DEVELOPER MANUAL

FLOOD APPLICATION

WP1.2

Document Filename:	**CG1.2-DM-v0.9-IISAS-FloodAppDeveloperManual.doc**
Work package:	**WP1.2**
Partner(s):	**IISAS**
Lead Partner:	**IISAS**
Config ID:	**CG1.2-DM-v0.9-IISAS-FloodAppDeveloperManual.doc**
Document classification:	**PUBLIC**

Document Log

Version	Date	Summary of changes	Author
1.0	2004-01-10	First Draft	Branislav Šimo, Emil Gatial, Martin Mališka, Petr Slížik, Ondrej Habala

CONTENTS

1 COPYRIGHT NOTICE

This research is partly funded by the European Commission IST-2001-32243 Project “CrossGrid”.

2 INTRODUCTION

This developer manual describes the implementation of components of the flood forecasting application developed by II SAS in the context of the CrossGrid project. The flood forecasting application consists of simulation models, supporting services and user interfaces.

The flood forecasting application framework with appropriate simulation models enable users to easily run desired sequence of simulations and respective post-processing tools, browse the results of simulations, register results into the replica management service and applicable metadata into the metadata catalog for later search and retrieval.

> **Note:** The manual does not describe the implementation of individual simulation models as they have been developed by third parties and such description is beyond scope of this manual.
>
> More information on the flood application can be found on the project web page http://www.crossgrid.org/main.html and in the user manual on the page http://www.eu-crossgrid.org/user_manuals.htm (the page contains also user manuals for other applications and tools developed in the Crossgrid project).

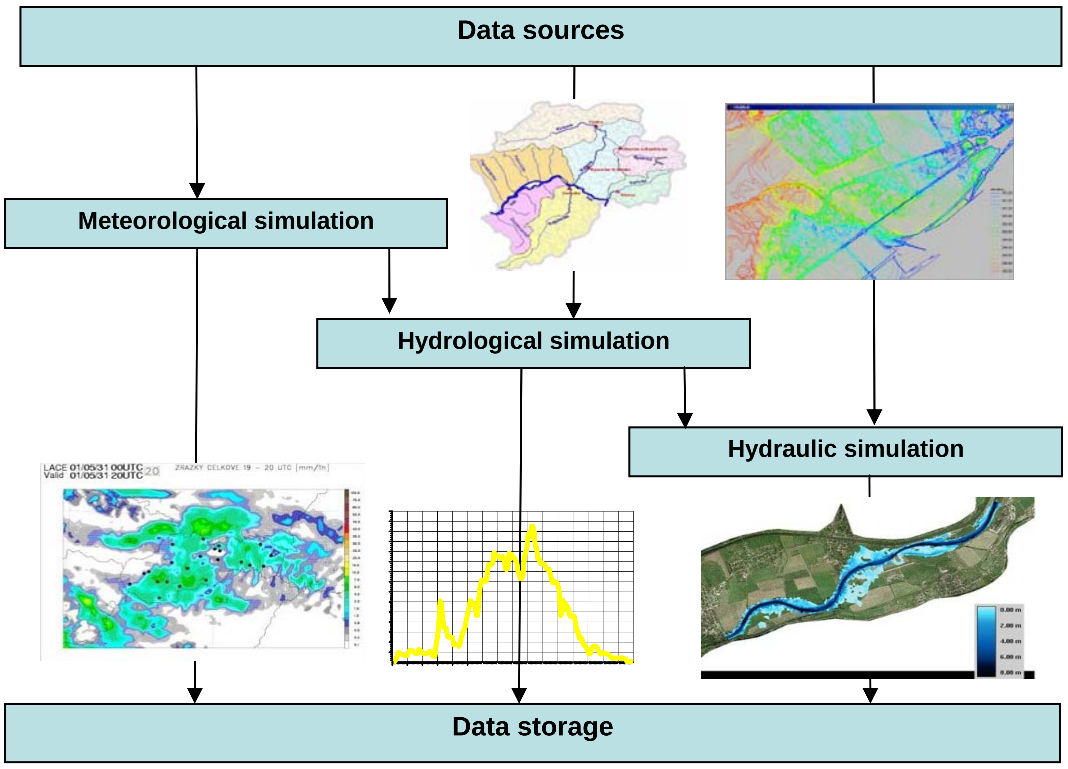


Fig. 2-1: Basic schema of flood application

The core of the application is formed by several simulation models (meteorological, hydrological and hydraulics) and appropriate post-processing tools connected together, so constituting a workflow. Meteorological model is used to forecast precipitation, which is used by hydrological model for computation of discharge of the river. That is used in the final step for the actual computation of possible flood by the hydraulics model. All the models generate binary output data, which are then used by post-processing tools to generate pictures visualizing the situation. Those pictures are then used by respective experts for situation evaluation. Overview of such workflow is shown on Fig. 1-1.

The core components are put together by a workflow service; metadata describing the datasets registered in the replica manager are stored in the metadata service. These two services are accessed from user interfaces implemented as:

- An application portal with standard web browser. It consists of a set of portlets – reusable web components – that are placed in the Jetspeed portlet portal framework.
- A Java plug-in for the Migrating Desktop.

While the portal interface focuses mainly on the flood application, MD is general tool that enables a user to work with grid in a flexible way. It also integrates other applications via its plug-in system. Schema of the whole system is shown on Fig. 1-2.

Note: For information about basic grid middleware tools like replica manager, job submission, user credential management, etc., see the LCG user manual and other documentation at http://grid-deployment.web.cern.ch/grid-deployment/cgi-bin/index.cgi?var=eis/docs.

In the following chapters we document the implementation of the application portal portlets, Migrating Desktop plug-in, workflow and metadata web services and visualization post-processing.

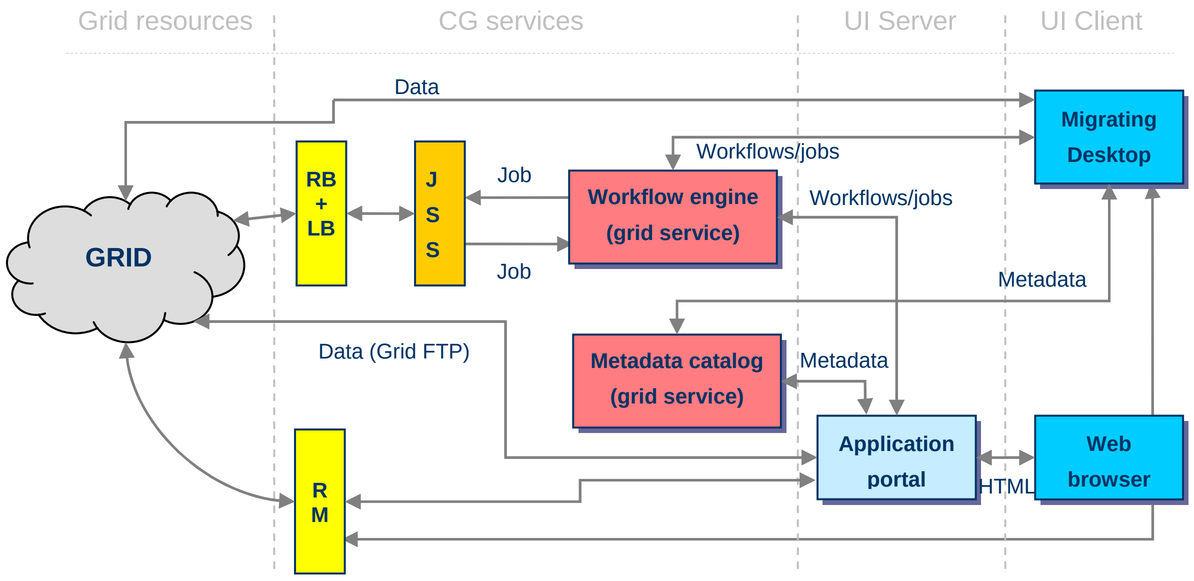


Fig. 2-2: Schema of the system components.

