

Deliverable 3.3 Appendix 2 **ViroLab Virtual Laboratory:** **Experiment Developers' Manual**

Project Start:	01-03-2006
Project Duration:	36 Months
Priority area	2.4.11
Contract No.:	INFSO-IST-027446
Website:	http://www.virolab.org

Due-Date:	31-08-2007
Delivery:	13-09-2007
Lead Partner:	CYFRONET
Coordinator	UvA, Prof. Dr P.M.A. Sloot
Dissemination Level:	Public
Status:	Final
Approved:	Quality Board, Steering Committee
Version:	1.0

Log of Document

Version	Date	Changes Summary	Authors
0.1	29/08/2007	Initial version of the manual	Tomasz Gubala
0.2	29/08/2007	Main contribution imported	Robert Pajak, Tomasz Gubala, Dariusz Krol, Marek Kasztelnik, Piotr Regiel, Eryk Ciepiela, Tomasz Bartynski, Joanna Kocot, Piotr Nowakowski
0.3	30/08/2007	Formatting sections 3.2.1, 3.2.2 and 3.2.3	Marek Kasztelnik
0.4	30/08/2007	DAC section reviewed and updated	Piotr Nowakowski
0.5	30/08/2007	Formatting sections 3.1.1 and 3.1.2	Dariusz Krol
0.6	30/08/2007	Input to section 4	Marek Kasztelnik
0.7	31/08/2007	Small changes	Tomasz Gubala
0.8	02/09/2007	Section 5.1 checked, refined, supplemented and formatted	Eryk Ciepiela
0.9	03/09/2007	Added sections: Example experiments, grid object abstractions, EPE intro	Tomasz Gubala
0.10	06/09/2007	Section 5.1 rechecked	Eryk Ciepiela
0.11	06/09/2007	Section 5.3 rechecked	Tomasz Bartyński
0.12	06/09/2007	Section 5.3 refined	Tomasz Bartyński
1.0	13/09/2007	Manual title changed, DAS section added, minor changes	Marian Bubak, Matthias Assel, Aenne Löhden

TABLE OF CONTENTS

COPYRIGHT NOTICE.....	8
1. INTRODUCTION.....	10
1.1. TARGET AUDIENCE.....	10
1.2. MORE INFORMATION.....	10
2. EXPERIMENT DEVELOPMENT ENVIRONMENT DESCRIPTION.....	12
2.1. EXPERIMENT PIPELINE IDEA.....	12
2.2. DEFINED CLASSES OF USERS.....	13
2.3. MORE DETAILED VIEW.....	13
2.4. THE EXPERIMENT PROCESS SCRIPT.....	15
2.5. GRID OBJECTS ABSTRACTIONS.....	16
2.5.1. <i>Explanation of abstraction levels</i>	16
2.5.2. <i>An example</i>	17
2.6. FUNCTION OF THE EXPERIMENT REPOSITORY.....	17
3. EXPERIMENT PLANNING ENVIRONMENT USER'S MANUAL.....	20
3.1. EXPERIMENT PLANNING ENVIRONMENT.....	20
3.1.1. <i>Installation and Configuration</i>	20
3.1.2. <i>Usage</i>	26
3.1.3. <i>Source Code Access, Bug Reporting and Authors Contact Information</i>	39
3.2. EXPERIMENT PLANNING PLUG-INS.....	40
3.2.1. <i>Installation</i>	40
3.2.2. <i>Virtual Organization Configuration Plug-in</i>	41
3.2.3. <i>Resources Browser Plug-in</i>	43
3.2.4. <i>Ontology Browser Plug-in</i>	53
3.2.5. <i>Source Code Access, Bug Reporting and Authors Contact Information</i>	58
4. GRID RESOURCES REGISTRY USER'S MANUAL.....	59
4.1. GRID RESOURCES WEB BROWSER.....	59
4.2. ADDING NEW GRID OBJECTS.....	60
4.2.1. <i>Preparing your Grid Object</i>	60
4.2.2. <i>Functionality</i>	61
4.2.3. <i>Interface</i>	61
4.2.4. <i>Interaction mode</i>	61
4.2.5. <i>Supported technologies and protocols to implement Grid Object</i>	62
5. GRIDSPACE EXPERIMENT DEVELOPER LIBRARY REFERENCE.....	65
5.1. LIBRARY CORE REFERENCE.....	65
5.2. DATA ACCESS REFERENCE.....	65
5.3. COMPUTATION ACCESS REFERENCE	67
6. EXAMPLE EXPERIMENTS.....	71
6.1. ECHO.....	71
6.1.1. <i>Short description</i>	71
6.1.2. <i>Detailed code explanation</i>	72
6.2. NUCLEOTIDE SEQUENCE.....	72
6.2.1. <i>Short description</i>	72
6.2.2. <i>Detailed code explanation</i>	72
6.3. DATA ACCESS.....	73
6.3.1. <i>Short description</i>	73
6.3.2. <i>Detailed code explanation</i>	74
6.4. ALIGNMENT.....	74
6.4.1. <i>Short description</i>	74
6.4.2. <i>Detailed code explanation</i>	75
6.5. LCG TESTBED TEST EXPERIMENT.....	75

6.5.1. Short description.....	76
6.5.2. Detailed code explanation.....	76
7. DATA ACCESS SERVICES PROTOTYPE MANUAL.....	78
7.1. INTRODUCTION.....	78
7.1.1. References and Source Code.....	78
7.2. PROTOTYPE USAGE.....	78
7.2.1. Running the Prototype.....	79
7.2.2. Basic Operations.....	81
7.2.3. Advanced Features.....	84
7.2.4. Known Problems.....	85
7.3. INTERFACE REFERENCE GUIDE.....	86
7.4. TROUBLESHOOTING Q&A.....	89
7.5. IMPLEMENTATION STRUCTURE.....	90
7.5.1. Product Use Cases.....	90
7.5.2. Product Component Model.....	92
7.5.3. Detailed Implementation Model.....	93
7.6. PRODUCT TESTING.....	96
7.7. CONTACT INFORMATION AND CREDITS.....	97
ABBREVIATIONS.....	99
REFERENCES.....	102

List of Figures

FIGURE -1: THE GENERIC, SIMPLEST VERSION OF THE EXPERIMENT PIPELINE.....	12
FIGURE -2: DESIGN AND USE OF EXPERIMENT WITH SUBSTRATES AND PRODUCTS.....	13
FIGURE -3: LEVELS OF ABSTRACT GRID OBJECT DESCRIPTION.....	16
FIGURE -4: EXPERIMENT PLAN BEING CONCEIVED, SHARED AND DEVELOPED.....	17
FIGURE -5: THE PROCESS OF RELEASING AND USING EXPERIMENT PLAN.....	18
FIGURE -6: SELECTING THE WINDOW -> PREFERENCES MENU OPTION.....	21
FIGURE -7: THE INSTALLED INTERPRETERS PROPERTIES PAGE.....	22
FIGURE -8: ADDING NEW INTERPRETER THROUGH THE ADD INTERPRETER DIALOG.....	23
FIGURE -9: SETTING THE NEWLY ADDED INTERPRETER AS DEFAULT FOR EVERY NEW PROJECTS.....	24
FIGURE -10: SELECTING THE RUN -> RUN... MENU OPTION.....	25
FIGURE -11: CHANGING THE INTERPRETER THAT IS USED FOR RUNNING THIS PARTICULAR EXPERIMENT.....	26
FIGURE -12: ECLIPSE RICH CLIENT PLATFORM (RCP)	28
FIGURE -13: EPE WELCOME SCREEN	28
FIGURE -14: NEW EXPERIMENT WIZARD	29
FIGURE -15: SHARE AN EXPERIMENT WIZARD – SELECT THE REPOSITORY LOCATION PAGES (LEFT – SELECTING THE REPOSITORY LOCATION PAGE, RIGHT – CREATING A NEW REPOSITORY LOCATION PAGE)	33
FIGURE -16: CHANGING THE LABEL OF THE EXPERIMENT PAGE AND ADDING A REVISION COMMENT PAGE (LEFT – CHOOSING AN EXPERIMENT LABEL, RIGHT – ADDING A COMMENT TO THE EXPERIMENT REVISION).....	34
FIGURE -17: SELECTING THE RESOURCES PAGE	35
FIGURE -18: “IMPORT AN EXPERIMENT” WIZARD	36
FIGURE -19: EXPERIMENT CHOOSER	37
FIGURE -20: RENAMING AN EXPERIMENT BEFORE DOWNLOADING	38
FIGURE -21: CHANGING LOCATION OF THE EXPERIMENT PROJECT	38
FIGURE -22: RELEASE A VERSION OF AN EXPERIMENT	39
FIGURE -23: EPE UPDATE SITES MANAGER WINDOW.....	41
FIGURE -24: OPENING PROPERTIES PAGES.....	42
FIGURE -25: VIRTUAL ORGANIZATION PROPERTIES PAGE.....	43

FIGURE -26: OPENING PROPERTIES PAGES.....	44
FIGURE -27: GRID RESOURCES REGISTRY PROPERTIES PAGE.....	44
FIGURE -28: USER DEFINED GRID RESOURCES REGISTRIES ENTRIES.....	45
FIGURE -29: OPENING AVAILABLE VIEWS BROWSER.....	46
FIGURE -30: OPENING GRID RESOURCES REGISTRY BROWER.....	46
FIGURE -31: BROWSING GRID RESOURCES REGISTRY.....	47
FIGURE -32: GRID RESOURCES REGISTRY BROWSER POP-UP.....	48
FIGURE -33: GRID RESOURCES REGISTRY BROWSER CONTEXT MENU.....	48
FIGURE -34: INSERTING CODE LINE TO EPE EXPERIMENT EDITOR.....	49
FIGURE -35: CODE LINE INSERTED TO EPE EXPERIMENT EDITOR.....	50
FIGURE -36: SHOW RESOURCE IN ONTOLOGY BROWSER.....	51
FIGURE -37: SHOW RESOURCE IN ONTOLOGY BROWSER RESULT.....	51
FIGURE -38: SEMANTIC SEARCH.....	52
FIGURE -39: SEMANTIC SEARCH CONTEXT MENU.....	52
FIGURE -40: SHOW SEMANTIC SEARCH RESULT IN GRID RESOURCES REGISTRY BROWSER.	53
FIGURE -41: THE MAIN ONTOLOGY BROWSER PANEL.....	54
FIGURE -42: THE SEARCH RESULT VIEW THAT APPEARS AFTER SUCCESSFUL GRID OPERATION SEARCH ACTION.....	57
FIGURE -43 WEB RESOURCES BROWSER.....	59
FIGURE -44: SELECTING THE TYPE OF RULE SETS.....	83
FIGURE -45: RECEIVED NOTIFICATION MESSAGE.....	84
FIGURE -46: SAVE THE NEWLY AVAILABLE RULE SETS.....	84
FIGURE -47: THE MAIN ‘DISTRIBUTED DATABASE BROWSER’ WINDOW.....	87
FIGURE -48: POP-UP WINDOW FOR ADDING A NEW DATA SERVICE INSTANCE.....	88
FIGURE -49: POP-UP WINDOW FOR UPDATING A SELECTED DATA SET.....	89
FIGURE -50: GENERAL ARCHITECTURAL OVERVIEW OF THE DAS.....	91
FIGURE -51: A TYPICAL USE CASE WITHIN THE VIROLAB SCENARIO.....	92
FIGURE -52: MAIN COMPONENTS OF DAS.....	92
FIGURE -53: USE CASE SHOWING A TYPICAL DATA ACCESS REQUEST AND CORRESPONDING INTERACTIONS WITH OGSA-DAI.....	94

FIGURE -54: CONTROL FLOW OF THE SPECIFIC REQUEST RULESETS METHOD.....	95
FIGURE -55: INTERNAL FLOW OF STORE APPLICATION DATA METHOD.....	96
FIGURE -56: VISUALIZATION OF DAS UNIT TEST CASES.....	97

COPYRIGHT NOTICE

Software described in sections 2-6:

Copyright (c) 2007 by **Academic Computer Centre CYFRONET AGH**. All rights reserved.

Any use of the products described in Sections 2-4 are subject to the terms stated in the GPL license agreement: <http://opensource.org/licenses/gpl-license.php>.

Software described in section 7:

Copyright (c) 2007 by **University of Stuttgart**. All rights reserved.

Use of the product described in Section 7 is subject to the terms and licenses stated in the GPL license agreement. Please refer to <http://www.gnu.org/licenses/gpl.html> for details.

The DAS product in its current version makes use of two freely available software frameworks and external libraries in its operations. Namely it depends on the following software and libraries:

1. the **Globus Toolkit 4.0** provided by The University of Chicago
2. the **OGSA-DAI 2.2** Framework provided by The University of Edinburgh
3. the **Addressing** library provided by The Apache WS_Addressng Foundation
4. the **Axis** libraries provided by The Apache Software Foundation
5. the **Commons** libraries provided by The Apache Software Foundation
6. the **Jaxrpc** library provided by The Apache Software Foundation
7. the **Log4j** library provided by The Apache Software Foundation
8. the **SAAJ** library provided by The Apache Software Foundation
9. the **Xalan** library provided by The Apache Xalan Foundation
10. the **XercesImpl** library provided by The Apache Xerces Foundation
11. the **Xml-apis** library provided by The Apache Software Foundation
12. the **Xmlsec** library provided by The Apache Software Foundation
13. the **Xerces** xml parsing libraries provided by W3C
14. the **Wsd4j** library provided by The Apache Software Foundation
15. the **JUnit** framework provided by JUnit.org

All the above software is provided for use free of charge on the basis of open source licenses, either the Apache Public License (1), OGSA-DAI Project Licence (2), Apache License (libraries 3 to 12), W3C Copyright Notice and License (library 13) or Common Public License (libraries 14 and 15).

Globus is a trademark held by the University of Chicago. All rights reserved.

OGSA-DAI is a trademark held by University of Edinburgh. All rights reserved.

Axis, log4j, wsdl4j are registered trademarks of The Apache Software Foundation. All rights reserved.

This research is partly funded by the European Commission IST-2005-027446 Project "ViroLab".

1. INTRODUCTION

This document contains a set of manuals and tutorials for a person that would like to design new scientific experiments for the ViroLab Virtual Laboratory. The sections inside contain instructions how to obtain, install, configure and use various tools provided for an experiment developer. The tools include the Experiment Planning Environment that combine the versatile Eclipse Rich Client Platform with richness of the JRuby programming language to provide an easy yet powerful programming environment. It is further augmented by a set of plug-ins dedicated specifically for virtual laboratory experiment planning and development.

The other important part of the manual is the runtime library reference that discusses the available APIs of various components of the laboratory runtime system (called GridSpace Engine). In order to extensively demonstrate the application of these libraries in experiment development a set of example experiments is described, including code listings and comments.

1.1. TARGET AUDIENCE

The intended audience of this document includes any person that works as a kind of (scientific) programmer in an institute devoted to infectious disease research. The function of an experiment developer requires good knowledge basic of programming techniques (mainly procedural programming with some fundamentals of object-oriented techniques) and understanding of the idea of interpreted scripting languages. It also assumes the future developer is familiar with the notion of source code repository, collaborative development and shared resources. While the main programming platform used is the Ruby interpreter, the knowledge of the syntax and semantics of this particular language is not required to understand this manual. In fact, the authors claim that a programmer that never used this language will have little trouble adapting to it due to its simple and straightforward nature.

1.2. MORE INFORMATION

This document is not the only source of information for future experiment developers. The ViroLab Virtual Laboratory web pages provide the most recent and frequently updated versions of the enclosed tutorials. Please check:

<http://virolab.cyfronet.pl>

for a thorough, complete introduction to Virtual Laboratory and its mechanisms, tools, runtime etc. In the upper right corner of the page you will find set of hyperlinks to development sites, where you may:

- obtain the latest releases of the virtual laboratory modules
- read about the development plans and future release time schedule
- report a bug or a feature request, discuss it and monitor its lifetime

The authors of this manual and the software it describes would like to ask for the assistance of all the developers that would like to use the virtual laboratory. Please don't hesitate to use the bug submission and feature request mechanism in the virtual laboratory development web pages to suggest the authors how to

refine the software. With this process the tools we provide will be more useful and productive for the future experiment developers.

2. EXPERIMENT DEVELOPMENT ENVIRONMENT DESCRIPTION

2.1. EXPERIMENT PIPELINE IDEA

The central idea behind any virtual laboratory is an *in-silico* experiment.

Experiment is a process that combines together data with a set of activities that act on that data in order to yield experiment results. The substrate data required for an experiment may be obtained from multiple resources in various possible forms. The activities may be manual, semi-manual or fully automatic, depending on their nature. No definite restrictions are imposed on the level of complexity of such an experiment: it might be as simple as listing data inside some remote database, or much more complex, such as a set of simulators combined together to obtain some insight into complex phenomena. Moreover, the experiments are not required to involve just a single, local machine – in fact, the power of the virtual laboratory comes from combining multiple distributed resources, dispersed over various geographical locations.

The purpose of the laboratory we present is to support collaborative work of all the people who are effectively involved in any stage of the experimentation process. For our purpose, we refer to this process as the experiment pipeline. Below is a section that explains our view of the subject.



Figure -1: The generic, simplest version of the experiment pipeline.

Figure -1 presents the simplest picture of the experiment pipeline. First of all, one has to decide what an experiment is about and how it should proceed. The part of the pipeline concerning design of the future experiment process is called experiment planning. During that phase the user has to decide upon the main subject of the planned experiment, its intended results and the means by which these results should be obtained. Such a detailed description of the experiment results in experiment execution (the middle part of the process). During this phase the user performs the experiment according to the plan developed at the planning stage, using all the resources provided to that user within the virtual laboratory. The usual outcome of such execution is the result of the experiment. Since the result itself is of the highest importance for the user, special attention is given to it in the last phase of the pipeline, called result management. Here the outcome of the experiment may be evaluated, described and stored. Given the strong collaboration aspect of a virtual laboratory, the results can also be shared among the users of the laboratory. This stage concludes a single experiment pipeline run.

It should be stressed that no single part of this pipeline has to be a one-off activity. It is very probable, and in fact expected that various stages will be repeated in order to achieve the correct effect or desired quality.

2.2. DEFINED CLASSES OF USERS

As the virtual laboratory is meant to combine together people of various levels of expertise, we define two main classes of users who take part in the experiment pipeline. Their descriptions are provided here, while another class of users (sharing a common phase in the experiment pipeline) will be introduced later on.

Experiment developer is a person who designs experiments in a specific domain (like e.g. virology). First of all, this means that the person is skillful enough to design and denote the way an experiment should proceed. Apart from technical skills, the developer also possesses a certain level of domain-related knowledge to understand the nature of the processes the experiment should model – otherwise, the designed experiments will never be valid. However, it is usually not required for a developer to fully comprehend all the data and results of a specific experiment, provided there is appropriate expert (scientist) assistance available to evaluate the developer's work. This assistance provides initial requirements and descriptions of the desired experiment, and also feedback on the developed experiment. The developer uses dedicated tools to – among others – search for available data sources, combine them with suitable computational activities and present the results in an appropriate form. The aim of the virtual laboratory is to give experiment developers a set of powerful tools making their task easier, while at the same time not constraining their skill and creativity in any way.

Experiment user is any person who runs a previously prepared experiment in order to obtain results. The user may or may not be involved in the process of experiment preparation – in the latter case it is probable that (future) experiment users would support developers with their expert knowledge about the modeled phenomenon. The main objective of the experiment user is to obtain valuable results that answer important scientific questions.

2.3. MORE DETAILED VIEW

In Figure -2 one may find the part of the previously described process concerning substrate/product information: what a specific phase requires and what it yields in the process.

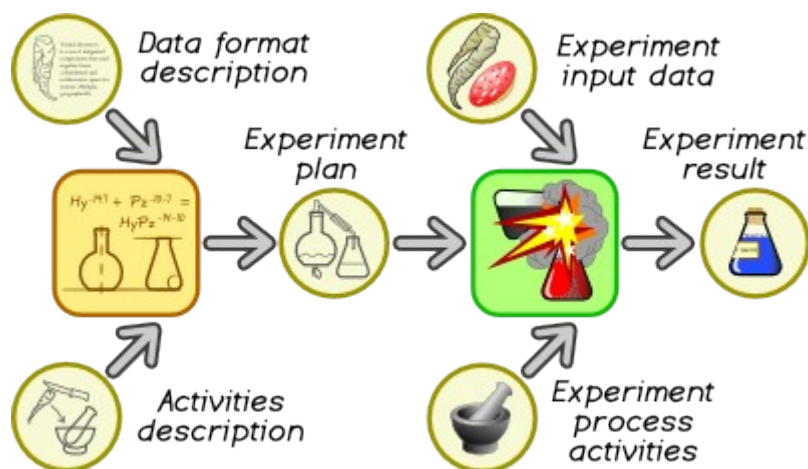


Figure -2: Design and use of experiment with substrates and products.

The person who develops the experiment requires some information on the resources needed to perform the future experiment. According to the definition, the experiment can involve both remote data sources and activities that need to be performed on the data in order to achieve the final result. Let us consider a simple example of an experiment that is meant to classify a set of patients into some illness classes. Here, the initial information we have at the beginning of the experiment pipeline is as follows:

- *Main idea*: given a set of patient data, assigning the most likely illness (from a set) to each patient
- *Input data*: we require certain reachable information on each patient (e.g. some measurements recorded on a hospital database)
- *Activities*: we also need a classifier resource, able to understand patient data and assign an illness class on the basis of this data (and, possibly, some previous training of the internal classification model)

As is depicted in Figure -2, the developer has to be aware of those requirements in order to design a valid experiment. However, the person ultimately needs the description of the data (e.g. its format, language, amount of data recorded) rather than the data itself - this is a very important distinction which we have to stress. For instance, in the example presented before, it can make a real difference, since patient data stored at a hospital is usually very confidential. However, provided the developer needs only the format of the data (and, perhaps, some toy example for testing purposes), it is possible that such an experiment could be developed by a person not even employed by the hospital. Given enough information regarding the data to be processed and the processing itself, the developer is capable of creating an experiment plan.

Experiment plan is a recipe that describes the process of certain experiment execution in the environment of the virtual laboratory. Physically, it is a set of files that, combined together, contain enough information for experiment users and their tools to proceed with (hopefully successful) experiment execution. The division of the plan into distinct parts is meant mainly to keep it well structured and easier to maintain by experiment developers. An important aspect of the virtualization part inside the laboratory is the fact that experiment plans are subject to storing, versioning, exchange, collaborative design and use etc. Experiment plans are the central focus of the virtual experimental space of the laboratory.

Once prepared, our experiment plan is ready to perform real classification (in this way we enter the second phase of the experiment pipeline). Now we need access to real data (the database of patients) and the classification unit. Please note that since the experiment user (in this case, the medical doctor who requires aid in illness classification) is a different person than the developer, their respective privileges may differ. Since the experiment is executed on behalf of its user, it is now able to access vital data about the patients (which constitute the experiment input data part of Figure -2). Furthermore, the virtual laboratory that actually (in a technical sense) runs the experiment plan, connects to a remote classification server to make it perform the required analysis. Figure -2 refers to this step in general as experiment process - those are the main functional blocks that constitute the experiment.

In the end, the experiment, successfully completed, should be able to provide experiment results. In the case of our sample experiment the result would be a table of patient-to-illness associations. Now, according to the generic experiment pipeline, the user who obtained that result may manage it any way he or she likes.

Experiment result in its most general meaning covers anything that is produced in the course of experiment execution. It could consist of some textual information, a generated image or a movie, a URL link to further information—almost anything. While an experiment outcome may be difficult to quantify, such quantification is usually useful for easier management of those results. There are no strict rules here — common sense of developers and users should be applied to decide what forms a separate result (for instance: a single image, a table of illness classifications, a file with a database table snapshot...) Since the result of an experiment is usually of the highest importance for the experiment user, the virtual laboratory devotes special attention to the management and post-experiment activities which act upon results (such as sharing, describing, storing etc.).

2.4. THE EXPERIMENT PROCESS SCRIPT

Each experiment is a process and therefore the most important part of the experiment plan is its script.

