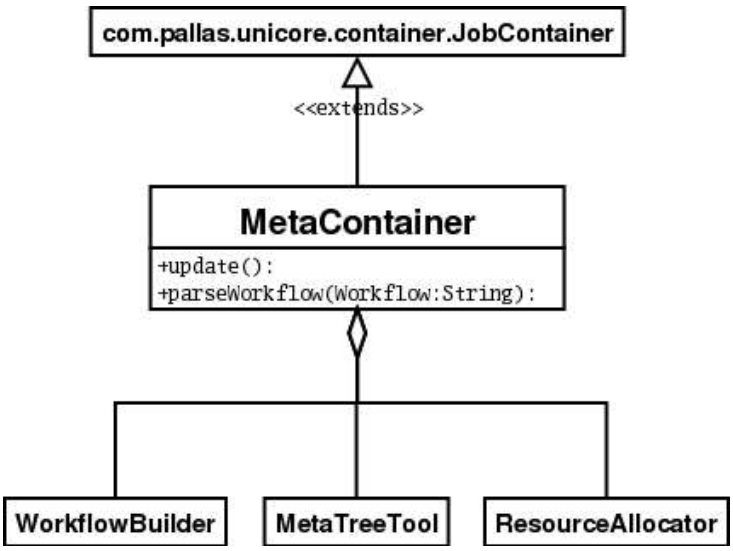


**Figure 6:** MetaPlugin container class hierarchy

### 3.6. The graphical user interface

The GUI for the MetaPlugin has three distinct areas.

1) Workflow control:
   - The user can select a workflow definition from a predefined list and load it into the job preparation area (and the job container)
   - The user can update the current contents of the job container.
   - The dependencies are shown
2) Job control, allowing access to basic UNICORE functions, such as manual USite/VSite selection and resource set editing
3) Information about the current status of the job preparation is shown to the user.

As usual in the UNICORE client, the state of the GUI is stored in the Container class, and can be saved and restored using the "File/Save(As...)" and "File/Load" functions of the client.

### 3.7. User Settings and Defaults Management

MetaPlugin default handling is done as usual in the UNICORE client: The main plugin class offers a function `getSettingsItem()` which is called by the client when the user selects "MetaPlugin defaults" in the "Settings" menu.

Defaults are used to store all user preferences relating to workflow management. These include, but are not limited to

- Preferred resource allocation schemes (see also section 6). These include preferred Sites, or preferred applications
- Predefined workflows. The MetaPlugin GUI offers a list of workflows. This list is editable and can be stored as part of the defaults.

# 4. OpenMolGRID application plugins

The MetaPlugin uses a special interface to the container classes designed for OpenMolGRID, in order to

- Query plugins for the tasks they support,
- Match input and output of subsequent tasks automatically,
- Set the application to be used
- Set and query the task that is to be executed
- Set additional options for a task

This interface, called `IChainable`, has to be implemented by application plugin container classes. It is used by various components in the workflow building and resource allocation process, as Figure 7 shows.