2.1 ABBREVIATIONS AND ACRONYMS

ALADIN	Meteorological model
ALADIN/LACE	ALADIN model for Central Europe
CAVE	Cave Automatic Virtual Environment
CE	Computing element (EDG)
CHAGAL	Graphical tool designed for visualization of 2D meteorological fields
CrossGrid/CG	The EU CrossGrid Project IST-2001-32243
DataGrid/EDG	The EU DataGrid Project IST-2000-25182
DaveF	Hydraulic model
DBMS	Database Management System
Flood-VO	Flood Virtual Organization
GRID	Grid framework for sharing of distributed resources
GUI	Graphical User Interface
GUID	Globally Unique Identifier
GVK	Grid Visualization Kernel
HSPF	Hydrological model
JDL	Job Description Language
Jetspeed	Java based Portal framework; uses portlets as components http://portals.apache.org/jetspeed-1/
LCG	LHC Computing Grid Project
MARMOT	MPI verification tool
MD	Migrating Desktop – grid desktop environment developed in Crossgrid project http://www.eu-crossgrid.org/user_manuals/MigratingDesktopUserGuide_200412.pdf
MM5	Meteorological simulation model
MPI	Message Passing Interface
MPICH-G2	Grid-enabled implementation of MPI
MPI-G2	Message Passing Interface with the globus2 device
NLC	Rainfall-runoff hydrological simulation model
OGCE	Open Grid Computing Environments Collaboratory
OGSI	Open Grid Services Infrastructure
PSE	Problem Solving Environment
RAS	Roaming Access Server
RB	Resource Broker
SE	Storage Element (EDG)
SHMI	Slovak HydroMeteorological Institute (Subcontractor of II SAS)
UI	User interface
WMO	World Meteorological Organization
workflow	Time and logical sequence of work items, which are processed by human agents or computers.
WN	Working Node
WS	Web services
XML	Extended Markup Language

2.2 REFERENCES AND SOURCE CODE

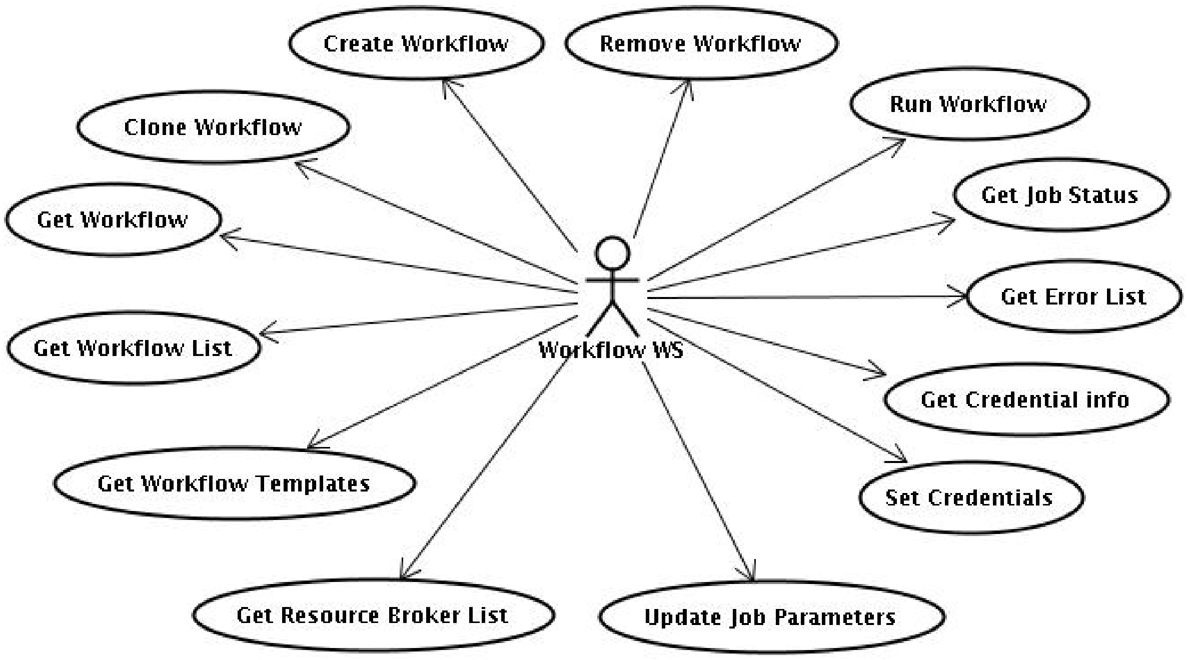
Source code of the application’s components is accessible on the public project CVS server and can be browsed via following https://savannah.fzk.de/cgi-bin/viewcvs.cgi/crossgrid/crossgrid/wp1/wp1_2-flood/.

There are several subdirectories in the repository:

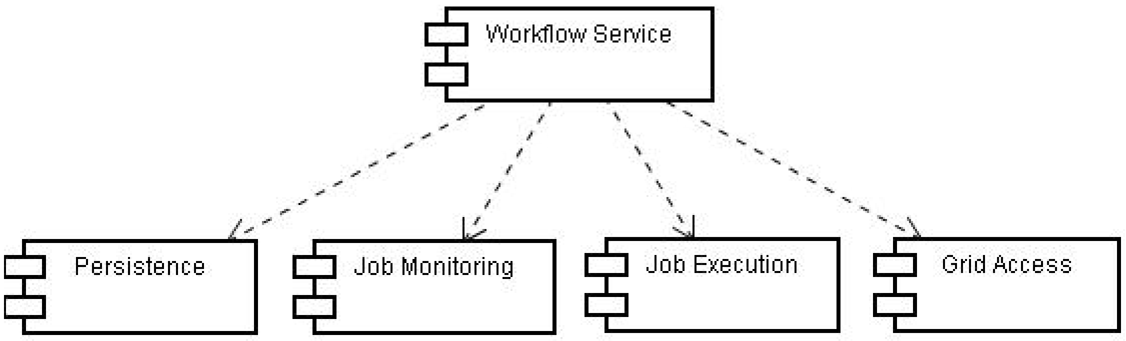
- FloodVoMetadata – metadata grid service
- OBSOLETE – obsolete components
- Commons – classes used by several components
- md_plugins – plug-ins for the Migrating desktop
- portlets – portlets for the Jetspeed portal framework
- utils – auxiliary tool(s)
- web_service – workflow grid service
- wrapper_scripts – wrapper scripts for simulations comprising the flood application

3 WORKFLOW SERVICE

3.1 PRODUCT USE CASES



3.2 PRODUCT COMPONENT MODEL



The workflow service is composite of four basic components. Component responsible for all operations on database is component *Persistence*. This component is based on Torque object to relational database mapping tool. Access to CrossGrid infrastructure is provided by component *Grid Access* through job submission service. Component *Job Monitoring* monitors executions of jobs in grid infrastructure and notifies Component *Job Execution* about job state change. Component *Job Execution* controls execution of workflow.