Experiment script is a piece of program in a computer language (physically located in one or more files) that the virtual laboratory execution components are able to interpret. The script defines the main steps of experiment process and the control flow between those steps. It also indicates what data is consumed, transformed and produced during the experiment execution. In case of ViroLab virtual laboratory, the JRuby programming language is used and augmented with a set of functions that are crucial for our approach. For exact list of features and further details please consult the [API](#) reference in later sections.

As you can see, the virtual laboratory uses a well-known scripting platform which immediately provides experiment developers with a considerable set of features and libraries. Hence, the first point to note is the possibility to use almost any kind of JRuby code within experiments. Apart from control flow statements (such as **ifs**, **fors**, assignments etc.) that every script supports, the three basic building blocks of every experiment are data, activities and results.

Data is the main substance that an experiment processes. There can be multiple types of data sources in an experiment script: local variables and constants, local or remote file systems, physical or virtual databases. In all of these cases data is imported into the experiment script execution space and may be analyzed, transformed and saved. Detailed information on ViroLab-specific means of acquiring data is contained in Section 5.2.

Activity is any kind of act that changes the internal state of the executed experiment script. Since this state is frequently defined by data, activities most of the time act upon experiment data (for analysis, transformation, printing, storage etc.) Basic (yet commonplace) activities are provided within the interpreter's standard library. However, more complex tools and services are available remotely

as Grid Objects. The reference on how to access such Grid Objects from the experiment script is in Section 5.3.

Results were already discussed in the previous section.

Samples of various experiment scripts may be found in Section 6.

2.5. GRID OBJECTS ABSTRACTIONS

Let us start the explanation of the idea of abstraction levels for computation description with the definition of Grid Object.

Grid Object is by definition any entity that is accessed remotely from the experiment execution environment that provides computation abilities for the experiment (to distinguish it from data sources). Examples of such objects: a service that transforms one scale of temperature to another, a data classifier component that assigns classes to data records according to their features, a dedicated cellular automata that simulates a given phenomena and so on. As one may see, there is uncountable amount of possibilities and in order to bring more order to the matter, we introduced different Grid Object abstractions.

2.5.1. Explanation of abstraction levels

Figure -3 shows in a simple way different levels of Grid Object abstractions and what information is associated with these levels. First of all we have the purely conceptual level of **Grid Object Class**. The class is used to group all the Grid Objects that have similar functionality with regard to their domain operations - this is also called *functional similarity*. By definition every Grid Object that belongs to a certain class exposes exactly the same program interface - and the declaration of such an interface is what really defines a given class.

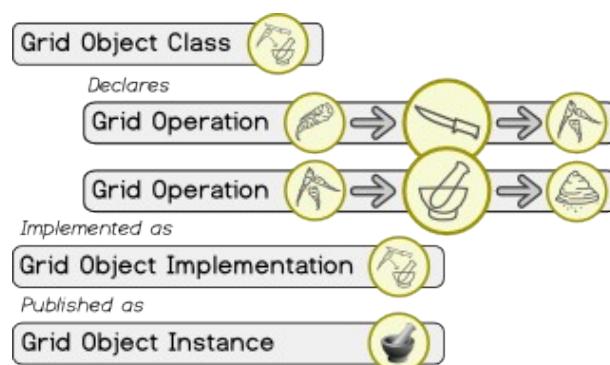


Figure -3: Levels of abstract Grid Object description.

So, within its interface every class defines a set of one or more **Grid Operations** (sometimes referred to as Grid Object Operations). Each operation describes what type of arguments (or input parameters) it requires and what kind of products (output results) it yields upon successful completion. Such a description of functionality is frequently called an *operation signature*.

Every Grid Object Class may be realized in a couple of different ways that are called **Grid Object Implementations**. Each implementation, while perhaps using different technological means, provides exactly the same functional behavior to the external world. In terms of programming, we say that every implementation of a given class *implements* or *realizes* the set of Grid Operations declared by that class. This is analogous to similar idea from object-oriented programming languages.

Finally, an implementation of a defined class is not sufficient. In order to have a real Grid Object ready to be invoked, we need a **Grid Object Instance**. An instance is an implementation that was executed and published so the Grid Object is accessible from the network. Such an instance, being a running and operational remote Grid Object, could be contacted by any experiment runtime and any Grid Operation that the implemented class declares may now be executed on behalf of some experiment. Of course, there may be multiple instances of the same implementation - then the choice of the best one is performed by the Grid Application Optimizer (GrAppO).

2.5.2. An example

As an example let us consider a *temperature service* that retrieves current air temperature in designated area (through a weather station's webpage report). We'll set up a Grid Object Class called *currentAirTemperature* - the class will declare two Grid Operations that we have designed, called *currentAirTemp* and *kelvinToCelcius*.

Now, we have decided to implement the tool as a Web Service, so name our new Grid Object Implementation *temperatureJavaWebService*. When later on we decide to implement this class in another technology (for instance as a Web Service in another framework, or as a MOCCA component) we will publish that implementation with a different name.

Finally, no doubt we have published our implementation as a physical, remotely accessible Web Service endpoint using some available web server. This concrete endpoint that could be called from outside is ours first Grid Object Instance. Since we are able to deploy the same implementation in several different location, there could be more than one instance.

More information on how to browse the current set of available Grid Objects and how to publish your own Grid Object to the ViroLab community, please see Section 4.

2.6. FUNCTION OF THE EXPERIMENT REPOSITORY

The purpose of the Experiment Repository within the ViroLab Virtual Laboratory is to store and provide experiment plans that are developed by experiment developers and that are used by experiment users. From this perspective the repository plays a meeting place for these two groups of users of the laboratory - they share among themselves the experiment plans.

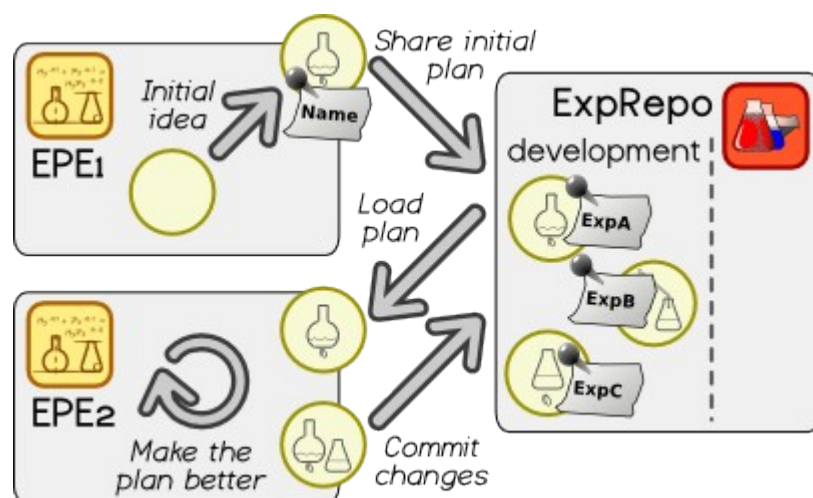


Figure -4: Experiment plan being conceived, shared and developed.

Usually, a typical life cycle of an experiment plan involves:

- initial conception and establishment of a new experiment plan
- shared, cooperative development of the plan by developers
- first release of the plan
- first uses (executions) of the plan by users
- gathering feedback and identifying shortcomings
- further refinement, next version releases
- and so on and so forth...

The Figure -4 shows us the first stages of this process. One developer gets a new idea about some exciting experiment that could be developed. She puts those initial thoughts in form of a sketch of an experiment script and shares that with fellow developers through the Experiment Repository. In terms of software engineering this activity of experiment sharing is referred to as initial import.

After that another experiment developer, interested in similar ideas, loads the experiment plan (in cooperative development this is usually called *checkout*) to his development environment and contributes his effort to make the experiment better. The new changes and additions are shared with the community through a commit operation that essentially synchronizes the content of the repository with the latest form of experiment developed by the given developer.

At this early stage of experiment planning, the plan in its current form is only stored in so-called development section of the repository. At the moment no user is able to obtain and execute this experiment - as it is assumed to be not mature enough.

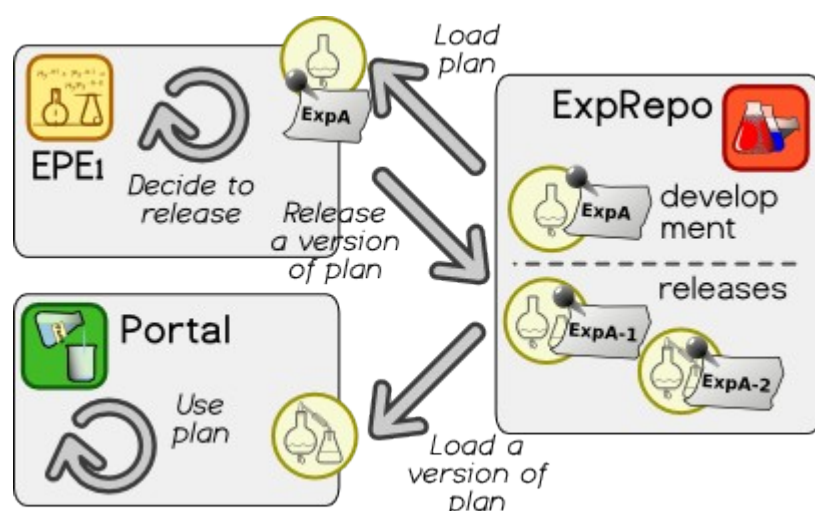


Figure -5: The process of releasing and using experiment plan.

After a certain amount of time and some effort put in the experiment plan development, one of its developers may decide to release a version of the experiment plan (see Figure -5). Such publication yields a copy of the current state of the plan in a dedicated space of the Experiment Repository. The two

important features of the releases space (in comparison to the development space) are: it is accessible by experiment users and the releases put there are frozen with respect to further changes.

The tool provided for the experiment user is now able to see the released versions of experiments and also, on request, is able to download a designated version of given experiment plan. The plan, using the functionality of the tool, could be now executed and the user may obtain results.

3. EXPERIMENT PLANNING ENVIRONMENT USER'S MANUAL

The ViroLab Experiment Planning Environment (EPE) is a tool for managing experiment development process. Beside the ViroLab Portal, EPE is the main component of the ViroLab presentation layer.

The main idea is to provide experiment developers with powerful Virtual Laboratory experiment editor for creating experiment plans in an easy way. Beside the most important scripting capabilities, EPE offers :

- storing experiments locally using workspaces mechanism,
- sharing experiments with other experiment developers using the Experiment Repository,
- releasing a new version of an experiment so the scientists may use it with their tools,
- executing experiment plans using GridSpace Engine.

However ViroLab EPE is not limited to these functionalities. Due to the fact that it is based on the Eclipse Rich Client Platform (RCP) ViroLab EPE can be extended by wrapping new functionality with an Eclipse plugin and plug it into EPE. For the description of a set of ViroLab-dedicated plugins supplied with this prototype, see Section 3.2.

3.1. EXPERIMENT PLANNING ENVIRONMENT

3.1.1. Installation and Configuration

This manual is written for the latest EPE versions: **0.2.4** and **0.2.4.x**.

Installation

Prerequisites

- Java Runtime JRE 1.5 (or higher), java executable needs to be in the PATH environment variable

Installation steps for Linux

- Download the archive containing ViroLabEPE release: ViroLabEPE-<version>.tar.gz from ViroLab EPE download page (http://gforge.cyfronet.pl/frs/?group_id=38&release_id=53)

- Extract the content of the archive:

```
tar zxvf ViroLabEPE-<version>.tar.gz
```

or if you downloaded the .zip version of the archive:

```
unzip ViroLabEPE-<version>.zip
```

Installation steps for Windows XP/Vista

- Download the archive containing ViroLabEPE release: ViroLabEPE-<version>.zip from ViroLab EPE download page (http://gforge.cyfronet.pl/frs/?group_id=38&release_id=53)

- Extract the content of the archive by clicking right mouse button on the archive, selecting "*Extract files...*" and following the wizard steps

Running

On either OS Linux or Windows, to run ViroLab EPE click on *virolabEPE* executable file, which is placed in ViroLab EPE root directory.

Configuration

This part contains step-by-step guide to set up GSEngine as the default experiment plan interpreter in EPE. For more information about EPE configuration options please visit ViroLab EPE User manual page (<http://virolab.cyfronet.pl/trac/vlwl/wiki/EpeManual>).

The following instructions describe how to set up GSEngine as the default experiment plan interpreter in EPE:

1. Open ViroLab EPE. Go to *Window -> Preferences*.

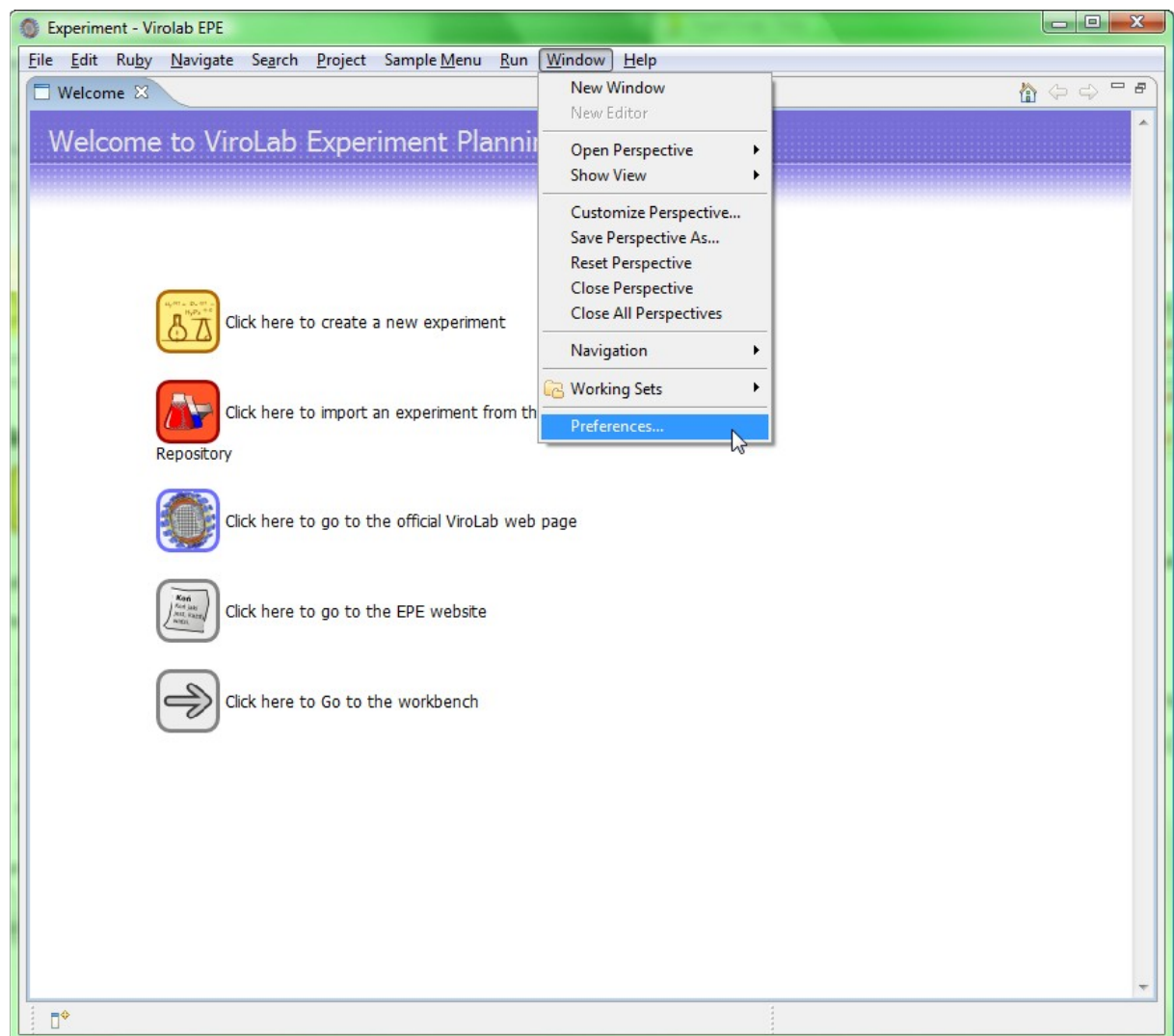


Figure -6: Selecting the *Window -> Preferences* menu option

2. Select *Experiment* -> *Installed Interpreters* preferences page.

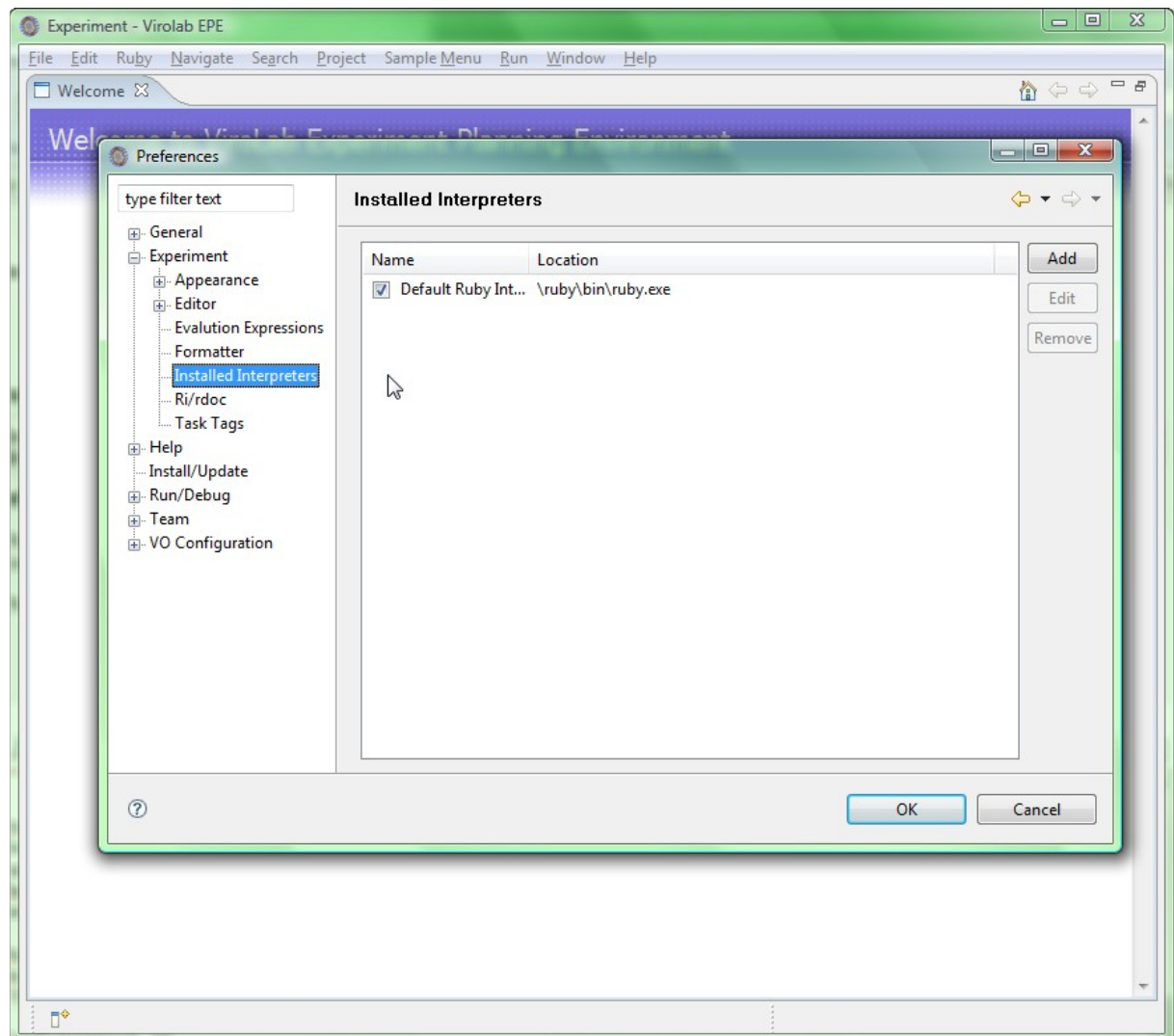


Figure -7: The *Installed Interpreters* properties page

3. Click "Add" button. Enter whatever you like for the "Interpreter Name" (e.g. GSEngine). For the "Location" enter a path to the gsengine executable (e.g. "C:\gsengine\gsengine.bat" on Windows or "/home/user/gengine/gengine" on Linux) or click "Browse..." to find the executable easily. Click "OK".

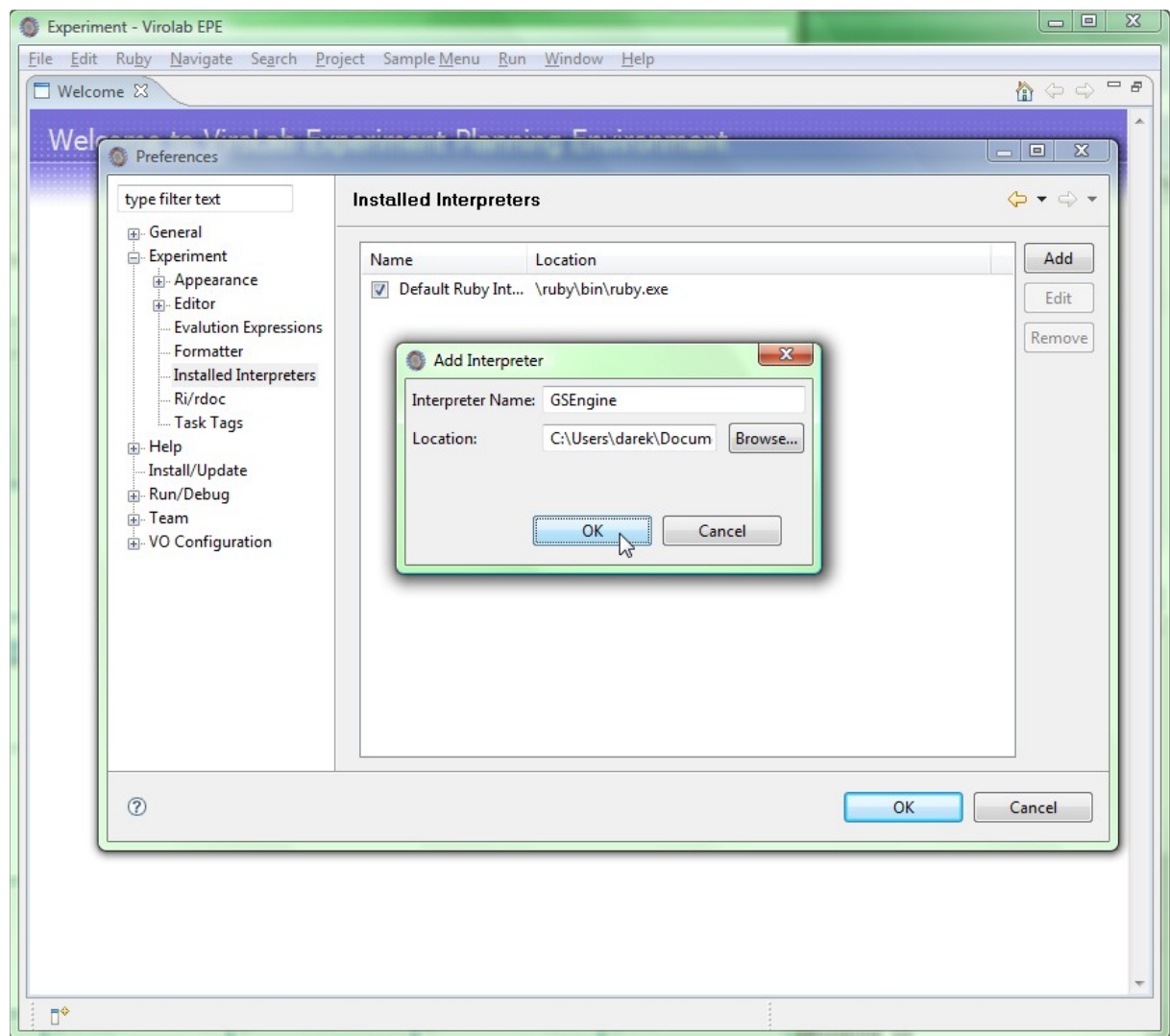


Figure -8: Adding new interpreter through the *Add interpreter* dialog

4. You should now see a new entry, which describes GSEngine as an experiment plan interpreter. To set it as the default experiment plan interpreter in EPE tick a checkbox, which is placed next to the interpreter name you have provided.