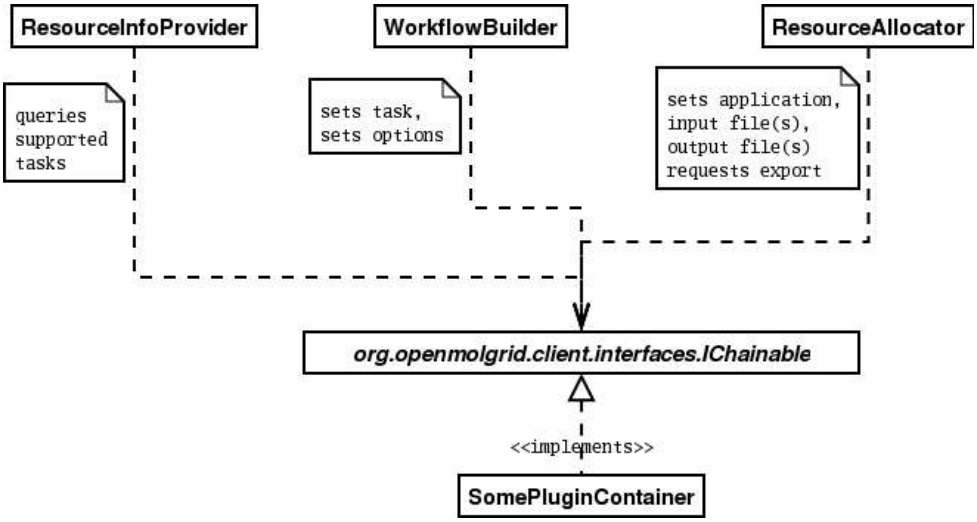


**Figure 7:** Uses of the IChainable interface

## 4.1. The IChainable interface

This interface deals with all the aspects necessary to set up a single task container in a complex job based on an abstract workflow description. The rest of this section provides a brief overview of the functionality offered by this interface. A more detailed description of the methods is given in the JavaDoc of the IChainable class, available on the project documentation server.

### 4.1.1. Task Control

In our terminology, a "task" is a specific, well-defined executable element of a workflow, such as "descriptor calculation or "Database request". A task is identified by its name only, and characterised by the input it needs and the output it produces.

A plugin has to advertise the tasks it supports. This is achieved by the function `getSupportedTasks()`. It returns an array of task names (as Strings).

The task to be executed can be set and queried using the `setTask()` and `getTask()` functions.

### 4.1.2. Application Control

The Application resource needed for executing the given task can be set (and queried again) using `setApplication()` and `getApplication()`.

### 4.1.3. Input Control

Name and type of input data files can be set using the function `addInputDataFile()`. A function `clearInputData()` is provided for resetting the input data.

### 4.1.4. Output control

An application produces zero or more files of one or more well-defined types. The methods `setOutfileName(String name, String fileType)` and `getOutfileName (String fileType)` are provided to set and get the name of the output file(s) of a given type. The method `getOutfileTypes()` can be used to query a list of all output file types.
The plugin is requested to export the outcome of the task by using the `doExport()` method, which takes a `boolean` parameter. Plugins are free to decide if and how they honor this request. In some cases, the plugin might export all results to the user's client machine, in some cases only one file might be exported.

# 5. Server side workflow support: Abstract Resource Interface

Software packages that are used to perform tasks in OpenMolGRID are wrapped with UNICORE Applications in a standardized way, in order to support automated workflow handling.
Metadata is used to convey information about the functionality offered by an Application (and thus a software package) to the UNICORE client. This information is used on the one hand by the workflow support components (WorkflowBuilder, ResourceAllocator), on the other hand it is used by the Application specific plugin.

## 5.1. Applications

In OpenMolGRID, software (SW) packages are integrated into UNICORE in a standardized way: a SW package is always accompanied by a wrapper (application) which takes standardized control input in XML format that tells the application how it has to proceed. The application gets the input for the SW package, which is in a standardized format wherever it makes sense. Output from the SW package has to be converted to a standard format analogously. The return codes from the SW package have to be provided to the outside world.