3.3 DETAILED IMPLEMENTATION MODEL

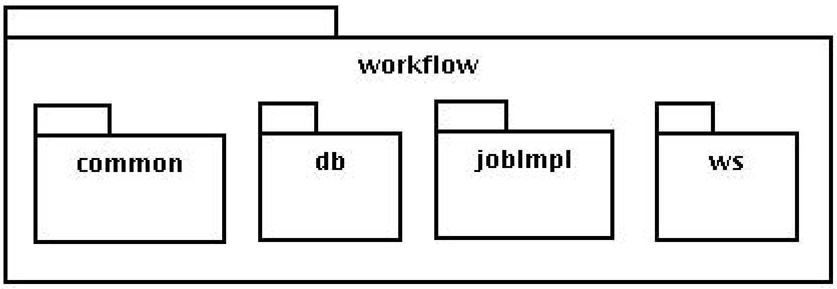


Fig. 3-1: workflow service package diagram

3.3.1 Workflow service packages overview

Main package Workflow comprises classes that cover all functionality provided by workflow service.

Class **ExecutionThread** is responsible for execution of one workflow. Each workflow has its own instance of *ExecutionThread*. This thread executes every job defined in workflow in order specified in this definition. State change driven policy is used, every job has its own object and that contains methods for determine state of job.

Class **JobMonitoringThread** is thread that periodically monitors execution of grid jobs. This class also holds map of all *ExecutionThread* instances accessible through workflow ID. When job state change occurs, *JobMonitoringThread* will notify appropriate *ExecutionThread* instance about this change.

Class **AbstractTask** – described in section 3.3.2

Class **ExceptionQueue** – this class controls the operations on queue of exceptions. Exception that can't be thrown directly to client (this are the exceptions that occurs in threads) are caught and stored into queue. Client should get list of these exceptions through workflow service interface.

Class **GridTools** – this class uses JSS (job submission service) for submitting job to grid, getting job status and cancelling job execution and provides this functionality for *ExecutionThread* and JobMonitoringThread.

Class **PersistenceLayer** – this class is responsible for storing all information about the workflow to database. Instead of a direct access to database, O/R mapping tool Torque was chosen. Because Torque objects have linkage to database, they cannot be migrated. To fix this dependency on the database, mapping mechanism is used to map Torque database object onto workflow service objects defined in packages *ws.*

Class **ServiceConfig** – this class encapsulates access to configuration file "service.properties" that contains default configuration for Grid and configuration of Log4J.

Class **DBtoWSMapper -** this class is responsible for mapping from Torque database objects which have dependencies on database to web service object.

3.3.1.1 Package common

This package contains classes with common meaning.

Class **StateName** provides functionality for conversion between grid job state ID and state name. This class also defines colour for each state used in GUI.

3.3.1.2 Package db

This package contains classes generated by Torque (Object to database mapping tool) that maps database tables onto java objects. Each database table has one class that holds data and one class (called Peer) that provides functionality for manipulation with data class.

3.3.1.3 Package jobImpl

This package contains classes inherited from *AbstractTask* that are implementations of workflow jobs. Basic implementation of job is *GridTask*, this is implementation for grid jobs. Use of this class is very simple, you only have to specify the name of script you want to execute and this should be done by defining job parameter *param_string* that have contain the name of script.

3.3.1.4 Package ws

This package contains workflow service interface definition and implementation, generated classes from workflow service WSDL.

Class WorkflowWS – this class contains implementation of workflow grid service interface, class is detailed described in section 3.4 (Product interfaces).

3.3.2 Job Implementation

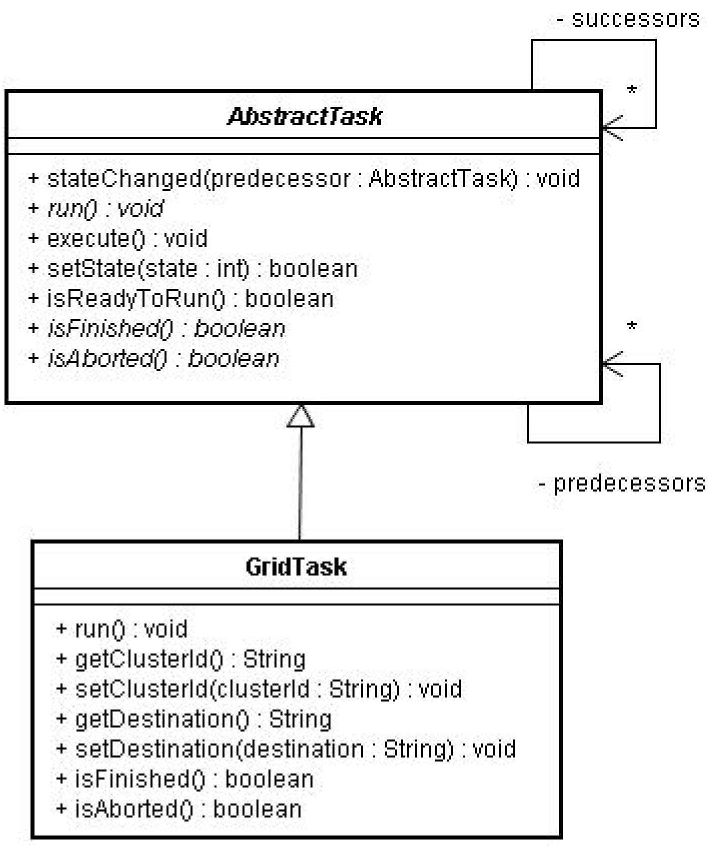
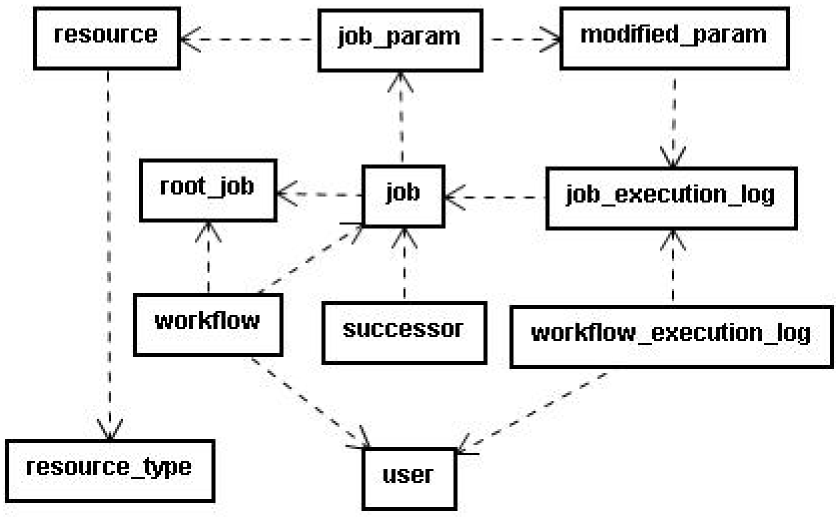

- successors
*
AbstractTask
+ stateChanged(predecessor : AbstractTask) : void
+ run() : void
+ execute() : void
+ setState(state : int) : boolean
+ isReadyToRun() : boolean
+ isFinished() : boolean
+ isAborted() : boolean
*
- predecessors
GridTask
+ run() : void
+ getClusterId() : String
+ setClusterId(clusterId : String) : void
+ getDestination() : String
+ setDestination(destination : String) : void
+ isFinished() : boolean
+ isAborted() : boolean

Fig. 3-2: Class diagram of Job implementation

On Figure 3-2 you can see class diagram of job implementation. Basic job class is class *AbstractTask*. Each job that wants to runnable by our workflow service has to inherit this class and implement this abstract methods: methods for identifying state of job – *isFinished()* (this method should return true value if job finished it's execution in normal way) and *isAborted*() (this method should return true value when job was cancelled or job finished it's execution abnormally), method *run()* that of course contains job action. We have made basic implementation of *AbstractTask* used for grid jobs – *GridTask*. To use this class, you need to specify the name of script to be executed by storing this name to job parameter called *param_string*.