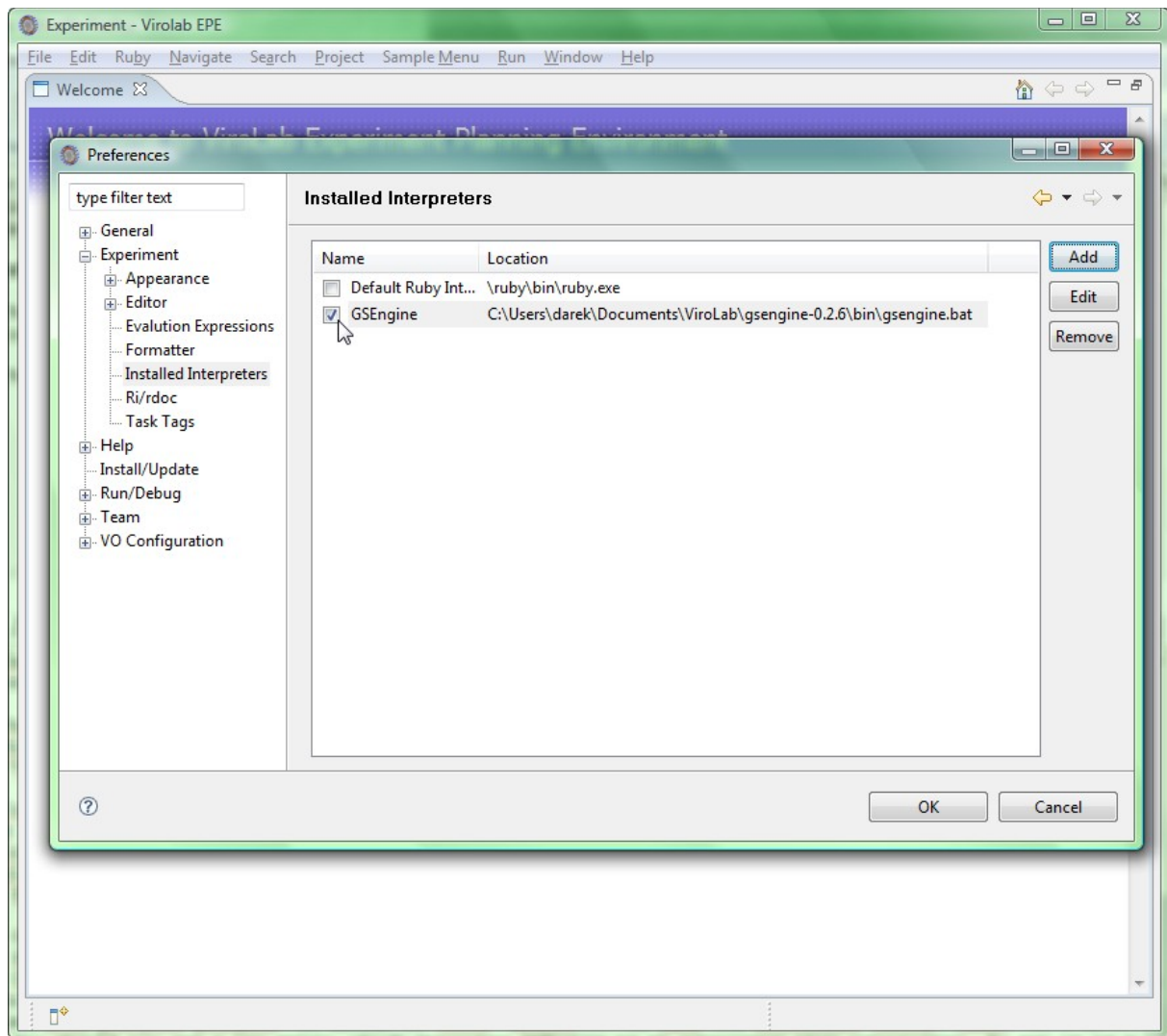


Figure -9: Setting the newly added interpreter as default for every new projects

Congratulations! Now you can use GSEngine to execute experiment plans.

In some cases (e.g. executing experiment plan using a different interpreter) it may be necessary to change interpreter manually. The following instructions describe how to do it :

1. Select *Run* -> *Run...* menu item.

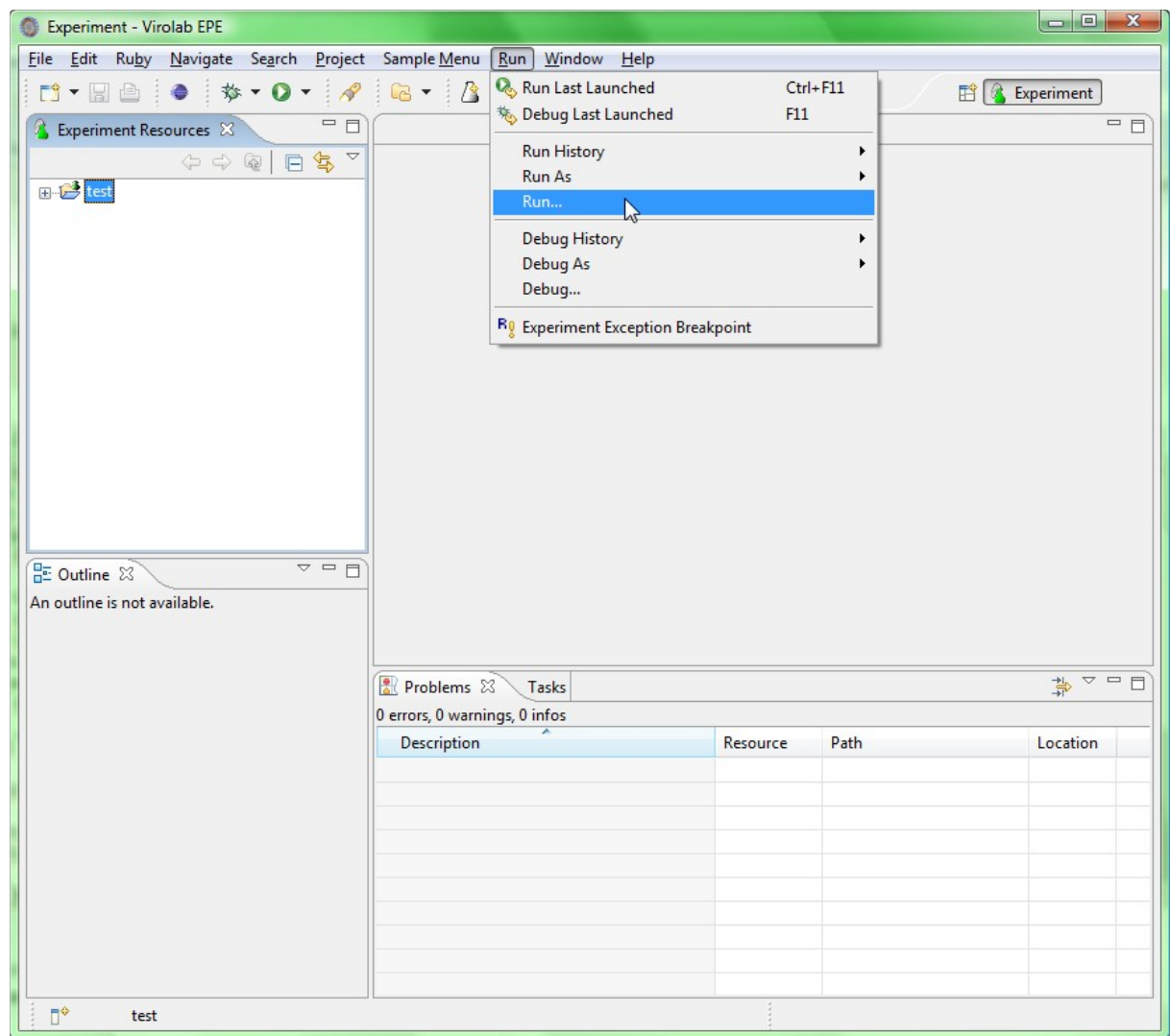


Figure -10: Selecting the *Run -> Run...* menu option

2. Choose experiment configuration, go to *Environment -> Interpreter* tab and select GSEngine.

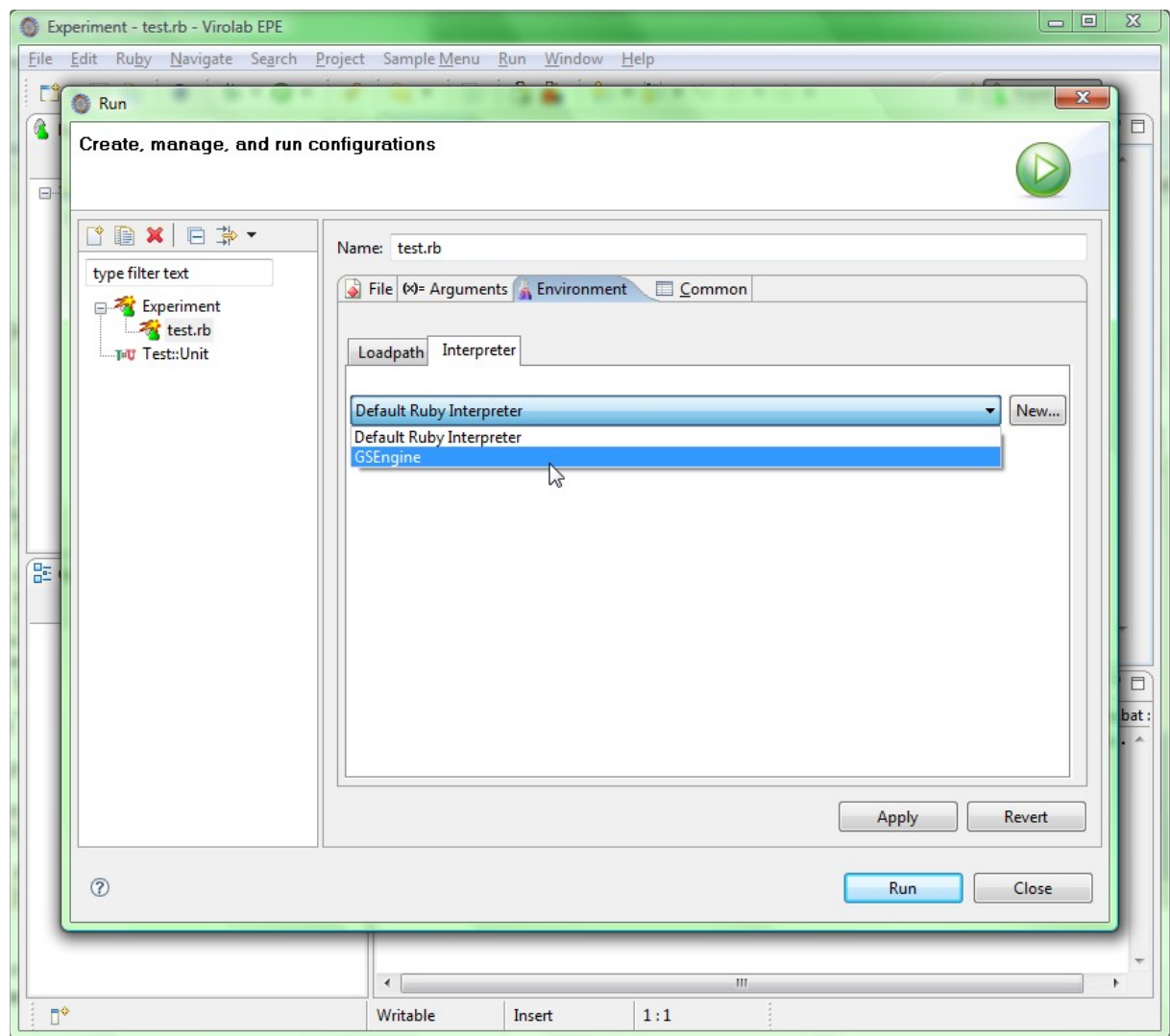


Figure -11: Changing the interpreter that is used for running this particular experiment

Comments

If you have any comments or critical remarks related to this manual please contact us. We would be grateful.

Dariusz Król dkrol AT student DOT agh DOT edu DOT pl

Piotr Pęgiel barca AT wp DOT pl

3.1.2.Usage

Introduction

This manual is based on the ViroLabEPE-0.2.4, which is available for both Linux and MS Windows OS. It is intended generally for everyone who participates in creating ViroLab applications using ViroLab EPE (e.g. experiment developers).

EPE is a tool that makes the development of experiments much easier. In order to achieve that, EPE provides support for most of the typical actions in an experiment development process, such as:

- creating new experiments with a proper structure
- sharing experiments using the Experiment Repository
- running experiment plans using a runtime system (e.g. GSEngine)
- releasing new versions of an experiment directly to the Experiment Repository
- providing relevant information about computational resources using the Grid Resource Registry plugin
- browsing ontologies using the Ontology Browser plugin.

Due to being based on the Eclipse RCP platform (for more information, please see the next section or visit RCP web page - <http://www.eclipse.org/home/categories/rcp.php>), EPE looks similar to the Eclipse IDE. Thus everyone, who is familiar with the Eclipse IDE, will have no problem with using EPE.

The following sections of the manual, firstly describe the Eclipse Rich Client Platform (RCP) in order to familiarize you with the EPE architecture background. Secondly, there will be shown how to perform the above-mentioned actions.

Eclipse Rich Client Platform (RCP)

The core of the [ViroLab](#) Experiment Planning Environment is based on Eclipse Rich Client Platform. It is a part of the Eclipse project created to simplify the process of developing a Rich Client Application. In short, the RCP is a subset of the Eclipse platform plugins (Figure 3-7), which enables developing Rich Client Applications.

This framework extends the plugin development process for Eclipse and allows developers to create a standalone application with the core Eclipse functionality. Therefore, after creating a plugin for Eclipse, there is no need to redesign it at a later stage. This plugin can be used in the RCP environment such as the EPE.

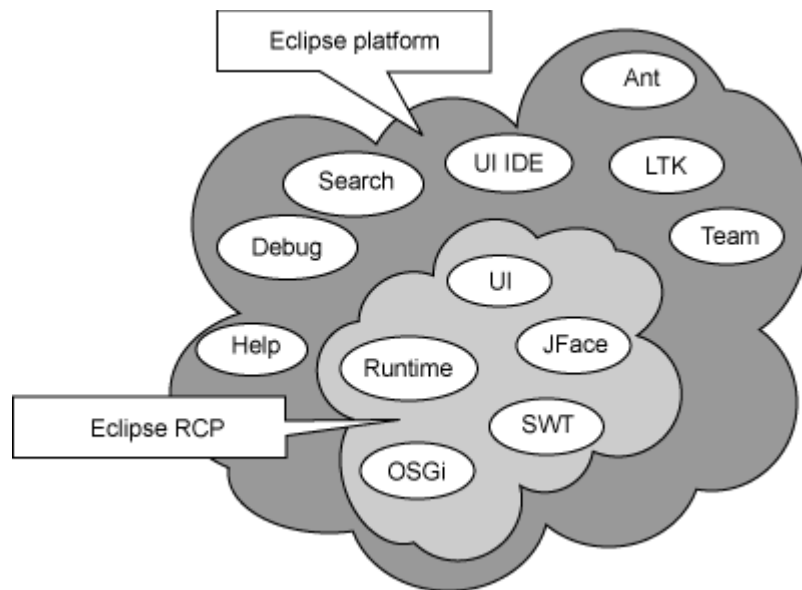


Figure -12: Eclipse Rich Client Platform (RCP)

The modular EPE architecture enables to extend the EPE by writing new plugins which provide new functionalities, and plug them into the EPE (for more information about writing plugins please check Section 3.2).

EPE main window

On having started EPE, the first screen you see is the Welcome screen, which is presented and described in Figure 3-8.

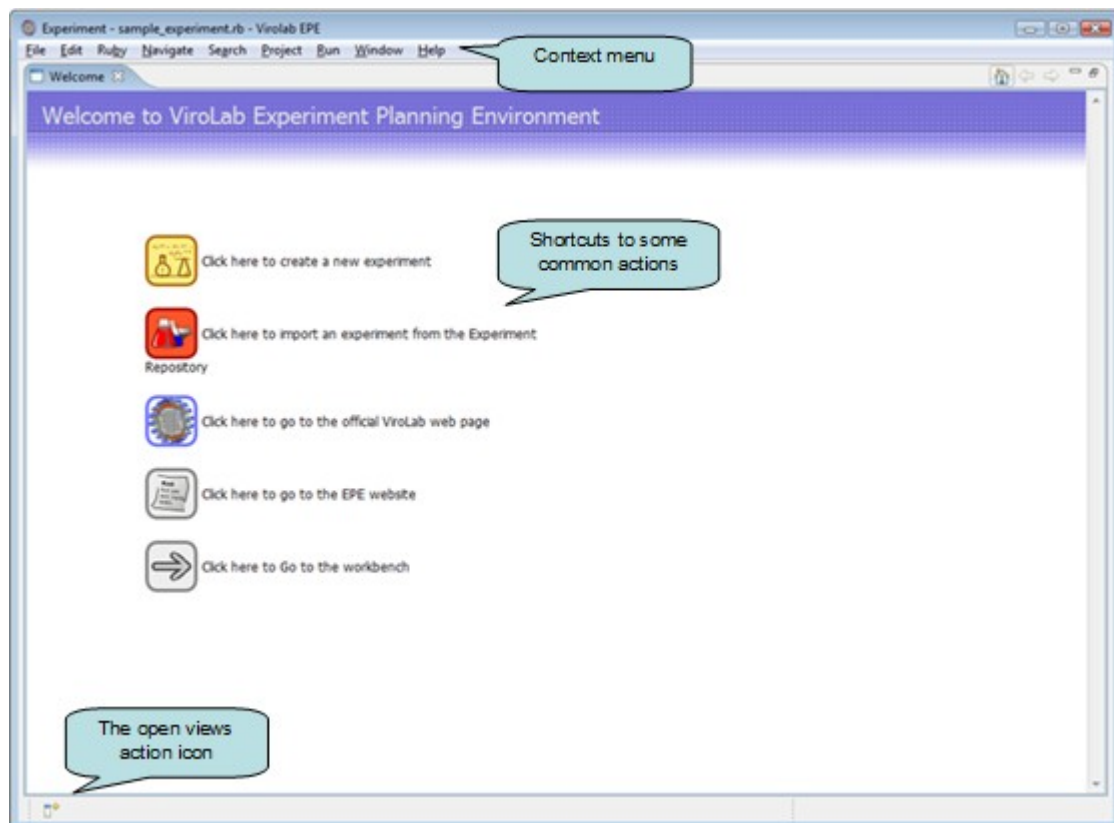


Figure -13: EPE welcome screen

The *Welcome screen* provides an easy way to start work with the EPE. By choosing different icons you can :

- create a new experiment
- import an experiment from the Experiment Repository
- go to either ViroLab or EPE web page
- go to the workbench.

Create the new experiment

Creating a new Experiment in ViroLab EPE is one of the most important functionality – it helps the user create a complete Experiment Project. This process is quite simple – it is implemented as a wizard. To open the wizard, you should select one of the following options:

- the first option (*"Click here to create a new experiment"*) from the Welcome screen
- the *"New experiment"* icon from the toolbar (the *Experiment perspective*)
- *File -> New -> New Experiment* option from menu

After choosing one of these ways, the new experiment wizard opens the window shown in Figure 3-9.

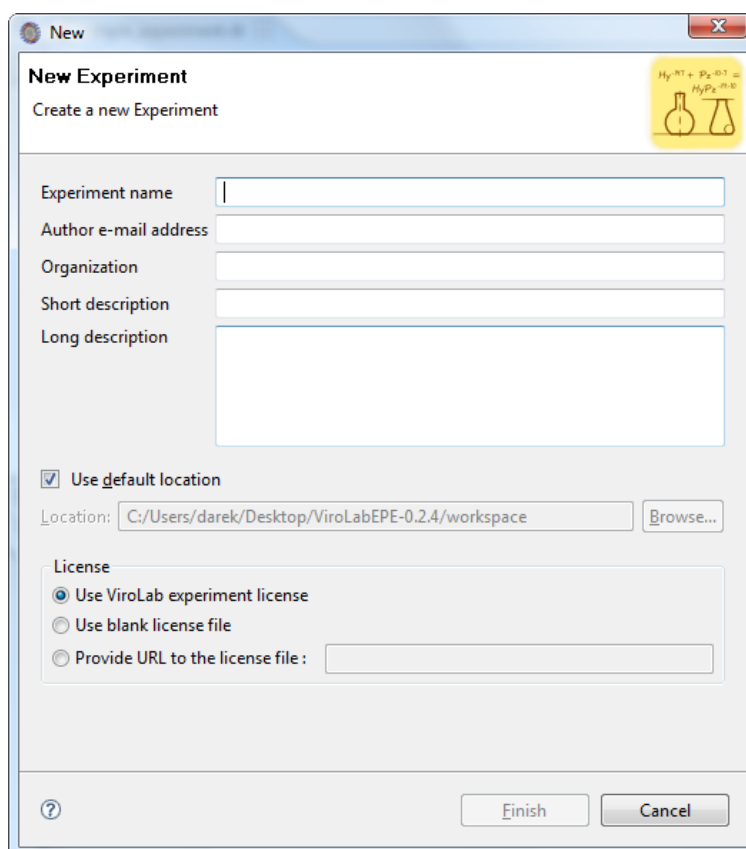


Figure -14: New experiment wizard

There are three information categories you have to provide in order to complete the wizard :

- information about the experiment, like an experiment name and description, author contact info and organization, they are important due to feedback and organizational matters
- path to the place where the experiment project will be stored
- information about the experiment project license. At the moment there are three options available:
 - default ViroLab experiment license – based on the MIT license
 - blank license – which should be filled by the experiment developer
 - URL to the license – no experiment license file is created.

After accepting provided information, by clicking on the Finish button, a new experiment project is created. In most cases, ViroLab Experiment will consist of (of course it may be extended in the future by new data):

- plan source files (GScript sources) – in the experiment src directory,
- license file – if created there should be `ExperimentLicense.txt` file in the experiment project main directory,
- an experiment meta-information file - `experiment.xml` – will contain information provided by the wizard (like the user who creates the experiment, a description of experiment),
- feedback file – contains information provided by ViroLab Portal from the experiment users.

Share local experiments

Storing an experiment to the repository is needed if the experiment is intended to be run by scientists (they simply could not use Experiment Planning Environment) or by other developers. The second reason is the situation when two or more developers are working simultaneously on the same Experiment. These are very common situations that occur in real life – when working, e.g., on Java, Ruby or C# project. Since many developers got used to version controlling systems (e.g. CVS), the ViroLab Experiment Repository will exploit one of the most known systems – SubVersion (SVN). It keeps track of all work over experiments and allows several (potentially widely separated) developers to collaborate.

To start the “Share an experiment” wizard, you should open a popup menu with mouse right-click on the experiment name and select Team -> Share experiment option. If you have connected to the Experiment Repository before, you would see a wizard page similar to the one presented in Figure 3-10 (the one at the top). There is a table with experiment repository locations that you are already connected to.

If this is the first time you connect to the Experiment Repository, you should be able to see a wizard page similar to the one presented in Figure 3-10 (the one at the bottom). You will see this page also, if you have connected to the Experiment Repository before but on the wizard page, which was described above, you select the "Create a new repository location" radio button.