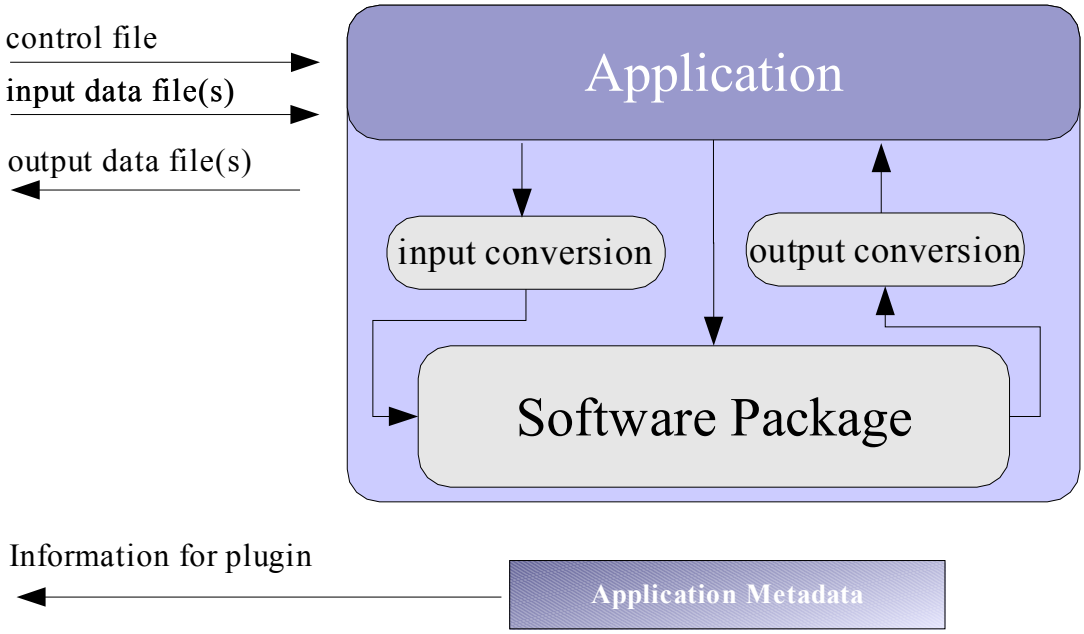


**Figure 8:** OpenMolGRID Application Architecture

An application supports one or more *tasks*, i.e. single steps of execution, characterised by input and output .

As *input,* it expects a "control file", and zero or more input data files of well-defined *types*.

As *output,* zero or more files of well-defined *types* are produced.

There are some points worth mentioning:

- "tasks" are identified by name only. i.e. tasks are considered equal when their names are equal.
- The Application "XML control files" are not considered in the list of input files. The rationale for this is that the XML control files are created by the Client plugin. This has a disadvantage: the description of the application on the server is not really complete. It is complete only in conjuction with the appropriate plugin. However, in the present state of UNICORE a more complete interface description does not make sense. As UNICORE moves towards the Open Grid Services

Architecture (OGSA), it will become clear how the interfaces to present services such as applications need to be described.

## 5.2. Application metadata specification

This section gives the present state of the metadata specification.

It is expected that the current metadata specification will be extended, especially with regard to resource description, and resource requirements of the applications.

### 5.2.1. Metadata overview

The following figure 9 gives an overview of the XML markup elements for application metadata.
The meaning of the tags used is given in section 5.2.3 below.

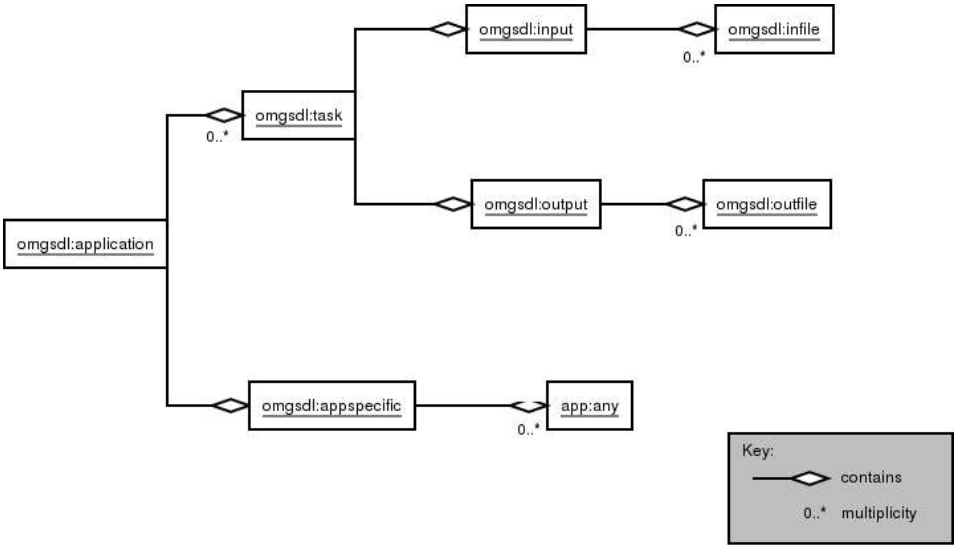


**Figure 9:** Metadata markup elements

### 5.2.2. Namespaces

Metadata contain information for workflow support *and* application plugin specific information, two namespaces are used.