3.3.3 Database Model



3.4 PRODUCT INTERFACES

3.4.1 setCredentials

Stores user credentials to database

Parameters:

`credentials` - credentials as byte array

3.4.2 setJobsParameters

Sets Job Parameters - if parameters with the same name exist, updates them.

Parameters:

`workflowID` - ID of workflow

`jobs` - array of Jobs

3.4.3 runWorkflow

Prepares all necessary things needed to run workflow - creates instance of ExecutionThread and assign choosen workflow to it, builds config files from job parameters for each job and finally starts execution of workflow.

Parameters:

`workflowID` - ID of workflow

3.4.4 getAllWorkflowTemplates

Gets array of workflow templates with included information about their jobs but without job parameters.

Returns:

array of Workflow Templates

3.4.5 getErrorList

Gets array of occured exceptions in JobMonitorinThread and ExecutionThread, because there is no other way how to throw exceptions to client from threads, client have to periodically call this method to identify if any error occured.

Returns:

array of exceptions

3.4.6 setJobParameters

Sets job parameters (updates existing job parameters)

Parameters:

`jobID` - ID of Job

`parameters` - array of Parameters

3.4.7 createWorkflow

Creates workflow from workflow template

Parameters:

`workflowTemplateID` - ID of workflow

`name` - name of workflow

Returns:

web service workflow object

3.4.8 getWorkflow

Gets Workflow by User ID and Workflow ID

Parameters:

`workflowID` - Workflow ID

Returns:

Workflow object

3.4.9 getCredentialInfo

Gets credential info

Returns:

WSCredentialInfo object

3.4.10 getStatus

Gets status of jobs

Parameters:

`workflowID` - ID of Workflow

Returns:

list of Jobs with current status

3.4.11 getRBList

Gets list of resource broker description

Returns:

array of descriptions

3.4.12 getWorkflowList

Gets list of Workflow

Returns:

array of Workflows

3.4.13 cloneWorkflow

Creates a copy of existing workflow.

Parameters:

`workflowId` - ID of workflow

`name` - name of workflow

Returns:

instance of copy

3.4.14 removeWorkflow

Removes workflow from database and stops execution thread

Parameters:

`workflowId` - ID of workflow

4 METADATA SERVICE

4.1 ARCHITECTURE

The CROSSGRID metadata service consists of 2 main components – the metadada database and the web service-based interface to this database. The interface allows for modification of the content of the database, as well as for object lookup by metadada constraints. All access to the database should be done entirely through the interface – no direct access to the database alone is necessary in order to use the metadata service.

The service interface is generic – it can connect to any underlying SQL database, providing this database conforms to certain structural rules (stated later). So the metadada service may be used for other applications than the CROSSGRID's Flood Forecasting Cascade – if the application of choice does not require any additional features not supported by the service interface.

The rest of this chapter describes the structure of the metadada database needed for correct work of the service interface, and the service interface itself.

4.2 METADADA DATABASE STRUCTURE

The database is deployed in the MySQL database server. For the OpenGIS geometry extensions to work, version 4.1 or later of MySQL database server is required. It contains a set of compulsory tables with specific structure, which in general describe the rest of the database. Another set of tables with specific structure may be needed for certain features of the Metadada Service, such as enumerated types. The Metadata Service supposes that the objects described by metadata are identified by a GUID and this paradigm is inherent part of the Metadata Service.

The main access point of the database is the table _METADATA_FIELDS. It describes all metadada fields present in this database.

Field Name	***Field Type***	***Field Description***
FieldID	int(11)	The ID number of the metadata field.
FieldName	varchar(250)	The name of the metadada field, for example „IdentificationInfo.Abstract“.
TableName	varchar(30)	The name of the table in which the data for this metadata field reside, for example „IDENTIFICATION_INFO“.
ColumnName	varchar(30)	The name of the column of the table *TableName* where the data for this metadata field reside, for example „Abstract“.
FieldType	varchar(10)	The type of the field, for example „string“. All available types will be described later.
MinOccurs	int(11)	Minimal required occurence of this metadata field. May be 0 (this field is not required) or 1 (this field is required). It is rarely necessary to use other values.

Field Name	*Field Type*	*Field Description*
MaxOccurs	int(11)	Maximal allowed occurence of this metadada field. It is usually 1 or 2147483647 (only one item for this field or any required number of items).

4.2.1 Field types

The column _METADATA_FIELDS.FieldType shows the type of the metadata field in question. It may be int, string, float, datetime, geometry or enum<name> - where name is the name of the custom enumerated type present in this database, for example „enumLang“ for the type of all available languages. The other types are standard MySQL types (see MySQL reference).

4.2.2 Enumerated types

The Metadata Service allows for custom datatypes, defined by enumeration. These types may be used for application-specific metadata fields, for example a database dealing with colors may use enumeration enumBColors consisting of the values „Black, White, Red, Green, Blue“. These enumerations are transparent to the user – he sees them as regular character strings, integer or floating point numbers, date/times or geographical values. They may be fixed or extensible – the fixed ones are not allowed to contain any item not already present in the enumeration definition table.

The enumeration definition table is another compulsory element of the database design. It is named _ENUMERATIONS:

Field Name	*Field Type*	*Field Description*
EnumName	char(100)	Name of the enumeration. For example „enumBColors“.
TableName	char(100)	Name of the table where the values of this type are enumerated. For example „BasicColorNames“.
ValueType	char(30)	Type of the values of this enumeration. For our color example, it would be probably „string“.
IsStatic	int(11)	This flag controls whether the enumeration is pre-defined and cannot be changed, or whether new values may be added. If this field is not zero, then the service interface will not add new rows to the <TableName> table.

Please note that the enumeration construct is useful not only for the creation of custom datatypes, but also for saving space and complexity of the database. Certain – otherwise cumbersome – metadata fields (for example names or Dns of people) may be defined as extensible (IsStatic = 0) enumerations and the actual metadata table will then contain only numbers instead of long, highly duplicit strings.

4.2.3 Compulosory, enumeration and other tables

There are two tables that have to be present in the metadata database - _METADATA_FIELDS and _ENUMERATIONS. All other tables are defined within these tables. The table

_METADATA_FIELDS contains references to all metadata fields, and the items of these fields are stored in separate tables, defined by the application. Also the table _ENUMERATIONS contains references to other application-defined tables with the actual enumerated types.

4.3 SERVICE INTERFACE

The Metadata Service is accesible through a web service-based interface, conforming to the OGSA standard and developed on top of the Globus Toolkit 3.0.2 WS Core (see Globus Toolkit 3). The interface provides several methods for metadata database access – modification, lookup, information retrieval. The service interface is implemented in the Java(tm) language.

4.3.1.1 Class FloodVoMetadata

This class implements the core of the service interface. It provides methods that the user may call – using WS/OGSA protocols – to access the metadata database. Some public methods return integer result code, where 0 means that the method completed succesfully and negative numbers represent error or warning codes.