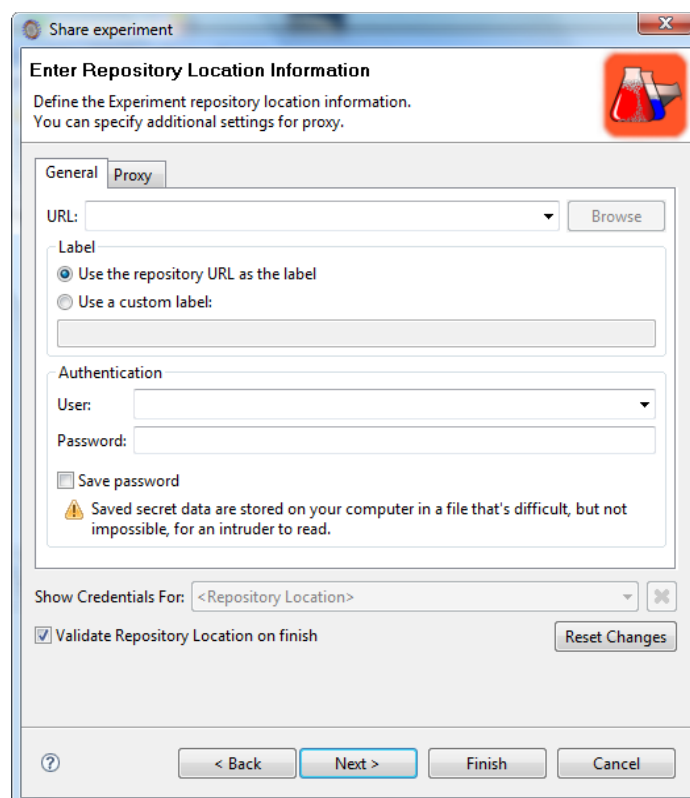
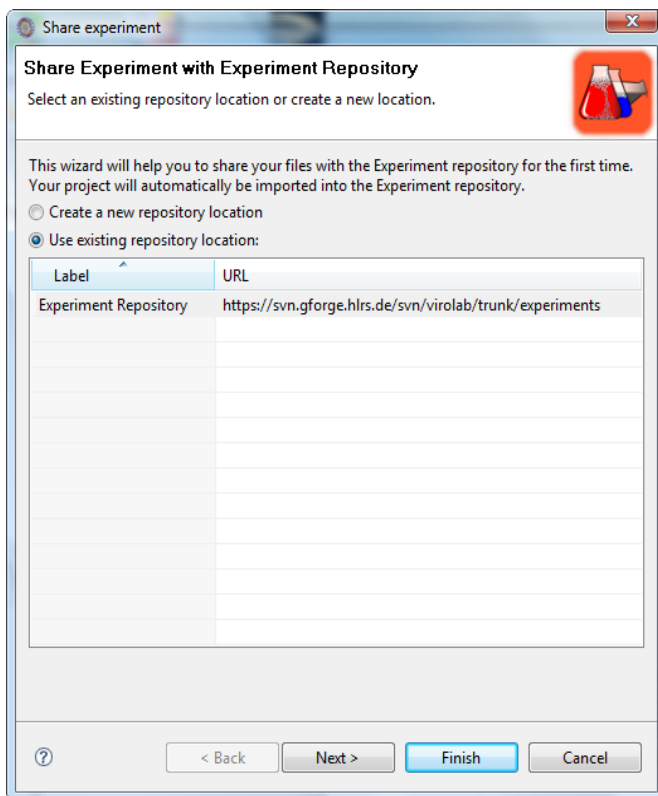


Figure -15: Share an experiment wizard – select the repository location pages (left – selecting the repository location page, right – creating a new repository location page)

In order to complete the wizard, the user has to provide information about a repository location (the default location has been added already to the list) and authentication info (a user name and a password are available at this time).

Optionally, you can provide label which will be used instead of a repository URL. There is also an option to store your authentication info locally, thus you do not need to provide it whenever you connect to the repository.

After choosing the repository location step you can either start sending an experiment to the repository, by clicking the "Finish" button, or go to the next page, by clicking the "Next" button.

There are two more pages to complete if you choose the "Next" button. The first one is about the label of the experiment, which will be used on the repository side (Figures 3-11, the one at the top). The second one is about comments you may want to add to the first revision of the experiment (Figures 3-11, the one at the bottom). At any time you may start sending an experiment to the repository, by clicking the "Finish" button or cancel the whole process by clicking the "Cancel" button.

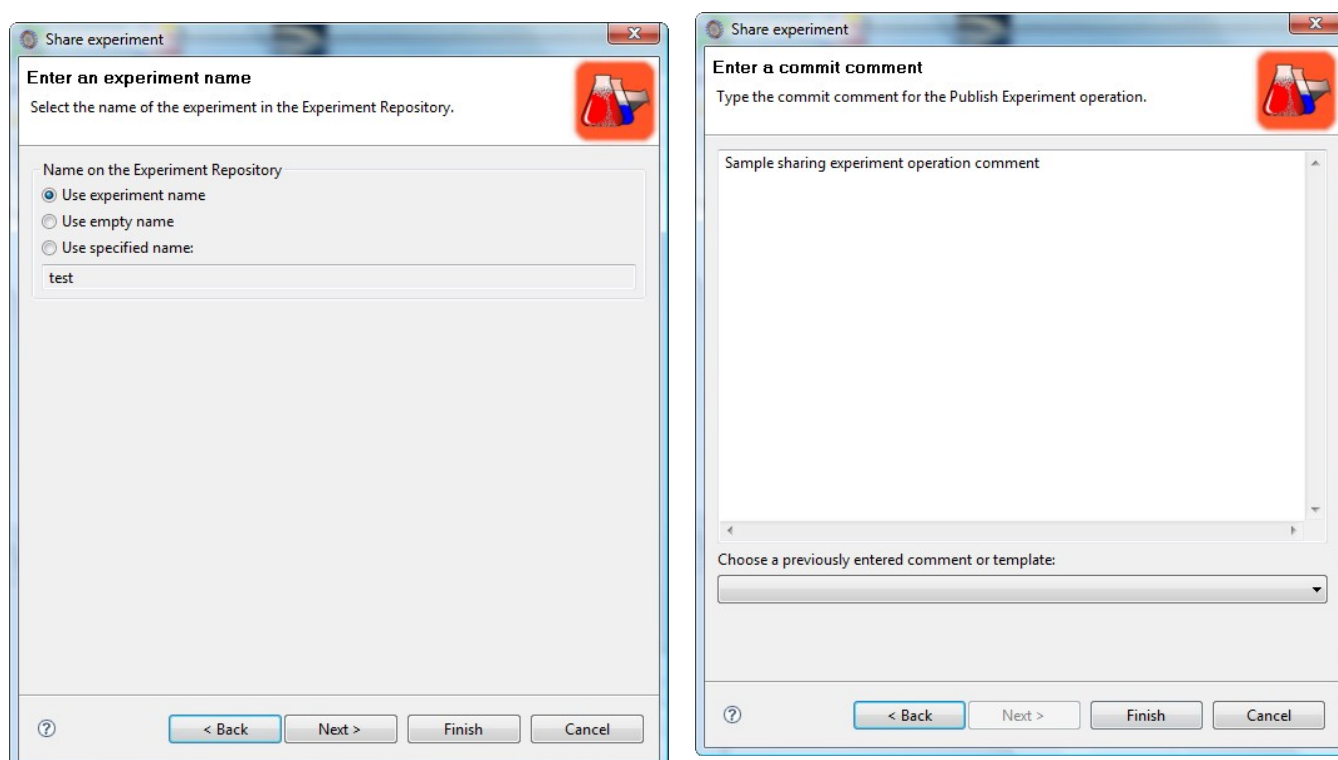


Figure -16: Changing the label of the experiment page and adding a revision comment page (left – choosing an experiment label, right – adding a comment to the experiment revision)

After clicking the "Finish" button eventually, the experiment sending process is about to begin but before it starts there is the last step you have to perform. In some situations you may not want to send the whole experiment project to the repository (e.g. some files are created locally during runtime and it does not make sense to share them). A simple dialog (presented in Figure 3-12) enables users to select the resources, which will actually be sent to the repository. By default, all files are selected.

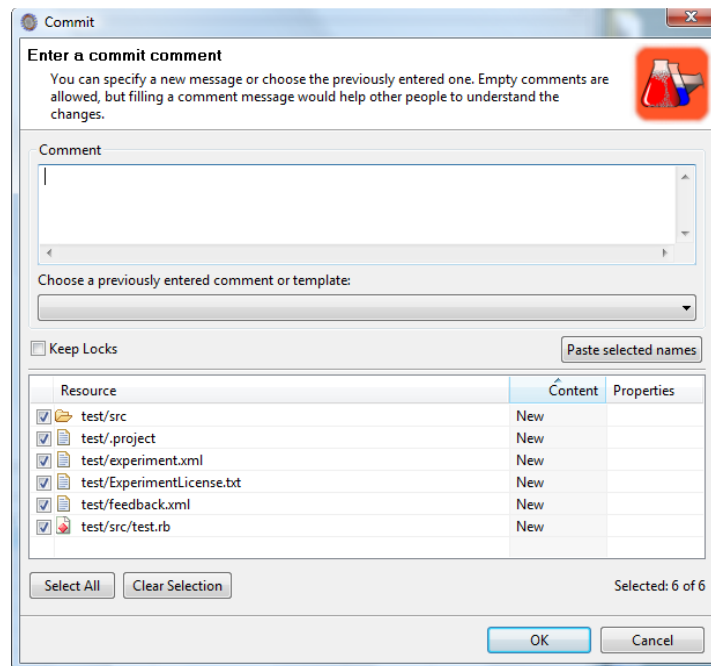


Figure -17: Selecting the resources page

After this step is completed, a proper structure of the experiment project will be created on the repository side and all selected resources will be placed on the repository. If any problems occur, you will be informed by an error dialog.

After sharing an experiment, you are able to perform some team operations on the experiment. They are accessible through the popup menu (mouse right-click on the experiment name) -> Team. The common team actions are:

- *Synchronize with Repository* – show differences between the local and the repository version of the experiment
- *Commit* – update the experiment with changes, which were made locally, on the repository side
- *Update* – retrieving all changes from the repository
- *New Release* – prepare a new release of the experiment (it will be described later)

Import an experiment from the Experiment Repository

If someone has stored some experiments to the Experiment Repository you can easily import them using ViroLab EPE. It is a common situation that occurs when e.g. a few developers work on the same experiment.

The "Import an experiment" operation is implemented as a wizard in order to make it as user friendly as possible. You can start it by choosing one of the following options :

- click the second position from the Welcome screen ("Click here to import an experiment from the Experiment Repository")

- click the “Import an experiment from the Experiment Repository” icon from the EPE toolbar
- File -> New -> Other (or mouse right-click and select “New -> Other” from the popup menu) and from the tree viewer select Experiment -> Import Experiment from Experiment Repository
- File -> Import (or mouse right-click and select “Import” from the popup menu), from the tree viewer select Team -> Import Experiment from Experiment Repository

After choosing one of these ways, the “import an experiment from the Experiment Repository” wizard opens. It is the same wizard you use when sharing an experiment (please see the previous section for details).

The screenshot shows a window titled "Checkout from Experiment Repository" with a close button in the top right. The main heading is "Enter Repository Location Information" with a subtext: "Define the Experiment repository location information. You can specify additional settings for proxy." There is a small icon of a flask with a red and blue liquid in the top right corner of the window.

The window has two tabs: "General" (selected) and "Proxy".

Under the "General" tab, there is a "URL:" label followed by a text input field and a "Browse" button. Below this is a "Label" section with two radio buttons: "Use the repository URL as the label" (selected) and "Use a custom label:" followed by a text input field. Below that is an "Authentication" section with "User:" and "Password:" labels, each followed by a text input field. There is a checkbox labeled "Save password" which is currently unchecked. Below the checkbox is a warning icon and text: "Saved secret data are stored on your computer in a file that's difficult, but not impossible, for an intruder to read." At the bottom of the form is a "Show Credentials For:" label followed by a dropdown menu showing "<Repository Location>" and a button with a cross icon. Below this is a checkbox labeled "Validate Repository Location on finish" which is checked, and a "Reset Changes" button. At the very bottom of the window are four buttons: a help button with a question mark, "< Back", "Next >" (highlighted in blue), "Finish", and "Cancel".

Figure -18: “Import an experiment” wizard

As you can see in Figure 3-13, in order to download an experiment from the Experiment Repository the user has to provide information about the repository location (the default location has been added already to the list) and authentication info (a user name and a password are available at this time). Optionally, you can provide a label which will be used instead of a repository URL. There is also an option to store your authentication info locally, thus you do not need to provide it whenever you connect to the repository. On having completed the wizard page, please click the “Next” button to connect to the Experiment Repository. If everything goes right you should see a page with an experiment chooser, similar to the one presented in Figure 3-14.

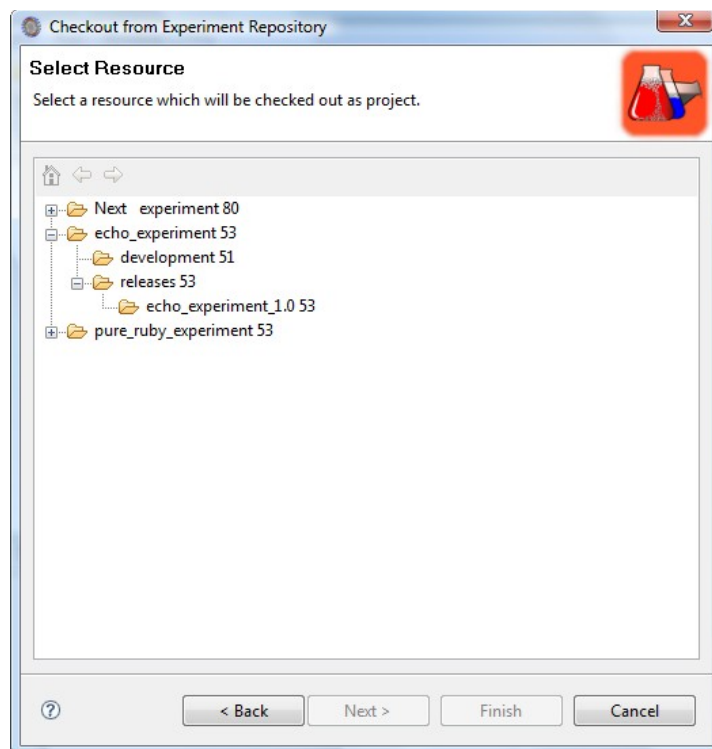


Figure -19: Experiment chooser

On this page you can select an experiment you want to download. As you may already know (if not, please visit the Experiment Repository web page - <http://virolab.cyfronet.pl/trac/vl vl/wiki/ExperimentRepository>), an experiment has two development stages: development and release and both of them are supported by the repository. By default, by clicking on an experiment name, the development branch is downloaded.

After having selected the experiment, click on the "Finish" button to accept your choice. The next screen you see should be similar to the one presented in Figure 3-15. You can change the name, which the experiment will be stored with.

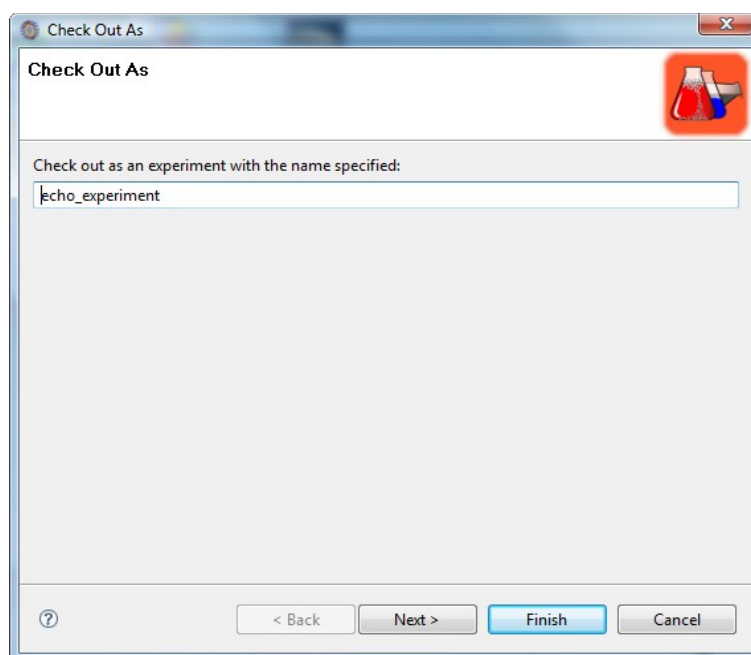


Figure -20: Renaming an experiment before downloading

After that you can, except of canceling the whole process, click either on the "Finish" button to download the experiment eventually or on the "Next" button to go to the next page (Figure 3-16). It is used to change the location of the place, where the experiment is actually stored. You can also select the working set for the experiment, which is a mechanism for grouping projects. This can be a very helpful feature, especially when you have many non-related experiments in the workspace. By using working sets you are able to organize them into a few categories without moving them to different workspaces.

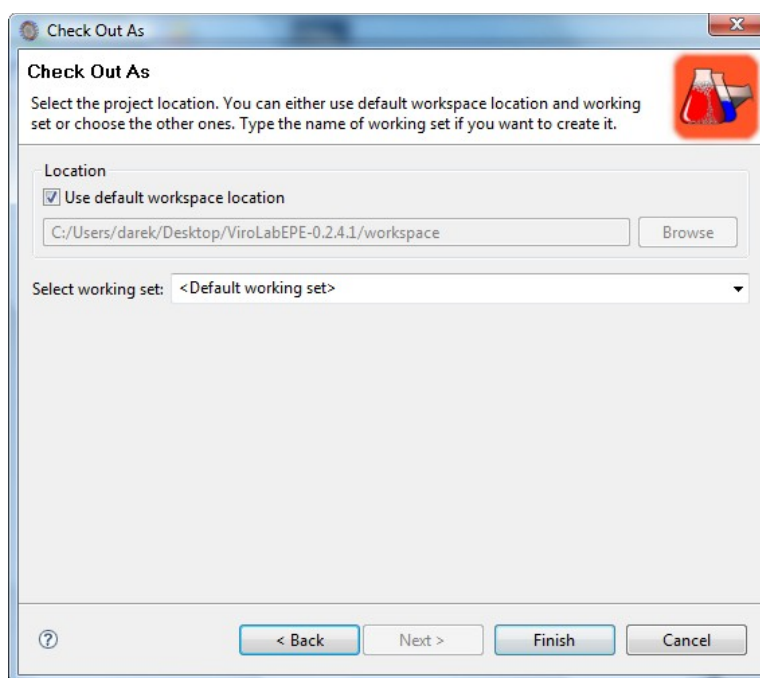


Figure -21: Changing location of the experiment project

After downloading an experiment you are able to perform team actions on the experiment. It is the same set of operations, which was described above in the “Share local experiments” section.

Release a new version of an experiment

As mentioned above, there are two stages of the experiment developing process: *development* and *release*. The *Development* branch of an experiment represents an unstable and untested experiment version, thus nobody should use it but developers. After a certain amount of time, it may, however, be decided that the experiment is stable and it is possible to create a release of the experiment. Releasing a version of the experiment is the only way to present the experiment in ViroLab portal.

The “release a version of the experiment” action is implemented as a wizard, like most of the common actions in EPE. The wizard is activated after selecting “Team -> New Release...” option from the experiment popup menu (mouse right-click on the experiment name). The wizard is very simple, the only thing you have to provide is a version number (or name) of the experiment (Figure 3-17). Optionally, you can add a release comment in the text area.

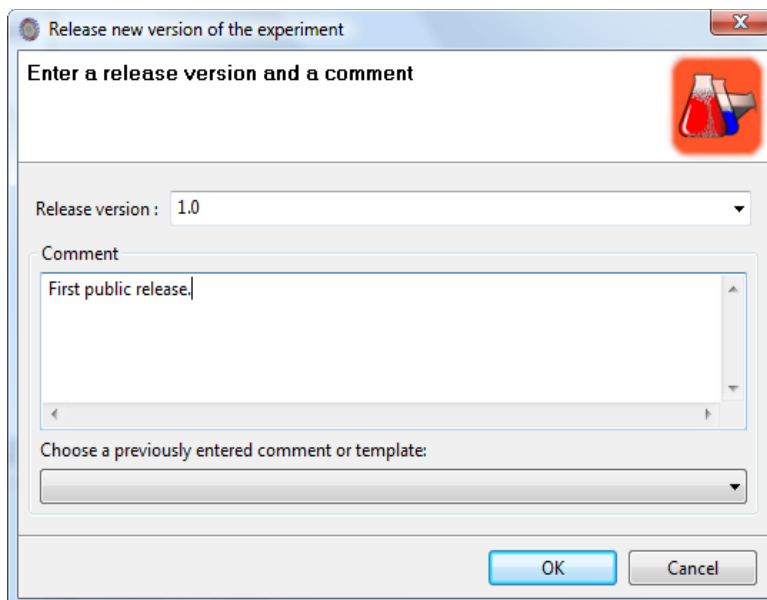


Figure -22: Release a version of an experiment

After clicking the “OK” button, a new release of the experiment will be added to the Experiment Repository, therefore it gets visible for the users of ViroLab portal.

3.1.3.Source Code Access, Bug Reporting and Authors Contact Information

The entire source code of EPE is accessible through the Subversion repository (the anonymouns read-only access is granted for everyone):

```
#> svn checkout https://gforge.cyfronet.pl/svn/epe
```

Should you find any bugs, missing functionality or you'd like to have some nice new features implemented, please use the ticket emission and management system on the Trac website of EPE:

- Viewing tickets: <http://virolab.cyfronet.pl/trac/epe/report>
- Issuing new tickets: <http://virolab.cyfronet.pl/trac/epe/newticket>

You do not need any account for that, tickets could be submitted anonymously.

Authors list: Dariusz Król, Piotr Pęgiel, Włodzimierz Funika.

Developers team contact person: Dariusz Król [dkrol@student.agh.edu.pl].

3.2. EXPERIMENT PLANNING PLUG-INS

3.2.1. Installation

ViroLab EPE plugins is a set of Eclipse plugins packaged as features. These features are hosted on a specialized website (called *update site*) which is accessed by Eclipse's update manager.

The following instructions describe how to install ViroLab EPE plugins via Eclipse's update manager:

1. Open Eclipse. Go to Help -> Software Updates -> Find and Install.
2. Select **"Search for new features to install"**. Click **"Next"**.
3. Click **"New Remote Site"**. Enter *"ViroLab plugins updatesite"* for the Name and *"http://virolab.cyfronet.pl/epe/plugins"* for the URL.
4. Click **"OK"**
5. You should now see a new entry name "ViroLab plugins updatesite" with a mark next to it (see Figure -1).

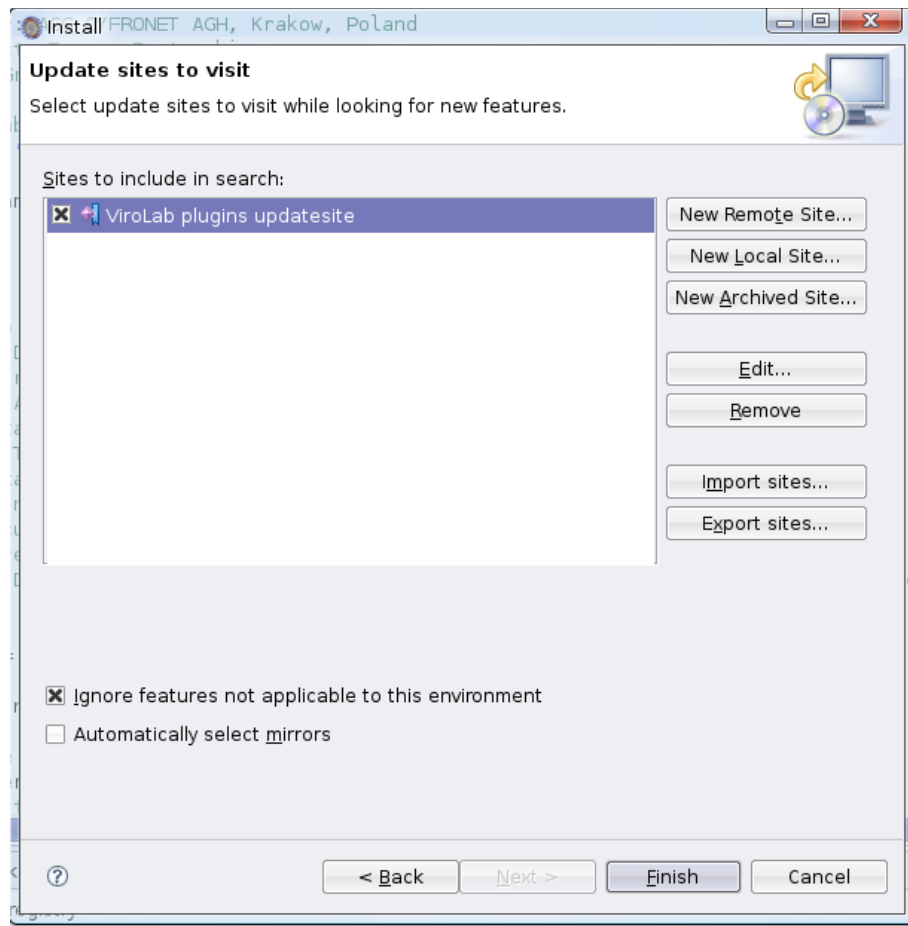


Figure -23: EPE update sites manager window.

