Grid Workflow Execution Service - Developer Manual

WP2

Document Filename:	**KWF-WP2-D2-FIRST-GWESDeveloperManual.doc**
Work package:	**WP2**
Partner(s):	**Fraunhofer FIRST**
Lead Partner:	**Fraunhofer FIRST**
Document classification:	**PUBLIC**

Abstract: This document describes the Grid workflow execution environment of the K-Wf Grid project, which consists of several system components, such as the Grid Workflow Execution Service (GWES), the Grid Workflow User Interface (GWUI), and the Grid Workflow Description Language (GWorkflowDL) Java Library. This document targets at developers that want to analyse, modify, or extend the source code of the above mentioned components.

Delivery Slip

	Name	Partner	Date	Signature
From	Andreas Hoheisel	FIRST	07/09/2006	
Verified by	Piotr Nowakowski	CYFRONET	08/10/2006	
Approved by	Steffen Unger	FIRST	08/10/2006	

Document Log

Version	Date	Summary of changes	Author
0.1	2006-09-04	First version, based on the WP2 appendix to D2_5.2 + updates	Andreas Hoheisel, Tilman Linden, Hans-Werner Pohl
0.2	2006-09-07	Version for the internal review	Andreas Hoheisel, Tilman Linden, Hans-Werner Pohl
0.3	2006-09-26	Including remarks of the internal review	Andreas Hoheisel
1.0	2006-10-08	QA check	Piotr Nowakowski

CONTENTS

1. COPYRIGHT NOTICE ... **5**

2. INTRODUCTION ... **7**

2.1. ABBREVIATIONS AND ACRONYMS ... 7
2.2. REFERENCES AND SOURCE CODE ... 8

3. DESCRIPTION OF PRODUCT ... **9**

3.1. K-WF GRID SETUP ... 9
3.2. INSTANT-GRID SETUP ... 11
3.3. THE WORKFLOW CONCEPT ... 12
3.3.1. From Abstract to Concrete Workflows ... *12*
3.3.2. High-Level Petri Nets ... *13*
3.3.3. Example ... *14*
3.4. GRID WORKFLOW USER INTERFACE (GWUI) ... 17
3.5. GRID WORKFLOW DESCRIPTION LANGUAGE (GWORKFLOWDL) ... 18

4. IMPLEMENTATION STRUCTURE ... **19**

4.1. PRODUCT USE CASES ... 19
4.2. PRODUCT COMPONENT MODEL ... 21
4.3. DETAILED IMPLEMENTATION MODEL ... 22
4.3.1. Grid Workflow Execution Service (GWES) ... *22*
4.3.2. GWorkflowDL Java library ... *24*
4.3.3. Static Workflow Analysis ... *28*
4.3.3.1. Theoretical Background ... 28
4.3.3.2. Analyzer Usage ... 29
4.3.4. GWorkflowDL Protocol ... *31*
4.3.4.1. API Layer Details ... 31
4.3.4.2. Codec Layer Details ... 34
4.3.4.3. Version Checking Layer Details ... 35
4.3.4.4. Transport Layer Details ... 36
4.3.4.5. Protocol Configuration ... 36
4.3.5. GWUI Servlets for customized Data Input Dialogs ... *37*
4.4. PRODUCT INTERFACES ... 43

5. PRODUCT TESTING ... **45**

5.1. GWES TEST CASES ... 45
5.1.1. GWESTest ... *45*
5.1.2. GWESWSTest ... *46*
5.1.3. WSClientTest ... *46*
5.1.4. GWES_SuspendResumeTest ... *46*
5.1.5. GWES_CTMTest ... *46*
5.1.6. GWES_GWUITest ... *46*
5.1.7. GWES_HierarchicalTest ... *46*
5.1.8. GWES_AABTest ... *47*
5.1.9. GWES_DecisionTest ... *47*
5.1.10. GWES_SchedulerTest ... *47*
5.1.11. GWES_WCTTest ... *47*

6. KNOWN ISSUES ... **47**

6.1.1. GWES ... *48*
6.1.2. GWUI ... *48*
6.1.3. GWorkflowDL ... *48*

7. CONTACT INFORMATION AND CREDITS 49

8. THE FRAUNHOFER FIRST LICENSE AGREEMENT 50

1. COPYRIGHT NOTICE

Copyright (c) 2005-2006 by **K-Wf Grid, Fraunhofer FIRST**. All rights reserved.

Use of this product is subject to the terms and licenses stated in the Fraunhofer FIRST license agreement. Please refer to Section 8 for details.

This research is partly funded by the European Commission IST-2002-511385 Project "K-WfGrid".

The components described in this manual make use of external Java libraries, which have their own license agreements:

Library	Version	License	Reference
castor	0.9.7	Apache License; Intalio license	http://www.castor.org/
commons-collections	3.0	Apache License	http://ws.apache.org/axis/
commons-digester	1.5	Apache License	http://ws.apache.org/axis/
commons-discovery	0.2	Apache License	http://ws.apache.org/axis/
commons-logging	1.0.4	Apache License	http://ws.apache.org/axis/
dotparser	1.0.1	GNU Lesser General Public License	http://www.netart-datenbank.org/glassbox/
glassbox	0.3.4	GNU Lesser General Public License	http://www.netart-datenbank.org/glassbox/
j2sewsoap	1.06	Free for commercial use	http://www.wingfoot.com/products.html
jaxen	1.1-beta-6	Werken Company License	http://jaxen.org/
jdom	1.0	JDOM License	http://www.jdom.org/
jug	1.1.2	GNU Lesser General Public License	http://jug.safehaus.org/
junit	3.8.1	Common Public License - v 1.0	http://www.junit.org/
addressing	1.0	Apache License	http://ws.apache.org/addressing/
Axis-url_gt	4.0.1	Apache License	http://www.globus.org/
Axis_gt	4.0.1	Apache License	http://www.globus.org/
cog-axis_gt	4.0.1	Globus Toolkit Public License, National Research Council of Canada (Contributor)	http://www.globus.org/
cog-jglobus_gt	4.0.1	Globus Toolkit Public License, National Research Council of Canada (Contributor)	http://www.globus.org/
cog-tomcat_gt	4.0.1	Globus Toolkit Public License, National Research Council of Canada (Contributor)	http://www.globus.org/
cog-url_gt	4.0.1	Globus Toolkit Public License, National Research Council of Canada (Contributor)	http://www.globus.org/
commonj_gt	4.0.1	IBM BEA License	
commons-beanutils_gt	4.0.1	Apache License	http://ws.apache.org/axis/
commons-cli	2.0	Apache License	http://ws.apache.org/axis/
concurrent_gt	4.0.1	Concurrent License	
cryptix-asn1_gt	4.0.1	Cryptix General License	http://www.globus.org/
cryptix32_gt	4.0.1	Cryptix General License	http://www.globus.org/

cryptix_gt	4.0.1	Cryptix General License	http://www.globus.org/
gemini	20060511	GNU General Public License	http://www.gridworkflow.org/kwfgrid/jars/
globus_delegation_service_gt	4.0.1	Apache License	http://www.globus.org/
globus_wsrf_mds_aggregator_stubs_gt	4.0.1	Apache License	http://www.globus.org/
gomclient	20051028	K-Wf Grid License	http://www.gridworkflow.org/kwfgrid/jars/
gram-client_gt	4.0.1	Apache License	http://www.globus.org/
gram-utils_gt	4.0.1	Apache License	http://www.globus.org/
gworkflowdl	1.0.5d	Fraunhofer FIRST License	http://www.gridworkflow.org/kwfgrid/jars/
jaxrpc_gt	4.0.1	Apache License	http://www.globus.org/
jce-jdk13	125	Bouncy Castle License	http://www.bouncycastle.org/
Jgss_gt	4.0.1	Apache License	http://www.globus.org/
jxupdate	0.7.1	Fraunhofer FIRST License	http://www.gridworkflow.org/kwfgrid/jars/
kwfgrid-dr	20051021	GNU General Public License	http://www.gridworkflow.org/kwfgrid/jars/
naming-common_gt	4.0.1	Apache License	http://www.globus.org/
naming-factory_gt	4.0.1	Apache License	http://www.globus.org/
naming-java_gt	4.0.1	Apache License	http://www.globus.org/
naming-resources_gt	4.0.1	Apache License	http://www.globus.org/
opensaml_gt	4.0.1	OpenSAML License, Version 1.1	http://www.globus.org/
puretls_gt	4.0.1	Claymore Systems License	http://www.globus.org/
resolver_gt	4.0.1	Apache License	http://www.globus.org/
saaj_gt	4.0.1	Apache License	http://www.globus.org/
servlet_gt	4.0.1	Apache License	http://www.globus.org/
wsdl4j_gt	4.0.1	Common Public License - v 1.0	http://www.globus.org/
wsrf_core_gt	4.0.1	Apache License	http://www.globus.org/
wsrf_core_stubs_gt	4.0.1	Apache License	http://www.globus.org/
wsrf_provider_jce_gt	4.0.1	Apache License	http://www.globus.org/
wsrf_tools_gt	4.0.1	Apache License	http://www.globus.org/
wss4j_gt	4.0.1	Apache License	http://ws.apache.org/wss4j/
xmlsec_gt	4.0.1	Apache License	http://ws.apache.org/wss4j/
log4j	1.2.8	Apache License	http://logging.apache.org/log4j/docs/
xercesImpl	2.6.2	Apache License	http://xml.apache.org/xerces2-j/
xmlParserAPIs	2.6.2	Apache License	http://xml.apache.org/xerces2-j/
Xpp3	1.1.3.3	Extreme! Lab License	http://www.extreme.indiana.edu/xgws/xsoap/xpp/

This product includes software developed by the

- K-Wf Grid Project (http://www.kwfgrid.eu/)
- Apache Software Foundation (http://www.apache.org/)
- Globus Alliance (http://www.globus.org/)
- ExoLab Project (http://www.exolab.org/) (castor).
- BEA IBM (http://dev2dev.bea.com/technologies/commonj/index.jsp) (commonj)
- JDOM Project (http://www.jdom.org/) (jdom)
- University Corporation for Advanced Internet Development http://www.ucaid.edu Internet2 Project (opensaml)
- Indiana University Extreme! Lab. For further information please visit http://www.extreme.indiana.edu/. (xpp3)
- And others…

2. INTRODUCTION

The Grid workflow orchestration and execution environment consists of several system components which work together in order to assist the user in building and controlling Grid applications in a distributed environment as shown in Figure 1. This developer manual describes the architecture of the components *Grid Workflow Execution Service (GWES 1.0.5)*, *Grid Workflow User Interface (GWUI 0.5.2)*, and *Grid Workflow Description Language (GWorkflowDL) Java Library (1.0.6)*. The Target audience of this manual are software developers who want to understand, modify, reuse, or change the code of the components. It is recommended to read the "GWES User Manual" first, which targets on system administrators and end users.

2.1. ABBREVIATIONS AND ACRONYMS

AAB	Automatic Application Builder (system component)
CTM	Coordinated Traffic Management (pilot application)
CVS	Concurrent Versions System (software development tool)
D-GRDL	D-Grid Resource Description Language (description language)
ERP	Enterprise Resource Planning (pilot application)
FFSC	Flood Forecasting Simulation Cascade (pilot application)
GEMINI	Generic Monitoring Infrastructure (system component)
GOM	Grid Organizational Memory (system component)
GS	Grid Service (remote procedure technology)
GWES	Grid Workflow Execution Service (system component) – *pronounced "ge-wes"*
GWorkflowDL	Grid Workflow Description Language (description language)
GWUI	Grid Workflow User Interface (system component) – *pronounced "gwui"*
RFT	Reliable File Transfer (file transfer technology)
SOAP	Protocol for exchanging XML-based messages (formally known as "Simple Object Access Protocol") (protocol standard)
WCT	Workflow Composition Tool (system component)
WS	Web Service (remote procedure technology)
WSDL	Web Services Description Language (description language) – *pronounced "wiz-dull"*
WS-GRAM	Web Service Grid Resource Allocation and Management (remote procedure technology)
WSRF	Web Service Resource Framework (standard)
XML	Extensible Markup Language (standard)

2.2. REFERENCES AND SOURCE CODE

Further online information about the Grid Workflow Execution Service (GWES) and the Grid Workflow User Interface (GWUI) is available at the following links:

- Source and binary software distributions: http://www.gridworkflow.org/kwfgrid/distributions/
- GWES software development site: http://www.gridworkflow.org/gwes/
- GWES source Xref: http://www.gridworkflow.org/kwfgrid/gwes/docs/xref/
- GWES Java Docs: http://www.gridworkflow.org/kwfgrid/gwes/docs/apidocs/
- GWES CVS (password protected): http://cvs.ui.sav.sk/cgi-bin/cvsweb.cgi/kwfgrid/gwes/
- GWUI software development site: http://www.gridworkflow.org/kwfgrid/gwui/docs/
- GWUI source Xref: http://www.gridworkflow.org/kwfgrid/gwui/docs/xref/
- GWUI Java Docs: http://www.gridworkflow.org/kwfgrid/gwui/docs/apidocs/
- GWUI CVS (password protected): http://cvs.ui.sav.sk/cgi-bin/cvsweb.cgi/kwfgrid/gwui/

3. DESCRIPTION OF PRODUCT

The components GWES, GWUI, and the GWorkflowDL Java library can either be used in combination with further K-Wf Grid components – such as the Automatic Application Builder (AAB), the Workflow Composition Tool (WCT), the Scheduler and the Web Portal (refer to the "*K-Wf Grid Setup*" – Section 3.1) – or as an easy-to-install "stand-alone" deployment with reduced functionality (refer to the "*Instant-Grid Setup*" – Section 3.2).

3.1. K-WF GRID SETUP

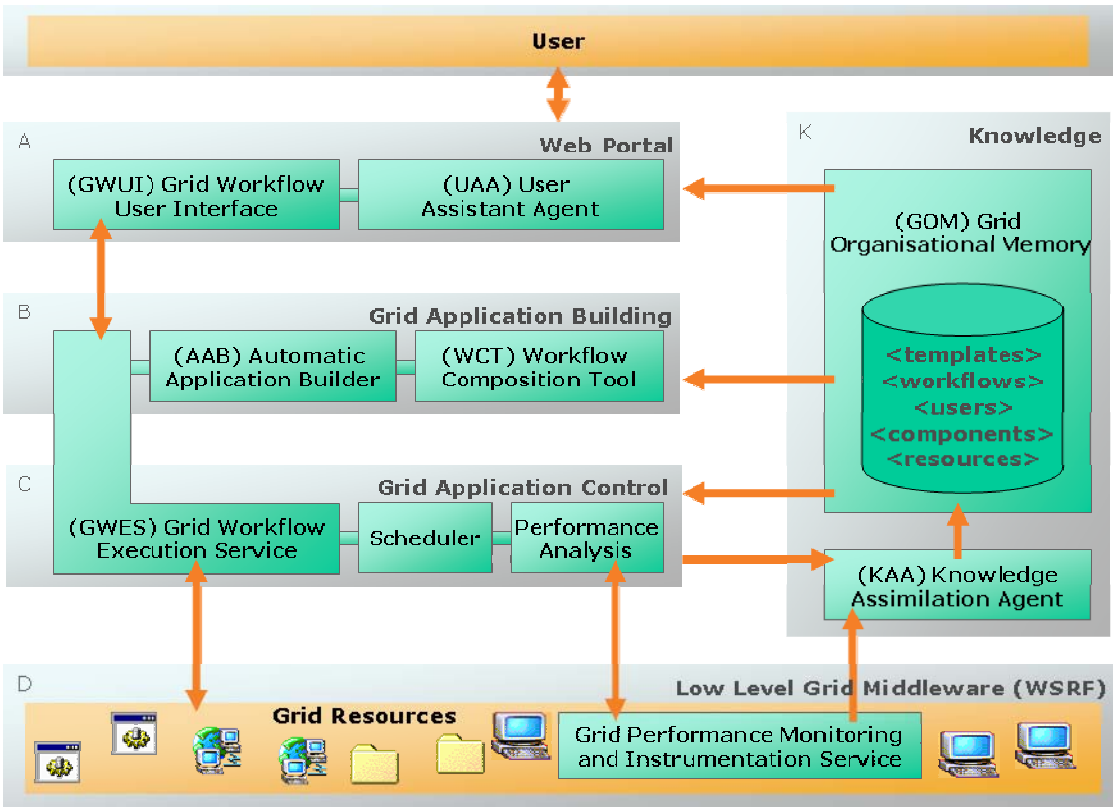


Figure 1. System architecture of the K-Wf Grid system.