4.3.1.1.1 FloodVoMetadata Methods

public int AddObject(

java.lang.String newGUID,

int numberOfProperties,

java.lang.Object[] properties)

This methods adds a new object, whose GUID is stated by the *newGUID* parameter. The metadata description of the object – its properties – are defined in the *properties* parameter as strings in the form <field name> = <value>, for example „IdentificationInfo.Language = English“. The parameter *numberOfProperties* defines number of (relevant) items in the *properties* field.

public int DeleteObject(java.lang.String GUIDtoDelete)

This method removes the object whose GUID is stated by the *GUIDtoDelete* parameter and all its metadata items from the database.

public java.lang.String FindObject(java.lang.String constraints)

This methods allows for searching of the metadata database for items conforming to a set of constraints. These constraints are described in the *constraints* parameter – one constraint per line. Each constraint is in the form <field name> <operator> <value>. Field name is the name of the metadata field to check for compliance. Operator may be one of relational operators =, !=, <=, >=. Value is the value to check for (SQL wildcards may be used). Example of such a constraint is „IdentificationInfo.Language < „%ish“ “.

The return value is a string with the list of conforming objects – one GUID per line.

public int ModifyObject(

java.lang.String GUIDtoModify,

int numberOfProperties,

java.lang.Object[] properties)

This method allows the description of an existing object to be modified. The object whose metadata is to be changed is identified by the *GUIDtoModify* parameter. Otherwise the method works the same way as the AddObject method – all existing metadata items are removed and new ones (in the *properties* parameter) are inserted into the database.

public java.lang.String PrintObject(java.lang.String GUIDtoPrint)

This method return a complete metadata description (all relevant metadata items) of an object. The object to display is identified by the *GUIDtoPrint* parameter.

The result of this method is a string with the list of metadata items describing the object in question. One metadata item per line, in the form <field name> = <value>.

public java.lang.String ShowStructure(java.lang.Object x)

This method return the complete structure of the database, so a user interface module may provide a comfortable way to access the metadata database. The parameter x is unused in the current version of the service interface.

The result of this method is a string with the list of all metadata fields defined in the database. Each field is on a new line in the form <field name>:<type><enumeration>. The <enumeration> element is present only on lines describing fields with enumerated values. The <enumeration> element is a list of all defined values, enclosed either in () or in <>, depending on whether the enumeration is extensible or static. Example is „IdentificationInfo.Language:string(„English“,“Slovak“,“German“).

5 PORTLETS

5.1 PRODUCT COMPONENT MODEL

This paragraph describes component model of portlets in the Flood portal and OGCE portal (OGCE). Both of the portals are build upon Jetspeed (Jetpseed) framework (an Open Source implementation of an Enterprise Information Portal). Flood portal contains portlets that work with flood application through developed Web services and GRID environment.

Each of the developed portlets are based upon Velocity template engine that provides a viable alternative to Java Server Pages (JSPs). It is designed especially for web developers to ease building process of Web pages.

Development of portlet is compound of three parts:

1. Configuration file: Each portlet has to register portlet resources within the portal framework, in our case Jetspeed. Configuration file is an XML document (see example below) that can contain multiple portlet entries in one configuration file. Each entry specifies properties of a protlet, such as tile and description (displayed during the portlet selection), used class and parameters such as template name, portlet's action class and other customizable parameters. Each parameter, except the customizable parameters, is obligatory. Be sure that you don't made any mistake in configuration file notation, because this is very common type of errors (this can be identified that your velocity template is not processed). Customizable parameters generally contain default prarameter required by the portlet. Its value could be different in different domain of deployment. Please, fill the correct value to ensure the correct operation of a particular portlet. An example of configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<registry>
<portlet-entry name="WorkflowPortletAction" hidden="false" type="ref" parent="Velocity"
                                            application="false">
       <meta-info>
              <title> Workflow Portlet </title>
              <description> Jetspeed portlet for job workflow management
              </description>
       </meta-info>