6. Click on that and select plugin (plugins) you want to install. Click **"Next"**.
 - You may see a warning when you try to install plugins that require other, not yet installed (or outdated) plugins. In order to check also the other, required plugins please click the **"Select required"** button.
7. Read the license (usually it's GPL) and, if you agree with its terms, tick the **"I accept..."** box and click **"Next"**.
8. Select or add the appropriate site to install the features (most of the cases you just accept the default). Click **"Finish"**.
9. Click **"Install"** (or **"Install All"**) on the warning dialogs during feature verification (the features are not digitally signed).
10. After successfully downloading and installing the features click **"Yes"** on the "Would you like to restart now?" dialog.

Congratulations. You have made it!

3.2.2.Virtual Organization Configuration Plug-in

This tutorial shows how to configure Experiment Planning Environment (EPE) to use properties specific for concrete Virtual Organization. It assumes you have already installed the plugin (see previous Section) inside your EPE.

For plugin version: 0.2.1

VO Configuration EPE Plug-in is responsible for storing virtual organization properties. It loads these properties from external properties file. These properties are used by other EPE plug-ins:

- Grid Resources Browser plug-in receives default Grid Resources Registry endpoint
- Ontology Browser plug-in receives default Domain Ontology Store endpoint
- Evaluation Request Builder receives VO properties necessary to create an Evaluation Request document (for more info about Evaluation Request please see the *GridSpace Experiment User Manual* Section in *Experiment Users' Manual*).

To open VO properties page click "Window->Preferences" (Figure -24) which opens preferences pages dialog.

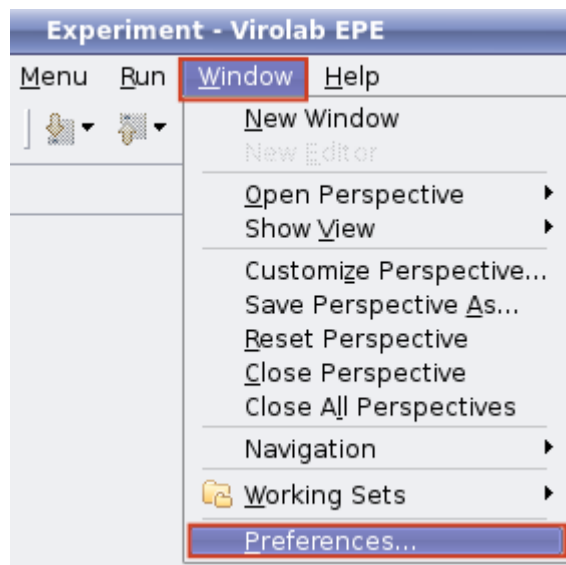


Figure -24: Opening properties pages.

Click "VO Configuration" from list (Figure -25).

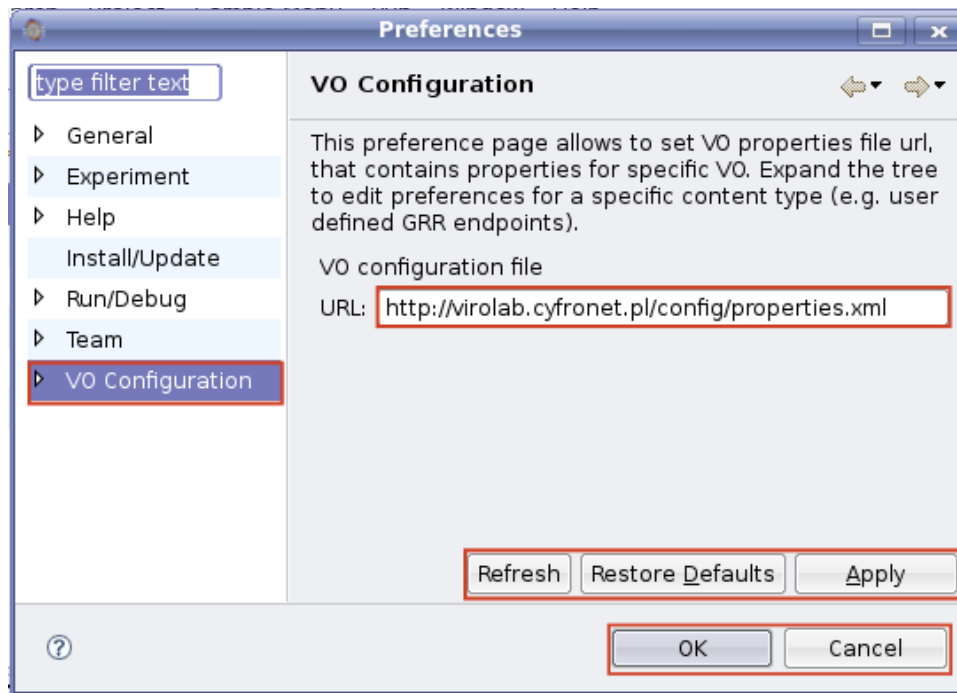


Figure -25: Virtual Organization properties page.

Using "VO Configuration" properties page user can select URL where virtual organization properties are stored. What is more there are two groups of functional buttons.

- First one doesn't close properties pages dialog:
 - **Refresh**: reload VO configurations
 - **Restore Defaults**: set default location of VO properties file (VO properties will not be loaded until user click **Refresh** or **OK** button)
 - **Apply**: apply all changes
- The second one does:
 - **OK**: apply all changes and close properties pages dialog.
 - **Cancel**: cancel all unsaved changes and close properties pages dialog.

3.2.3.Resources Browser Plug-in

his tutorial shows how to configure and use Grid Resources Browser plug-in. It assumes you have already installed the plug-in inside your EPE (see Section 3.2.1).

For plugin version: 0.2.3

3.2.3.1. Configuring Grid Resources Registry browser

Grid Resources plug-in is able to present, in user-friendly way, resources registered inside Grid Resources Registry. What is more, it is able to browse more than one registry at the same time. To configure registries that browser should browse dedicated properties page is created. To open this preferences page click **Window->Preferences** (see Figure -26).

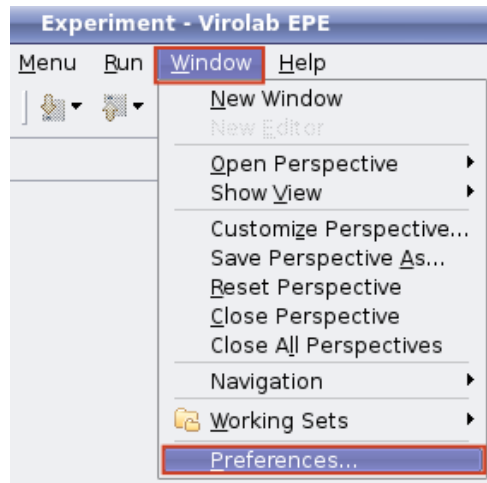


Figure -26: Opening properties pages.

After that preferences pages dialog is opened. Expand "VO Configuration" node from the list and select "Registry Entries" (see Figure -27).

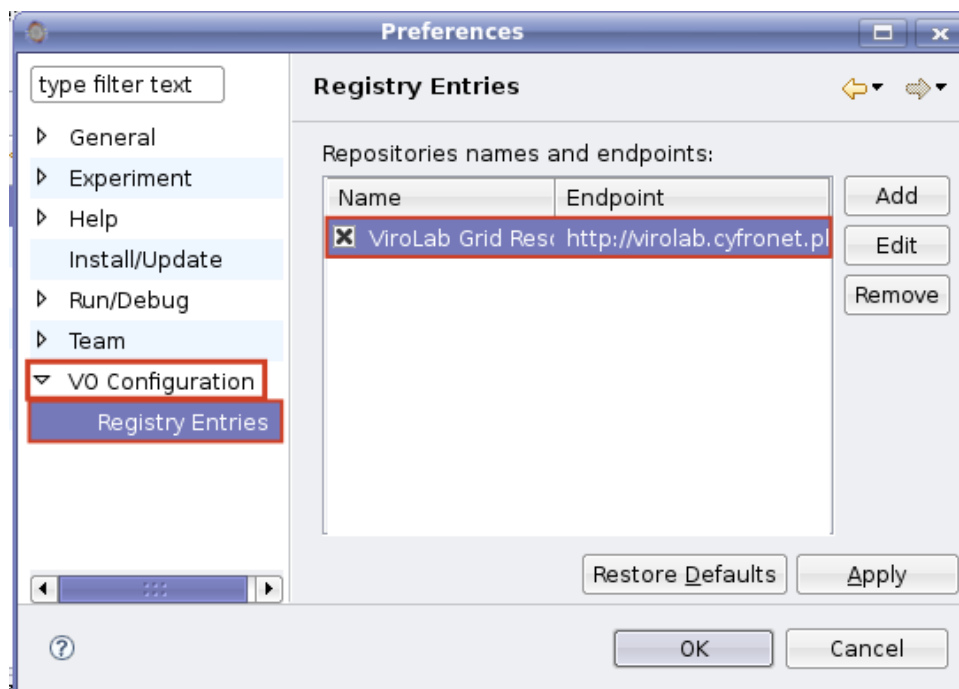


Figure -27: Grid Resources Registry properties page.

By the default Grid Resources Registries defined in virtual organization properties are added and user is able to add additional registries endpoints. There are two different colors that presents registries from VO and user defined (see Figure -28). User is able to add, edit or removed user-defined registries, but he or she

is not able to remove registry defined in VO (this registry can be only disabled by switching check box off).

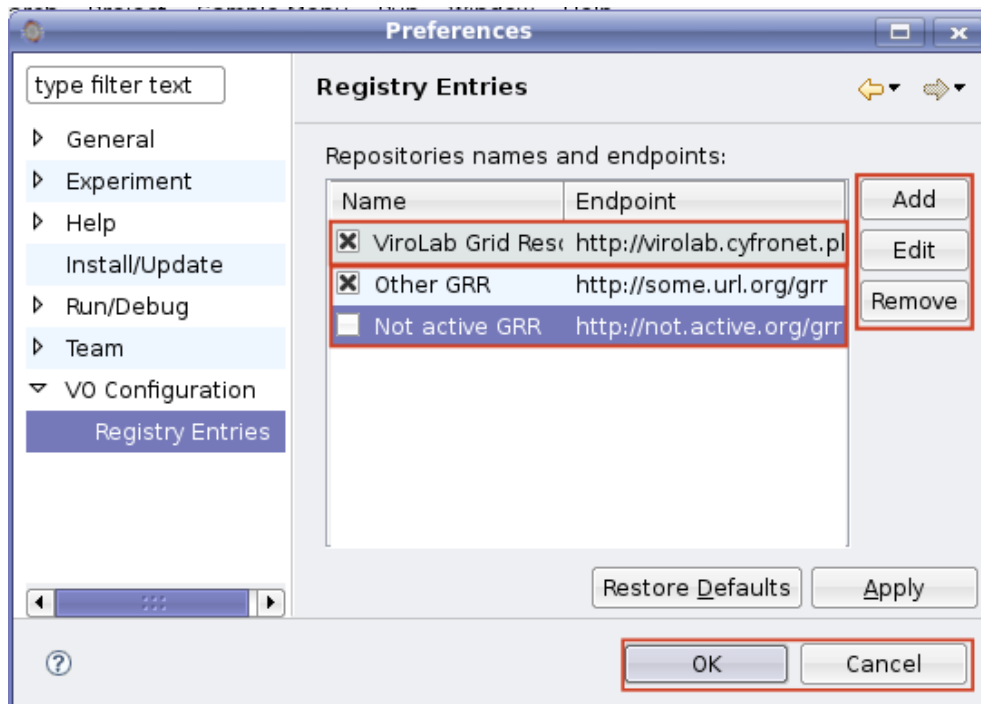


Figure -28: User defined Grid Resources Registries entries.

These properties page contains three groups of buttons:

- the first one that allows to manage registries entries:
 - **Add** add new registry entry (user defines registry name and its endpoint)
 - **Edit** edit selected registry entry (if more than one entries is selected, the first one is chosen). Editing VO defined registries entries (this with gray background) is not allowed
 - **Remove** remove selected entry(ies). Similar to **Edit** button registries entries defined by VO are not allowed to be removed.
- the second one:
 - **Restore Defaults** remove all registries added by the user (only registries defined by VO stayed)
 - **Apply** save all changes
- the third one that closes properties pages dialog:
 - **OK** save all changes and close properties pages dialog
 - **Cancel** cancel all unsaved changes and close properties pages dialog

When user saves changes, Grid Resources browser is refreshed.

3.2.3.2. Opening Grid Resources Registry browser

To open Grid Resources browser please click **Window->Show View->Other...** (see Figure -29) then find and expand **Grid Resources Registry** node, select **Resources Browser** and click **OK** button (see Figure -30). After view is opened, it stays active even between stopping and starting EPE again (unless you definitely close Grid Resources browser by yourself).

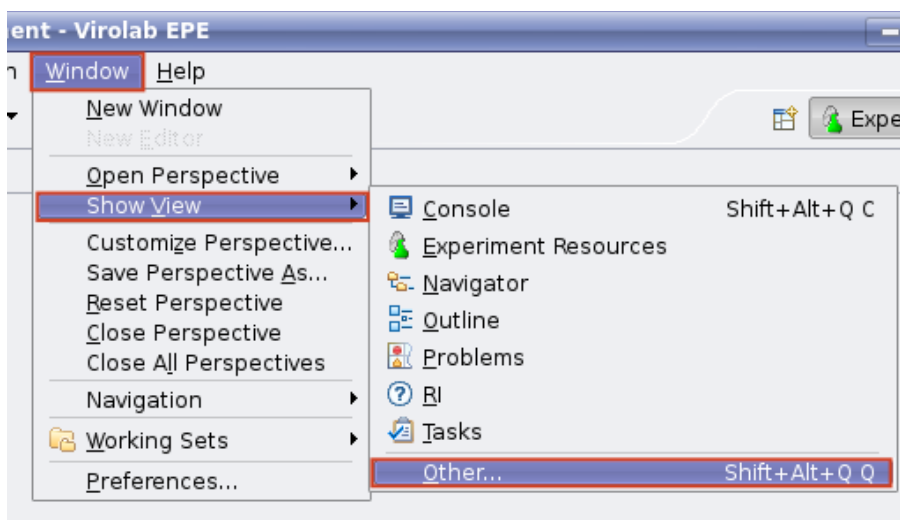


Figure -29: Opening available views browser.

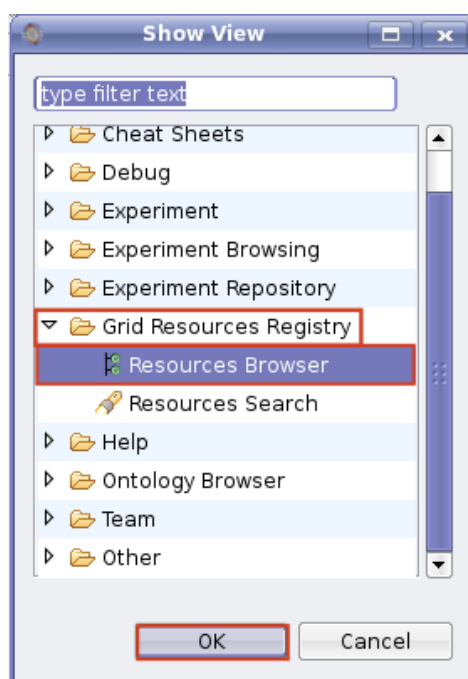


Figure -30: Opening Grid Resources Registry browser.

3.2.3.3. Browsing Grid Resources Registry

Before going into detail user should be familiar with three resources description layers paradigm (see the online Grid Object Abstractions tutorial - <http://virolab.cyfronet.pl/trac/vlwl/wiki/GridObjectAbstractions>).

Figure -31 presents Grid Resources Registry browser. Browser is able to connect to remote Grid Resources Registry (or more than one registries) and presents resources stored in it. The highest layer of resources description (technological independent) is Grid Objects. Grid Objects are grouped in packages (this idea is similar to java interfaces and packages). Every Grid Object has operations with input and output parameters. Browsing this layer is enough to write full functional experiment (gsengine takes care of choosing the most optimal technology and resource instance) but if user wants to take more control during creating script, he or she is able to browse information about existing technologies and instances of Grid Object.

During script development information inside registry(ies) can hanged (e.g. you ask developer to create and register a new gem), that is why refresh button is available in resources browser.

To manage resources browser window there is minimize, maximalize and close button.

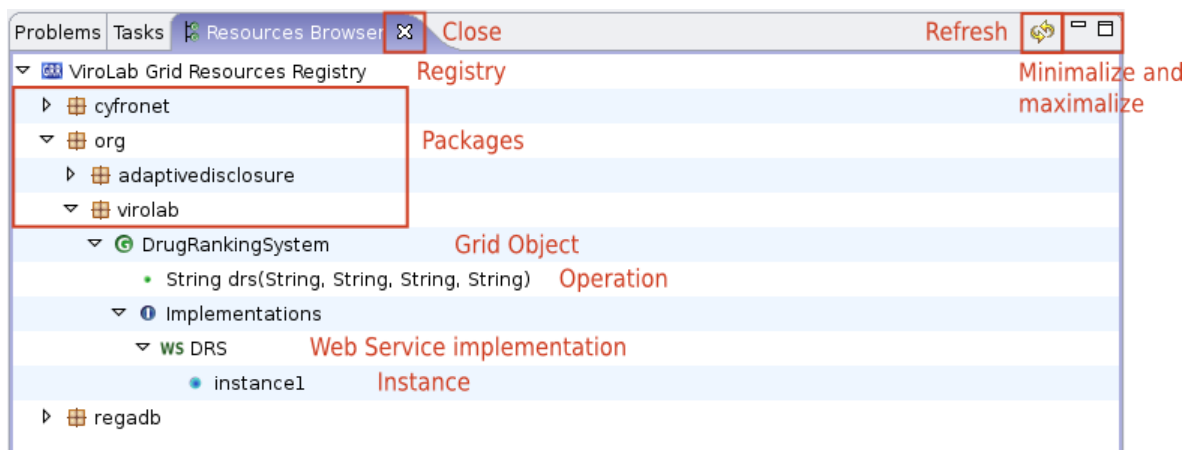


Figure -31: Browsing Grid Resources Registry.

Browsing Grid Object and its operation is not enough to create experiment script in easy and user friendly way. That is why every operation and its input and output parameters should be described in understandable way. Resources browser allows to present this information to the user. If you move cursor on the operation, in short delay pop-up with description is shown (see Figure -32).

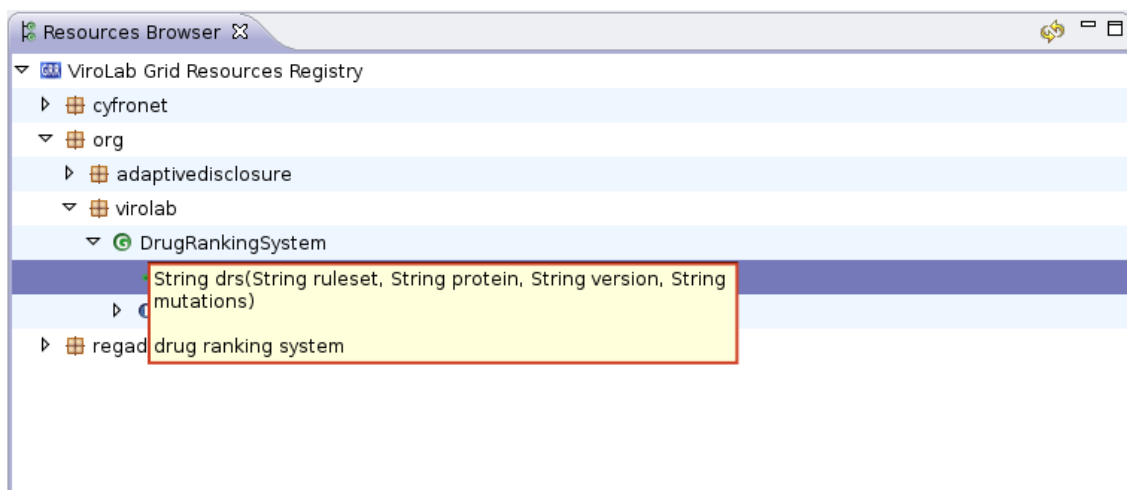


Figure -32: Grid Resources Registry browser pop-up.

Every element shown in resources browser can have context menu. Context menu items can be different for Grid Object, operations, etc. (see Figure -33). Currently there are available following context items for:

- *Grid Object* - inserting code line to script editor that create selected Grid Object.
- *Grid Object Operation* - show input or output parameters meaning in ontology browser.

Context menu items can be dynamically added to resources browser by implementing 'IctxMenuItemsProvider' interface (for further details see [RESBROWDEV])

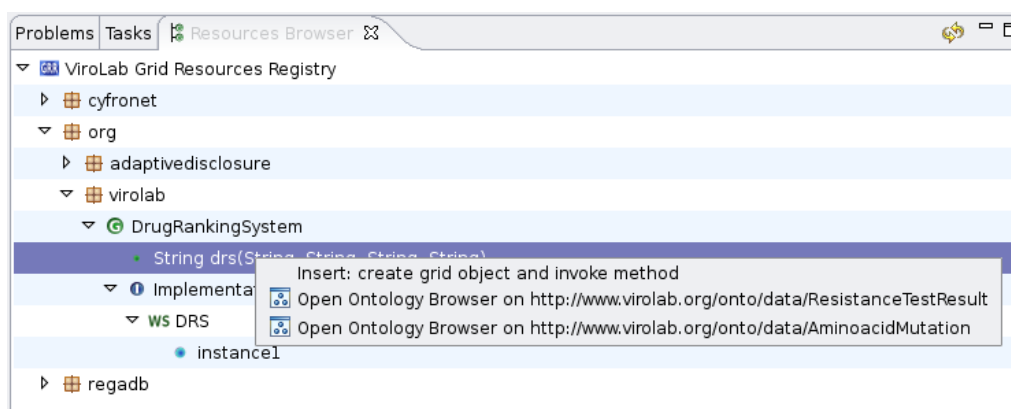


Figure -33: Grid Resources Registry browser context menu.

3.2.3.4. Inserting code line to EPE experiment editor

Resources registry allows user to add creating Grid Object code line to script editor in two ways: by double click on Grid Object or by choosing correct context menu item (see Figure -34 and Figure -35).