The *Grid Workflow Execution Service (GWES)* is the Grid workflow enactment engine of the K-Wf Grid system, which coordinates the composition and execution process of Grid workflows. It implements a highly dynamic workflow concept based on the *Grid Workflow Description Language (GWorkflowDL)*. It provides interfaces to the Web Portal for user interaction and to the Low-Level Grid Middleware for the invocation of application operations. The mapping of abstract to concrete workflows is mainly delegated to further system components, such as WCT, AAB, and Scheduler (refer to the corresponding user manuals).

The main purpose of the GWES and its user interface is to:

1. define user requests,
2. initiate Grid workflows based on these user requests,
3. monitor and inspect running and finished workflows,
4. interact with the workflow building and execution process
5. download and upload files from/to the Grid, and
6. provide assistance during the orchestration of workflows.

Therefore the GWES possesses the following features:

- Analysis and verification of the workflow descriptions (delegated to the GWorkflowDL Java library).
- Can act as a workflow engine, cycling through the workflow graph, searching for activated transitions, evaluating arbitrary conditions, and triggering related activities.
- Detection of conflicts within the workflow and delegation of workflow building decisions to the user via the Grid Workflow User Interface (GWUI) in case of conflicts or annotations that request user decisions.
- Invocation of the Workflow Composition Tool (WCT) if abstract workflow elements need to be mapped onto operations of Web Service classes (refer to separate user manual).
- Invocation of the Automatic Application Builder (AAB) if operations of Web Service classes need to be mapped onto lists of concrete Web Service operations (refer to separate user manual).
- Invocation of the Scheduler if a list of concrete Web Service operations needs to be mapped onto a single instance of Web Service operation (refer to separate user manual).
- Reliable invocation of target Web Service operations.
- Transfer of data from one Web Service to another as specified in the Grid workflow description.
- Control of the execution of the Web Service operations and throwing workflow-related events to the Event System.
- Execution of remote command line programs using WS-GRAM.
- File transfer by means of RFT

The GWUI (Grid Workflow User Interface) is a client interface for the GWES. The GWUI mainly implements the client part of interfaces 1 and 2 of the Workflow Reference Model of the WfMC (http://www.wfmc.org/standards/model.htm):

- User interface for process definition
- Workflow client application

Additionally it has capabilities for:

- Monitoring of workflows

- User interaction with workflows (start, pause, stop, cancel, modify, ...)

For details about how to install and use the Grid Workflow Execution Service please refer to the separate GWES User Manual.

3.2. INSTANT-GRID SETUP

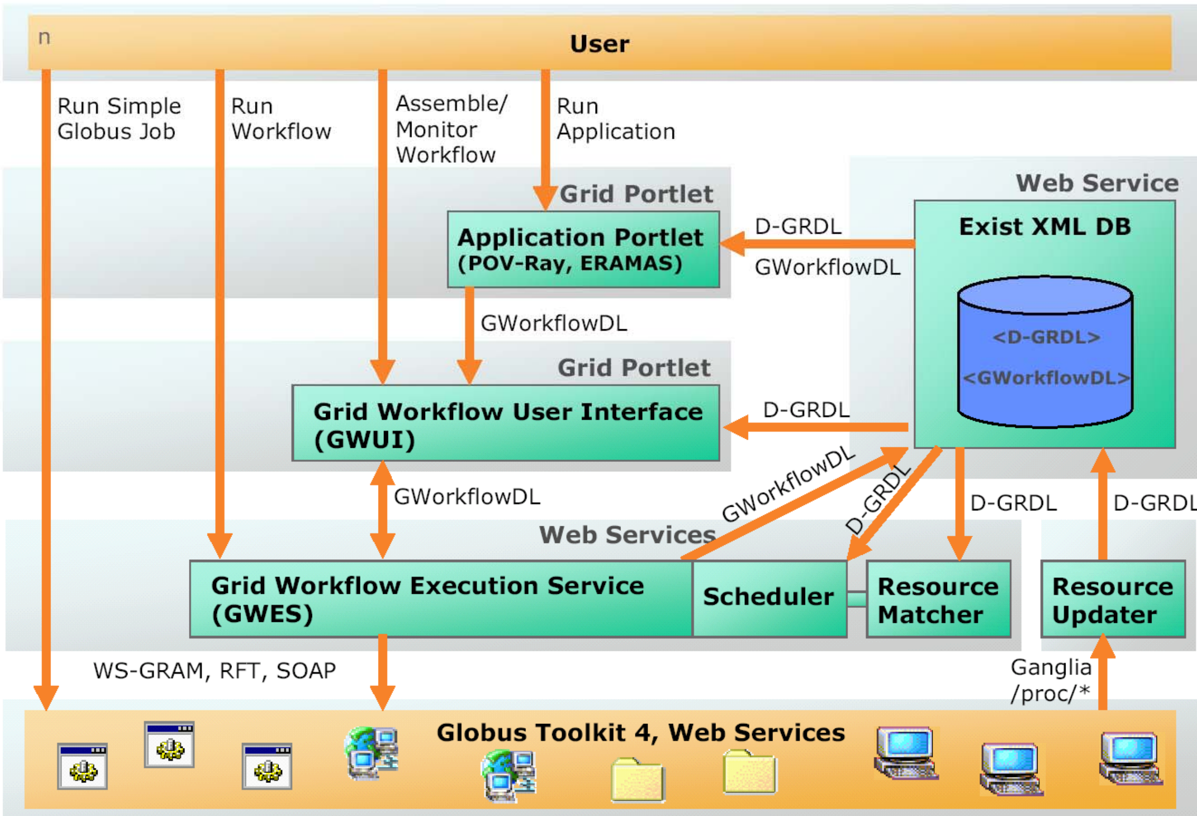


Figure 2. System architecture of the Instant-Grid workflow management system.

Within the K-Wf Grid setup, the GWES depends on further services, in order to provide full automation of the workflow orchestration process. An alternative GWES setup with reduced functionality, using only few components of K-Wf Grid, is realized within the Instant-Grid project (http://instant-grid.de/). The Instant-Grid project aims at developing a Knoppix-based live-CD bundled with a pre-configured Grid environment based on the Globus Toolkit. In order to provide Grid workflow management features, it reuses some components developed by Fraunhofer FIRST within the K-Wf Grid project.

Figure 2 shows the architecture of the Instant-Grid setup, where an XML database replaces the K-Wf Grid Organizational Memory. A resource matcher service maps abstract workflows onto service candidates as replacement for the AAB component. While the K-Wf Grid setup takes into account Web and Grid Service invocation, the Instant-Grid setup focuses on the remote execution of programs using WS-GRAM.

The user may use the Instant-Grid setup as a starting point for evaluating the workflow management environment without having to provide the ontologies for the services and applications. This is the

reason why we include some documentation related to the Instant-Grid setup in this manual, although parts of the work has not been performed in the scope of the K-Wf Grid project.

3.3. THE WORKFLOW CONCEPT

The workflow concept realized within the Grid Workflow Execution Service has some innovative features: It supports several levels of abstraction within one workflow description language, it is simple but universal (in the sense of being Turing complete), it supports data as well as control flow, and it enables dynamic workflows where the structure of the workflow may change during runtime. This section explains the underlying workflow concept.

The basis for the workflow management is the Grid Workflow Description Language (GWorkflowDL), which is a Petri Net-based standard for describing workflows using XML. For the features and the specification of the Grid Workflow Description Language please refer to the GWorkflowDL software development site at http://www.gridworkflow.org/gworkflowdl/.

3.3.1. From Abstract to Concrete Workflows

The GWorkflowDL and the GWES support several levels of workflow abstraction levels that are displayed in Figure 3. An initial input workflow provided by the user or by the user assistant may possess different abstraction levels, ranging from an abstract user requests to the concrete (executable) workflow which can be invoked directly on the available Grid resources. The supported abstraction levels are:

- **User Request (Red):** The user request represents an abstract operation which has still not been mapped onto potential workflows.
- **Abstract Workflow (Yellow):** An abstract (non-executable) workflow consists of operations of Web Service or program execution classes. The WCT automatically composes abstract workflows from the user requests, if the corresponding ontologies are represented within the Grid Organizational Memory (GOM). The user can also directly provide abstract workflows without using the WCT (refer to *Instant-Grid setup*).
- **Workflow of Service Candidates (Blue):** Consists of lists of Web Service or program execution candidates, which match the Web Service (or program execution) classes. The mapping is done by the AAB (or the resource matcher).
- **Workflow of Service Instances (Green):** Consists of concrete (executable) instances of Web Service operations or program executions. The selection of the optimal instance out of the list of candidates is delegated to the Scheduler.
- **Control Flow (Black):** Black transitions within the workflow denote a control flow (separate from the data flow) which is not connected to any real operation.

Only green or black workflow nodes can directly be executed by the GWES. If the workflow contains red, yellow or blue nodes, the WCT, AAB, Resource Matcher, or Scheduler is invoked, in order to refine the workflow. If these components are not able to provide a solution, then the GWES will suspend the workflow and the user has to refine the workflow.

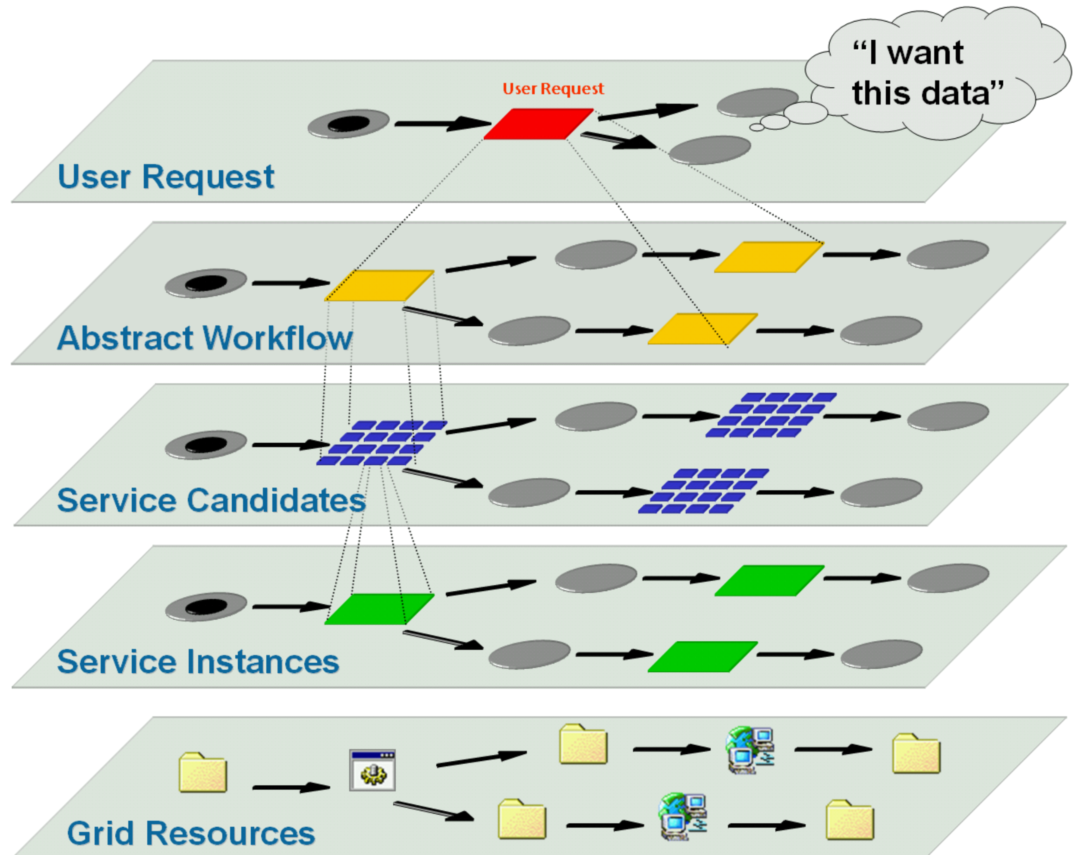


Figure 3. The abstraction levels that are supported by the GWorkflowDL and the GWES.

3.3.2. High-Level Petri Nets

The workflow description model is based on the expressive formalism of High-Level Petri nets. Petri nets are directed graphs, containing two distinct sets of nodes: *transitions* - represented by rectangles - and *places* - denoted by circles. Places and transitions are connected by directed edges. An edge from a place *p* to a transition *t* is called an *incoming edge* of *t*, and *p* is called *input place* of *t*. *Outgoing edges* and *output places* are defined accordingly. Each place can hold a certain number of individual *tokens* that represent data items flowing through the net. The maximum number of tokens on a place is denoted by its *capacity*. A transition is called *enabled* if there is a token present at each of its input places, and no output place has reached its capacity. Enabled transitions can *fire* by consuming one token from each of the input places and putting a new token on each of the output places. The number and values of tokens each place holds during the workflow execution is called *marking*. The *initial marking* specifies the marking at the beginning and consecutive markings are obtained by firing transitions. Each edge in the Petri net can be assigned an *edge expression*. For incoming edges variable names are used as edge expressions, which assign the token value obtained through this edge to a variable of that name. Additionally, each transition can have a set of *conditions*. A transition can only fire if all of its conditions evaluate to true for the input tokens.

Petri nets are suitable to describe the sequential and parallel execution of tasks with or without synchronization; it is possible to define loops and the conditional execution of tasks. As mentioned above, we use Petri nets not only to model, but furthermore to control the execution of the workflow.

In most cases, the workflow within Grid jobs is equivalent to the dataflow, i.e., the decision when to execute a software component is taken by means of availability of the input data. Therefore, the tokens of the Petri net represent real data that is exchanged between the software components. In this case, we use Petri nets to model the interaction between software resources represented by transitions that are linked to Web Service operations, and data resources represented by places linked to SOAP parameters. In other cases, however, the workflow should be independent from the dataflow, and in addition we have to introduce control places and control transitions. Control transitions only evaluate logical conditions. In the case that a transition is linked to a Web service operation, the tokens store the output parameter returned by the method call. The state of the workflow is represented by the marking of the Petri net, which makes it easy to implement tools for workflow check pointing and recovery.

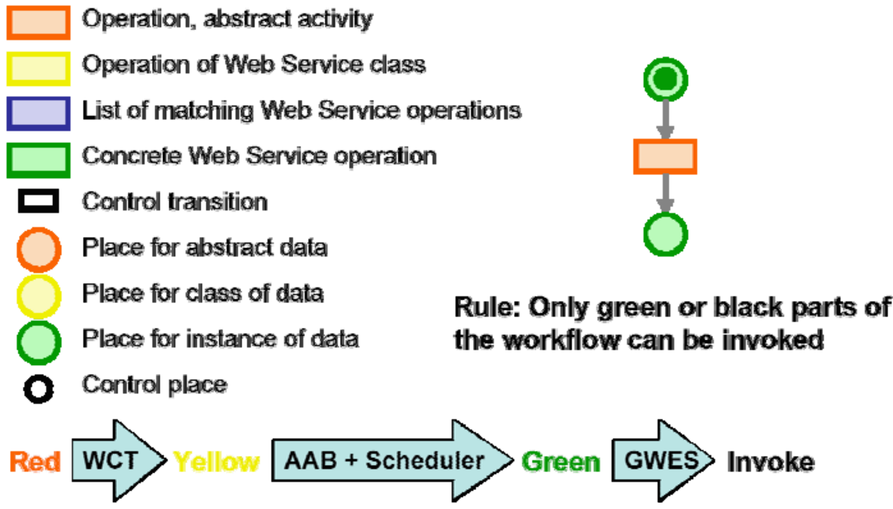


Figure 4: The workflow description language specifies transitions (rectangles representing activities), and places (circles representing data and the state), in different levels of abstraction. The transitions and places are connected by directed edges that are annotated by edge expressions

3.3.3. Example

Figure 5 to Figure 9 show an exemplary life cycle of a Grid workflow in our concept – beginning with the creation of an abstract workflow in Figure 5 and ending with a complete workflow that has been executed on real Grid resources in Figure 9. In the first step the user creates a very abstract workflow just consisting of one transition (red) related to an (unknown) operation without specifying further details. In this example the transition is linked to an input place and an output place, both referring to well known data (green) (see Figure 5). While the input data is available (place with token), the output data has to be produced by the workflow (place without token). In a next step the Workflow is then sent to the Workflow Composition Tool (WCT), which proposes two different workflows (yellow), which produce the same type of required output data. These two workflows are merged into one single workflow by surrounding it with an "XOR split" and an "XOR join" control construct (black).