| *Prefix* | *Namespace* | *Description* |
|---|---|---|
| omgsdl | http://www.openmolgrid.org/namespaces/omgsdl | Information for the MetaPlugin(workflow support) |
| app | application dependent | Application plugin specific information |

The "application dependent" metadata (in the "app" namespace) is ignored by the workflow support components. However, application plugins can use this metadata for their own purposes. It is up to the application and plugin developers to specify if and how application specific metadata are used.

### 5.2.3. Description of tags used in metadata

The following table gives tag names, their attributes and a description of the tag's meaning.

| ***Tag*** | ***Meaning*** |
|---|---|
| application | Base tag for the metadata file |
| task (subtag of "application")<br>attributes: name, description | Task description, name and description as attributes. |
| input (subtag of "task") | Input specification, can contain "infile" tags |
| output (subtag of "task") | Output specification, can contain "outfile" tags |
| infile (subtag of "input")<br>attributes: "type"<br>"use" | Input file specification. The "type" attribute defines the type. A list of types used within OMG will be compiled by the project partners.<br>The "use" attribute can be "required" or "optional" and indicates whether or not the file needs to be present. |
| outfile (subtag of "output")<br>attributes: "type"<br>"occurs" | Output file specification. The "type" attribute defines the type. The "occurs" attribute specifies that one or more of these files may be present in the output, occurs="multiple", or that output may be missing, i.e. occurs="optional". |
| appspecific (subtag of "application") | Contains application specific info, i.e. info intended for the Application plugin. This will usually be XML as well, in the "app" namespace. |

### 5.2.4. Example metadata file

This is a commented metadata example. It (partly) describes an application providing data type conversions.

```
<?xml version="1.0"?>

<omgsdl:application xmlns:omgsdl="http://www.openmolgrid.org/namespaces/omgsdl"
                    xmlns:app="http://www.openmolgrid.org/namespaces/fileop">
```

The namespaces used in the XML file are defined. the “omgsdl” namespace defines the metadata schema, the “app” namespace defines the elements pertaining to the application specific plugin.

```
  <omgsdl:task
    name="ExtractDataFromDataBaseRequest"
    description="Splits data base request output file into subfiles">
```

This defines a task by its name. A human-readable description is added.

```
    <omgsdl:input> <!-- describe input of tool -->
         <omgsdl:infile
              type= "http://www.openmolgrid.org/namespaces/dbat_output"
              use="required"/>
    </omgsdl:input>
```

This is an input definition. An input file of the given type (identified by an XML namespace) is required for the task.

```
    <omgsdl:output> <!-- describe output of tool -->
        <omgsdl:outfile type="OMG_ANYTYPE" occurs="multiple"/>
    </omgsdl:output>
```

The task produces one or more' output files of one, unspecified type.

```
  </omgsdl:task>

<!-- application specific information follows, can be left empty if not applic
    able -->
<!-- this part will be evaluated by the Client plugin handling the application
-->
  <omgsdl:appspecific>
```

The “appspecific” element contains content that is to be evaluated by the application specific plugin. The definition of the format is up to the application developers.

```
    <app:fileop_meta>
    </app:fileop_meta>
```

The content of the “appspecific” tag belongs to a different XML namespace. In this example there is no content:

```
  </omgsdl:appspecific>
</omgsdl:application>
```

# 6. Resource Management

The MetaPlugin needs resource information to find out where (on which Vsites) tasks can be run. On the client side, information about the capabilities of the available plugins has to be evaluated. This information is handled by a component called ResourceInformationProvider. It interfaces to the MetaPlugin (or other interested parties) via an interface IResourceInfoProvider.

## 6.1. The IResourceInfoProvider interface

This interface provides methods that clients can use to request information about server side resources and client plugin capabilities.