       <classname>org.apache.jetspeed.portal.portlets.VelocityPortlet</classname>
       <parameter name="template" value="workflow-portlet" hidden="true"
cachedOnName="true" cachedOnValue="true"/>
       <parameter name="action" value="portlets.WorkflowPortletAction" hidden="true"
cachedOnName="true" cachedOnValue="true"/>
       <parameter name="serviceHandle"
value="http://portal.ui.sav.sk:8090/ogsa/services/org/crossgrid/wp12/WorkflowService"
hidden="false" cachedOnName="true" cachedOnValue="true"/>
       <media-type ref="html"/>
       <url cachedOnURL="true"/>
</portlet-entry>
</registry>
```

2. Action class: The buildNormalContext method must be extended to pass variables into template context. This method is called whenever the content of portlet must be refreshed. The class can contain action methods, that processes actions invoked from generated portlet page. The possible invocation methods are described later.

3. Velocity template: Velocity template is made of simplified scripting language (conditions, cycles) that is embedded into HTML code. Full description of Velocity framework is available on (Velocity).

Instead of describing every particular portlet's behaviour, it is convenient to describe general invocation methods, because every portlet implementation is based upon common features. Further detailed description of particular portlet will be provided in section 5.4 – Detailed implementation model.

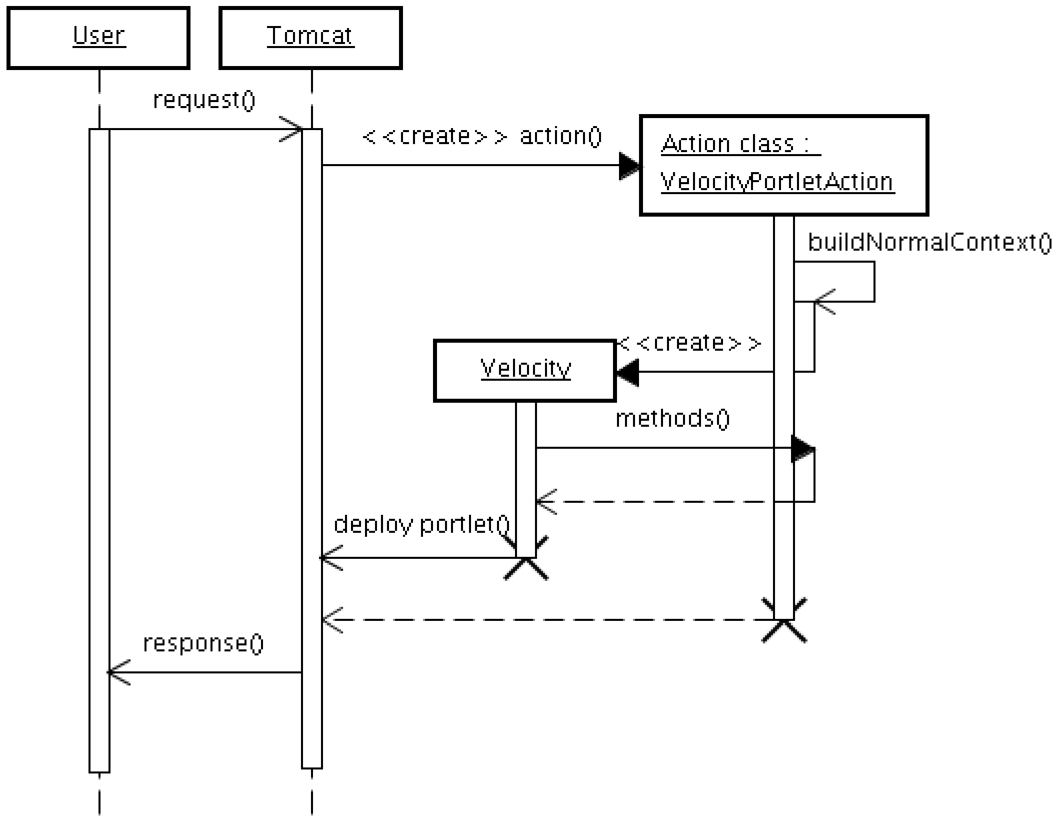


Fig. 5-1: Sequence diagram of general portlet method invocation.

5.2 GENERAL PORTLET INVOCATION METHODS

Every time the user sends an event to server (usually by clicking any portlet button), the appropriate action is invoked to process the action and action's parameters. From the client browser point of view, the action can be invoked by using the following two methods:

- POST method: On one hand, this method is suitable when we need to process many of form's attributes. An action that has to be performed is stored in the action parameter of form tag, for example:
```
<form method=”POST” action=”$jslink.getAction(“portlets.TestPortletAction”)”>
...
<input type=”hidden” name=”hidden_test_value” value=”20”/>
<input type=”text” name=”test_text” size=”20”/>
<input type=”submit” name=”eventSubmit_doProcess_action” value=”Process Test Action”/>
</form>
```
After the form is submitted to server, the action class is able to find out every parameter written within the scope of form tag.
- GET method: On the other hand, the GET method is used to transfer only several parameters to be processed by the server. Usually, the portlet only tell the server to invoke the specified action with some parameters, for example:
```
<a href=”$jslink.getAction(“porlets.TestPortletAction”)
.addQueryData(“eventSubmit_doProcess_action”, “hello”)
.addQueryData(“value”, 10)”>Process Test Action</a>
```

As it is written in previous example, such action could contain more than one parameter, but it is unusual to transfer grater amount of data by this way.

Both of the methods invoke the same action *doProcess_action*. Note, that the prefix *eventSubmit_* is a standard notation that is followed with the identifier of an action.

After the action finishes its execution, the method *buildNormalContext* is invoked to generate portlet's content. This method has to be invoked every time it is necessary to refresh the portlet.

In practise it means that the *buildNormalContext* must pass parameters to the Velocity context. The particular portlet (piece of HTML code) is generated according to such parameters.

5.3 DIFERENCIES IN OGCE PORTAL

Portlets in OGCE portal differs from that ones deployed in Jetspeed framework in several details. The first very important difference covers the different processing of configuration file. In Jetspeed framework, it could be done by checking for change in the portlet configuration by calling the method *PortletSessionState.getPortletConfigChanged*. OGCE portal API doesn't provide such interface, so the configuration must be done by extending the method *buildConfigureContext*, which is invoked whenever the customization portlet is generated (usually by pressing the “magic wand” in the upper-right corner of a portlet). This method is responsible for changing the appropriate customize template, initializing and passing the variables into the context of customize template. Consequently, the user can change parameters of the portlet and submit the change. It is done by calling the action method, usually it is called *doUpdate* (but it is not necessary fulfil such notation). The *doUpdate* action could validate changed parameters and store them into persistent storage for user. Described configuration technique is much more complicated according to the one used in Jetspeed portal, but it provides developer with better control over the customize process and makes the portlet code clearer.

Other useful documents and source codes for Jetspeed can be found at (Alliance) and (OGCE). We also recommend to take close up look into the source code of OGCE portlets to find out another useful techniques and gain new experiences in the domain of portlet development.

5.4 DETAILED IMPLEMENTATION MODEL

Description of functionalities of particular porlet (buildNormalContext and every action – very briefly, because there are a lot of actions).

5.4.1 Wrokflow Template Portlet

Configuration file

Parameter name	***Parameter value***
Template	workflow-template-portlet
Action	portlets.WorkflowTemplatePortletAction
serviceHandle	http://portal.ui.sav.sk:8090/ogsa/services/org/crossgrid/wp12/WorkflowService

Action class

The method *buildNormalContext* checks whether the *serviceHandle* has changed according to the last remembered state. If it does, new configuration value is loaded from configuration file and it is established new connection to the web service. Note, that serviceHandler is stored in *httpSession* environment that is because the portlet must ensure the valid connection to workflow web service every time the portlet session is created.

The rest of the method is quite straightforward, it only exports values, such as error message, web service handle and utilities (provides a helpful methods for Velocity template) into the context.

The very important actions are *doRoll_down* and *doRoll_up*. The first one remembers the state for the workflow template (rolled or unrolled) by marking the identifier of currently rolled workflow template into the httpSession environment and the second removes such identifier. This provides the most convenient mechanism how to unambiguously mark the roll and unroll states. The same mechanism is used for the WorkflowPortlet that is described in the section 5.4.2.

Finally, the action that selects the desired workflow is *doSubmit_wft*. It takes an identifier of the workflow template and calls the client method of web service with such identifier to create concrete workflow.

There is another one private method *connectToWfService* that connects to the web service, which is used only by the buildNormalContext.

Velocity template

At first, the Velocity template checks for an *error* variable and if it exists then some error occurred in portlet, which had generated the context of velocity template and the description of the error is stored as the value of the *error* variable.

If no error occurred then the presence of *ws_interface* variable is checked, consequently the main block is executed to generate workflow templates. Every workflow template is displayed in HTML table as the name of workflow template.

The small arrow is placed on the right side of the workflow template row that provides mechanism to view the content on workflow template (a set of job templates). The action for an arrow picture is invoked by the GET method (as described in section 5.2.1 b.). If the arrow button is pressed the event *doRoll_down* or *doRoll_up* is invoked its last state (as described in Action class paragraph). Such state of arrow button is determine the shown picture (arrow up or arrow down).

5.4.2 WROKFLOW PORTLET

Configuration file

Parameter name	***Parameter value***
Template	workflow-portlet
Action	portlets.WorkflowPortletAction
serviceHandle	http://portal.ui.sav.sk:8090/ogsa/services/org/crossgrid/wp12/WorkflowService

Action class

The method *buildNormalContext* is similar to the one described for Workflow Template Portlet, in the way of processing the configuration parameter *serviceHandle*. Moreover, it exposes the same parameters, because the portlet is built upon the object provided by the workflow service (*serviceHandle*).

There is the same mechanism for remembering the state of portlet's rolling feature (rolled or unrolled) as described in 3.3.1. The rest of methods of action class provide actions for cloning (method *doClone*), removing (method *doRemove*) and updating (method *doUpdate*) the concrete workflow displayed in the portlet. The functionality of the first two actions is clear, because it only takes a parameter from the action event a calls the appropriate method of workflow service. The *doUpdate* method is designed to process displayed parameters. Such parameters must be identified by the unique

Id, that is compounded of three elements (see Figure 5-2), the identifier of the particular workflow (wf_id), the identifier of a job (job_id) and the indentifier of a parameter (param_id). The ID for displayed parameter is a string value delimited by the dash character "-", for example:

If the *doUpdate* action is invoked, then it processes every parameter for given workflow and saves it to the persistent storage via a workflow service.

Of course, there is the clear action called *doSubmit_wf* that submits the workflow to workflow service.

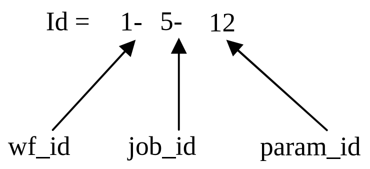


Fig. 5-2: Identifier structure of workflow parameters.

Velocity template

Velocity template for the WorkflowPortlet generates the portlet's content by the similar way as the template for WorkflowTemplatePortlet. Here, the functionality is extended by possibility to display the jobs of workflow and parameters of every job through rolling mechanism described above. There are two hidden input HTML tags, that serves as identifiers of workflow and its name for parameter processing.

5.4.3 VISUALIZATION/BROWSER PORTLET

Configuration file

Parameter name	***Parameter value***
template	visualization-portlet
action	portlets.WorkflowPortletAction
paneId	P-f979ae48f8-10000
basePath	/flatfiles/SE00/flood-vo/portal/jobs
storageHost	flood-vo.ui.sav.sk
metadataServiceHandle	http://portal.ui.sav.sk:8090/ogsa/services/org/crossgrid/wp12 /metadata/FloodVoMetadataService

Action class