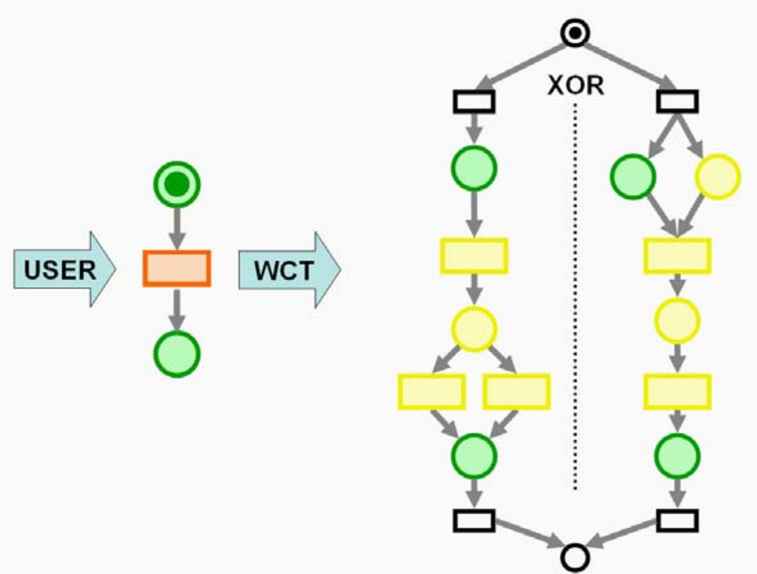


Figure 5: The user defines a very abstract workflow, just defining the available input and the required output data. The WCT proposes two workflows - surrounded by an XOR construct - that produce the required data

In the Petri net theory, each region where two or more transitions are enabled and compete for the same token is called *conflict*. In our case we have such a conflict at the XOR split construct at the top of the workflow. If this conflict cannot be resolved automatically (e.g., by additional conditions), the workflow is passed back to the user in order to take the decision. In our example the user decides to invoke the left branch of the workflow (see Figure 6).

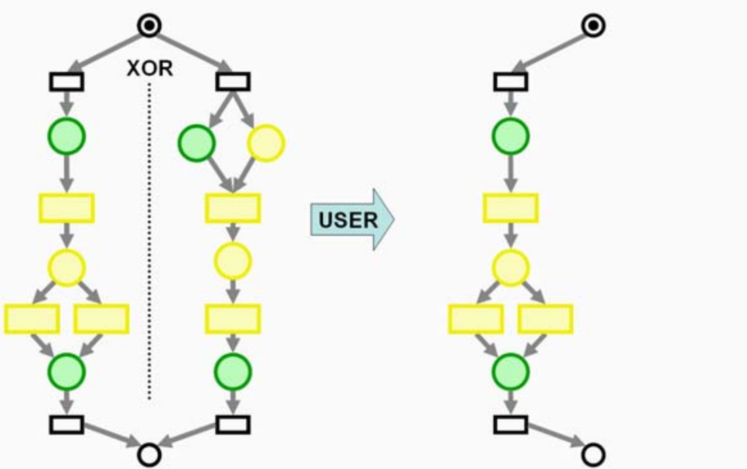


Figure 6: The workflow contains a conflict where two transitions compete for the same token and the user decides to follow the left branch of the workflow

After the user resolved the conflict by deciding to invoke the left branch of the workflow, the Grid Workflow Execution Service (GWES) can now fire the first control transition (black) at the top of the workflow (see Figure 7). A control transition is not related to any kind of Web Service activity; it only

removes one token from each input place and adds one token to each output place. Together with the control places this kind of transition just models the control flow that is required to synchronize activities. The next transition that is enabled now is the yellow transition in the middle of the workflow. This transition is linked to specific operation of a Web Service class, which has first to be mapped onto real, existing Web Service operations in order to be invoked. In the next step the Automatic Application Builder (AAB) maps this operation of a Web Service class (yellow) onto a list of matching Web Service operations (blue).

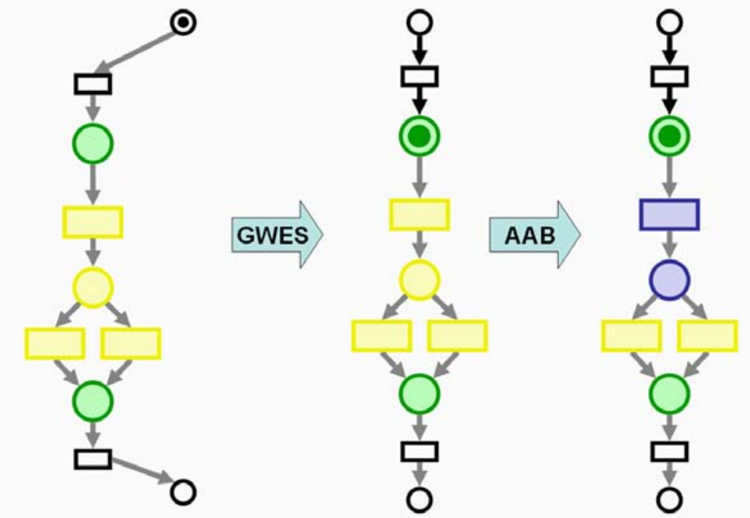


Figure 7: The GWES fires the first transition of this workflow. In the next step, the AAB maps the following yellow part of the workflow onto a list of real existing Web Service operations

Figure 8 shows the subsequent concretion and execution of the workflow. The Scheduler selects one Web Service operation (green) out of the list of matching operations (blue). This selection is done concerning some metric that represents the user requirements, such as fastest, cheapest, or the most secure. Then the GWES enacts the following green part of the workflow, calling the Web Service operation that is linked with the corresponding transition. Now the workflow enactment again reaches a conflict region where two transitions (yellow) compete for the same token (green). In this example the decision could not be done in advance because it depends on the recently produced input data that is represented by the green token. Here the user decides to follow the left branch of the sub-workflow.

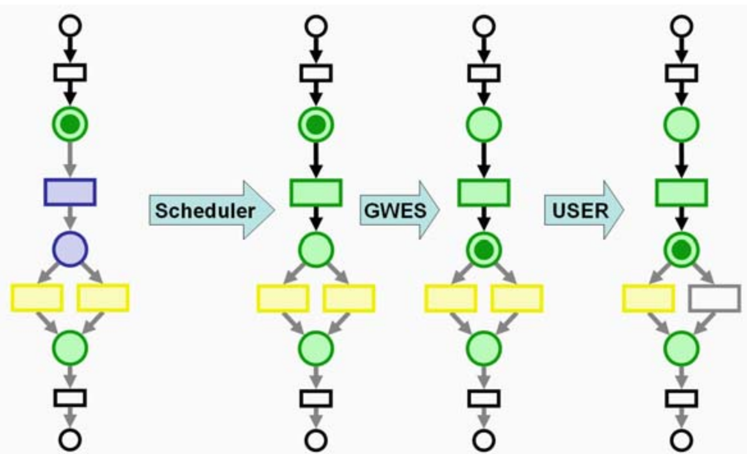


Figure 8: The Scheduler selects one Web Service operation, which is then invoked by the GWES. The following conflict is resolved by user interaction

The AAB and the Scheduler now concretise the yellow transition that has been selected by the user in the previous step (refer to Figure 9). The GWES executes the remainder of the workflow that now completely consists of transitions that are linked to real Web Service operations and that can be invoked by means of the low-level Grid middleware.

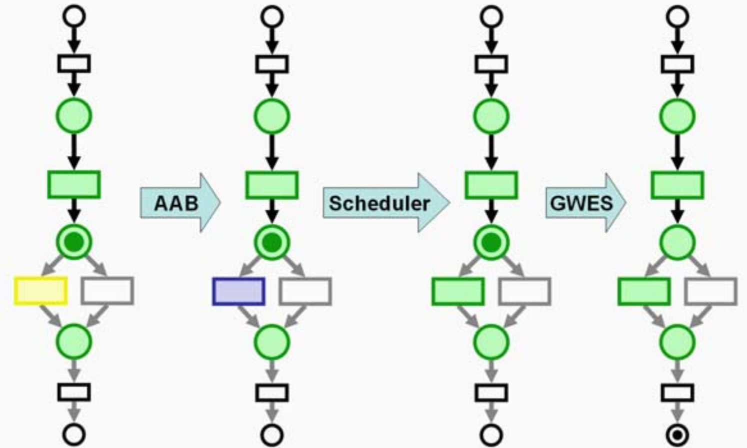


Figure 9: The AAB and the Scheduler concretise the yellow transition. The GWES invokes the Web Service operations that are linked to the green transitions

3.4. GRID WORKFLOW USER INTERFACE (GWUI)

The GWUI is an interactive graphical interface for the Grid Workflow Execution Service. It offers a set of features to initiate, monitor and manipulate workflows. It also includes mechanisms to support user interactions during workflow execution like input data specification and decision making.

The intention of GWUI is to provide a library of GUI components suitable for workflow monitoring, manipulation and execution which can be differently combined so that different GUI setups are possible. Thus graphical interfaces for different user requirements can be constructed on the basis of the GWUI library.

The central GUI component is a dynamic visualization of the Petri net of a running workflow. Further components include inspectors for the details of single elements inside the workflow. The inspectors can be used for monitoring and manipulation of diverse workflow properties. Furthermore dialogs for workflow status control are offered as well as the possibility to upload and run workflows on the Grid. GWUI also offers a dynamic task list showing all tasks the user has to accomplish in order to execute a certain workflow. This task list interfaces with a generic data input dialog framework so that application and data specific input dialogs can be triggered from within GWUI.

GWUI has been designed regarding usability guidelines like flexibility and consistency as well as guidelines for information visualization. The diverse needs of different user groups have been addressed in the design. Furthermore GWUI is suitable for collaborative workflow execution and manipulation. Several instances of GWUI can be run on different computers and share the same workflow instance. By means of a distributed object model which is part of the GWorkflowDL library (see 3.5) all changes made to the workflow by one party will be instantly distributed to all other's working on the same workflow instance.

The GWUI is implemented in Java 1.4.2 based on the Java Swing GUI framework. The single GUI components are included in several Java applets which run inside the web browser of a client computer. In the K-Wf Grid portal these applets are located inside single portlets offering the user the flexibility to construct a custom arrangement of the GUI components he/she requires.

3.5. GRID WORKFLOW DESCRIPTION LANGUAGE (GWORKFLOWDL)

The module GWorkflowDL comprises a Java library for dealing with workflow representations such as construction, refinement and reading from and writing to XML workflow representations. Especially the operations of the enactment engine (GWES) are based on the GWorkflowDL library. Interfaces and implementation classes are designed to ease implementation exchange to support future optimizations.

A special feature of the GWorkflowDL is the support of different abstraction layers of transitions or operations respectively. The most abstract ones represent pure control functionality and the least abstract ones describe executable grid operations that are constructed by higher-level tools as WCT, AAB, and Scheduler.

Moreover, a tool is included to perform a static analysis of workflows at design-time and at run-time as well. The tool is based on the Karp-Miller-Tree construction and answers questions like “Has the workflow an infinite run?”, “Is a certain place unbounded?”, or “Can a certain transition fire at all?”.

On the one hand K-Wf Grid workflows are described by XML documents and on the other hand by internal Java representations. The XML representation supports the (e.g. remote) exchange of workflow information and persistency issues, the internal Java representation eases construction, analysis, refinement and execution of workflows. Additional, there is a third representation (a second internal one) due to the utilized Java-XML-library, in our case JDOM.

In the actual version the XML-schema for workflow descriptions is to find under:

http://www.gridworkflow.org/kwfgrid/src/xsd/gworkflowdl_1_0.xsd

The XML-description of workflows defined by an XML schema or the Java structure library can be used for workflow construction and refinement; there is nearly a one-to-one correspondence in between. (Violations are of no practical meaning, e.g. the shuffling if input and output places in XML descriptions that has no Java counterpart).

The GWorkflowDL library furthermore contains the implementation of a protocol which enables the sharing of workflow instances across multiple computers. The protocol follows a client-server

approach and is based purely on polling for changes by the clients. The protocol's design is similar to the well known CVS (Concurrent Versioning System) approach. This rather conservative approach makes the protocol easy to adapt to different transport standards like SOAP, Java RMI, XML-RPC and the likes. The protocol is hidden behind the common GWorkflowDL API and can be easily enabled and disabled. It is therefore nearly transparent for clients of the GWorkflowDL library. A workflow in GWorkflowDL can be regarded as a distributed object tree. The protocol makes use of XUpdate and XPath standards for the serialization of changes of the workflow structure.

4. IMPLEMENTATION STRUCTURE

4.1. PRODUCT USE CASES

The main functional requirements related to the overall workflow orchestration and execution process are to:

1. enable users and application developers to customize the K-WfGrid system,
2. create workflows and results descriptions,
3. monitor and inspect running and finished workflows,
4. interact with the workflow building and execution process
5. download and upload files from/to the Grid, and
6. get assistance during the orchestration of workflows.

The most relevant non-functional requirements are interoperability, flexibility, usability, reusability, extensibility, and reliability. Other non-functional requirements, such as availability, scalability, security, performance, maintainability, and portability, are regarded less relevant within the scope of the K-WfGrid project. While the performance of the workflow execution may certainly be important, the performance and scalability of the system components themselves is – apart from few exceptions – less relevant for the pilot operations of K-WfGrid. Security related issues are currently not addressed by the experimental prototype implementation but will be included in the stable version.