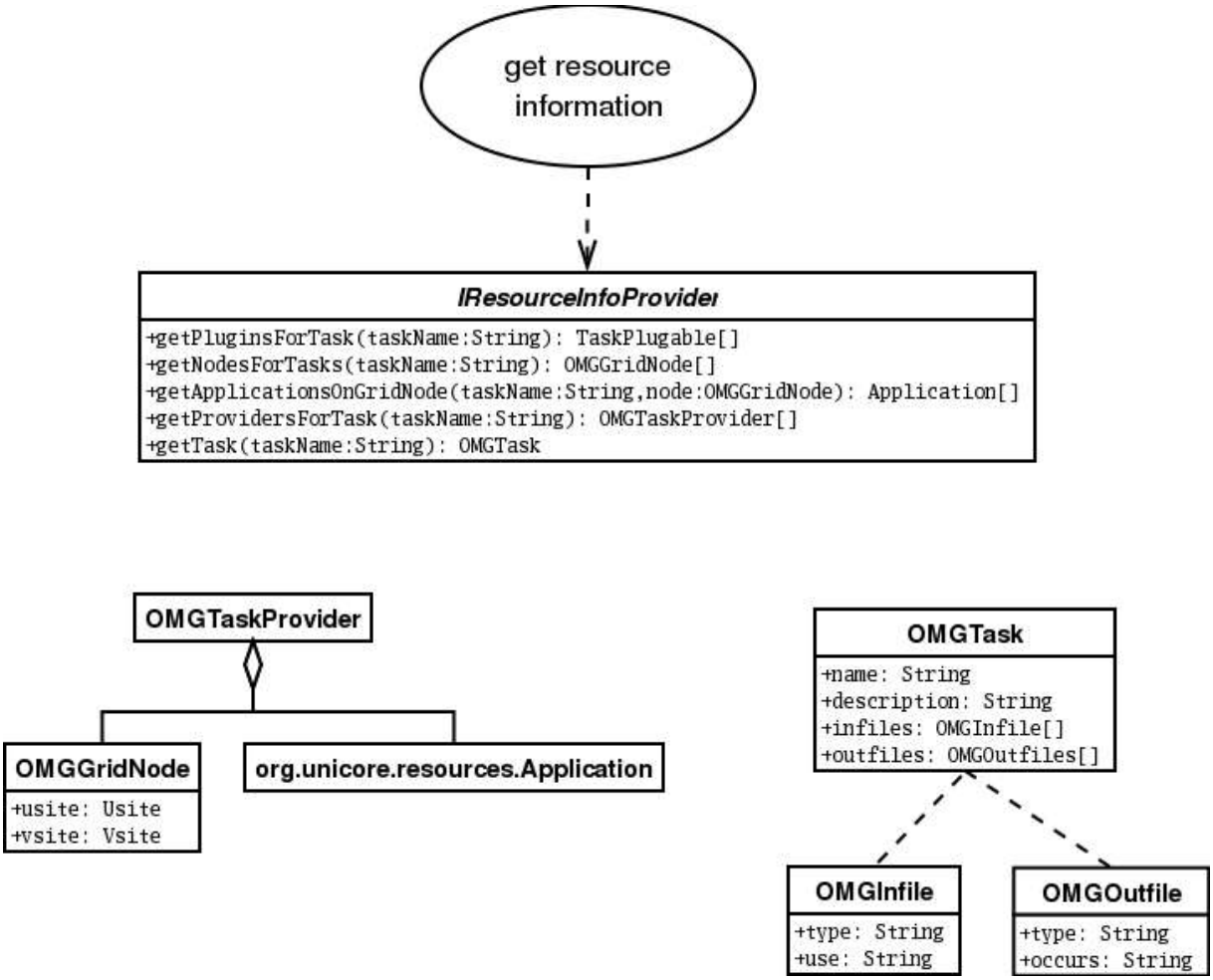


**Figure 10:** The IResourceInfoProvider interface and related classes

The following methods provide information about a given task identified by name:

- `getPluginsForTask()`: returns a list of client plugins that support the task
- `getProvidersForTask()`: returns a list of OMGTaskProvider objects containing information on which servers the task can be executed

A `OMGTaskProvider` specifies the grid node (UNICORE Usite and Vsite) and the Application that support the given task.