Fisrt of all, the method *buildNormalContext* must initialize (if they weren't initialized earlier) variables such as processed template, storage host, base path and metadata service handler from configuration file. This method provide variables for two templates a visualzation-portlet (main functionality of this portlet) and a visualization-portlet-register (for metadata creation of replica file). For each of them, the *buildNormalContext* method must put correct variables to its context. Therefore, the *ACTUAL_TEMPLATE* static variable is used to recognize the currently processed template and choose the parameters that are passed to its context.

In this action class, there are a lot of action methods and its full description could produce huge amount unclear text, therefore the next paragraph considers only interesting or complicated features of some of notable methods. The others, usually, take the parameters from user's request and pass them to the appropriate method of service.

The goal of *doChange_dir* method is to change the directory, which content is shown in the portlet. Base directory is given by the *basePath* parameter. User is able to browse the files within the subdirectory structure of *basePath*. Any attempt to change the directory, that is not subdirectory of *basePath* (by applying the ".." in the path) is restricted.

The doView_output is very interesting method, because it is called from the action class of WrokflowPortlet. It is responsible for skipping to the portlet within the portal. To do this, it is inevitable to know the *paneId* parameter. Consequently, the portal is redirected to this pane.

Visualization portlet is used to register specified file in the replica catalog, therefore it must use some features implemented in metadata portlet such as Records class, and actions that are described in section 3.3.4 more in detail.

Velocity templates

Main portlet template displays the edit box for manual change of directory path and list of files and directories for actual directory, on the left side. User is able to view the graphical or textual files, in a window located on the right side of the portlet, by clicking on its file name (the *doChange_dir* method is called). If actual directory contains any pictures, than its list (*anim_files)* is put into the context of visualization-template and user is able to view such animation in main window. It is done by JavaScript language.

5.4.4 METADATA PORTLET

Configuration file

Parameter name	*Parameter value*
template	metadata-portlet
action	portlets.MetadataPortletAction
metadataServiceHandle	http://portal.ui.sav.sk:8090/ogsa/services/org/crossgrid/wp12/ metadata/FloodVoMetadataService

Action class

The *buildNormalContext* method initializes variable of metadata service handle in the same way as in VisualizationPortlet. Besides, it initializes the table-like data structure (*Records* class) that provides storage for metadata records obtained from metadata service. This is useful, because it is not always necessary to request the same information twice. The *buildNormalContext* is responsible for passing the correct variables to the context of three Velocity templates (metadata-portlet, metadata-portlet-add and metadata-portlet-query) and switching over them.

One of the most important actions is *doView_record*, which provides list of replicas for given metadata record along with the metadata records. The method *daesWrapper.listReplicasDAB* finds the replicas with data access bandwidth values. As one of its parameters, it requires user proxy stored in file x509up_<user_name> in the /tmp directory.

The set of conditions, restricting the selection only for such records that conform to any of the condition, could be applied to find desired metadata list. The most important methods that enable such selection feature are the *doAdd_constraint_query*, *doRemove_constraint_query*, *doFind_query*, which goal is straightforward according to its name. They commonly use the *METADATA_CONSTRAINT* static variable to pass the list of constraints through the user's temporary storage (the Turbine service). The *FloodVoMetadataClient.getObjectsByConstraints* method is called with the constraint as the parameter to find or preview the list of records.

The last type of the action methods enable to create new metadata record and store it via the metadata service. At first, the *Record* data structure is created to store metadata items. When the user fills every desired items of the form, the action *doSubmit_add* is invoked, that parse, verify and store the inserted values. Maybe it could be unclear, what this method really does, so let's explain the functionality a bit.

It iterates through the list of possible parameters and in every iteration it processes a one parameter. Parameters are composed of several elements (see Figure 4), similarly as it was described in section 5.5.2. Following figure shows, how are the elements organized:

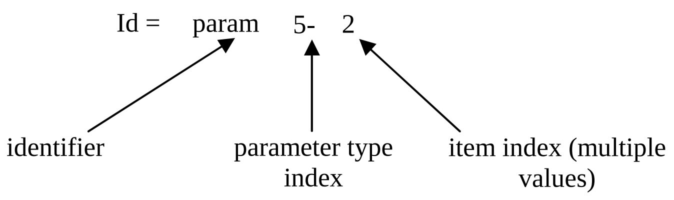


Fig. 5-3: Identifier structure of items in metadata creation form.

If the parameter could contain multiple values, than the method iterates the values of the list, until no value was found in the list. If it couldn't contain multiple values, the last index is not present, therefore the last element in the item identifier is missing. Simple validation of cardinality restrictions (minimal and maximal number of elements in the item) are done during iteration. At the end of successful iteration, the created structure is parsed and stored via metadata service.

Velocity templates

The main Velocity template serves as a main panel for metadata navigation, that enables to browse metadata resords, downloading replicas, removing replicas and metadata records. It enables to select desired metadata records by giving the restrictive conditions on the records in the metadata-template-query. Here, a user is able to preview and select the particular record. The metadata-template-add template enables to create of new metadata. In this stage the JavaScript functions (located in directory ${PORTLET_APPLICATION}/javascript) stores some information, for example the selected records, into the browser cookie to remember last state.

6 MIGRATING DESKTOP PLUGIN

The plugin for Migrating Desktop (MD) provides a user interface for workflow and metadata services.

7 VRMLVIS

The aim of the `vrmlvis` program is to produce 3D VRML scenes from the DaveF simulations. The whole program is written in pure Python, avoiding thus any dependance on other software packages. The Python version 2.3.0 or later is required. For work with images, Python Imaging Library (PIL) has been used.

The `vrmlvis` program employs a few interesting underlying concepts. This guide deals only with these concepts; the actual implementation of classes and functions is described in comments in the source code in detail.

7.1 REGISTRY

The registry is a simple persistent data object, designed to permanently store arbitrary values. Technically speaking, it is an ordinary dictionary and it is implemented straightforwardly with the Python `shelve` standard module. The data stored in the registry are available between consecutive program executions. Externally, the persistent data are stored in the `vrmlvis.reg` file (located in the program's current directory). The registry can store every value that can be dealt with by the Python's standard `pickle` module.

7.2 BATCH PROCESSING AND BUFFERING

The execution of the program resembles batch processing of files. During the execution, the input data are transformed into an intermediate form. The intermediate files are then processed, and transformed into another intermediate form. This procedure can be repeated several times, depending on the format of the input data or the required format of the output data.

As the execution of the program takes too long (approx. tens of minutes), a sophisticated buffering system has been designed to save as much computational time as possible. The intermediate results of each processing step that took place so far are stored on the hard drive. During the next run of the program, the intermediate files that left from the previous run are checked for their accuracy and reliability. As far as the stored data are up-to-date, they are used. Only the first occurrence of inaccurate data triggers new computations.

During the development, the buffering mechanism has been redesigned several times. In its current version, it supports work with intermediate data stored in external files and with intermediate data stored in internal variables and/or registry. All the functionality, related to the buffering mechanism, is defined in the `DepsChecker` class. A script, that wishes to use the buffering mechanism, has to instantiate this class first. The whole complex computation should than be split into several steps. The following computation steps should be dependent on each other only in terms of data stored in variables and/or files. The `checkDeps()` function then checks, if the number of inputs the output is dependent on has changed. If the number remained the same, then the check is carried out, if the sources are exactly the same (in case of files) and then, their actual accuracy is tested. Variables are checked according to their values; in case of external files, their timestamps (dates of last change maintained by the underlying operating system) are used. All the necessary data (old variable values, old file timestamps) are stored in the persistent registry.

Each function, that implements one computation step, should follow few important rules. The function should be a “real” function, i.e., it should accept its input data only as its parameters and return the results only with the **`return`** statement. If the data are too complex, they should be passed to/returned from the function as file names, where the files contain the real data. The functions are required not to have any side effects. Usage of global variables is unacceptable.

The data passed to the function should be passed through the *“kwargs”* mechanism, i.e., the mechanism for passing variable number of arguments identified by keywords. This allows for convenient checking of the number and type of required arguments and makes the process of passing them to deps checking functions much easier.

The arguments to the function must be in one of the following special forms:

arguments passing file names must begin with the `FILE_` prefix; and

arguments passing ordinary values must begin with the `VAL_` prefix.

At least one source must be given. Arguments not starting either with `FILE_` or `VAL_`, are silently ignored. This way, the function can accept additional parameters, that are not tested by the `checkDeps()` function.

What follows is the typical way of defining a function that uses the buffering mechanism.

(The following function returns a single value.)