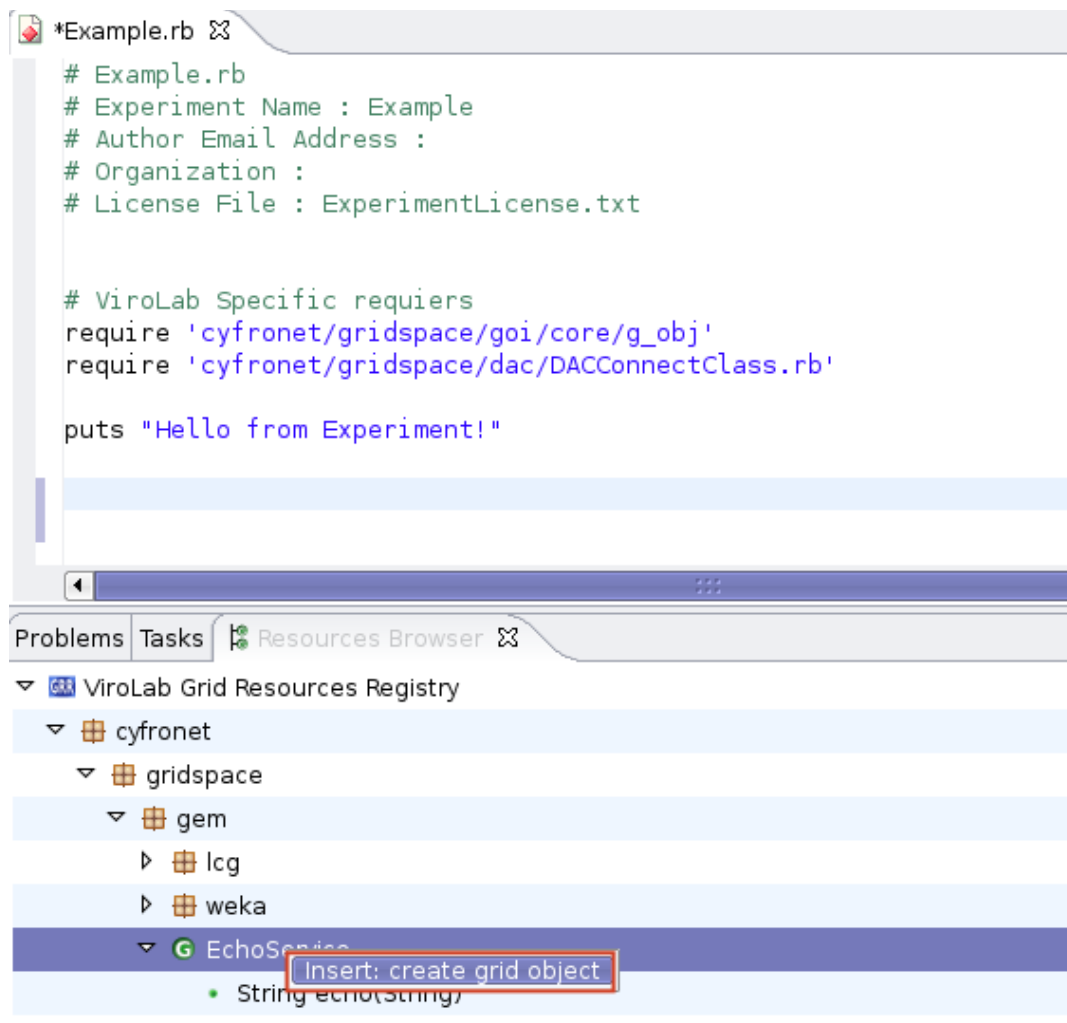


Figure -34: Inserting code line to EPE experiment editor.

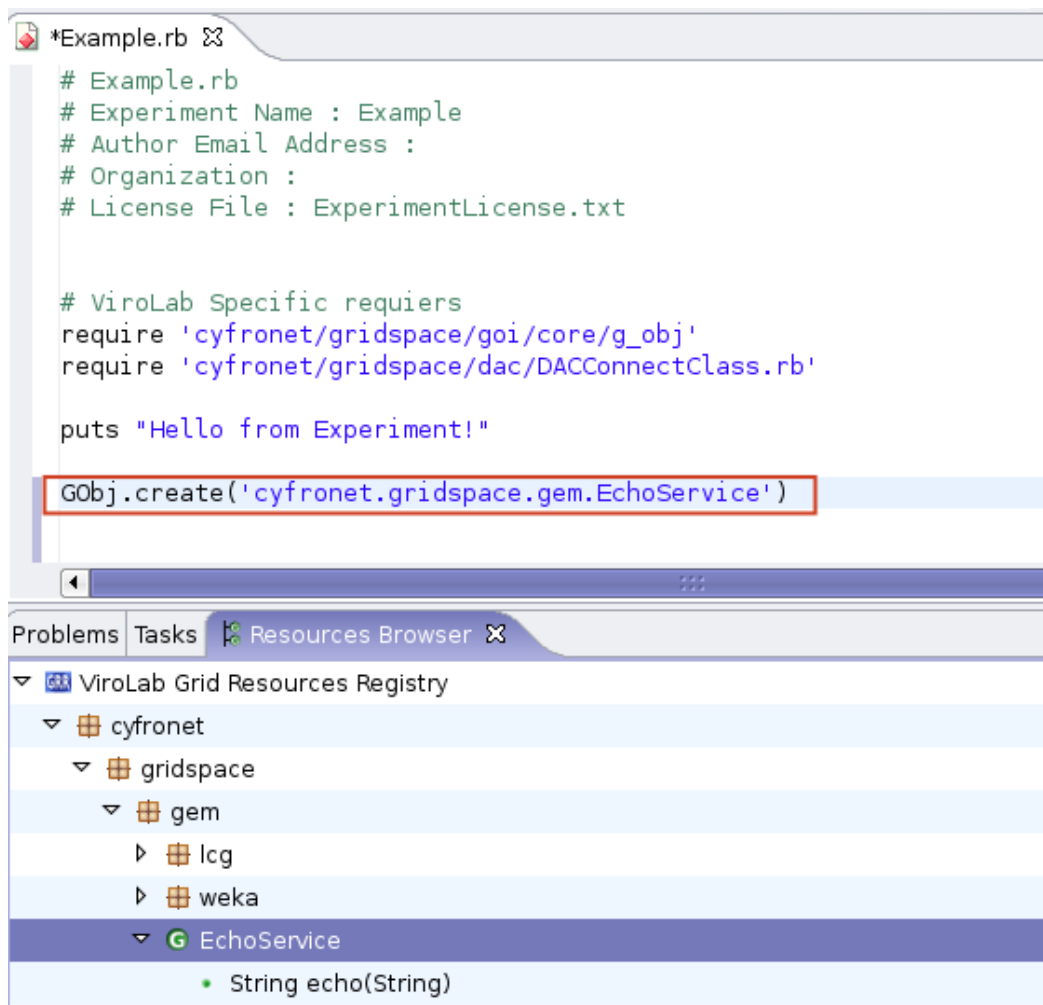


Figure -35: Code line inserted to EPE experiment editor.

3.2.3.5. Interaction between Grid Resources Registry browser and Ontology browser plug-ins

This paragraph presents interaction between Grid Resources Registry browser and Ontology browser plug-in.

From GRR browser to Ontology browser

Every input and output parameter can have ontology meaning that can be presented in Ontology browser plug-in. To open ontology parameter meaning you have to chose correct context menu item – *"Open Ontology Browser on ..."* (see Figure -36 and Figure -37).

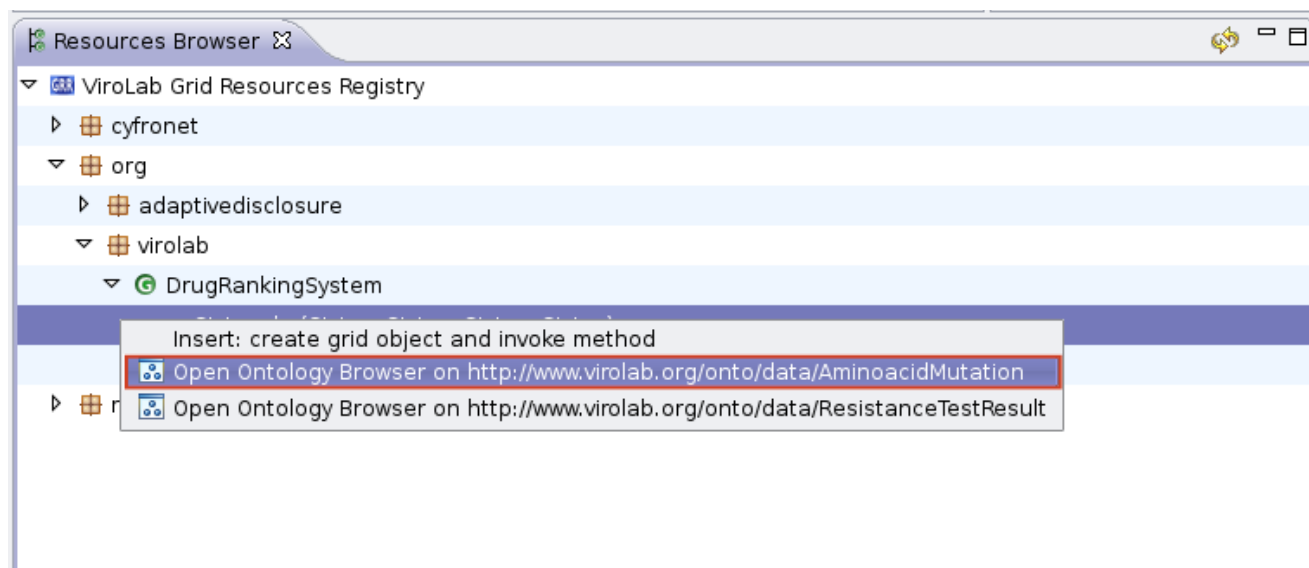


Figure -36: Show resource in Ontology browser.

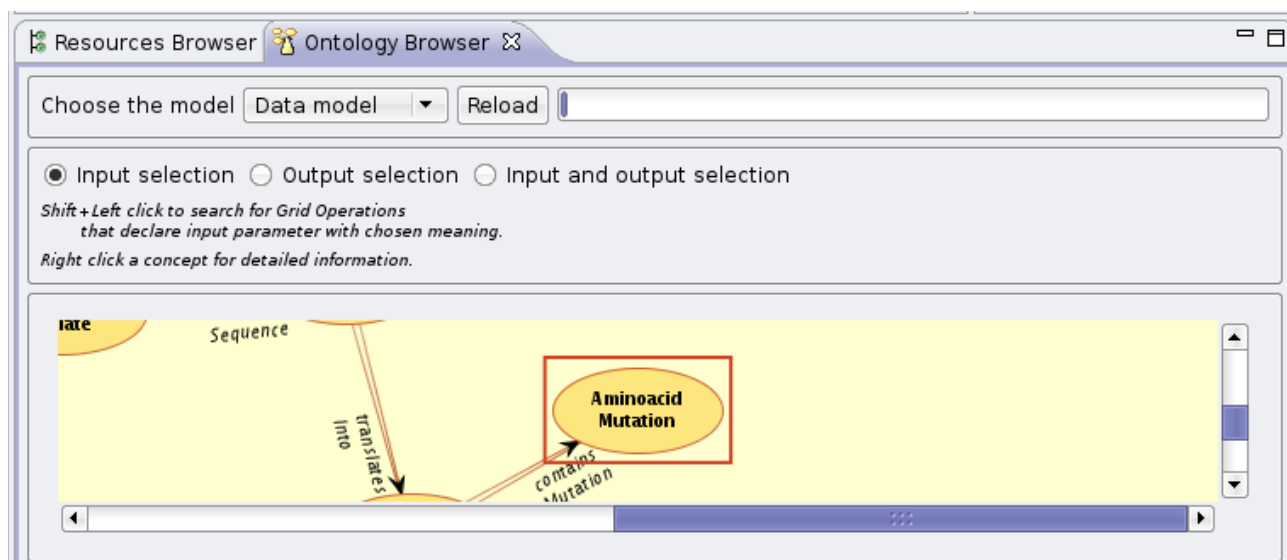


Figure -37: Show resource in Ontology browser result.

Semantic search in Grid Resources Registry browser

There is possibility to find all operations that have input or output parameters that fulfill ontology meaning. For more information how to switch from Ontology Browser to Grid Resources Browser plug-in see the Ontology Browser plug-in manual in the following Section. Figure -38 presents result of semantic search of operation that has input parameters with "Aminoacid Mutation" meanings. As you can see there are only this Grid Object and operation that fulfill this query.



Figure -38: Semantic search

User can be interested in browsing packages of found Grid Object. That is why there is a possibility to show found Grid Object (and other found elements) in resources browser. To show found Grid Object or its operation in resources browser user should choose *Show in Resources Browser* context menu item (see Figure -39). After that resources tree is expanded to selected item (see Figure -40).

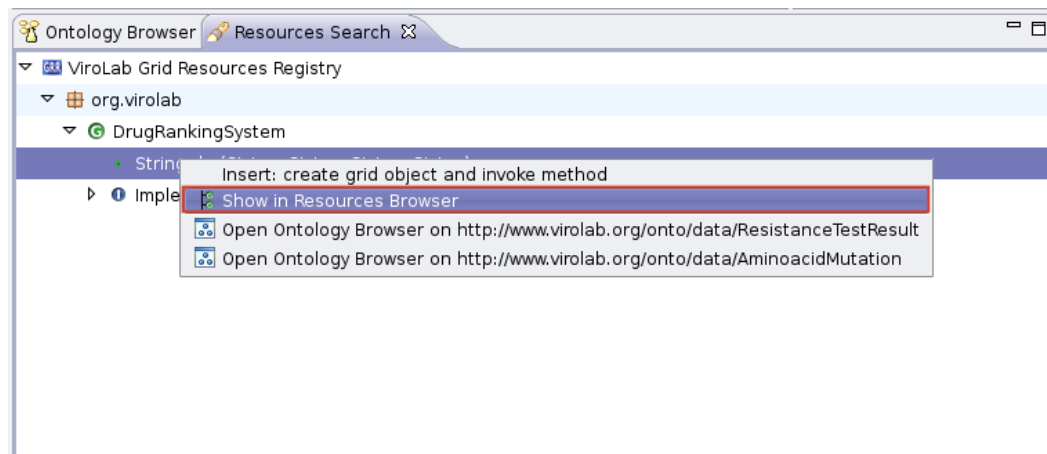


Figure -39: Semantic search context menu

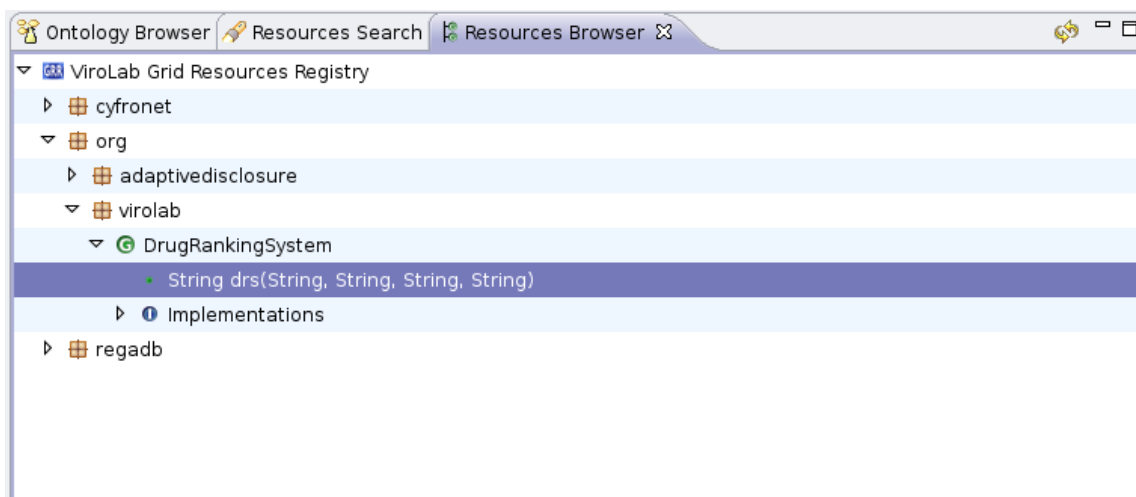


Figure -40: Show semantic search result in Grid Resources Registry browser.

3.2.4.Ontology Browser Plug-in

This manual shows how to operate the **Ontology Browser Plugin** in order to use its functionality for experiment development inside the Experiment Planning Environment (EPE). It assumes you have already installed the plugin inside your EPE (see Section 3.2.1).

For plugin version: 0.1.2

Configuration

Currently there is neither need nor any possibility to configure the plugin. In the future releases we plan to add configuration for another Domain Ontology Store and some further options connected to the method of viewing and using the ontological taxonomies.

Opening

After successful installation, in order to invoke the Ontology Browser view to your EPE workspace, please:

- choose **Window -> Show View -> Other...** from the main upper menu
- find **Ontology Browser** section and double-click on the **Ontology Browser** view inside that section

When you do that once the view will always stay active within your current perspective and you don't have to repeat this procedure after you shutdown/reopen EPE (unless, of course, you deliberately close the Ontology Browser view yourself).

The Ontology Browser view window

simple triple of **<subject, predicate, object>** and could be easily and naturally visualized as a directed graph arc going from a concept in the role of **subject** to the (usually different) concept in the role of **object** with the label (type) of the arc denoting the **predicate**.

As a single concept may be a subject (or object) of many such statement, the entire model immediately forms a kind of network, where the concepts are nodes of the network and the relations are connections. Such a network is (in somewhat informal way) called a **taxonomy**. Most of the taxonomies could be visualized as a graph where nodes (concepts) become vertices and connections (relations) become directed arcs. In fact, such a visualization is presented to you by the *Taxonomy panel* of the Ontology Browser.

Theoretically ontological models (and their taxonomies) may represent anything. In the case of our plugin, they represent a part of knowledge from the chosen domain of science. They model concepts and relations that are frequently used in the domain's terminology. In fact, one of the main functions of these models is to form a kind of bridge between scientists of that domain and the virtual laboratory infrastructure: they allow higher level of the human-to-computer understanding.

Basic operations.

Changing and reloading model.

The taxonomical models that are presented by the Ontology Browser are stored inside a special repository called the **Domain Ontology Store**. The repository is a remote http-protocol service accessible from anywhere around the world - the plugin simply loads the models on demand from that source. In order to distinguish between the models they have descriptive names and the name of the currently loaded model is given in the *Model section*. The default model loaded for the first time when you open the Ontology Browser view should be the *Data model*.

In order to load a different model just choose a different model name in the drop-down edit box with the model name. The model is instantly loaded into the browser when you click your choice. For instance, if you choose to load the Activity model, after a second or two of delay the *Taxonomy panel* will change to the new model (it is possible that this one is empty - it simply means that there are no statements in this model). To go back to the *Data model* please use the same mechanism.

If you want to just to **reload** the present model (e.g. when you know there are some changes made to it) just press the **Reload** button in the *Model section*. Apart from the reload of the taxonomy this may also change the content of the *Taxonomy panel* since even if there are no changes the layouting mechanism for the graph is not deterministic and may produce a different-looking taxonomy (don't worry - still, this is exactly the same taxonomy, just some things could be relocated in the panel). As you see, you may also use this button when the taxonomy gets too cluttered.

Viewing the model taxonomy.

There are a couple of basic actions that you can do with the Taxonomy panel to browse the model:

- *zooming in/out*: please use the scroll button of your mouse to make the graph larger or smaller,
- *moving around*: click the left mouse button on a place where there are no concepts and drag the mouse while holding the button,
- *rotating*: SHIFT-click the left mouse button on a place where there are no concepts and drag the mouse while holding the button,
- *more information*: click the right mouse button on a concept to learn more about it (currently nothing more than the full id of the concept is given).

Searching for Grid Operations.

First of all for this functionality you need to install the **Resources Browser** as well (see previous Sections). You don't need to have the Resources Browser view opened for the feature described below - just make sure that the plugin is installed.

As you'll find described in the Resources Browser manual, the Grid Objects (that are important building blocks of many experiments) have Grid Operations (just like objects in object-oriented programming languages have methods). The operations may have input and output parameters and some of them (not necessarily all of them) may have **semantic meanings**.

Semantic meaning is a piece of meta information that could be attached to virtually anything that is identifiable. It tells both artificial systems and human users what is the real-world meaning of this particular object, usually in the terms of some specific domain of human knowledge. For instance, a simple string of letters "CCTCAAATCACTCTTTGGCAAC" could have a semantic meaning of "nucleotide sequence". This mechanism is usually used in computer systems to improve the common understanding between the system and its human users.

The searching capability allows to find registered Grid Operations that have parameters with semantic meanings similar to a concept from the taxonomy (an example is given below). There are three modes of operation here activated through the *Grid Operations search section*. When you switch to another model a short information on how to use it appears in the same section, just below the choices. The modes allow you to search for:

- Grid Operations that have at least one input parameter with the semantic meaning equal to the chosen concept
- Grid Operations that have at least one output parameter with the semantic meaning equal to the chosen concept
- Grid Operations that have at least one input and one output parameters with their semantic meanings equal to the indicated pair of concepts

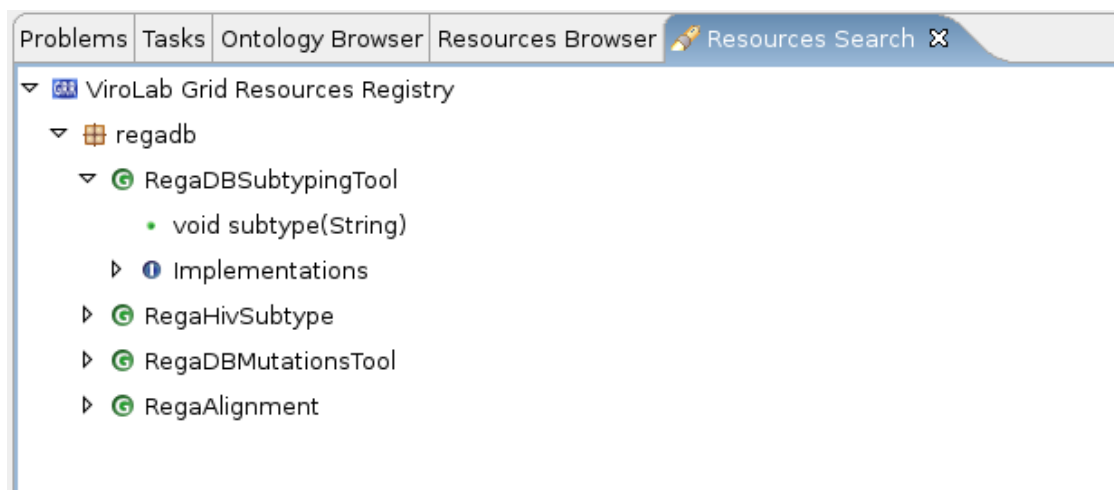


Figure -42: The search result view that appears after successful Grid Operation search action.

Example. I have acquired a couple of nucleotide sequences and I'm interested in the Grid Objects that are able to analyze them. My first try will be to find all the operations that accept such nucleotide sequences as one of their input arguments. Therefore I open the Ontology Browser view, load the *Data model*, switch to the *Input selection* mode of search and, according to the instructions given, SHIFT-click on the *Virus Nucleotide Sequence* concept (as it is the one that seems semantically closest to what I mean). The *Resources Search* window appears right away that presents to me the set of the four Grid Objects that fulfill my requirements (see Figure -42). Now I may use this windows to further pursue my search.

It is possible that there are no Grid Operations that would meet the search requirements. In that case the search results window does not appear and the user is simply notified of the fact by a message.

Future functionality.

The Ontology Browser in its current form has a status of a prototype. While there are many functionalities that are not present in it, the release is mainly for testing purposes and to gather feedback from the users of the tool. Still, the main functionality of Grid Operation search is already provided and may be effectively used for faster search of appropriate Grid Objects.

The features that are in our short and long term plans:

- adding the activity model to the Domain Ontology Store are implement the search for Grid Operation by the type of the activity performed
- integrating the data model with Data Access Client library for data query formulation assistance
- align the data model taxonomy to converge with the ViroLab data schema (when the schema is finally decided)
- add a separate properties page so a user may customize the browser a little bit.

3.2.5. Source Code Access, Bug Reporting and Authors Contact Information

The entire source code of the EPE plug-ins is accessible through the Subversion repository (the anonymous read-only access is granted for everyone):

```
#> svn checkout https://gforge.cyfronet.pl/svn/plugins
```

Should you find any bugs, missing functionality or you'd like to have some nice new features implemented, please use the ticket emission and management system on the Trac EPE website:

- Viewing tickets: <http://virolab.cyfronet.pl/trac/epe/report>
- Issuing new tickets: <http://virolab.cyfronet.pl/trac/epe/newticket>

You do not need any account for that, tickets could be submitted anonymously.

Authors list and contact information: Marek Kasztelnik (VO Configuration and Resources Browser plug-ins) [m.kasztelnik@cyfronet.pl] and Tomasz Gubała (Ontology Browser plug-in) [gubala@science.uva.nl].

4. GRID RESOURCES REGISTRY USER'S MANUAL

The Grid Resources Registry is a central place where information about ViroLab virtual laboratory resources is stored. This component is responsible for storing two types of information that describe the following resources: technological independent and technology specific. The first type of information is presented to the user and is very important during scenario script development (see Section 3.2.3). The second type of the information (technology specific) is used by the GSEngine mechanism during executing remote resources.