## 6.2. Getting a ResourceInfoProvider Instance

To make the MetaPlugin components independent of a specific ResourceInfoProvider implementation, a management component called RIPManager is introduced. A RIPManager instance can be provided as e.g. another UNICORE client plugin. The use of a manager for ResourceInfoProvider instances gives added flexibility and extensibility of the system.

The interface to a RIPManager offers two functions:

- `getResourceInfoProvider()` returns a ResourceInfoProvider instance
- `refreshResourceInfo()` updates the given ResourceInfoProvider instance

Figure 11 shows the process of getting a ResourceInfoProvider instance.

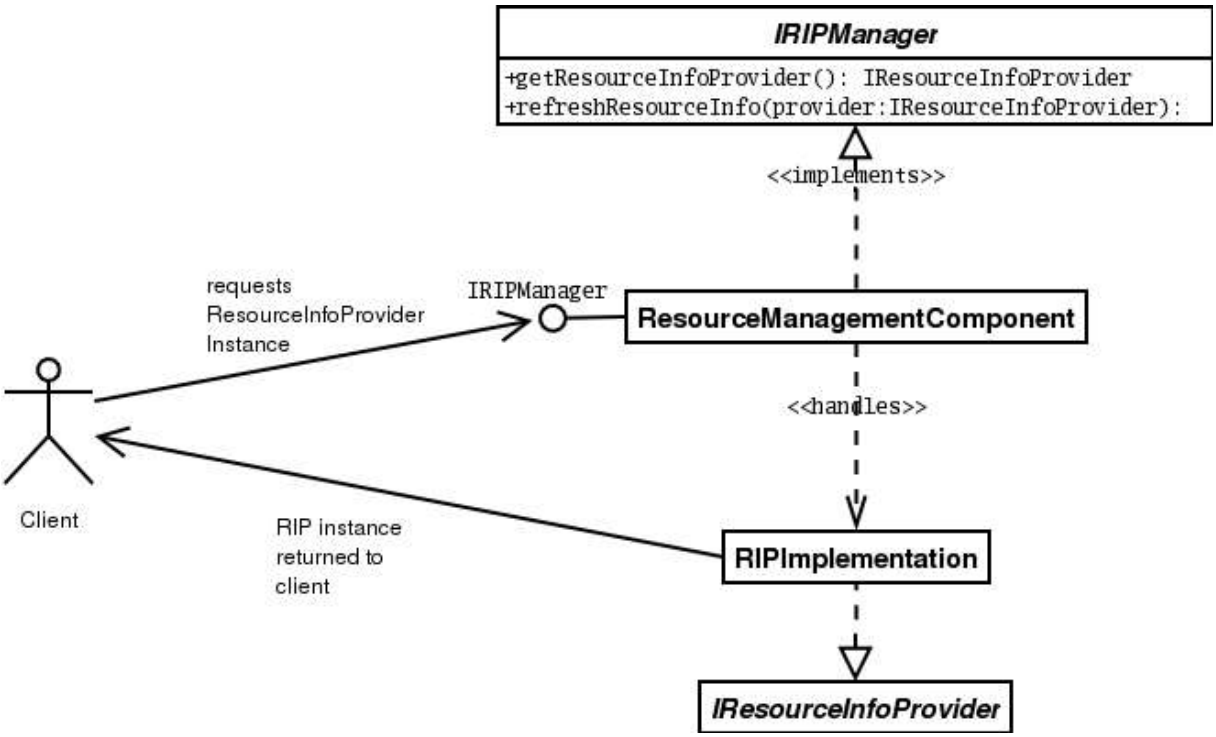


**Figure 11:** Getting a ResourceInfoProvider instance

## 7. References

[1]Deliverable D4.5a
Description of the OpenMolGRID Grid architecture, security architecture, and infrastructure and the deployment of the project's testbed

[2] UNICORE Plugin Programming Guide, Ralf Ratering (Intel Corp.)
available at http://www.unicore.org/downloads.htm (section "Plugins")