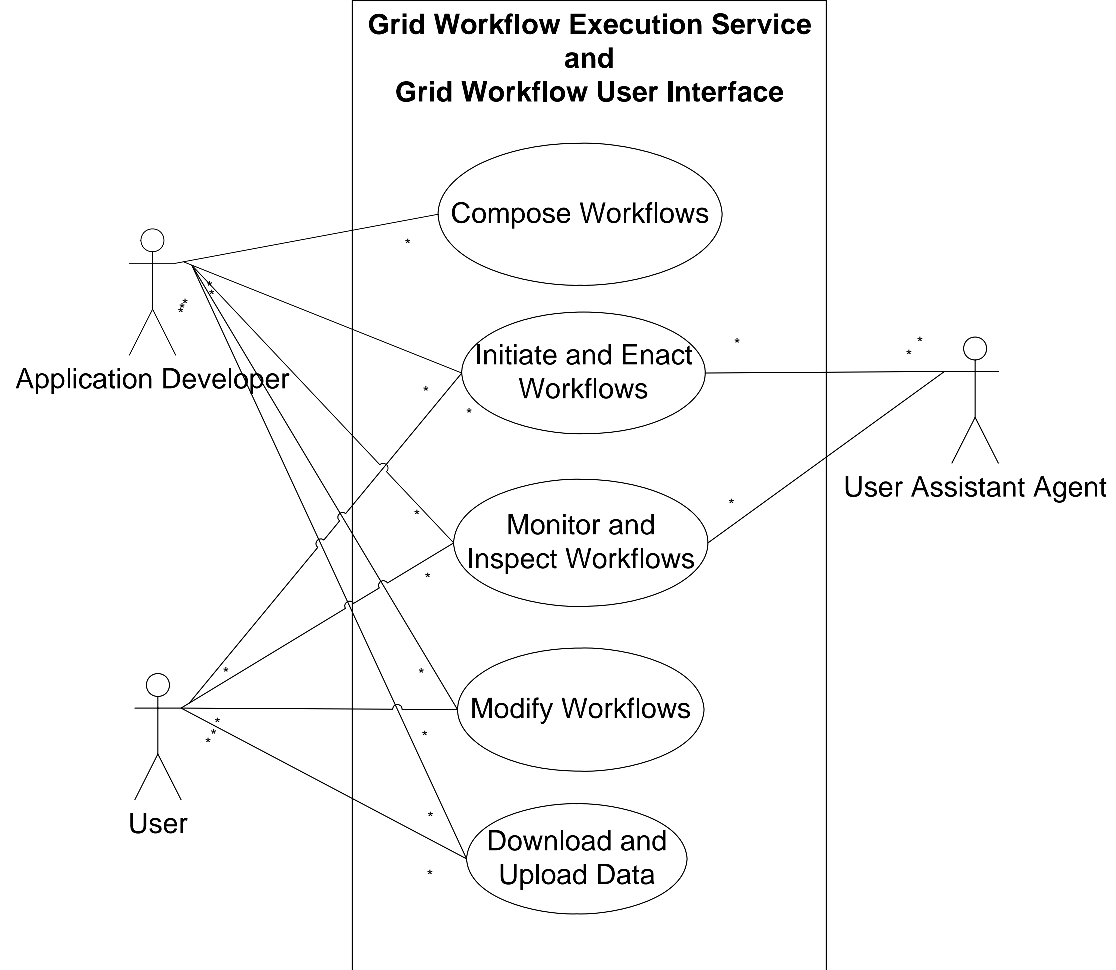


Figure 10. Use case diagram of the Grid Workflow Execution Service and the corresponding user interface.

There are three main actors supposed of using the GWES: The application developer composes, modifies and tests new workflows, and the user invokes all the tasks related to the workflow enactment, including modifying, monitoring, and inspecting workflows as well as uploading initial data and downloading result data. The user assistant agent is a portal component, which assists the user in initiating and enacting workflows that are created from predefined workflow templates.

4.2. PRODUCT COMPONENT MODEL

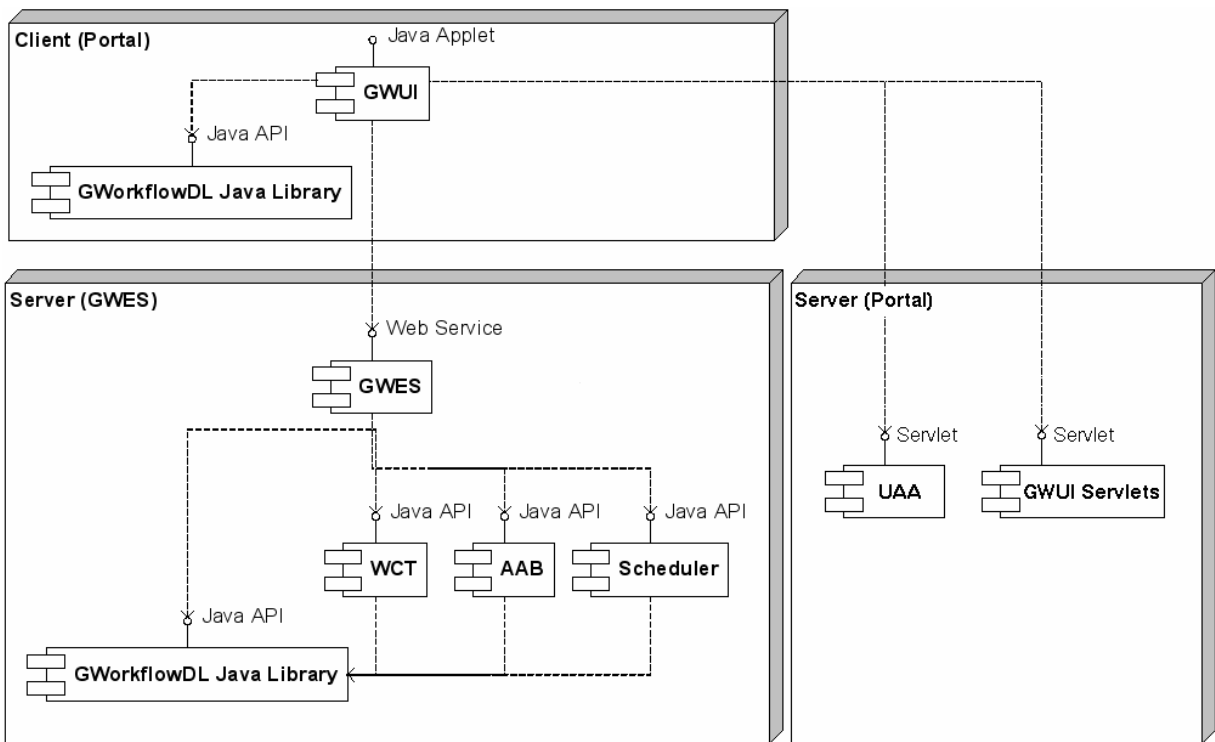


Figure 11. Component model of the Grid workflow orchestration and execution environment. The Java libraries WCT, AAB, and Scheduler can also be deployed as separate, stand-alone Web Services (not shown here).

The **GWES** and the **GWUI** are two separate system components that share workflow instances by means of the GWorkflowDL protocol. The communication is based on SOAP and standard web service calls. The **GWES** uses some external components, such as the **GWorkflowDL** Java library for creating, modifying, parsing and analysing Workflow Descriptions, the WCT for the automatic composition of abstract workflows based on user requests, the AAB for mapping abstract workflows onto concrete Web Service candidates, and the Scheduler for optimizing the selection of the concrete Web Service instance out of the list of candidates (refer to the corresponding manuals for details).

Whereas for the envisioned pilot operations it will be enough to have one centralized instance of the **GWES**, in principle it is possible to have multiple instances deployed on different sites realizing a distributed workflow enactment system, where each instance controls a separate sub-workflow of a Grid workflow. This is possible because the **GWES** itself can be regarded as an arbitrary Web Service application that may be invoked within a Grid workflow.

The **GWUI** is the main user interface for the application developer as well as the user and runs as a set of Java applets inside the client browser. The GWUI triggers the User Assistant Agent (UAA) and a set of servlets for custom data input dialogs which are located on the K-Wf Grid portal server.

4.3. DETAILED IMPLEMENTATION MODEL

4.3.1. Grid Workflow Execution Service (GWES)

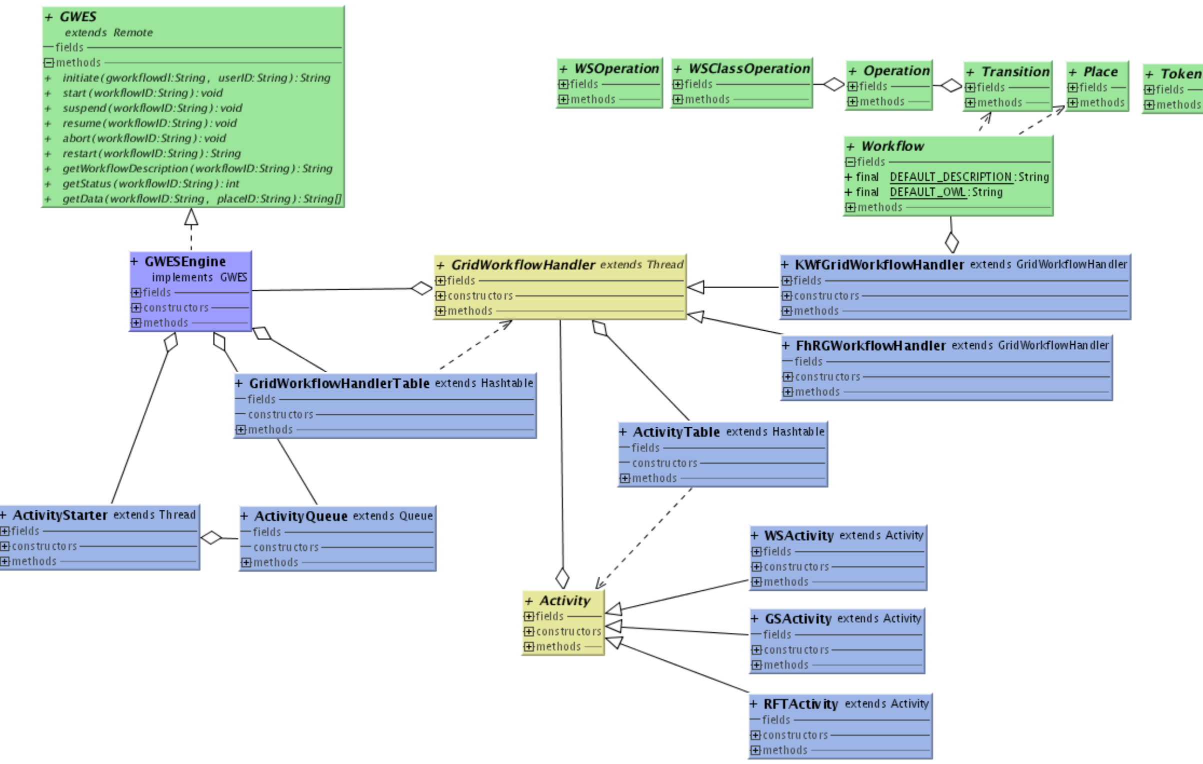


Figure 12. UML class diagram of the basic classes of the Grid Workflow Execution Service.

The figure above shows the UML class diagram of the basic GWES classes. The main external interface is the `GWES` which specifies the main methods for managing workflows. The `GWESEngine` implements `GWES` and contains a table of Grid Workflow Handlers. Each `GridWorkflowHandler` handles one workflow. The `KWfGridWorkflowHandler` is one implementation of a `GridWorkflowHandler` which supports the Petri Net-based Grid Workflow Description Language (GWorkflowDL). During the enactment, a workflow triggers a set of Activities, such as the invocation of Web Service (`WSActivity`) or Grid Service (`GSActivity`) operations. Activities are enqueued to an `ActivityQueue` and processed in parallel by several `ActivityStarters`.

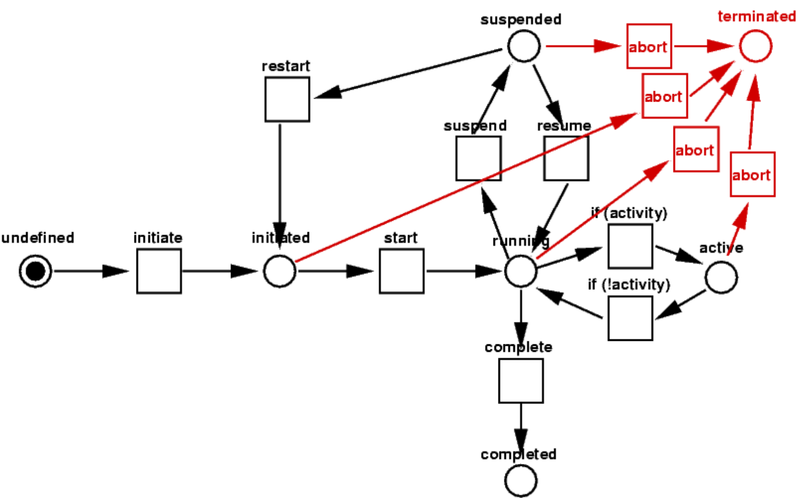


Figure 13. State transition diagram for workflows and their activities.

Workflows as well as activities possess the states UNDEFINED, INITIATED, RUNNING, ACTIVE, SUSPENDED, COMPLETED and TERMINATED. The figure above shows all possible state transitions. The initiate method initiates a new workflow (or activity) by means of its workflow (or activity) description. When invoking the start method, the state switches to RUNNING. If there exist one or more instances of activities related to the workflow (or sub activities related to the activity) the corresponding state switches to ACTIVE. In the case of an activity, this means that the pre and post processing (e.g. building the input parameters and processing the return values) as well as the waiting for required resources is done during the RUNNING state. The state switches to ACTIVE right before accessing external resources (e.g. a Web Service operation). After releasing the external resources, the state switches back to RUNNING. The suspend method suspends a running workflow (or activity). If the current state is ACTIVE, then the GWES first waits until the state RUNNING is reached, before suspending the workflow. If there does not exist any further activity to invoke and the workflow (or activity) has not been aborted yet, then the state switches to COMPLETED. If the workflow (or activity) fails or is aborted by the user, then the state switches to TERMINATED.

In general, workflows and activities possess the same possible states, because it is planned to support hierarchical workflows in the future, i.e. activities could then represent (sub) workflows.

For the communication between the GWES and the GWUI a specific protocol has been implemented, in order to exchange workflow modification events. These events denote changes of the workflow model and are currently generated by the GWES. Modifications are initiated by calling certain methods on the GWorkflowDL Java API. These method calls are be encoded into XUpdate statements denoting the modification in terms of the XML model. The XML representation of these XUpdate statements – enriched by certain header information – is used to notify the modifications to other services. A service consuming these events has to interpret the XUpdate statement and update its workflow model accordingly. An API for these interpretation steps is provided. While this protocol is included in the default implementation, other protocols can be added by extending suitable APIs (see below).

4.3.2. GWorkflowDL Java library

The GWorkflowDL Java library is composed of:

- A set of Java interfaces
- A corresponding set of implementation classes
- A factory class to enable application programming to be independent of special implementation variants.
- Utility classes that transform internal Java objects to the corresponding internal Jdom elements and vice versa.
- Reading and writing utilities of XML document-files.
- A set of implementations of the Java interfaces which trigger the GWorkflowDL protocol to share workflow instances across multiple computers.
- A protocol kernel which implements a client-server approach based on a polling-for-changes strategy.
- A set of interfaces used inside the protocol kernel which makes certain parts of the protocol implementation easily exchangeable.
- An encoder for all method calls of the GWorkflowDL object model. The provided implementation encodes method calls into XUpdate modifications of the underlying XML document.
- A framework to apply XUpdate modifications to an XML grounded Java object model.

Figure 14 shows a simplified UML class diagram of the internal Java representation of workflows. For simplicity reason the splitting of entities in *interface* and *implementation class* is ignored.