This section shows the alternative way of browsing the GRR resources (not connected with EPE) and presents the procedure of creating and registering the new resources.

4.1. GRID RESOURCES WEB BROWSER

This tutorial shows how to use the web version of Grid Resources Registry browser.

For web browser version: 0.2.3

Web Resources browser is very similar to Resources Browser EPE plug-in (see Section 3.2.3), but with limited functionality. It allows only to browse resources stored inside GRR. Figure -43 presents Web Resources browser. The most important component in this tool is tree that presents all resources available in browsed GRR. By expanding tree items you are able to browse available packages, Grid Objects, Grid Objects operations, implementations and instances.

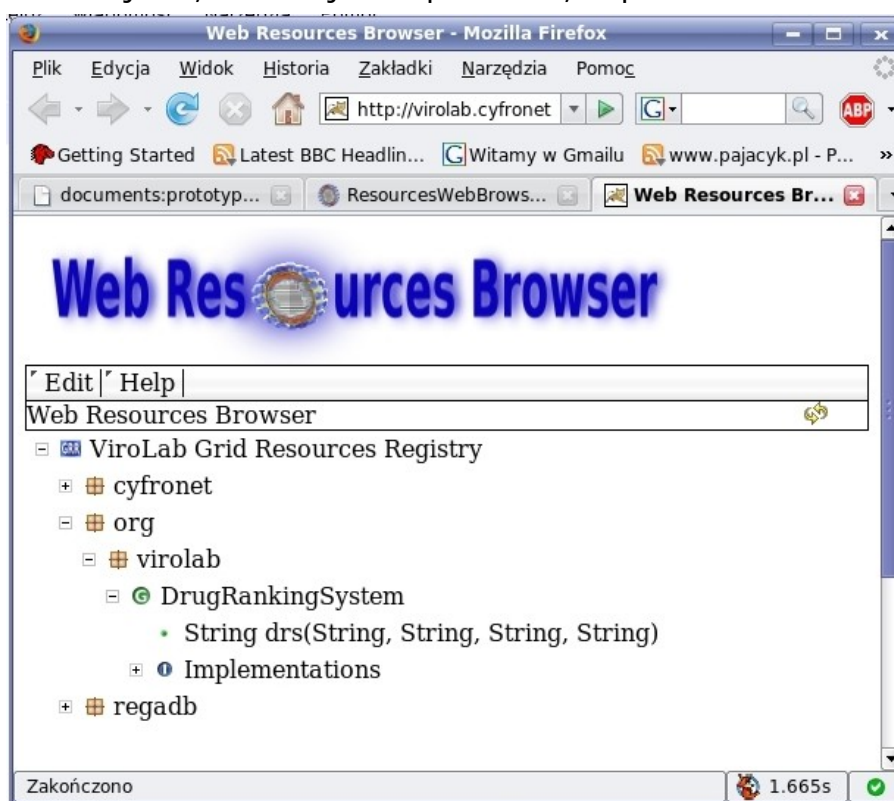


Figure -43 Web Resources Browser

Currently Web Resources Browser allows to browse resources from registries defined by Virtual Organization. User is not able to configure browser (this functionality will be available in the next version of Web Resources Browser). For more information about future functionalities that will be added to this tool, please see [D3.3] Section 3.2.3.

4.2. ADDING NEW GRID OBJECTS

It is fairly obvious that the usefulness of any virtual collaborative space, and the ViroLab Virtual Laboratory is no exception, is the amount and quality of the resources one may use through them. One of the crucial resources that the laboratory is built around are computational activities, called Grid Objects (pages <http://virolab.cyfronet.pl/trac/vlvi/wiki/ExecutionMechanism> and <http://virolab.cyfronet.pl/trac/vlvi/wiki/GridObjectAbstractions> are good places to visit first to learn about the Grid Object idea). Here we would like to introduce possible methods of adding your own building block to the space in order to allow fellow users to include your tool in their experiments.

If you have your new Grid Object operational and accessible from outside, the last thing is to spread the word. To make the tool visible to other members of the virtual laboratory community (and to components of the laboratory itself) you need to publish it in Grid Resources Registry (GRR). For more information how GRR is build see [D3.2] Section 6.1.2 and what is the implementation status see [D3.3] Section 3.4.

4.2.1. Preparing your Grid Object

So you have a nice piece of software that you'd like to share with your collaborators. There is a couple of questions (or steps) you should try to answer to efficiently and suitably provide your tool:

1. What is your tool doing? To what purpose it could be used?
2. What functionality and how the tool will provide to the virtual laboratory users?
3. How the interaction between the tool and its future user should be organized?
4. What would be the best technological solution to realize previous assumptions?

Example. Together with my colleagues we work on a climate modeling application and I'd like to donate a temperature provision service. In this Section we will follow this example in a series of steps given in the separated rectangles like this one.

Step by step let us assist you in this process. Also, from time to time, an example will be presented to further picture the ideas. However, please keep in mind

this is not the only way of preparing publication of your own Grid Object and you are welcome to follow your own methodology if you find it suits you more.

4.2.2.Functionality

Step 1. The tool should provide current air temperature in pre-designated 30 weather stations. All this stations publish periodically their climate reports and the task would be to grab the newest ones (from web pages of stations), parse them and return the requested results.

This is simple yet crucial step. It is important to describe to fellow collaborators what the tool does in terms of its respective field or domain. This also helps the author himself or herself to properly and precisely define the purpose of the tool: what exactly will it provide. The result of this step should be a high-level description of the tool (this will be useful during the publication phase later on).

4.2.3.Interface

Step 2. It seems it is far too inefficient to grab all the 30 results every time the temperature is needed as it seems our experiment will require one location per time. So, one operation could be called *currentAirTemp* and provided with the location name it will return the number (Kelvin scale). Since some of other users may need this information in centigrade, I'll provide the translation *kelvinToCelcius* operation as well.

After general definition of functionality the next step is to decide how the functionality will be exposed for the experiment developers. You have to define its interface. The approach we propose is to divide all the functions of the tool into fairly independent operations, define their signatures and transform them into Grid Operations in the process of publication. Please also consider what input information will be required to call operations of your tool and what are possible sources of this information.

4.2.4.Interaction mode

Step 3a. There are two different approach I consider for my tool. The straightforward one is to provide two operation in blocking, synchronous mode and do not store any internal state of the tool. The *currentAirTemp* operation will get the report of specified station, parse it and return the result in Kelvin scale. The other operation may be called to transform it into Celcius scale (by the way, this way the other operation could be used completely independent from the first one for simple transformation purposes).

At this stage you have to think on the correlation of different operations of our tool (having defined the operations in the previous step). Are they completely independent, or perhaps they should be called in some specific order. If the later, there is a good chance that some data is passed between them - is that data passed explicitly (a value returned by one operation is consumed by another) or is it maintained by the tool internally (an operation saves something that another operation will later use)? If the second is the case, the tool is probably stateful and so should be the interaction with it (in case of stateful tools that require

specific order of operations we use the specific term of *conversation*). The decisions you take regarding the interaction mode of your tool will probably highly impact the choice of implementation (of wrapping) technology you'll use in subsequent steps of this scenario.

Step 3b. Another possibility would be to allow for more stateful, conversation-like interaction mode. For instance, to have an initialization method that one may use e.g. to constrain the list of meteo stations (for optimization). As the air temperature does not change every second, periodic check for latest reports could be made in background and a call to the main operation would return the latest result obtained for certain station.

Keep also in mind that there could be non-functional reasons for choosing specific solution. For instance, consider a tool that has just one operation that is called frequently but that requires time-consuming computation to be performed once before. One may choose the synchronous and stateless interaction mode, but that introduce a huge delay during the first execution of the operation (as the computation is performed). In that case, one may consider adding another operation (like *startComputation*) that would in a non-blocking mode start the computation (non-blocking means the calling client does not have to wait for the completion) and the first time the original operation is called the computation results are already present (in the worst case, when the operation is invoked right after the initialization, the result can not be any worse that in the simple, former scenario).

Having the tool prepared and knowing how its functionality will be presented to other virtual laboratory users (that is, mainly experiment developers), the choice of the wrapping technology that will make the tool remotely accessible, has to be made.

4.2.5. Supported technologies and protocols to implement Grid Object

The set of remote processing technologies supported by the virtual laboratory is changing with time as its components are developed - so please note the information given below could change (hopefully grow rather than shrink) in the future.

Technology	Overall characteristics	Programming platform
Web services	Useful especially for stateless services of relatively fine-grained communication delays (at most 3-4 minutes to response); mainly used in blocking, synchronous mode.	Various languages and command line
MOCCA	Better suited for <i>heavier</i> , more time consuming computations; internal state could be maintained; dynamic deployment possible.	Java and command line

In the first case you basically use the SOAP framework that seems to fit your needs (programming platform, performance, simpleness) to publish your own tool as a Web Service. In the second case, please consult the on-line MOCCA component tutorial [MOCCATUT].

Step 4. Depending on the choice I made in the previous step, I would follow the Web Service scenario in the first case or I would wrap the tool as a MOCCA component for the other, stateful solution.

Below please find some resources (tutorials, guides) that will help you proceed with wrapping your tool in the chosen remote processing technology. Apart from these that we provide ourselves, there are probably many more tutorials elsewhere.

- MOCCA component development and wrapping tutorial
<http://mocca.icsr.agh.edu.pl/doku.php?id=doc>
- Quick Java Web Services wrapping with Xfire and Maven
<http://virolab.cyfronet.pl/trac/vlwl/wiki/WsXfireTutorial>
- Command line tool application wrapping with Java
<http://virolab.cyfronet.pl/trac/vlwl/wiki/ShellJavaTutorial>
- Wrapping WTS services and EGEE-installed applications as Gems
<http://virolab.cyfronet.pl/trac/vlwl/wiki/WritingWrappers>
- Command line tool wrapping as Web Service with Ruby
<http://virolab.cyfronet.pl/trac/vlwl/wiki/ShellWsRubyTutorial>

Currently the registry has no administration entry point so in order to publish your tool you have to do it traditionally by inserting new records to relational database. Please send an e-mail message to Marek Kasztelnik from CYFRONET team (m.kasztelnik@cyfronet.pl) with following information:

- To register Grid Object (very similar to java interface):
 - package (e.g. cyfronet.gridspace.gem)
 - Grid Object name and description
 - Methods:
 - name and description
 - input and output parameters names, type and descriptions
- To register Grid Object Implementation (very similar to java class):
 - Grid Object that this implementation implements
 - Instance name and description
 - Technology specific information:
 - WS:
 - Code base URL (optional).
 - WS type (XML RCP or Document)
 - MOCCA (if MOCCA component has many ports than it is modeled by many Grid Objects):

- Component code base URL
- Component class name
- Port name
- Component port class name
- JOB:
 - wrapping script
- WTS
 - wrapping script
- To register Grid Object Instance (concrete resources installed in some container, e.g. xFire, H2O kernel)
 - Grid Object Instance
 - Instance name and description
 - Copyright (optional)
 - Instance endpoint

For more information how to write WTS and JOB scripts see <http://virolab.cyfronet.pl/trac/vlwl/wiki/WritingWrappers>

5. GRIDSPACE EXPERIMENT DEVELOPER LIBRARY REFERENCE

5.1. LIBRARY CORE REFERENCE

In order to enable a conversation between GridSpace Engine application and application user the means for interaction are provided by GridSpace engine library. Such a library is used explicitly by an application developer in the application code and allows him programming behavior of data input request sent towards the application user.

Sending data input request from GridSpace Engine towards the command line tool, EPE, EMI, or whatever that submitted the evaluation request is actually some kind of callback. It takes place when an evaluation request call processing is in progress, however suspended until the requested data is provided. After that evaluation request call processing is being continued.

On the submitter side, after intercepting callback data request, the appropriate form is generated and shown to the application user. After form submission the user input is sent back towards GridSpace Engine.

From the application developer perspective the `DataRequester` class is fundamental. It offers method `getData(dataRequest)` that takes a string parameter `dataRequest` that will be shown as a data request caption in a form and returns the user input in a string format.

For the present, `DataRequester` class supports the basic functionality that is to be significantly extended in the future version that will result in a robust UI library for GridSpace Engine.

The example usage for `DataRequest` class is shown in below.

```
require 'DataRequester'

puts "Geno-to-drug resistance: start"

region = DataRequester.new.getData("Region (lowercase)")

puts "region " + region
```

Submission and sample conversation in such an application in the GSEngine command line tool will result in the following output.

```
Geno-to-drug resistance: start
Region (lowercase):
rt
region rt
```

5.2. DATA ACCESS REFERENCE

The DAC is capable of interfacing with data sources represented by standalone databases and by the OGSA-DAI Data Access Client developed at HLRS.

The Data Access Client base class is called **DACConnectClass**. This class should be used for interfacing with VL data sources.

DACConnectClass provides the following methods:

`new(db_tech, db_address, db_name, username, password)` This creates a new DAC handle linked to the data source. At present the following technologies are supported:

- **mysql** (for MySQL databases)
- **pgsql** (for PostgreSQL databases)
- **hsql** (for local HSQL databases)
- **das** (for the VL Data Access Service)

The remaining parameters should be input as strings:

- `db_address` is the URI where the data source (or service) resides
- `db_name` is the name of the data source which we wish to access
- `username` is the login name for access to the data source (does not apply for DAS)
- `password` is the password for access to the data source with a given login name (does not apply for DAS)

Furthermore, **DACConnectClass** also provides a configurable parameter called `objective_references`. This is 0 by default. If set to 1, results will be output as JRuby objects rather than plain arrays (see below). Use `requestObjectiveReferences()` and `requestPlainReferences()` on **DACConnectClass** objects to set this parameter.

- **executeQuery(query_string)** This executes a query on the given data source. The query string should contain a valid SQL expression. Results are returned as a 2d Ruby array (actually a list of lists) if the `objective_references` parameter is set to 0. Otherwise they are returned as a **DACResult** class object (see below).
- **executeUpdate(query_string)** DAC will automatically attempt to disambiguate query isolation level basing on its contents but this functionality is still in its prototype stages and should be used with care. If you are sure that you need to submit a blocking query (i.e. an update), please use this method instead of `executeQuery`.
- For both types of queries, DAC will automatically sanitize input, i.e. it will check if the queries contain native escape characters. If so, a **DACException** will be thrown, to protect the data source against SQL injection attacks.
- **describeDataSource()** This method returns the names of tables (i.e. entities) available in the given data source.

- `describeTable(table_name)` This method returns the schema of the given table (i.e. entity).

DACResult is a Java class with the following methods:

- `String[][] getResultArray()` - Returns output data as a Java array of strings
- `String[][] getResultArrayNoHeaders()` - Strips leading column headers, then returns output data array
- `int getLength()` - Returns number of result tuples
- `String[] getRow()` - Returns single result row at current position of iterator (if array length exceeded, a `DACException` object is thrown)
- `void resetIterator()` - Resets result list iterator to position 0

5.3. COMPUTATION ACCESS REFERENCE

Grid Operation Invoker is a component of GSEngine and is responsible for computation access. It is implemented in Jruby (see [JRUBY]) and is included in the GSEngine release, as well as all Java libraries that GOI requires. Its source can be found in the `$GS_HOME/ruby/cyfronet/gridspace/goi` and its Java libraries are placed in the `$GS_HOME/java` directory.

This document is intended for developers who want to create complex experiments in virology domain. Using GOI enables them not only to exploit the power of modern object-oriented scripting language Jruby, but also to take advantage of the Grid environment.

This manual covers the topic of using Grid Operation Invoker API inside Jruby scripts (a.k.a experiments) in order to invoke remote operations on Grid Object Instances all over the world, regardless of the communication protocol. It explains how to create Grid Objects in three different manners and introduce code examples. It is assumed that GSEngine is already installed and configured (if not please refer to *Experiment Users' Manual* Section 5.1) and that developer is familiar with the concept of Grid Object Abstraction (see Section 2.5). Please remember that GOI requires EDG UI (see [EDG documentation](#)) to be installed in order to support job submission on EGEE infrastructure.

Grid Operation Invoker provides the uniform interface for creating Grid Objects. To fulfill its responsibilities GOI performs the following activities while creating Grid Object:

- querying Optimizer (see [D3.3] Section 3.6) for id of optimal Grid Object Instance of the class requested in the script (experiment).
- querying Registry (see [D3.3] Section 3.2) for technical information describing selected instance.
- loading appropriate technology adapter which creates Grid Object

Once created, Grid Object can be used just like any other JRuby object. It covers the burden associated with interfacing specific middleware and makes invoking remote operation identical to calling a method on a local object.

It is possible for developer to bypass listed steps if she/he wishes to use low-level API provided by adapters.

GOI provides uniform interface to invoke operations on Grid Objects Instances, that can be published using various middleware technologies, such as Web Services, WSRFs, jobs etc.

There are three possible ways of GOI usage:

- create Grid Object of a given class
- create Grid Object for a given Grid Object Instance
- create Grid Object using low level API by providing all necessary technical information

Whichever manner is used, the returned object is always an object representative of a piece of software deployed on a remote or local computational resource.

GObj API

The easiest way to create Grid Object representatives is to use the GObj factory class methods. First of all, developer must require necessary Ruby files. GObj class is the most essential and in most cases, excluding the third scenario, it is the only class that needs to be loaded explicit in the experiment source code. To do so, developer must include the following code:

```
require 'cyfronet/gridspace/goi/core/g_obj'
```

After that it is possible to create Grid Objects using GObj class methods: *create* and *create_instance*. The former method take the name of the Grid Object Class as an argument. Such invocation performs all three steps mentioned in the paragraph describing how GOI works. For instance, to create a representative for a Grid Object of class named *cyfronet.gridspace.gem.EchoService* the following code would be used:

```
echo1 = GObj.create('cyfronet.gridspace.gem.EchoService')
```

Now, let us do the same using the latter method, *create_instance*, which takes id the Grid Object Instance.

```
echo2 = GObj.create_instance(5)
```

In this case, querying Optimizer for an optimal instance is omitted, but developer must be sure that the id of desired Grid Object Instance, which is stored in the Registry, equals 5. Otherwise it is possible that later in experiment there will be an attempt to invoke operation which is not provided by the Grid Object and an error will be raised.

Adapters API

This API enables developer to create representatives for Grid Object Instances directly. In this case, developer must know the exact Ruby class that is capable for acting as a representative, as well as technology data describing the instance. Using adapter API requires in-depth understanding of GOI and involve much more effort than the *GOBJ* API.

Now let us create the same object using the low level API. Firstly, we must load the appropriate resource class. In this manual we will create a representative for Web Service, so we need *WsResource* class:

```
require 'cyfronet/gridspace/goi/adapters/ws_resource'
```

Next the technical information is needed. Below a Hash containing the full information about Grid Object Instance is defined.

```
techInfo = {'instId' => 5, 'name' => 'instance1',  
            'endpoint' => 'http://virolab.cyfronet.pl:18080/',  
            'type' => 'WS', 'wsType' => 'RPC',  
            'method#0' => 'echo', 'in#0#0' => 'echoString',  
            'out#0#0' => 'echoReturn',  
            'namespace' => 'http://virolab.cyfronet.pl/echo',  
            'codebase' => 'url'}
```

Please remember, that technology information is specific for every middleware. Finally, let us create the resource representing the EchoService:

```
echo3 = WsResource.new(techInfo)
```

Using Grid Object representatives

Representatives of the same Grid Object, can be used identically, albeit they were produced using distinguishable methods. Moreover, invoking operation on a representative is analogous to calling method on an ordinary JRuby object.

```
ordinary = String.new('I am local object')  
l = ordinary.length  
puts l  
msg1 = echo1.echo('I am easy to use!')  
puts msg1  
msg2 = echo2.echo('I am easy to use too!')  
puts msg2
```

```
msg3 =echo3.echo('So am I!')  
puts msg3
```

Choosing the appropriate API

As proved in this document, using different APIs involves various levels of knowledge and understanding of GOI. Although all methods for creating Grid Objects representatives provide the same functionality and produce the same result, they provide different non-functional capabilities. For instance, the *create* method is the most convenient to use and the most universal, while the *create_instance* can be used to ensure that a specific instance will be used, because of accounting issues, reliability and numerical quality of the software installed on a concrete node, etc.

In the end, the low level API can be used for testing new GEMs, before they will be registered in GRR, as well as using external Grid Objects Instances.

Sample experiment

Below there is full and runnable source code of a sample experiment using the API described in this document. A representative for EchoService is created, which reflects given message.

```
require 'cyfronet/gridspace/goi/core/g_obj'  
  
echo = GObj.create('cyfronet.gridspace.gem.EchoService')  
reflected = echo.echo('Hear me roar!')  
puts 'Reflected message: ' + reflected
```

Expected output for this experiment is:

```
Reflected message: Hear me roar!
```

For more sample scripts please refer to example experiments (Section 6).

6. EXAMPLE EXPERIMENTS

This section describes a set of experiments that could be used to test and learn the ViroLab Virtual Laboratory. Since the experiments presented here require the GSEngine we assume you have downloaded the latest release of this software package and you are familiar with the *Experiment Users' Manual*.

Each experiment subsection below gives information on the purpose of the experiment being described and how is it realized using the means of the Virtual Laboratory.

All the examples are compliant with GridSpace Engine version *0.3.0*. The package with the latest version of the experiments could be obtained from the GridSpace Engine development page: <http://virolab.cyfronet.pl/trac/vlruntime>. In fact, due to reasons of brevity, the explanations usually list just some parts of experiment plans (so you have to refer to the *SampleExperiments* package contents in order to get the whole experiment plan scripts).

Some of the experiment below runs on EGEE testbed [EGEE]. In order to execute these, one needs :

- a valid grid certificate,
- the EDG User Interface installation (with its executables in the \$PATH environmental variable).

Before an experiment is started the user needs to create his proxy certificate (by executing the *grid-proxy-init* command and provide the passphrase if requested). For more instructions on acquiring a grid certificate and using LCG job submission middleware, please refer to [LCG-2 User Guide](#) [LCG2].

6.1. ECHO

6.1.1. Short description

Functionality. This is the simplest experiment in this set. It produces a message to a remote *echo* server, which in turn responds with the same message. The returned message is printed out.

Realization.

1. The *Echo* server is available - it is a Web Service running on a remote machine that is able to reflect a received message
2. The server has a single *echo* operation and the server's endpoint and its operation signature is registered inside the Grid Resources Registry (GRR) - after that operation the server becomes a Grid Object instance
3. On the local side (that is, within the GSEngine where the experiment is being executed) the Grid Operation Invoker (GOI) serves as a *generic* client for remote Grid Object instances (including the ones with SOAP WS interface)
4. Using the specific libraries the experiment script instantiates a local representation of the remote *Echo* server (called a *stub*)

5. The local stub now has the same operations as the remote *Echo* server does, so it is just a matter of simple method invocation on the stub to get the work done.

6.1.2.Detailed code explanation

The entire code is written in the Ruby programming language and is executed with the JRuby interpreter.

```
require 'cyfronet/gridspace/goi/core/g_obj'
```

This includes the main part of the Grid Operation Invoker (GOI) to be used later on.

```
echo = GObj.create('cyfronet.gridspace.gem.EchoService')
```

Here is the main part - the Grid Object instantiation. Using a special **GObj** class and its **create** method one creates a local stub of the remote *Echo* server. The name used as a parameter is the name by which the *Echo* server is registered within the Grid Resources Registry (GRR). After the successful execution of this line of code, the **echo** variable should contain the local stub of the remote Grid Object along with all the operations it exposes.

```
reflected = echo.echo('Hear me roar!')
puts 'Reflected message: ' + reflected
```

One may see (the first line of the code snippet above) that the local **echo** stub of the *Echo* Grid Object is a regular representation of the server. The simple execution of **echo** method gives as the expected result that may be printed out for the user.

6.2. NUCLEOTIDE SEQUENCE

6.2.1.Short description

Functionality. This experiment shows a simple yet very useful data retrieval scenario. It uses remote access to a relational data base server to query it for data. The result data is printed out.

Realization.