# 8. Terminology / Glossary

| | |
|---|---|
| **AJO** | Abstract Job Object |
| **Application** | Wrapper for a Software package, corresponds to UNICORE Application Resource |
| **CGX** | ComGenex |
| **FZJ** | Forschungszentrum Jülich |
| **GUI** | Graphical User Interface |
| **JMC** | Job Monitor Controller |
| **JPA** | Job Preparation Area |
| **JRE** | Java Runtime Environment |
| **MetaPlugin** | UNICORE Client Plugin supporting workflows |
| **NJS** | Network Job Supervisor |
| **SW package** | Software package, e.g. Codessa, Gaussian |
| **TSI** | Target System Interface |
| **UNICORE** | Uniform Interface to Computer Resources |
| **Usite** | UNICORE site, containing one or more Vsites |
| **UT** | University of Tartu |
| **UU** | University of Ulster |
| **Vsite** | UNICORE Virtual Site |
| **WP** | Work Package |
| **XML** | Extensible Markup Language |

# 9. Appendix A: XML document style for workflow description

Workflows consist of tasks, groups and dependencies. Every task and group element within a specific XML workflow document has a unique ID of type String. The ID is used to indicate dependencies.
The main tag (XML document type) is `<workflow>`.

## 9.1. Tasks

A task is given by name, identifier and ID, for example

```
<task name="DataBaseRequest" identifier="SomeQuery" id="1">
    ...
</task>
```

- The `name` attribute is used to identify the task within the system, i.e. to find the correct plugin and the correct server to handle the task.
- The `identifier` attribute will be used to identify this instance of the task to the user.
- The `id` must be unique within this workflow, otherwise an error will occur while the system parses the workflow file.

Tasks have optional attributes.

- `export="true"` requests that the output from the task shall be exported from the Uspace, and thus preserved after the job has finished,
- `split="true"` requests that the task shall be distributed to multiple Vsites.

### 9.1.1. Subtags

The `<task>` element can contain multiple `<option>` subtags, giving name/value pairs, as in

```
<task name="DataBaseRequest" identifier="SomeQuery" id="1">
    <option name="Option A" value="a"/>
    <option name="Option B" value="some other value"/>
    ...
</task>
```

- The supported options are dependent on the `name` of the task.

## 9.2. Options

Options are used to further refine `<task>` and `<group>` elements, depending on their `type`.
Options have the syntax

```
<option name="optionName" value="optionValue"/>
```

with `name` and `value` of type string.

## 9.3. Groups

Groups are given by type, name and id, for example

```
<group type="subjob" identifier="SubJob1" id="2">
    ...
</group>
```

- The `type` attribute corresponds to the group containers available in the UNICORE client: "subjob", "repeat", "doN", "if", "then", "else".
- the `identifier` attribute will identify this instance of the group to the user
- the `id` must be unique within the workflow

Groups can be distributed to multiple Vsites by setting the `split` attribute:

- `split="true"` requests that the group shall be distributed to multiple Vsites.

### 9.3.1. Subtags of `<group>`

Groups can have the following subtags:

- multiple <task> elements
- multiple <group> elements
- multiple <dependency> elements
- multiple <option> elements, where supported options depend on the type of group.

### 9.3.2. DoN group

In groups of type "doN", the number of iterations can be indicated as follows

```
<group type="doN" name="..." id="...">
<option name="iterations" value="N"/>
...
</group>
```

where N is a n integer

### 9.3.3. If group

In UNICORE, "If" groups are special, since they always have "then" and "else" subgroups, and the "if" group must not contain executable tasks.

The workflow description reflects this, since in <group type="if"> elements, "then" and "else" are mandatory, even if one of the branches is empty.

```
<group type="if" name="..." id="...">
...
   <group type="then" name="..." id="...">
    ...
   </group>
   <group type="else" name="..." id="...">
    ...
   </group>
</group>
```

### 9.3.4. Testing return codes

The group types "if" and "repeat" use a return code test, corresponding to the ReturnCodeTest in UNICORE.