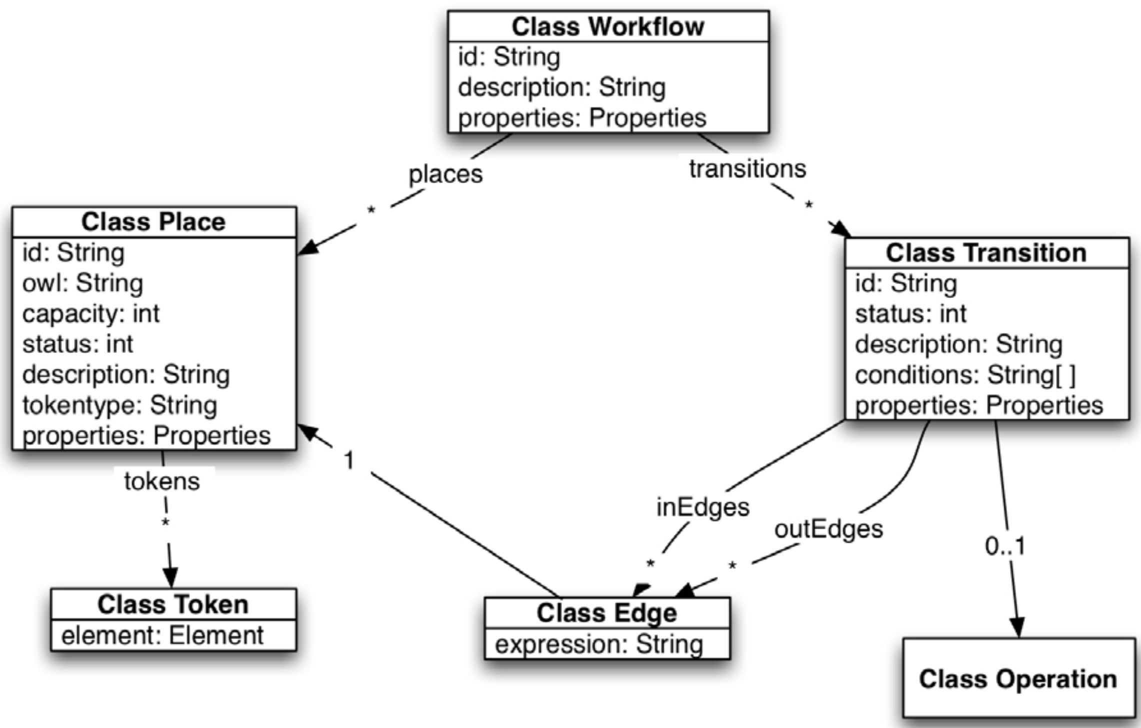


Figure 14: Simplified class diagram of internal Java workflow representation

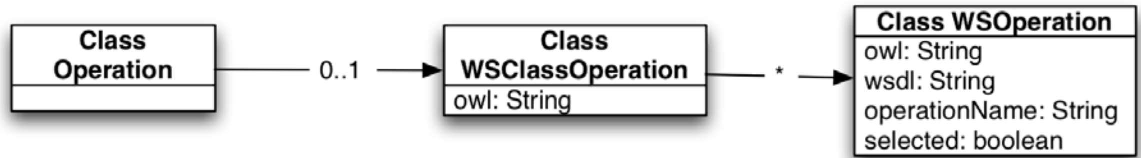


Figure 15: Class diagram of the operation part

Figure 16 shows the different abstraction levels of workflow actions (or in other words transitions and operations). “BLACK”-transitions represent pure control or steering functionality. For the refinement “RED to YELLOW” WCT (Workflow Construction Tool) is responsible, for “YELLOW to BLUE” ABB (automated Application Builder) and for “BLUE to GREEN” the Scheduler.

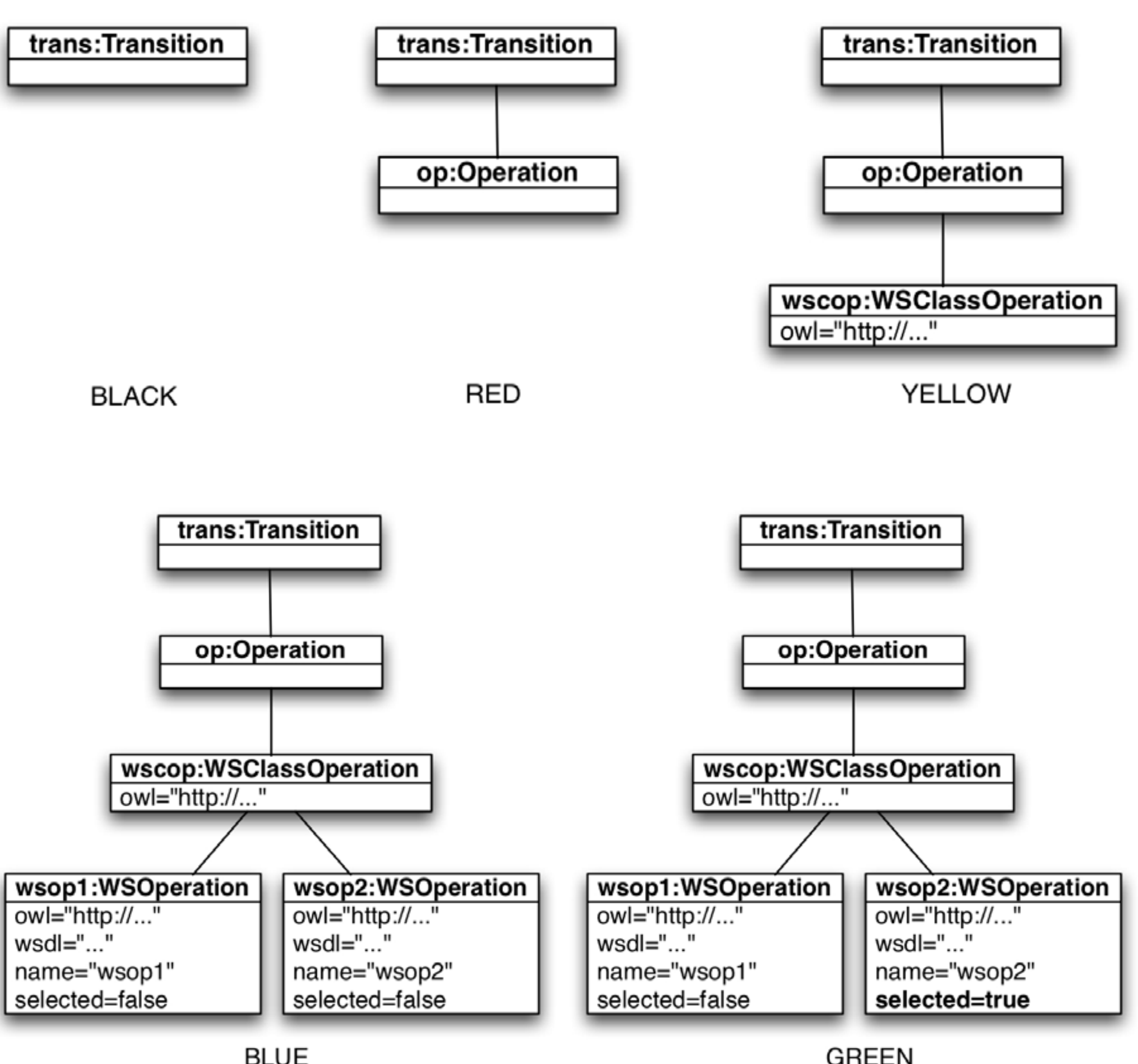


Figure 16: Abstraction levels of transitions and operations

We explain the idea behind the GWorkflowDL Java library by the transition example.

For the implementer of WCT, AAB, and Scheduler only the transition interface Transition is of interest. All what can be done with transitions is declared by methods of this interface. In our illustration only some methods are mentioned.

```
interface Transition {
   String getID();
   Edge getInEdges();
   // ...
}
```

A corresponding implementation is shown next. The name `ArrayListTransition` is a hint to the means used for implementation. In our case `java.util.ArrayList` is used for containers of input and output edges. Another implementation, say `ArrayTransition`, could use Java arrays as containers. The intention is to make the exchange of implementation as simple as possible such that (e.g. performance-) optimization in later development phases easily can be done.

```
public class  ArrayListTransition implements Transition {
    private String id;
    private ArrayList inEdges;
    // ...
    public String getID() {
        return id;
    }
    public Edge[] getInEdges() {
        Edge[] ret = new Edge[inEdges.size()];
        for (int i = 0; i < ret.length; i++) {
            ret[i] = (Edge) inEdges.get(i);
        }
        return ret;
    }
    // ...
}
```

The GWorkflowDL factory can be configured with an implementation of the interface `net.kwfgrid.gworkflowdl.structure.Creator`. The factory simply delegates all method calls to it's creator. To achieve a global exchange of the implementations of the GWorkflowDL interfaces it is sufficient to configure the factory with a different creator which creates the desired implementations. The method to configure the factory is `Factory.setCreator(Creator c)`. An example is the GWorkflowDL protocol which includes different implementations for all interfaces and also includes the according creator which creates these implementations.

Therefore, to achieve a global exchange of an implementation it is sufficient to exchange the constructor in the corresponding factory method, in our example `ArrayListTransition()` by `ArrayTransition()`.

Between the internal JDOM-representation of workflows and the XML-text-representation there is (naturally) a one-to-one isomorphic correspondence of trees. This isomorphism is not completely preserved in relation to the Java internal representation, which is implemented as DAG (directed acyclic graph). The main point in this context is that two different edges (of different transitions) can refer to the same place.

As mentioned above the transformation of the internal Java-representation to the internal JDOM-representation and vice versa is implemented by special utility classes. The class for transitions looks like:

```
public class  Factory {
   private static Creator _creator;

   public static Transition newTransition() {
       return _creator.newTransition();
   }
   //...
}

public class DefaultCreator implements Creator {
   public Transition newTransition() {
       return new ArrayListTransition();
   }
   // ...
}
```

Note, `TransitionJdom.element2java(…)` calls `OperationJdom.element2java(…)` and `TransitionJdom.java2element(…)` calls `OperationJdom.java2element(…)`. This way the translation from XML to Java (and vice versa) follows the hierarchical structure of workflows.

One may ask why `element2java()` in case of transitions has the additional workflow argument. The reason is that in the XML-representation of transitions the places are given by identifiers only. So the workflow must be asked by its method `getPlace(String id)` to find the real places.

4.3.3. Static Workflow Analysis

The workflow analyzer gives answers to questions concerning the long run of KWFGrid workflows. Such questions are for example:

- Can a place get a token?
- Can a transition fire?
- Are there potential conflicts (in the sense of Petri-nets)?
- Must a workflow come to an end or is there an infinite run?
- Is the number of tokens for a place bounded?

The analyzer is based on the Petri-net structure of workflows only. Conditions are not taken into account. Otherwise (nearly) every question would get undecidable (Petri-nets with test-at-zero are mighty like Turing-machines). Hopefully, this will be no serious restriction. (Up to now I didn't see any K-WFGrid workflow with conditions).

4.3.3.1. Theoretical Background

Our static analyzer is based on the Karp-Miller-Tree, well known in the Petri-net-community. The Karp-Miller-Tree construction is a controlled unwinding of the Petri-net that always stops, even in case the net has an infinite run. The main question that can be answered by the Karp-Miller-Tree is whether there is a reachable marking (a token distribution at places) that covers a given marking. This simply can be used to decide whether a place is dead (can never get a token). To do this we define a marking with one token exactly at this given place and ask whether this marking can be covered.

4.3.3.2. Analyzer Usage

Class WorkflowAnalyzer
+ WorkflowAnalyzer(Workflow wf)
+ isQuasiLive(Transition t): boolean + isQuasiLive(Place p): boolean + hasInfiniteRun(): boolean + isUnbounded(Place p): boolean + is Unbounded(): boolean + isFinallyMarked(Place p): boolean + isPersistent(): boolean + getDecisions(): Decision[ ] + analysis2string(): String
+ isComplete(): boolean + getComplexity(): int

Class Decision
+ TAKE_CONFLICT = 0 + TAKE_CHOICE = 1 + PUT_CONFLICT = 2 + PUT_CHOICE = 3
+ type: int + place: Place + transitions: Transition[ ]

Figure 17: Class diagrams for the workflow analyzer

- `WorkflowAnalyzer wfa = new WorkflowAnalyzer(wf);` constructs for a given workflow a new analyzer by building the Karp-Miller-Tree.
- `wfa.isQuasiLive(Transition t)` and `isQuasiLive(Place p)` answer the question whether a transition can fire or, respectively, a place can get a token.
- `wfa.hasInfiniteRun()` tells whether the workflow has an infinite run or must come to an end.
- `wfa.isUnbounded(Place p)`. Is the place `p` unbounded?
- `wfa.isUnbounded()`. Is there an unbounded place?
- `wfa.isFinallyMarked(Place p)` gives answer whether a place at end must be marked. It is (always) correct only in case the net has finite runs only.
- `wfa.isPersistent()`. A net is said to be persistent in case there are no conflicts in the long run. Every transition can loose concession to fire by firing itself only.
- `wfa.getDecisions()`. Returns the array of possible decision situations, explanation follows later on.
- `wfa.analyse2string()` gives a string representation of the performed workflow analysis. Figure 18 shows an example, all places and transitions are quasi-live and there are no conflicts at all.

```
 == general information ==
 Karp-Miller-Tree complete
 complexity=108
 bounded
 stopping
 persistent

== places ==
 begin marking=1
 OrderInserted finally_marked
 wct-place--000--1892231862951968287 finally_marked
 wct-place--001--1892231862951968287 finally_marked
 wct-place--010--1892231862951968287 finally_marked
 wct-place--018--1892231862951968287 finally_marked

== transitions ==
 wct-trans--000--1892231862951968287 enabled

== decisions / conflicts ==
```

Figure 18: Example output of `analyse2string()`

- `wfa.isComplete()` returns whether the Karp-Miller-Tree construction was completed or whether a given complexity bound was exceeded such that the construction was stopped. In the latter case the results of analysis my still give hints, however, there is a reliability miss. This loss of reliability happens with respect to the "far future" because the construction of the Karp-Miller-Tree is performed breadth-first.
- `Wfa.getComplexity()` returns a measure for the Karp-Miller-Tree complexity.

It remains to explain decisions:

Let a decision be given by a type `type`, a place `place` and an array of transitions `transitions[]`. The meaning is that there is a reachable marking such that:

- All transitions in `transitions[]` have concession.
- `type == TAKE_XXX`: All the transitions take a token from the given place.
- `type == PUT_XXX` : All the transitions put a token to the given place
- `type == XXX_CONFLICT`: All the other transitions loose concession in case one of the transitions fires.
- `type == XXX_CHOICE:` There is no conflict.
- The transition array is maximal with respect to these properties.

Decisions occur in situations where someone, the user or the system, has to decide what should be done to continue the workflow execution.

4.3.4. GWorkflowDL Protocol

The GWorkflowDL library contains the implementation of a protocol which can be used to share workflow instances across multiple computers. Due to this protocol workflows can be regarded as distributed Java object trees. The GWorkflowDL protocol is hidden behind implementations of the GWorkflowDL Java interfaces. Therefore in order to make use of the protocol it is sufficient to configure the GWorkflowDL factory with the according creator, either for server or client side of the protocol.

The architecture of the protocol is illustrated in Figure 19. The protocol consists of four layers. The top layer is the API layer. This is the implementation of the GWorkflowDL library. Method calls to the API layer that change the structure of the workflow are handed down to the codec layer. The codec layer encodes the method call. The encoded method call is then processed by the protocol kernel where version checking of distributed workflow instances is implemented. The version checking layer takes care for the synchronization of the distributed instances. The lowest layer is the transport layer. The protocol can be implemented on top of an arbitrary transport mechanism. An example is the communication between GWES and GWUI where SOAP and standard web service calls are used.

The protocol is designed so that the implementation of the codec layer and the transport layer can easily be exchanged. While an XUpdate codec is included in the GWorkflowDL library it is therefore possible to provide a different encoding mechanism for the method calls. By exchanging the implementation of the transport layer the protocol can be easily reused in different networked environments.

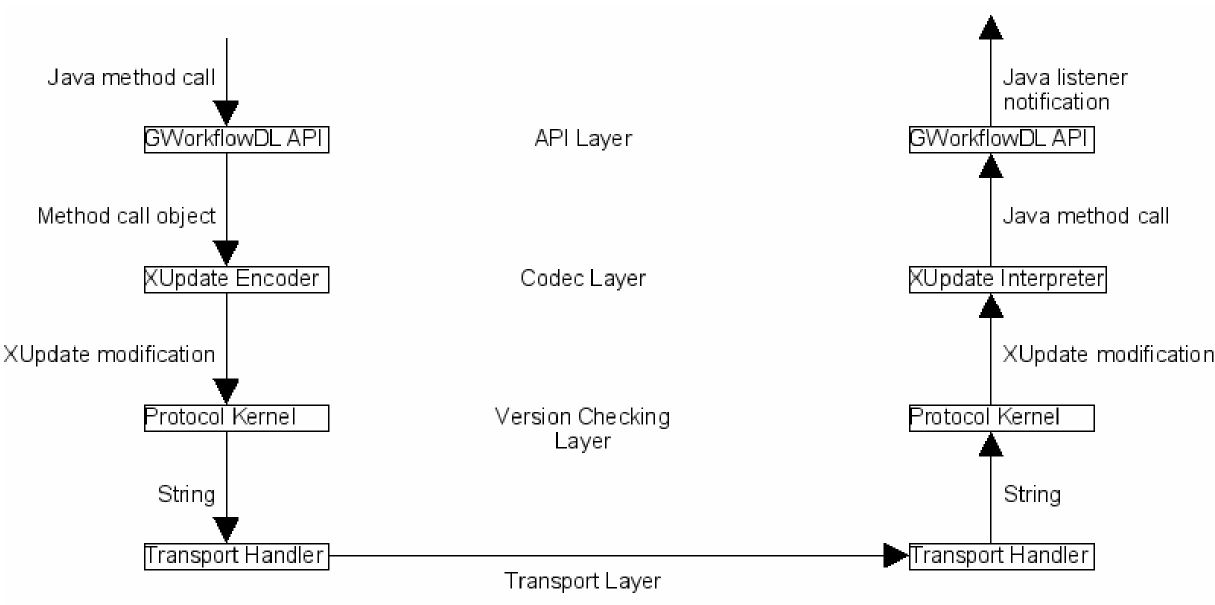


Figure 19: Architecture of the GWorkflowDL protocol.