```
    def my_function(**kwargs):
        global depsChecker

        if not depsChecker.checkDeps('my_function', **kwargs):
            # Rebuild isn't necessary
            return depsChecker.getResult('my_function')

        # Else - compute the result
        <compute>
        <compute>
        <compute>
        depsChecker.storeResult('my_function', 'val', result)
        return result
```

(The following function returns a file.)

```
    def my_function2(**kwargs):
        global depsChecker

        if not depsChecker.checkDeps('my_function2', **kwargs):
            # Rebuild isn't necessary
            return depsChecker.getResult('my_function2')

        # Else - compute the result
        outFileName = depsChecker.makeTempFile()
        <compute>
        <compute>
        <compute>
        depsChecker.storeResult('my_function2', 'val', outFileName)
        return outFileName
```

The typical way of calling such a function is:

```
    my_val = my_function(FILE_infile1 = 'file1.txt',
        FILE_infile2 = 'file2.txt', FILE_infile3 = 'file3.txt',
        VAL_someoption = options.someoption,
        ign = 'this will be ingored by checkDeps()',
        forceRebuild = True)
```

Or, in case of a file-returning function:

```
    my_tmp_file = my_function2(FILE_infile1 = 'file1.txt',
        FILE_infile2 = 'file2.txt', FILE_infile3 = 'file3.txt',
        VAL_someoption = options.someoption,
        ign = 'this will be ingored by checkDeps()',
        forceRebuild = True)
```

7.3 OPTIONS & CONFIG FILE

The program in its current stage is complex and customizable. As such, it supports customizing the various parts of the execution and passing input arguments with the command-line options. Because there are so many command-line options, the options have reasonable default values wherever possible. To save the user from typing the same options over and over again, the program supports reading command-line options from the configuration file.

The command-line option parser is implemented with Python's standard module `optparse`. The `optparse` module is easy to use and offers a rich set of possibilities. The options are passed in the standard UNIX manner, e.g.:

```
    vrmlvis -d --lidar=myfile.asc --blurlidar
```

Frequently used options can be stored in the configuration file. The configuration file has the name `vrmlvis.cfg`. It must be located in the program's current directory. The configuration file is an ordinary Python file. It is run (parsed) by the Python interpreter in restricted environment, so it cannot interact with the main program. Although it can contain arbitrary Python expression, it is recommended to put only keyword/value pairs there. The string values have to be enlosed within single or double quotes, according to the Python parsing rules. The names of the keywords are identical with the options names. Wherever any single option has more forms, the longest form will be recognized by the config file parsing engine.

The options are looked for in the following places (in given order):

- command-line options
- configuration file
- default values

Example of a configuration file:

```
    lidar = 'mylidar.asc'
    blurlidar = True
    orthophoto = 'map.png'
    tiffworld = 'map.tfw'
    mesh2dm = 'mymesh.2dm'
    region = (500, 500, 2000, 1500)
    maxpoints = 20000
```

Following is the description of the most important options. (The list is not exhaustive; the program features many other options. The complete list is available with the `-h` or `--help` option.)

--quiet, -q	Suppresses all messages, except fatal errors.
--delinterm, -d	Deletes all intermediate files (forces their re-creation).
--lidar	Input LIDAR file (ASC format).
--lidarpgm	LIDAR data in the PGM format. Allows to fetch corrected (preprocessed) data.

--orthophoto	The orthophotomap (texture).
--tiffworld	The TFW file belonging to the given orthophotomap.
--mesh2dm	Water mesh in the 2dm format.
--scl	Water level heights (in different timesteps).
--region	Boundaries of the clip region.
--startframe	The first frame of the animation (default: from beginning).
--endframe	The last frame of the animation (default: to the end).

7.4 INPUTS & OUTPUTS

Inputs:

- Input LIDAR file (ASC or PGM format);
- Orthophotomap;
- TIFF World file;
- 2dm water mesh;
- Water level heights (SCL format).

Outputs:

- VRML terrain;
- Terrain texture;
- Water VRML file;
- Water texture (if required);
- Unifying VRML file.

7.5 PROGRAM FLOW OVERVIEW

The program can be given either a file in the raw LIDAR format, or a preprocessed file in the PGM format. In first case, the raw LIDAR data are transformed into the internal PGM representation. The second case allows to supply corrected or otherwise preprocessed terrain data. In both cases, the raw LIDAR data are used to extract their metadata (min. and max. geographical coordinates). That means, even in case of working with preprocessed LIDAR data, the original (raw) LIDAR file has to be present.

After the LIDAR data have been imported, they can be blurred (with the `--blurlidar` option). Blur flattens roughnesses in the terrain surface. If advanced blur is required, it is recommended to blur the image in an external program and supply the data with the `--lidarpgm` option.

In the next step, the *TIFF World* file is read and the coordinates of the orthophotomap are aligned with the LIDAR geographical data. If the *TIFF World* (TFW) file is missing, their lower left points are unified and only their sizes are aligned.

In the next step, the required region in taken into account. If the region is required, both the LIDAR data and the orthophotomap are cut into the required dimensions. If the region is not required, they are left unchanged.

The cut region is then triangulated with the external `terra` program (http://www-2.cs.cmu.edu/afs/cs/user/garland/www/scape/terra.html). After triangulation, the 3D mesh data are directly transformed into the VRML format.

When the terrain is transformed, the program begins to work with water. First, the water is imported (from the 2dm water mesh). Then, the actual water level heights (in different time steps) are extracted from the SCL file. When all the necessary data are loaded, the animation is created. The user can choose from two types of animation: the `switch`-based (jumpy), which utilizes the VRML's `switch` node; and the fluent animation (utilizing the `CoordinateInterpolator` node).

8 PRODUCT TESTING

9 CONTACT INFORMATION AND CREDITS

In case of any questions, please contact the developers:

- Branislav Simo, branislav.simo@savba.sk, (portal, MD plug-in, web services)
- Martin Maliska, martin.maliska@savba.sk (portal, MD plug-in, web services)
- Emil Gatial, emil.gatial@savba.sk (portal, MD plug-in, web services)
- Peter Slizik, peter.slizik@savba.sk (visualization of hydraulic simulation output)
- Jan Astalos, astalos.ui@savba.sk (testbed administration, ALADIN model)
- Miroslav Dobrucky, miroslav.dobrucky@savba.sk (HSPF, NLC models)
- Viet Tran, viet.ui@savba.sk (DaveF model)

Management:

- Dr. Ladislav Hluchy, hluchy.ui.@savba.sk (task leader, director of II SAS)

10 THE EDG LICENSE AGREEMENT

6. LIMITATION OF LIABILITY