This is specified using three options, as follows:

```
<group type="repeat" name="..." id="...">
   <option name="testTask" value="someTaskID"/>
   <option name="testType" value="equal|not_equal|successful|not_successful"/>
   <option name="testValue" value="someInteger"/>
...
</group>
```

The testValue is ignored in case testType is successful or not_successful.

## 9.4. Dependencies

Dependencies are used on top-level or within groups to indicate order of execution for tasks and groups. Dependencies are directed from a predecessor task or group to a successor task or group, identified by their id , respectively. The syntax is

```
<dependency pred="1" succ="2"/>
```

which indicates a dependency "1"->"2". Dependencies must be defined in the group that contains also the tasks or groups they refer to.

## 9.5. Summary

Figure 12 summarises the markup language used for workflow descriptions.

A document type definition (DTD) that can be used to validate workflows follows.

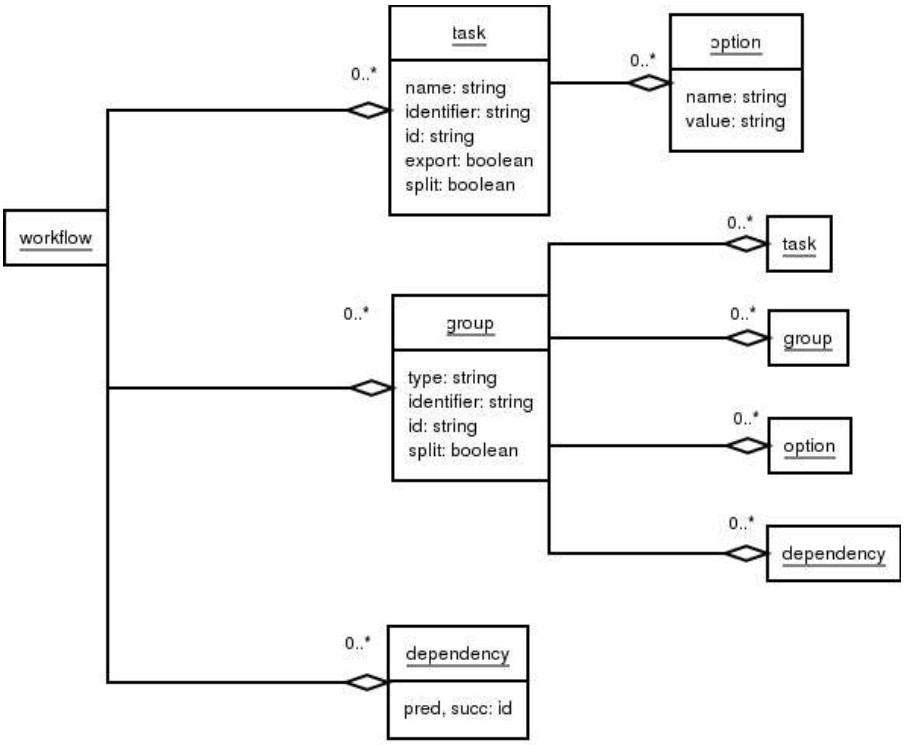


Figure 12: Workflow markup summary

```
<!ELEMENT workflow (task*, group*, dependency*)>
    <!ELEMENT task (option*)>
      <!ELEMENT option EMPTY>
        <!ATTLIST option
             name     CDATA #REQUIRED
             value    CDATA #REQUIRED
        >
      <!ATTLIST task
           name        CDATA #REQUIRED
           identifier  CDATA #REQUIRED
           id          CDATA #REQUIRED
           export      (true | false) #REQUIRED
           split       (true | false) #REQUIRED
      >
    <!ELEMENT group (option*, task*, group*, dependency*)>
      <!ATTLIST group
           type        (subjob | repeat | doN | if | then | else) #REQUIRED
           identifier  CDATA #REQUIRED
           id          CDATA #REQUIRED
           split       (true | false) #REQUIRED
      >
    <!ELEMENT dependency EMPTY>
      <!ATTLIST dependency
           pred        CDATA #REQUIRED
           succ        CDATA #REQUIRED
      >
```

Document Type Definition of workflow markup XML style