4.3.4.1. API Layer Details

The API layer includes an implementation of the GWorkflowDL interfaces which trigger the protocol for client and server. The general class diagram of this implementation is illustrated in Figure 20.

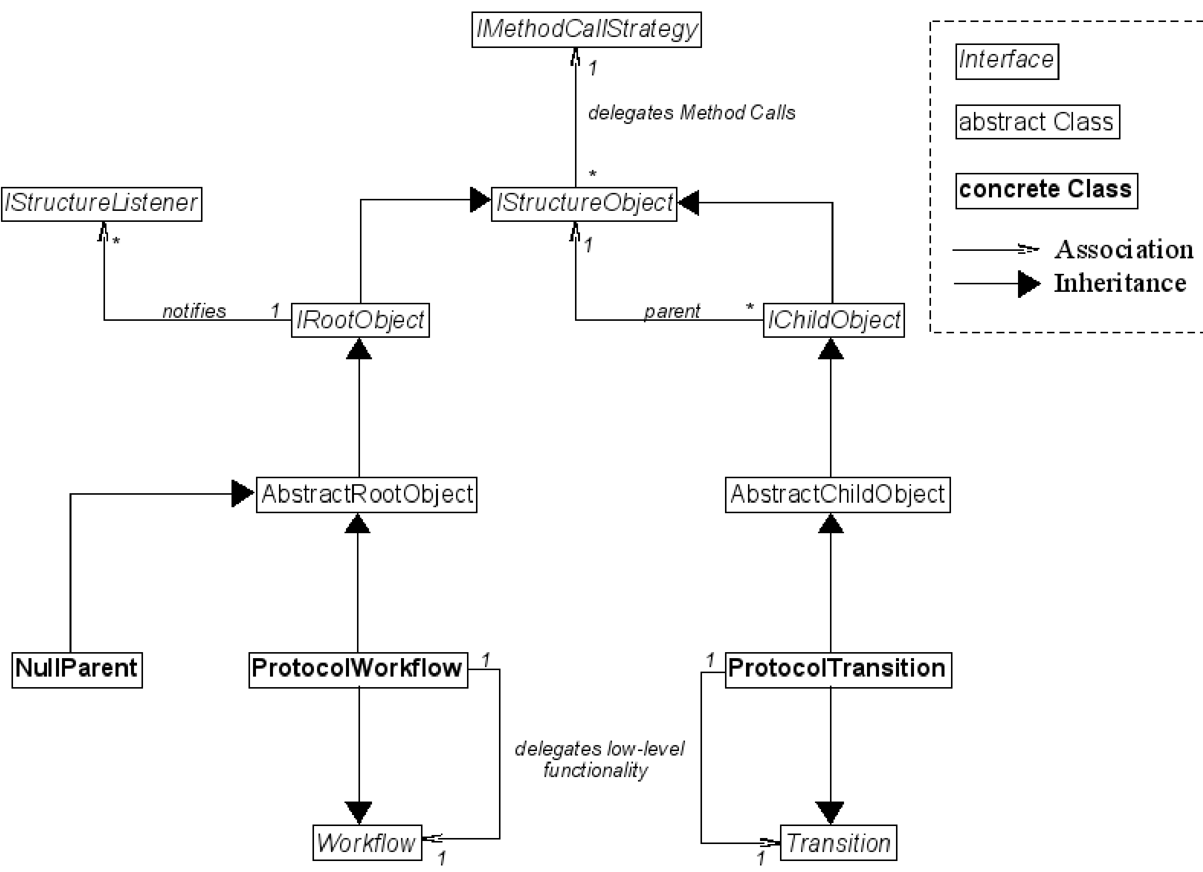


Figure 20: Class diagram of the API layer of the GWorkflowDL protocol.

The API layer provides an abstract tree structure defined by means of the interfaces `IStructureObject`, `IRootObject` and `IChildObject`. These interfaces are first implemented by abstract classes `AbstractChildObject` and `AbstractRootObject` which are further extended by the actual GWorkflowDL protocol classes like `ProtocolWorkflow` and `ProtocolTransition`. These concrete classes in turn also implement the standard GWorkflowDL interfaces like Workflow and Transition. Additionally they all use an instance of their corresponding GWorkflowDL interface as a delegation object. The protocol implementations only add the protocol functionality to the API, while all basic operations are directly delegated to the delegation object. By this usage of the decorator design pattern the protocol can be used with different low-level implementations of the GWorkflowDL API.

The decisive point in the abstract tree structure is that additionally to the child references already defined in standard GWorkflowDL API a parent reference is defined for all child objects in a workflow structure (for example a Transition has a reference to the workflow it is defined in). This makes it possible to communicate changes in lower levels of the workflow structure to the root object (which is the generally the Workflow instance itself) which is important to handle complete workflow structures as distributed entities. In situations where child objects are used which are not already referenced within a workflow structure an instance of the `NullParent` class is used as their root

object. Here the null-object design pattern is implemented to make it transparent for child objects if they actually are referenced inside a workflow or not.

Depending on the status of an object of the API – if it is referenced inside a workflow or not – different strategies for method calls are executed. Every object of a workflow structure has access to an instance of an implementation of the interface `IMethodCallStrategy`. While child objects only use the method-call-strategy of their parent object, the root object actual define the strategy to execute. By using dynamic method-call-strategies it is possible to use the same classes of the GWorkflowDL API for client and server side of the protocol. Additionally the `NullParent` provides a default method-call-strategy which does not trigger the protocol at all. Therefore objects of the protocol API change their behaviour from local to distributed objects as soon as they are added to an actual workflow structure. The interface `IMethodCallStrategy` represents the link from the API layer to the codec layer of the protocol (see next chapter).

Finally the API layer defines the interface `IStructureListener` which enables the observation of a complete workflow structure. This is especially important in GUI environments where an MVC (Model-View-Controller) architecture is implemented.

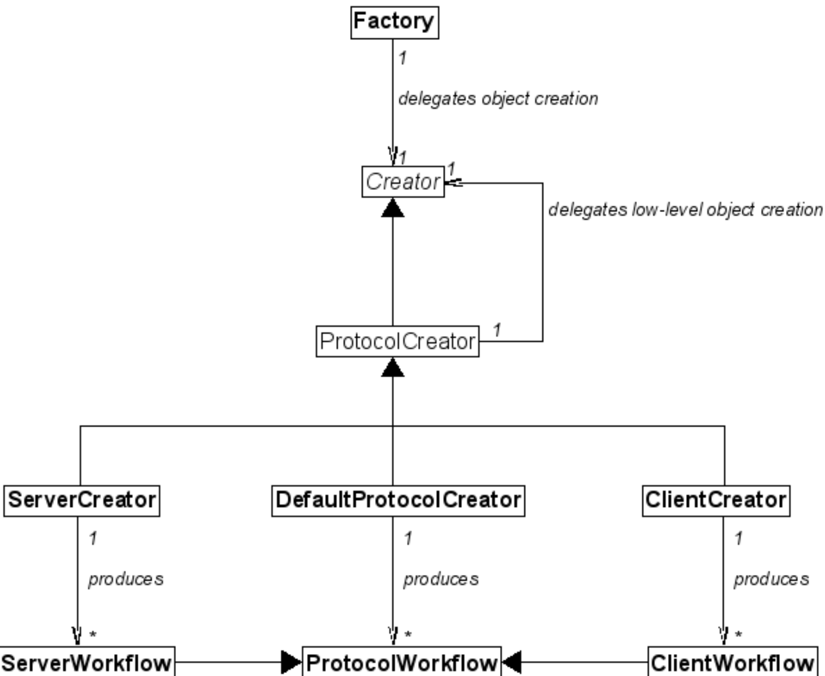


Figure 21: The class diagram showing the creational objects of the GWorkflowDL protocol API layer.

Figure 21 shows the relationships of the creational objects of the protocol API layer. Implementations of the Creator interface are provided for server and client as well as for situations where the protocol should be disabled (`DefaultProtocolCreator`). By setting the right creator as a delegate of the global GWorkflowDL factory the desired implementations can be created. Every instance of `ProtocolCreator` or one of its subclasses uses another implementation of the Creator interface as a delegate to create the low-level implementations of the GWorkflowDL interfaces. This enables the above mentioned decorator pattern for all GWorkflowDL classes of the protocol.

4.3.4.2. Codec Layer Details

The codec layer of the protocol mainly consists of an encoder for all method-calls that change the workflow structure and a handler which decodes these changes and applies them to a remote instance of the same workflow structure. Additionally it includes classes that represent all possible modifying method-calls of the GWorkflowDL API. The class diagram of the codec layer can be found in Figure 22.

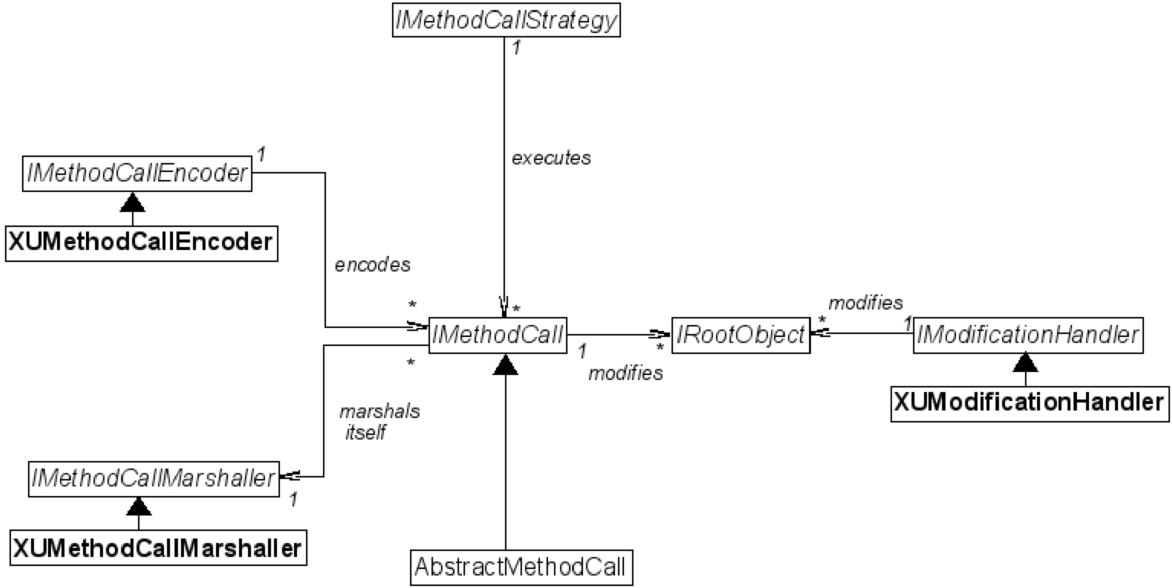


Figure 22: The class diagram showing the most important relationships of the codec layer of the GWorkflowDL protocol.

The core of the codec layer is made up of the three interfaces `IMethodCallEncoder`, `IMethodCallMarshaller` and `IModificationHandler`. Implementations of these interfaces are dedicated for the encoding and decoding of modifications of the workflow structure. The class diagram shows that for all three interfaces an implementation is provided which use the XUpdate standard as encoding syntax (XU stands for XUpdate). A different encoding syntax can easily be integrated into the protocol if desired by providing according implementations of these interfaces.

The method-call-encoder takes instances of the interface `IMethodCall` and hands them over an instance of the method-call-marshaller which is then used by the method-call to marshal itself. The encoded method-call is handed down to the lower layers of the protocol and after all will be handled by an appropriate implementation of the interface `IModificationHandler` on a remote computer. So on the computer where a method-call was originally executed the according implementation of `IMethodCall` is responsible for modifying the workflow structure represented by the interface `IRootObject`; on all remote computers which receive the change originating from the method-call the modification will be handled and applied to the workflow structure by the implementation of `IModificationHandler`.

Finally the class diagram shows the interface `IMethodCallStrategy` which represents the link from the API layer of the protocol to the codec layer.

4.3.4.3. Version Checking Layer Details

The version checking layer is responsible for synchronizing remote instances of the same workflow. A basic client-server architecture based on polling-for-changes by the clients is implemented here which is similar to the well known CVS architecture. Changes made on a client computer are committed to the server. The server checks if the client's version of the workflow is synchronized with the server before the change is executed on the server; if this is not the case the server refuses the modification. If a change is committed the workflow structure on the server is modified accordingly and the difference between the last version and the new version is buffered on the server. In order to synchronize with the server clients can update their workflow structure from the server. The server decides if it needs to send the client the whole workflow structure because the client's version is too old or if it is sufficient to send a list of differences between the current version on the server side and the current version on the client side. The client then either rebuilds the whole workflow structure or applies the list of changes to it's existing workflow structure. One difference to CVS is that also on the server side the workflow structure can directly be manipulated by means of the API layer and the codec layer implementation for the server.

Figure 23 shows the class diagram of the version checking layer. The interfaces `IClientRootObject` and `IServerRootObject` define all properties needed on a per-workflow basis to implement the above described mechanism. Most important is a version number on the client and the server side as well as an instance of a `ModificationBuffer` on the server side. The interfaces are implemented by the GWorkflowDL structure classes `ClientWorkflow` and `ServerWorkflow`. These implementations are rather lightweight themselves and only bundle the required properties. All protocol intelligence is delegated to an instance of `IClientDelegate` and `IServerDelegate` respectively which is shared by all workflow instances on the client or server machine. Default implementations are provided by means of the classes `DefaultClientDelegate` and `DefaultServerDelegate`.

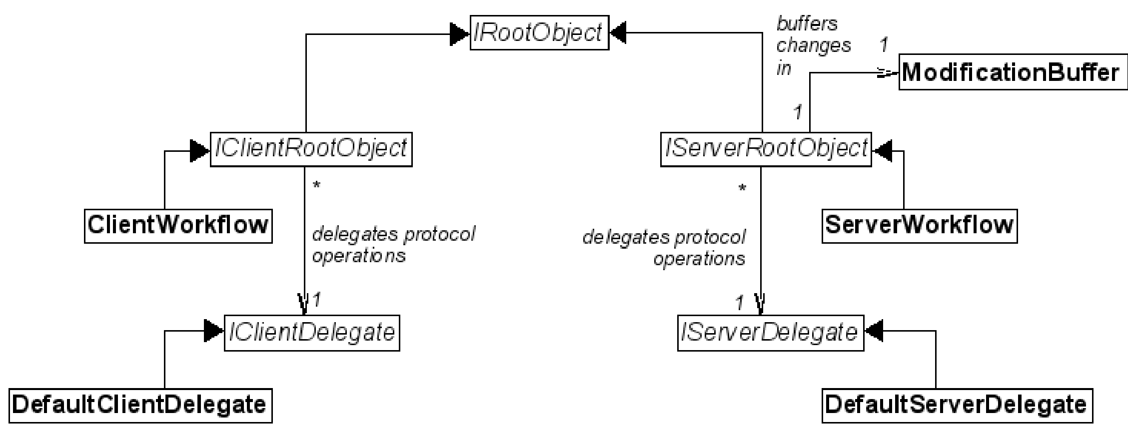


Figure 23: The class diagram of the version checking layer of the GWorkflowDL protocol.

On the client side some strategy for updating the workflow instances needs to be implemented. Such a strategy is not provided as part of the GWorkflowDL API as it's functioning depends on the type and purpose of the client. In some situations it is sufficient to update the client only infrequently; however clients exposing a GUI which visualizes workflow properties should frequently update the workflow

instances by means of a thread running in the background. To update a client workflow the method `void update()` has to be called on the according instance of `ClientWorkflow`.

4.3.4.4. Transport Layer Details

The transport layer is responsible for the actual remote communication. The GWorkflowDL library does not contain a specific implementation of the transport layer; this has to be provided by the users of the API. The transport layer is solely specified by means of the interface `IStructureService` (see Figure 24).

On the client side the interface `IStructureService` needs to be implemented as a stub for a remote service offering the functionality of the two defined methods. An instance of the interface is referenced by the `DefaultClientDelegate` instance; this represents the link between version checking layer and transport layer. On the server side it is up to the user of the API if he/she implements this interface or if he/she chooses different method signatures. The interface is not explicitly referenced on the server side so it must just match the implementation of the stub the user provides on the client side.

IStructureService Method Summary	
`int`	`commitModification(java.lang.String structureid, int clientversion, java.lang.String modification)` Commit a modification to the server.
`java.lang.String[][]`	`getModificationsForUpdate(java.lang.String structureid, int clientversion)` Get the modifications necessary to update the client of a distributed structure.

Figure 24: Interface specifying the transport layer.

The server only needs to map workflow IDs to the according workflow instances. When one of the remote methods is called by a client all that has to be done is looking up the correct workflow instance, casting it to `ServerWorkflow` and calling the according method with the data provided by the client. The methods in `ServerWorkflow` have the signatures

`int commitModification(int clientversion, String modification)` and

`String[][] getModificationsForUpdate(int clientversion)`

which are equal to the signatures of the service interface despite the missing ID parameter.

4.3.4.5. Protocol Configuration

To make use of the GWorkflowDL protocol it has to be configured on the client and on the server side of the application. The configuration must only to be executed once and must be executed before any part of the GWorkflowDL library is used. The initialization part of the application is therefore the right place for protocol configuration.

The server configuration is executed using the following lines of code:

```
import net.kwfgrid.gworkflowdl.structure.Factory;
import net.kwfgrid.gworkflowdl.structure.Creator;
import net.kwfgrid.gworkflowdl.structure.DefaultCreator;
import net.kwfgrid.gworkflowdl.protocol.*;
import net.kwfgrid.gworkflowdl.protocol.xupdate.*;
import net.kwfgrid.gworkflowdl.protocol.server.*;

IMethodCallStrategy defaultStrategy = new DefaultMethodCallStrategy();
IMethodCallStrategy protocolStrategy = new DefaultServerDelegate();
IMethodCallEncoder encoder = new XUMethodCallEncoder();
IModificationHandler handler = new XUModificationHandler();

Protocol.setDefaultMethodCallStrategy(defaultStrategy);
Protocol.setProtocolMethodCallStrategy(protocolStrategy);
Protocol.setMethodCallEncoder(encoder);
Protocol.setModificationHandler(handler);

Creator creatorDelegate = new DefaultCreator();
Creator serverCreator = new ServerCreator(creatorDelegate);
Factory.setCreator(serverCreator);
```

The client configuration is executed using the following lines of code assuming that the implementation of the interface `IStructureService` is given by the class `MyStructureService`:

```
import net.kwfgrid.gworkflowdl.structure.Factory;
import net.kwfgrid.gworkflowdl.structure.Creator;
import net.kwfgrid.gworkflowdl.structure.DefaultCreator;
import net.kwfgrid.gworkflowdl.protocol.*;
import net.kwfgrid.gworkflowdl.protocol.xupdate.*;
import net.kwfgrid.gworkflowdl.protocol.client.*;

IStructureService serviceStub = new MyStructureService();

IMethodCallStrategy defaultStrategy = new DefaultMethodCallStrategy();
IMethodCallStrategy protocolStrategy =
        new DefaultClientDelegate(serviceStub);
IMethodCallEncoder encoder = new XUMethodCallEncoder();
IModificationHandler handler = new XUModificationHandler();

Protocol.setDefaultMethodCallStrategy(defaultStrategy);
Protocol.setProtocolMethodCallStrategy(protocolStrategy);
Protocol.setMethodCallEncoder(encoder);
Protocol.setModificationHandler(handler);

Creator creatorDelegate = new DefaultCreator();
Creator serverCreator = new ClientCreator(creatorDelegate);
Factory.setCreator(serverCreator);
```

4.3.5. GWUI Servlets for customized Data Input Dialogs

GWUI includes a set of servlets which are used to trigger dialogs for application and data specific input dialogs. Together with these servlets GWUI offers a framework for application developers to include sophisticated dialogs tailored for specific data elements the user has to supply during workflow execution. Following is a tutorial for application developers how such dialogs can be implemented as HTML forms or Java applets. The tutorial also describes how the dialogs are registered in the K-Wf Grid system so that they are accessible from within the generic applets offered by GWUI.

Example: Form for person's ID

The tutorial will demonstrate the implementation of a form which will retrieve data about a person like name, prename, age, etc. Let's assume the data needed inside the workflow consists of the following elements:

- Person's name
- Person's prename
- Person's age
- Person's sex
- Person's nationality

We assume that chunks of this information shall be transformed into the following XML representation in order to use them as tokens inside a running workflow:

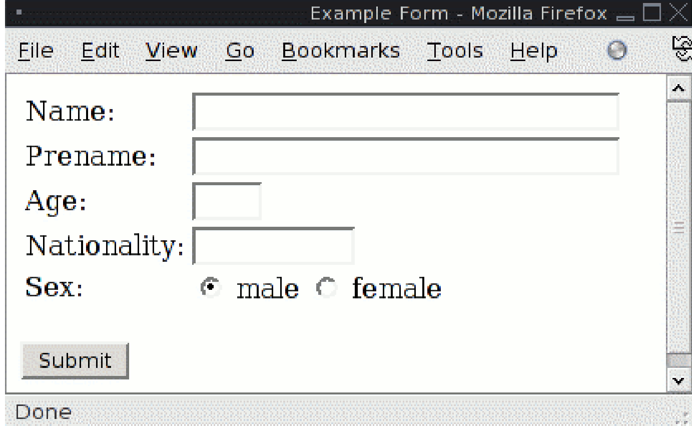
```
<myns:person xmlns:myns="http://www.mynamespace.com/person" sex="female">
    <myns:name>Miller</myns:name>
    <myns:prename>Jane</myns:prename>
    <myns:age>32</myns:age>
    <myns:nationality>UK</myns:nationality>
</myns:person>
```

Providing a form for this type of data consists of three steps:

1. Design the form and implement an HTML template for it or alternatively offer a Java applet for data input.
2. Deploy the template so that it is accessible through an URL.
3. Register the form in the KWf-Grid system to take user input for the according data type.

Design the form and implement an HTML template for it

The form for our example could look something like this:



Note that this is just a simple example. Input forms can make use of the full power of HTML and CSS features in order to give them a sophisticated look and feel. The implementation of the form's HTML template is straight forward. It is just a usual HTML file with the exception that certain placeholders need to be used inside. Also the single form elements need to follow a naming convention in order to map the form elements to XML later on. Let's first have a look at the HTML for the above example. We will then learn all about the placeholders and go deeper into the naming convention.

```
<html>
<head>
<title>Example Form</title>
</head>
<body>
<form action="$SERVLET_URL" method="POST">
    <!-- Namespace declarations in hidden form elements. -->
    <input type="hidden" name="NAMESPACE_myns" value="http://www.mynamespace.com/person">
<table>
    <!-- Input elements for the actual data. -->
    <tr><td>Name:</td><td><input name=".myns:person.myns:name" type="text"
value="$VALUE_.myns:person.myns:name" size="30"></td></tr>
    <tr><td>Prename:</td><td><input name=".myns:person.myns:prename" type="text"
value="$VALUE_.myns:person.myns:prename" size="30"></td></tr>
    <tr><td>Age:</td><td><input name=".myns:person.myns:age" type="text"
value="$VALUE_.myns:person.myns:age" size="3"></td></tr>
    <tr><td>Nationality:</td><td><input name=".myns:person.myns:nationality" type="text"
value="$VALUE_.myns:person.myns:nationality" size="10"></td></tr>
    <tr><td>Sex:</td><td><input name=".myns:person_sex" type="radio" value="male"> male <input
name=".myns:person_sex" type="radio" value="female"> female</td></tr>
</table><br>
    <!-- And finally the submit button. -->
    <input type="submit" value="Submit">
</form>
</body>
</html>
```

Let's go through the HTML file step by step...

As you can see in the opening "<form ...>" tag the form's template needs to include a placeholder for the URL of the servlet which consumes the user's input. That placeholder has the name "$SERVLET_URL". It will be replaced automatically with an appropriate URL. The form can use the POST or the GET method, the servlets are designed to deal with both of them.

Looking at the "name" attributes of the single input elements you can already get a clue about the naming scheme. Here are the details:

1. **Namespaces**
   If any namespaces are used inside the XML specification of your data, you need to define them using hidden HTML inputs. The name of the hidden field is build up of the placeholder "NAMESPACE_" and the prefix of the namespace attached to it. The value of the input is just the URI of the namespace. In our example we use the prefix "myns" for the namespace "http://www.mynamespace.org/person". This leads to the input's name "NAMESPACE_myns" as you can see in the example.
2. **Names of input fields**
   To map the values of the single input fields to an XML element the names of the inputs need to match the following scheme: Element names are prefixed with ".", attribute names are prefixed with "_". If an element or attribute is defined in a certain namespace the node's name needs to be prefixed by the namespace's prefix. (And the namespace must be defined in an hidden input field as described in 1.) By using this scheme the name of an input field can be constructed by following the path from the XML root node to the node which is edited by the according input field.
   For example take a look at the attribute "sex" of our "person" element. The path to this attribute from the root node is "person" -> "sex". Now concatenate these two regarding the naming scheme and the namespace definition for the element "person". The resulting name for the input field is ".myns:person_sex".

Finally we come to the "value" attributes of our input fields. The form shall also be used for data display so we need some placeholders here in order to fill in actual data. The placeholders will be automatically replaced by appropriate data values if the form is being used for data display.

The placeholders for the values have the prefix "$VALUE_". Attached to this prefix is the name of the input field which is constructed as described above.

For example the placeholder for the "sex" attribute of the "person" element consists of the "$VALUE_" prefix concatenated with the name of the according input field ".myns:person_sex". Thus the placeholder is "$VALUE_.myns:person_sex".

Using these information you can design the complete form for your data element. Don't forget the "submit" button and you are done.

Alternative: Offer a Java applet for data input

It is also possible to offer the users a Java applet for data input. The implementation in this case involves creation of the applet and the implementation of the HTML template which includes the <applet ...> tag to launch your applet. In the HTML template you will have to use the same placeholders as if you would write a HTML template for an HTML form (see above). In your applet you will need to construct the correct URL to call the servlet consuming the data input. This is done automatically if you provide an HTML form.

The following sub sections describe the single steps that are required to implement data input applets for a specific workflow activity:

1. Implement the HTML template to launch the applet

This is straight forward while you have to take care about some placeholders in your template. Please read section "Design the form and implement an HTML template for it" if you have not already done so. It contains some valuable information about the placeholders.

Here is the example code for an HTML template to launch an applet.

```
<html>
<head>
<title>Example Applet</title>
</head>
<body>
<applet code="MyApplet.class"
        codebase="http://www.myserver.com/"
        width="300"
        height="200">
        <!-- Applet parameter holding the servlet URL. -->
        <param name="servletURL" value="$SERVLET_URL">
        <!-- Applet parameters to transfer the data values to the applet. -->
        <param name="sex" value="$VALUE_.myns:person_sex">
        <param name="name" value="$VALUE_.myns:person_myns:name">
        <param name="prename" value="$VALUE_.myns:person_myns:prename">
        <param name="age" value="$VALUE_.myns:person_myns:age">
        <param name="nationality" value="$VALUE_.myns:person_myns:nationality">
</applet>
</body>
</html>
```

We use applet parameters to transfer some information from the servlet to the applet. In your template you need to prepare certain parameters and put placeholders into their value attributes. These parameters are required:

1. A parameter for the servlet URL (value must be the placeholder "$SERVLET_URL").
2. A set of parameters holding data values if the applet is launched to display a data object. Here you must use the placeholders according to the naming scheme described in the above chapter.

Note that the names of those parameters can be chosen arbitrarily. It is required only that you use the correct placeholders inside their value attributes.

2. Implement your applet

Inside your applet you have to process the parameters that hold data values. Their content should be displayed to the user in a way appropriate for the actual data type. When the user finishes his input and confirms to commit the data he entered to the workflow, the applet needs to construct the correct URL of the servlet and call this URL in the browser. The URL must contain the data the user entered just as if the URL would result from a HTTP "GET" call. Here is the example code for our person data:

```
import java.net.URLEncoder;
import java.net.MalformedURLException;
import java.applet.*;
import javax.swing.JApplet;

public class MyApplet extends JApplet {
    private String _servletURL;
    private String _name;
    private String _prename;
    private String _sex;
    private String _nationality;
    private int _age;

    public void init() {
        // read the parameters and store them inside member variables
        // the values should be used when the GUI is constructed.

        _servletURL = getParameter("servletURL");
        _name = getParameter("name");
        _prename = getParameter("prename");
        _sex = getParameter("sex");
        _nationality = getParameter("nationality");
        _age = Integer.parseInt(getParameter("age"));

       // … Do other initialization, build the GUI etc…
    }

    // ....

    /**
        This method can be called to commit the data entered by the user to the server.
        @param name The name of the person.
        @param prename The prename of the person.
        @param sex The person's sex.
        @param age The age of the person.
        @param nationality The nationality of the person
     */
    private String commitInput(String name,
                              String prename,
                              String sex,
                              int age,
                              String nationality) throws UnsupportedEncodingException,
MalformedURLException {
        String url = encodeURL(_servletURL, name, prename, sex, age, nationality);
        getAppletContext().showDocument(new URL(url));
    }

    /**
        This method will encode the URL for the servlet call as if it would result from a HTTP
"GET".
        @param servletURL The URL of the servlet to call.
        @param name The name of the person.
        @param prename The prename of the person.
        @param sex The person's sex.
        @param age The age of the person.
        @param nationality The nationality of the person
     */
```

```
    private String encodeURL(String servletURL,
                            String name,
                            String prename,
                            String sex,
                            int age,
                            String nationality) throws UnsupportedEncodingException {
        StringBuffer url = new StringBuffer();
        // start with the servlet URL and append the "?"
        url.append(servletURL).append("?");
        // now namespace declarations
        url.append("NAMESPACE_myns=").
            append(URLEncoder.encode("http://www.mynamespace.com/person", "UTF-8")).
            append("&");
        // now the data values
        url.append(".myns:person.myns:name=").
            append(URLEncoder.encode(name, "UTF-8")).
            append("&");
        url.append(".myns:person.myns:prename=").
            append(URLEncoder.encode(prename, "UTF-8")).
            append("&");
        url.append(".myns:person.myns:age=").
            append(age).
            append("&");
        url.append(".myns:person.myns:nationality=").
            append(URLEncoder.encode(nationality, "UTF-8")).
            append("&");
        url.append(".myns:person_sex=").
            append(URLEncoder.encode(sex, "UTF-8")).
            append("&");
        return url.toString();
    }
}
```

In the example you should take a closer look at the methods "encodeURL" and "commitInput".

The "encodeURL" method constructs the complete URL which needs to be called to commit the data entered by the user to the server. This URL has to include parameters for the namespace declarations of your data element and all the values. Please reread the naming scheme described above to understand how namespaces are declared and how the names of the single data elements have to be choosen in order to be mapped to an XML element later on. *Please regard that every data value (not name) you append to the URL needs to run through URLEncoder.encode(...).*

The "commitInput" method takes input from the user, calls "encodeURL" to encode the URL and then uses the browser accessible through "getAppletContext().showDocument(...)" to call that URL.

When you are finished with your applet you can continue with …

3. Deploy the template so that it is accessible through an URL

The next step is to deploy your template and all the requisites needed (like images, applet, CSS files etc.). You need to upload your template and all other files to a publicly available webserver. A servlet will download the template from there, parse it and replace all placeholders and will then send the resulting page to the user. Other requisites will be directly loaded from your server by the client's browser.

NOTE: As the template is not directly loaded from a browser you need to care about relative links and relative references to images and such inside your template. The best practice is to avoid relative references at all and only use absolute references to your requisites.

4. Register the form in the KWf-Grid system to take user input for the according data type

Finally you need to register the form inside the KWf-Grid system.

To enable discovery and use of the form template for data input, you have to provide:

- URL of the form template
- Human-readable description of the data input task (e.g. "Please provide input data for...")
- Recommended width and height of the form window
- Optional URL of the consumer servlet (for specialised forms that can't be serviced by the standard form servlet)

These parameters can be added in two ways:

- Through the service WSDL file (recommended way)
- Through a dedicated GOM browser tab in the K-Wf Grid portal

5. Annotation of WSDL file

To add form description into the WSDL file of the service for which the form is intended, you have to add documentation to its *operation* description in the *porttype* part of the WSDL document. You do this by using the *<documentation>* tag, with the actual description having the following syntax:

```
INPUTFORM=URL of the input form;LABEL=human-readable label;WIDTH=integer number;HEIGHT=integer
number;CONSUMER=URL of the consumer servlet
```

If you intend to provide several input forms, you just chain the descriptions - the **INPUTFORM** of the second form will follow the **CONSUMER** of the first form (or other tag, if you order them differently). Example:

```
<portType name="UserProxyServicePortType"
      wsdlpp:extends="kwfgs:KWfGridServicePortType
        wsrpw:GetResourceProperty
        wsrpw:GetMultipleResourceProperties
        wsrpw:SetResourceProperties
        wsrpw:QueryResourceProperties
        wsntw:NotificationProducer"
      wsrp:ResourceProperties="tns:UserProxyProperties">
    <operation name="configureFromProperties">
      <input message="tns:ConfigureInputMessage">
        <documentation>
           INPUTFORM=http://www.nothing.eu/upsform.html;LABEL=Please provide user
name;WIDTH=500;HEIGHT=300;
           INPUTFORM=http://www.nothing.eu/superupsform.html;LABEL=Please select user from
directory;WIDTH=400;HEIGHT=600;CONSUMER=http://www.kwfgrid.eu/portal/userdirform.jsp
        </documentation>
        </input>
      <output message="types:VoidOutputMessage"/>
    </operation>
        <operation name="userHasFinished">
                <input message="tns:UserHasFinishedInputMessage"/>
                <output message="types:VoidOutputMessage"/>
        </operation>
  </portType>
```

4.4. PRODUCT INTERFACES

The Grid Workflow Execution Service can be deployed as a Web Service, e.g. by means of Apache Tomcat and Axis. Therefore the main external interface of the GWES is a Web Service interface, described by its wsdl, which can be downloaded at http://fhrg.first.fraunhofer.de:8080/gwes/services/GWES?wsdl . The GWES web service implements two Java interfaces: GWES and IStructureService.

The GWES interface contains methods for the management of Grid workflows:

GWES Method Summary	
void	**abort**(java.lang.String workflowID) Abort a workflow with a certain identifier.
java.lang.String[]	**getData**(java.lang.String workflowID, java.lang.String placeID) Get some data that is hold inside a specific workflow and that is referenced by a data place identifier.
int	**getStatus**(java.lang.String workflowID) Get the current status code of the workflow specified by its identifier.
java.lang.String	**getWorkflowDescription**(java.lang.String workflowID) Get the current Grid Workflow Description document of the workflow specified by its identifier.
java.lang.String[]	**getWorkflowIDs**() Get the identifiers of all the workflows that are handled by this Grid Workflow Execution Service.
java.lang.String[][]	**getWorkflowStatusArray**() Get the workflow status of all currently available workflows.
java.lang.String	**initiate**(java.lang.String gworkflowdl, java.lang.String userID) Initiates a Grid workflow with a certain userID.
java.lang.String	**restart**(java.lang.String workflowID, java.lang.String userID) Restart a workflow with a certain identifier from the beginning.
java.lang.String	**restore**(java.lang.String workflowID, java.lang.String userID) Restores a workflow from the XML database.
void	**resume**(java.lang.String workflowID) Resume a workflow with a certain identifier.
void	**start**(java.lang.String workflowID) Start a workflow with a certain identifier.
java.lang.String	**store**(java.lang.String workflowID) Store a workflow in the XML workflow database.
void	**suspend**(java.lang.String workflowID) Suspend a workflow with a certain identifier.

The IStructureService interface specifies methods that are used for the communication between the Grid Workflow Execution Service (GWES) and the Grid Workflow User Interface (GWUI) which are implemented in the GWorkflowDL protocol (refer to Section 4.3.4):

For a more detailed description of the interfaces refer to the JavaDoc of the classes net.kwfgrid.gworkflowdl.protocol.service.IStructureService and net.kwfgrid.gwes.GWES.

5. PRODUCT TESTING

Several JUnit tests have been invoked in order to validate the certain functionalities of the GWES as well as the integrated system. The current reports of the JUnit tests are available online at http://www.gridworkflow.org/kwfgrid/gwes/docs/junit-report.html . The sources of the test cases are available at http://www.gridworkflow.org/kwfgrid/gwes/docs/xref-test/ .

5.1. GWES TEST CASES

5.1.1. GWESTest

	Test	Description
	testSortWorkflow	Test the execution of the example workflow “gworkflowdl_sort.xml”, which invokes one external Web Service (sort). The test is only successful if the workflow completes with the correct output data.
	testSortTailWorkflow	Test the execution of the example workflow “gworkflowdl_sort-tail.xml”, which sequentially invokes two external Web Services (sort and tail). The test is only successful if the workflow completes with the correct output data.
	testGrepWorkflow	Test the execution of the example workflow “gworkflowdl_grep.xml”, which invokes one external Web Service (grep). The test is only successful if the workflow completes with the correct output data.
	testDuplicateWorkflow	Test the execution of the example workflow “gworkflowdl_duplicate.xml”, which duplicates tokens. The test is only successful if the workflow completes with the correct output data.
	testWaitWorkflow	Test the execution of the example workflow “gworkflowdl_wait.xml”, which invokes one external Web Service (wait). The test is only successful if the workflow completes with the correct output data.
	testWebServiceFault	Test the exception handling of the GWES by trying to enact an incorrect workflow “gworkflowdl_sortFault.xml”, which invokes one external Web Service (sort) with wrong type of input parameter. The test is only successful if the workflow terminates with the corresponding soap fault.
	testLoop	Test the execution of the example workflow “gworkflowdl_loop.xml”, which contains a loop that iterates ten times. The test is only successful if the workflow completes with the correct output data.

5.1.2. GWESWSTest

	testGWESWS	Test the Web Service Interface of the GWES by executing the example workflow "gworkflowdl_sort.xml" using the remote Web Service Interface. This test validates the initiate() and start() methods of the GWES Web Service.
	testGWES_SOAPFault	Test the exception handling of the GWES Web Service by trying to start a workflow with a wrong workflow identifier. The test is only successful if the client catches a soap fault.

5.1.3. WSClientTest

	testWSClientToken	Test the WSClient class, which invokes arbitrary Web Service operations on remote hosts. The test is performed by invoking the remote "sort" Web service with example input, and by evaluating the return values.

5.1.4. GWES_SuspendResumeTest

	testWaitWait	Test the GWES methods suspend() and resume() using the "gworkflowdl_waitwait.xml" example workflow. The test is only successful if the workflow completes with the correct output data.
	testAbort	Test the GWES method abort() using the "gworkflowdl_waitwait.xml" example workflow. The test is only successful if the workflow terminates after invoking abort().

5.1.5. GWES_CTMTest

	testCTM	Test the enactment of a concrete workflow from the CTM pilot application ("gworkflowdl_CTM_UC_workflow_v0.3.xml"). This workflow invokes three remote Web Services at Softeco in Italy. The test is only successful if the workflow completes with the correct type of output data.

5.1.6. GWES_GWUITest

	testWithListener	Test the subscription of a client to the GWES UIProxy.

5.1.7. GWES_HierarchicalTest

	testHierarchicalWorkflow	Test the hierarchical enactment of a workflow that enacts a workflow ("gworkflowdl_hierarchical.xml"). The test is only successful if the workflow completes with the correct type of output data.

5.1.8. GWES_AABTest

	testAABwithCTMWorkflow	Test the GWES invocation of the AAB with a workflow from the CTM pilot application ("ctm_aab-input-wf.xml").
	testAABwithERPWorkflow	Test the GWES invocation of the AAB with a workflow from the ERP pilot application ("erp_aab-input-wf.xml").
	testAABwithFFSCWorkflow	Test the GWES invocation of the AAB with a workflow from the FFSC pilot application ("ffsc_aab-input-wf.xml").

5.1.9. GWES_DecisionTest

	testUnresolvedDecisions	Test the automatic detection of unresolved decisions using the example workflow "gworkflowdl_decision.xml". The test is successful only if the workflow conflict is detected during the execution and the GWES suspends the workflow.

5.1.10. GWES_SchedulerTest

	testSchedulerWithCTMWorkflow	Test the GWES invocation of the Scheduler with a workflow from the CTM pilot application ("ctm_aab-output-wf.xml")
	testSchedulerWithERPWorkflow	Test the GWES invocation of the Scheduler with a workflow from the ERP pilot application ("erp_aab-output-wf.xml")

5.1.11. GWES_WCTTest

	testWCTwithCTMWorkflow	Test the GWES invocation of the WCT with a workflow from the CTM pilot application ("ctm-new-red-wf.xml")
	testWCTwithERPWorkflow	Test the GWES invocation of the WCT with a workflow from the ERP pilot application ("erp-red-wf.xml")
	testWCTwithFFSCWorkflow	Test the GWES invocation of the WCT with a workflow from the FFSC pilot application ("ffsc-red-wf.xml")

6. KNOWN ISSUES

If you have access to the K-Wf Grid issue tracking system (currently only for K-Wf Grid developers), you will find more details by following the link connected to each issue ID. For issue/bug reporting please contact Andreas Hoheisel (andreas.hoheisel@first.fraunhofer.de) or report it directly to the issue tracking system at http://cvs.ui.sav.sk/mantis/.

6.1.1. GWES

ID	Category	Severity	Summary
0000085	gwes	**major**	GWES should fire the transition AFTER completing the activity
0000094	gwes	minor	GWES: Exclude JUnit tests in project.xml which depend on external resources
0000092	gwes	minor	GWES: GRAMActivity NullPointerException when aborting workflow
0000089	gwes	minor	GWES source distribution contains confidential data
0000088	gwes	minor	GWES: The RFTService should throw an execption if the output token contains a soap fault
0000054	gwes	minor	GWES Activity Statistics are sometimes wrong
0000032	gwes	minor	Exception Management for Activities
0000014	gwes	minor	GWES sends activity_terminated without activity_initiated
0000019	gwes	minor	error when getting workflowdescription when workflow is finished
0000008	gwes	minor	GWES: Problem when output place has capacity=1 and there are several input tokens
0000061	gwes	text	Document GWorkflowDL <property> elements processed by GWES
0000013	gwes	trivial	GWES: Use wsdl4j for parsing the wsdl file
0000080	gwes	feature	GWES should store all workflow tokens in XML database
0000066	gwes	feature	Test and optimize the scalability of the GWES
0000063	gwes	feature	Testscript for Remote GWES Test
0000031	gwes	feature	GUI for configuration of GWES
0000033	gwes	feature	Include Fault Tolerance Mechanisms

6.1.2. GWUI

0000075	gwui	**crash**	Portal GWUI applet sometimes hangs (firefox too)
0000037	gwui	**major**	upload new workflow after termination of old workflow hangs
0000093	gwui	minor	GWUI-0.5.2 sometimes does not display the workflow after calling graph layouter
0000071	gwui	minor	GWUI should center the workflow after loading it.
0000064	gwui	minor	Scrollbar of Workflow Inspector is deactivated if workflow is running, terminated or completed
0000027	gwui	minor	Invisible arrow heads
0000028	gwui	feature	visualize the state of the operations related to transistions
0000067	gwui	feature	GWUI: Provide buttons/user interface for all main GWES Web Service operations
0000043	gwui	feature	Button for storing workflows
0000060	gwui	feature	GWUI: Tokens with SOAPFaults

6.1.3. GWorkflowDL

ID	Category	Severity	Summary
0000025	gworkflowdl	minor	NullpointerException when parsing wrong

			GWorkflowDL
0000069	gworkflowdl	feature	Implement service for creating huge parallel and/or sequential workflows

7. CONTACT INFORMATION AND CREDITS

The following persons were involved in the development of the GWES, GWUI, and GWorkflowDL:

- Andreas Hoheisel, Fraunhofer FIRST (developer and workpackage leader)
- Hans-Werner Pohl, Fraunhofer FIRST (developer)
- Tilman Linden, Fraunhofer FIRST (developer)
- Steffen Loos, Fraunhofer FIRST (developer)

For further information or bug reporting please contact Andreas Hoheisel (andreas.hoheisel@first.fraunhofer.de)

8. THE FRAUNHOFER FIRST LICENSE AGREEMENT

Copyright (c) 2005 Fraunhofer Gesellschaft/Fraunhofer FIRST. All rights reserved.

This software includes voluntary contributions made by Fraunhofer FIRST under the K-WfGrid-project. For more information on K-WfGrid, please see http://www.kwfgrid.eu.

Installation, use, reproduction, display, and modification of this software, with or without modification, in source and binary forms, are permitted, however only for Licensee's own scientific or educational purposes. Any use beyond – especially but not limited for commercial purposes – and/or distribution to third parties requires K-WfGrid's prior written consent. Any exercise of rights under this license by you is subject to the following conditions:

1. Any use of this software, with or without modification, must contain the above copyright notice and the above license statement as well as this list of conditions, in the software, the user documentation and any other materials provided with the software.

2. The user documentation, if any, must include the following notice: "This product includes software developed by K-WfGrid (www.kwfgrid.eu)." Alternatively, if that is where third-party acknowledgments normally appear, this acknowledgment must be reproduced in the software itself.

3. The names "K-WfGrid" and "Knowledge Workflow Grid" may not be used to endorse or promote software, or products derived therefrom, except with prior written permission by steffen.unger@first.fraunhofer.de.

4. You are under no obligation to provide anyone with any bug fixes, patches, upgrades or other modifications, enhancements or derivatives of the features, functionality or performance of this software that you may develop. However, if you publish or distribute your modifications, enhancements or derivative works without contemporaneously requiring users to enter into a separate written license agreement, then you are deemed to have granted participants in K-WfGrid a worldwide, non-exclusive, royalty-free, perpetual license to install, use, reproduce, display, modify, redistribute and sub-license your modifications, enhancements or derivative works, whether in binary or source code form, under the license conditions stated in this list of conditions.

5. DISCLAIMER

THIS SOFTWARE IS PROVIDED BY Fraunhofer FIRST AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, OF SATISFACTORY QUALITY, AND FITNESS FOR A PARTICULAR PURPOSE OR USE ARE DISCLAIMED. Fraunhofer FIRST AND CONTRIBUTORS MAKE NO REPRESENTATION THAT THE SOFTWARE, MODIFICATIONS, ENHANCEMENTS OR DERIVATIVE WORKS THEREOF, WILL NOT INFRINGE ANY PATENT, COPYRIGHT, TRADE SECRET OR OTHER PROPRIETARY RIGHT.

6. LIMITATION OF LIABILITY

Fraunhofer FIRST AND CONTRIBUTORS SHALL HAVE NO LIABILITY TO LICENSEE OR OTHER PERSONS FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, EXEMPLARY, OR PUNITIVE DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, LOSS OF USE, DATA OR PROFITS, OR BUSINESS INTERRUPTION, HOWEVER CAUSED AND ON ANY THEORY OF CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE, ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Fraunhofer FIRST takes no maintenance obligations.