1. The data is retrieved from the *virolab.cyfronet.pl* MySql server
2. The DB has a *test* database inside with read access for the user *testuser* (no password required)
3. The Data Access Client (DAC) on the GSEngine side is used to get a working connector to the remote database and the connector is used to perform querying

6.2.2.Detailed code explanation

The entire code is written in the Ruby programming language and is executed with the JRuby interpreter.

```
require 'cyfronet/gridspace/dac/DACConnectClass.rb'
```


This turns on the Data Access Client part of the GSEngine.

```
db = DACConnector.new
    ("mysql", "virolab.cyfronet.pl", "test", "testuser", "")
```

Instantiation. Here the developer passes to the *DACConnector* class constructor all the needed identification and authorization data:

- data source type (here: *mysql* rdbms)
- data source location (here: the DNS name, assuming the standard mysql port)
- data base name inside the designated server (here: *test*)
- login and password to access the server (here: non-anonymous *testuser* with no password is used)

```
db.executeQuery("select column_name from information_schema.columns
    where table_name='nt_sequence'")
# (...)
result = dbzeus.executeQuery("select nucleotides from nt_sequence
    where patient_ii=6;")
```

Here are examples of using the created connector to ask the server some typical SQL queries. The first one shows some structure information of a chosen table, the other one gets the actual data from the table. This is the pure SQL so you probably has a good idea what could be done here.

One may see (the first line of the code snippet above) that the local **echo** stub of the *Echo* Grid Object is a regular representation of the server. The simple execution of **echo** method gives as the expected result that may be printed out for the user.

```
result.each_index {|ind|
    puts ind.to_s + ": " + result.flatten[ind]
}
```

Finally, the query result is returned in two-dimensional Ruby array (SQL-type of data source usually return the query result in a table) - so you may use internal Ruby constructs to analyze, parse, transform the obtained data.

6.3. DATA ACCESS

6.3.1. Short description

Functionality. Similar to the *Nucleotide sequence* scenario above, this one is about data retrieval. The main difference is, however, the fact that now we get data from the ViroLab **Data Access Service (DAS)** that federates many data from various institutes in Europe and presents them as unified, virtual data base. The Data Access Client (DAC) that is shipped with GSEngine is able to contact this type of data source as well.

Realization. Checking the source code of this example you will quickly find out that it is very similar to the *Nucleotide sequence* retrieval scenario - and this is the main idea behind DAC: to have multiple types of data sources accessible for programmers in a unified way.

6.3.2.Detailed code explanation

The entire code is written in the Ruby programming language and is executed with the JRuby interpreter.

```
require 'cyfronet/gridspace/dac/DACConnectClass.rb'
```

This *turns on* the Data Access Client part of the GSEngine.

```
dbhls = DACConnector.new  
("das", "angelina.hlrs.de:8080/wsrf/services/DataResourceService", ""  
, "", "")
```

Instantiation. Here the developer passes to the *DACConnector* class constructor all the needed identification and authorization data:

- data source type (here: *das* Data Access Service type)
- data source location (here: the DNS name)
- other parameters are not needed right now (in the case of this data, no authorization mechanism is required)

```
dbhls.describeDataSource()  
dbhls.describeTable("viralload")
```

Those are two new methods of DAC API that allow to obtain some structure, meta-data on the data source we access. The *describeDataSource* shows some general information while the *describeTable* returns more specific data on a chosen table (mainly the column names).

```
query_result=dbhls.executeQuery(  
  "select IDpatient,ViralLoad from viralload limit 20")  
query_result.each {|row|  
  p row  
}
```

The target query is executed here to get some examples of viral load indicators (this data base sample is completely anonymized, so no mapping of the viral load level to specific person could be made). As the query result is returned in two-dimensional Ruby array one uses internal Ruby constructs print it out as the main experiment result.

6.4. ALIGNMENT

6.4.1.Short description

Functionality. This experiment aligns the nucleotides sequence using a RegaDB tool published with WTS (*Witty Services* [WTS]) middleware. The nucleotide sequence is hardcoded as *nt_seq* variable. The obtained alignment is printed out.

Realization.

1. The *regadb.RegAlignment* is a WTS service. It can be accessed with the *wtsc_client* Java library.
2. This service has a single *align* operation which takes nucleotide sequence and region as input parameters and returns alignment.
3. The service is registered in GRR, which contains all technical information about this Grid Object Instance.
4. GOI support WTS technology, therefore representative for the WTS service can be created using GObj API.
5. A representative, which has the *align* operation, is created using *WtsAdapter* class.
6. The *align* method is called on a representative that delegates the computation to WTS service.

6.4.2.Detailed code explanation

The entire code is written in the JRuby programming language and is executed with the JRuby interpreter.

```
require 'cyfronet/gridspace/goi/core/g_obj'
```

This *includes* the main part of the Grid Operation Invoker (GOI).

```
alignTool = GObj.create('regadb.RegAlignment')
```

Using a special *GObj* class and its *create* method one creates a representative for a WTS service. The name used as a parameter is the name by which the WTS service is registered within the Grid Resources Registry (GRR). After the successful execution of this line of code, the *alignTool* variable represent the Grid Object Instance along with all the operations it exposes.

```
result = alignTool.align(nt_seq, 'PRO')
```

As shown in the code snippet above, invocation of remote operation is simply calling a method on a representative. Now let us create the same representative using the *create_instance* method. The difference is only the input parameter, which is now an instance id, instead of a Grid Object class name. Invocation of remote operation is the same as in the previous case. See Section 2.5 for explanation of different levels of abstractions of Grid Objects.

```
alignTool2 = GObj.create_instance(9)
result = alignTool2.align(nt_seq, 'PRO')
```

6.5. LCG TESTBED TEST EXPERIMENT

In order to run this experiment you need a valid grid certificate for the EGEE testbed (see the note in the introduction to the example experiment section).

6.5.1.Short description

Functionality. This experiment executes a bash shell command on the EGEE infrastructure. It submits it as a job using EDG User Interface, which must be installed on the same machine as GSEngine. The command that will be executed must be in a \$PATH variable on the EGEE testbed machine or a full path to the executable should be provided.

Realization.

1. EGEE infrastructure offers a great amount of computational resources with standard Linux distributions.
2. User running the script has a valid proxy certificate, therefore can submit jobs.
3. Shell command is specified in the experiment.
4. While executing the experiment, GOI creates a JDL file describing the job and submit it through EDG UI.
5. Output of the shell command is downloaded and printed.

6.5.2.Detailed code explanation

The entire code is written in the Ruby programming language and is executed with the JRuby interpreter.

```
require 'cyfronet/gridspace/goi/core/g_obj'
```

This includes the main part of the Grid Operation Invoker (GOI).

```
sample = GObj.create('cyfronet.gridspace.gems.lcg.LcgSample')
```

A representative is created with *create* class method of *GObj*. It enables execution a bash shell command on the testbed by calling the *execute_shell_cmd* method.

```
result = sample.execute_shell_cmd('ping -c 5 $1 | grep -v 64', 'virolab.cyfronet.pl')
```

Output of the shell command, which in this case is *ping -c 5 virolab.cyfronet.pl | grep -v 64*, is assigned to the *result* variable.

```
puts result
```

Result is printed. If job completed successfully similar output should be displayed:

```
PING virolab1.cyfronet.pl (149.156.9.2) 56(84) bytes of data.  
--- virolab1.cyfronet.pl ping statistics ---  
5 packets transmitted, 5 received, 0% packet loss, time 4004ms  
rtt min/avg/max/mdev = 0.942/0.977/1.008/0.021 ms, pipe 2
```


7. DATA ACCESS SERVICES PROTOTYPE MANUAL

7.1. INTRODUCTION

The complexity of data management on a Grid arises from the scale, dynamism, autonomy, heterogeneity, and distribution of resources. To conceal these complexities of the underlying infrastructure, a sophisticated management system needs to be developed, which ensures that the resources appear transparent to their users. This could be achieved by hiding the different data resources and their internals behind a layer of virtualization services that guarantees data access in a consistent, data resource-independent way.

The Data Access Services (DAS) consist of a set of such virtualization services that provide interfaces for querying, updating, transforming and delivering data to various data resources via standard web services. The current version, which is described in detail by this document, allows users to collect any kind of (meta) information of an underlying resource including the used data management technology (database system), schema specification, and availability of the resource itself. This information can then be used by clients to specify a query that accesses the corresponding database similarly as they would proceed within their local environment. Not only single data resources can be accessed but also queries to multiple databases can be simultaneously performed using specific interfaces provided by the DAS. There are currently some restrictions while sending requests to more than one resource but the DAS is in constant development and the functionalities are permanently extended.

This documentation is intended to support other developers working in the ViroLab project to connect their components and applications with the DAS in order to get an unified entry point for distributed data access. It describes the main steps to set up and use the services developed and offers a deeper insight into the implementation structure of the first prototype.

7.1.1. References and Source Code

The current DAS JavaDoc source code documentation can be found at the following web address:

<http://www.hlr.de/organization/ds/projects/virolab/dasapi/index.html>

The latest source code of the DAS and appropriate test applications can be downloaded from the ViroLab SVN hosted by University of Stuttgart:

https://svn.gforge.hlr.de/svn/virolab/trunk/modules/DataAccess/ViroLab_v.0.2/

The access to the repository is restricted only to members of the ViroLab consortium. An open source release of the DAS is planned to be provided at the end of the project.

7.2. PROTOTYPE USAGE

In order to deal with the DAS, there are some technical prerequisites for components and applications regarding hardware and software. In a typical Grid

infrastructure, the DAS is usually installed at one central location where it can be accessed from any external machine. This section gives an overview on how to interact with the DAS from a particular machine and also explains basic functionalities and some advanced features.

7.2.1. Running the Prototype

7.2.1.1. Operating Requirements

To communicate with the DAS, which is based on standard web service interfaces, an user - could be an application or a component - has to ensure that the latest interface description (WSDL specification) is used. These descriptions are normally used to automatically generate so-called client stubs, a set of Java classes, which allow the interaction with the corresponding services in a smooth way.

The following sections shall provide setup information that is required to use the generated classes and to finally interact with the DAS.

Local Hardware Requirements

The hardware requirements for using the client stubs are not very high as any today's computer hardware able to run Java 5 software should be sufficient. An Internet connection is needed in order to call the DAS. As the services are quite communication dependent, the optimal configuration will have a low-latency Internet connection.

Local Software Requirements

The software may run on almost every 32-bit operating system like Windows XP, Vista or Server 2003 as well as on different Linux distributions like Ubuntu or SuSe Linux. It only requires a running installation of the Java 5 Runtime Environment, which can be directly downloaded free of charge from SUN's website (<http://java.sun.com/j2se/1.5.0/>).

Furthermore, to ensure a proper working with the DAS, a number of third-party Java libraries (compiled and zipped in *jar* files) is needed:

- activation.jar (<http://java.sun.com/products/javabeans/jaf/index.jsp>)
- addressing-1.0.jar (<http://jakarta.apache.org/addressing/>)
- axis.jar (<http://ws.apache.org/axis/>)
- commons-discovery.jar (<http://jakarta.apache.org/commons/discovery/>)
- commons-logging.jar (<http://jakarta.apache.org/commons/logging/>)
- jaxrpc.jar (<http://ws.apache.org/axis/>)
- log4j.jar (<http://jakarta.apache.org/log4j/>)
- saaj.jar (<http://ws.apache.org/axis/>)
- servlet.jar (<http://jakarta.apache.org/tomcat/>)
- wsdl4j.jar (<http://sourceforge.net/projects/wsdl4j/>)
- wss4j.jar (<http://ws.apache.org/wss4j/>)
- xalan.jar (<http://xml.apache.org/xalan-j/>)
- xercesImpl.jar (<http://xml.apache.org/xerces2-j/>)
- xml-apis.jar (<http://xml.apache.org/xerces2-j/>)

- xmlsec.jar (<http://xml.apache.org/security/>)

Principally, there are two ways to get the latest client stubs. Firstly, one can download them from the SVN repository (see part 1.1) or secondly one can create them dynamically everytime before using the DAS. The second option would be more flexible but requires additional effort from the developer. He needs to modify the source code to create the Java classes on-the-fly during program execution. The developer can for example use the Axis tool *WSDL2JAVA* available within the Axis library earlier mentioned or he can use the *CreateDASStubs* class shipped with the DAS, which is also available within the SVN repository.

Grid infrastructure requirements

The main requirement for using the DAS is that it has to be operational and that it publishes its web service interfaces (WSDL) externally. Being unavailable, the user's requests will be rejected by the DAS.

If additional security between a client and the DAS is required due to some specific reasons, there is an optional feature that allows secure communication with the services - the messages are encrypted and signed. To turn these security functionalities on, the provider of the DAS needs to set some property values within the DAS configuration files and a valid X.509 certificate needs to be in place as well. The client also needs one of these certificates in order to encrypt and decrypt the corresponding messages. Both certificates must be trusted by the same certificate authority (CA), otherwise the integrity of the exchanged messages is not guaranteed.

Further information on security principles and mechanisms can be found at the following website:

<http://gdp.globus.org/gt4-tutorial/multiplehtml/pt03.html>

7.2.1.2.Step-by-Step User Setup

Step 1: Download the precompiled stubs from the SVN or create them on-the-fly using the following WSDL file:

<http://angelina.hlr.de:8080/wsrf/services/DataAccessService?wsdl>

Step 2: Make sure to have a proper installation of the Java 5 Runtime Environment on the local site (simply search for java executable). If not, download and install it properly.

Step 3: Get all the third-party libraries of the software. Please check with *Local Software Requirements* section where to download these files.

Step 4: Use the stubs together with all the other libraries to include DAS interactions within your own source code. One can proceed like the following example where a connection to the DAS is established and different resource IDs are requested and simply printed out.

```
DataAccessServiceAddressingLocator locator = new DataAccessServiceAddressingLocator();
EndpointReferenceType endpoint = new EndpointReferenceType();
try
{
```



```

String serviceURI = "http://angelina.hlr.de:8080/wsrf/services/DataAccessService";
String ogsaService = "http://csharp.hlr.de:9090/wsrf/services/hospitals";
endpoint.setAddress(new Address(serviceURI));
DataAccessPortType dataService = locator.getDataAccessPortTypePort(endpoint);
if(dataService != null)
{
    System.out.println("Connecting to service at: "+serviceURI);
    DataResourceList result = dataService.getAvailableDataResources(ogsaService);
    if(result != null)
    {
        for(int i = 0; i < result.getResources().length; i++)
        {
            ResourceParams currentResource = result.getResources(i);
            if(currentResource != null)
            {
                System.out.println("Found resource: "+currentResource.getResourceID());
            }
        }
    }
}
else
{
    System.out.println("Cannot connect to service at: "+serviceURI);
}
}
catch(RemoteException e)
{
    e.printStackTrace();
}

```

Step 5: Compile the source code and execute the program.

7.2.2. Basic Operations

The DAS currently offers different functionalities for querying distributed data resources. The basic features allow the interaction with underlying databases in a common way but also provide specific methods such as distributed queries, download of publicly available rule sets, and more. Details on the interfaces currently available are described on the following website:

<http://www.hlr.de/organization/ds/projects/virolab/dasapi/index.html>

In order to give users an idea of how to use the basic functionalities and also in which order these operations must be invoked, the following parts shall simply list the single methods and explain their basic features. For more details on the implementation, please refer to section 7.5.

Part 1 will explain the main interfaces for accessing remote data resources. Part 2 will concentrate on the submission of distributed queries to various resources concurrently. Finally, part 3 presents one specifically implemented routine to download publicly available rule sets while part 4 describes the principle how to store specific application data.

Part 1: Connecting to and querying from remote databases

- Initialize the corresponding service and resource. The `InitParams` argument of the method requires two parameters, the service URL and the abstract resource ID (typically an unique name).

`boolean init(InitParams params)` where `params` is used the following way:

```
InitParams initParams = new InitParams();
initParams.setServiceLocation("http://csharp.hlrs.de:9090/wsrf/services/hospitals");
initParams.setResourceID("ROME");
```

- Collect the data resource information of a particular resource. This information includes the database technology (e.g. MySQL, Postgres) and a list of keywords indicating available tables - this is currently a prerequisite, refer to section 7.2.4.

```
DataResourceInformation getDataResourceInformation(String resourceID) where
resourceID is "ROME"
```

- Query for the schema definition. This principal is currently based on plain SQL statements. To request schemes from different databases, please refer to the according SQL statements.

```
DataResult getDataFromQuery("Describe currentTableName") -> currentTableName
obtained from previous request
```

- Perform a concrete query. To query different databases, please refer to the according SQL statements.

```
DataResult getDataFromQuery("Select * From currentTableName") or whatever you want...
```

Part 2: Submitting distributed queries

Submitting a distributed query to multiple resources is currently restricted to one specific method:

```
DataResult submitDistributedQuery(java.lang.String queryString)
```

This method requires a real SQL query as input and then automatically performs the following actions:

- Checks which resources are available
- Requests data resource information of each available resource
- Compares the keywords to the tables given by the query
- If corresponding resources are found, each of them is queried using the input statement
- Finally, the results are merged and the resource ID is added to each new data row as an additional primary key

Part 3: Requesting publicly available rule sets

To deal with such rule sets, one specific method was implemented that handles all relevant activities (download, version checking, submission to requester) based on the requirements of the DRS application.

The method can be directly called by the DRS or any other component. The only necessary argument to be passed to the function is the type of rule sets wanted while the version argument is optional. These arguments are simple string types and must have the following input values.

- String type: *ANRS|HIVDB|REGA*
- String version: e.g. *4.1.0* (optional)

The result can either be a notification message (String) saying that the current rule set version is up-to-date ('Your current version is up-to-date'), or an XML document (String) including the latest rule set.

To point out the functionality, there is a simple test application named *AccessRulesetsClient*. It is a Java Swing application, which allows the user to input the relevant information (required by the above explained method *RequestRuleSets*) and then starts to process this input by downloading a new version or to find out that the current one is up-to-date. The test application is available in the SVN at

https://svn.gforge.hlr.de/svn/virolab/trunk/modules/DataAccess/ViroLab_v.0.2/TestClients.

The application can be started by typing the following into a command line:
ant runRuleSetClient

The following window appears on the desktop.

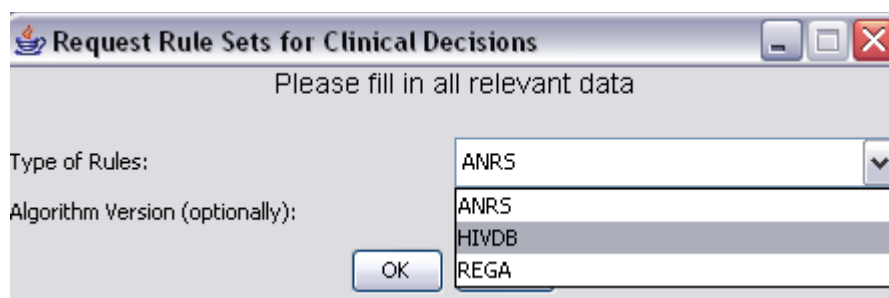


Figure -44: Selecting the type of rule sets

When the version is up-to-date, the following notification will be shown.

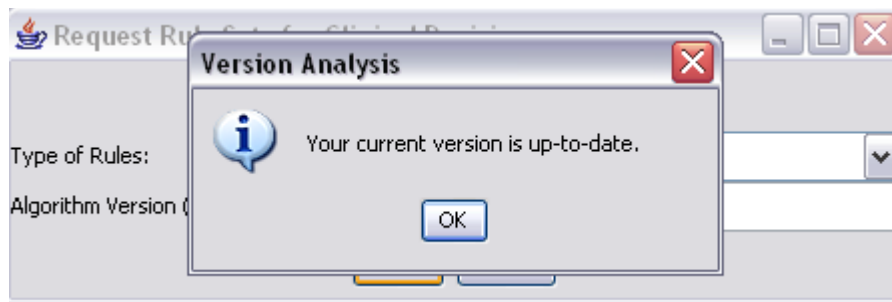


Figure -45: Received notification message

If a newer rule set version is available, a message box will pop up and ask whether to save the newer version or not (it will be saved as an XML file).

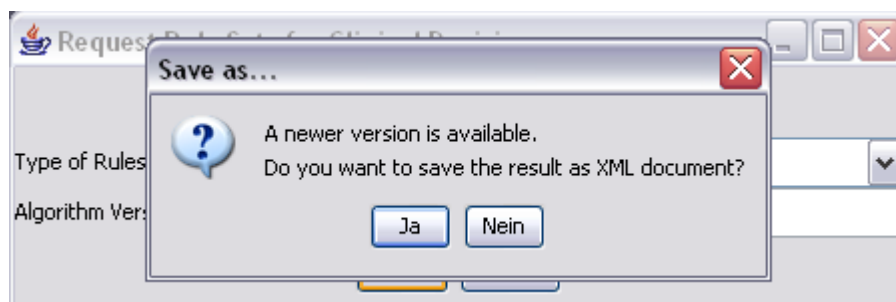


Figure -46: Save the newly available rule sets

Part 4: Storing relevant application data in a specific database

In order to track specific experiment results and user inputs related to these experiments, every application should store their input as well as output data in a corresponding database. In the current version, the DAS provides one particular method that allows to store appropriate application data:

```
boolean storeApplicationData(AppStoringParams appStoringparams)
```

It should be called whenever an application wants to store its data. The input argument required by the function consists of two parameters, which should be of the following input format:

- AppType appType: *DRS|RegaAlignment|RegaHIVSubType*
- String[] values: an array of values to be stored

Internally, the method firstly analyzes the application type and based on type value, the particular application-dependent function is called and all passed values are stored in the corresponding database (refer to section 7.5.3).

7.2.3.Advanced Features

One advanced feature in the current release of the DAS is the integration of the services together with the authorization principle of Shibboleth. It is in an early stage of development so that the final user authorization is more or less

implemented in a static way, meaning that as long as a user carries a specific attribute such as a role, institution, etc. he might be allowed to access a particular resource. There is not a real dynamic procedure for access control available yet but the future releases of the DAS will also contain such a dynamic authorization model where a so-called Policy Decision Point (PDP) can be asked whether a user has the necessary attributes to enable his access to a particular resource. For more information on Shibboleth and the principle used in ViroLab, please refer to the deliverables D2.2 and D3.2 submitted in month 12 and 9 respectively.

7.2.4. Known Problems

The current version of the DAS software is in its first release but most of the basic functionality is considered stable. There are actually no known bugs but this might change while the software is in daily usage. Nevertheless, the DAS has got some limitations and yet unimplemented features, which shall be roughly listed and explained, and which will be covered by future releases.

1. Yet unimplemented features:

- Dynamic user authorization: Using a PDP and user-defined policies (usually defined and managed by data providers themselves) to control the access to different resources
- Meta query language to simplify communication with services: Developing a higher-level language that allows application users as well as developers to query resources without using real SQL statements but rather common well-known terms
- Parallelization of distributed queries: Develop an algorithm that enables parallel processing of a distributed query – currently done in a sequential order - to increase performance and reliability of user queries
- Automatic registration of newly available resources: Design a wizard for data providers that facilitates automatic registration of their resources at the DAS
- Functionalities that enable easy and efficient requesting of schema specifications
- Application-specific transformations that transform data or data formats for specific needs of ViroLab applications

2. Current limitations

- Submission of distributed queries:
 - All tables must have the same schema
 - The database technologies used should be the same
 - The keywords identifying the content of a data resource must be equal to the database table names (requirement for OGSA-DAI only)
 - The results contain a new primary column: the resource ID

- Schema retrieval: Schema specifications can only be requested using corresponding SQL statements like *Describe table* for MySQL databases
- Authorization: Authorisation is more or less static, granting either full access or no access to all service methods (access denied!)
- Query language: DAS interfaces must be queried using concrete SQL statements instead of an abstract query language
- Application data storage: The current implementation is limited to one specific ViroLab application – Drug Ranking System (DRS) – but further applications will follow

7.3. INTERFACE REFERENCE GUIDE

Since the current release of the DAS provides standard web service interfaces, one typically needs a client to interactively test its functionality. Therefore, a distributed database browser was implemented, which allows users not only to browse the resource's content like its schema and data but also to manipulate its entries.

The GUI of the 'Distributed Database Browser', which can be seen in Figure -47, is a simple but very helpful application to quickly overview available resources and their contents. It presents on one screen all information about the selected resource including available tables, table schemes, and table contents. One can perform specific selections on the tables and if a user has permission to insert, update, or delete data sets, this can also be directly done via this application.

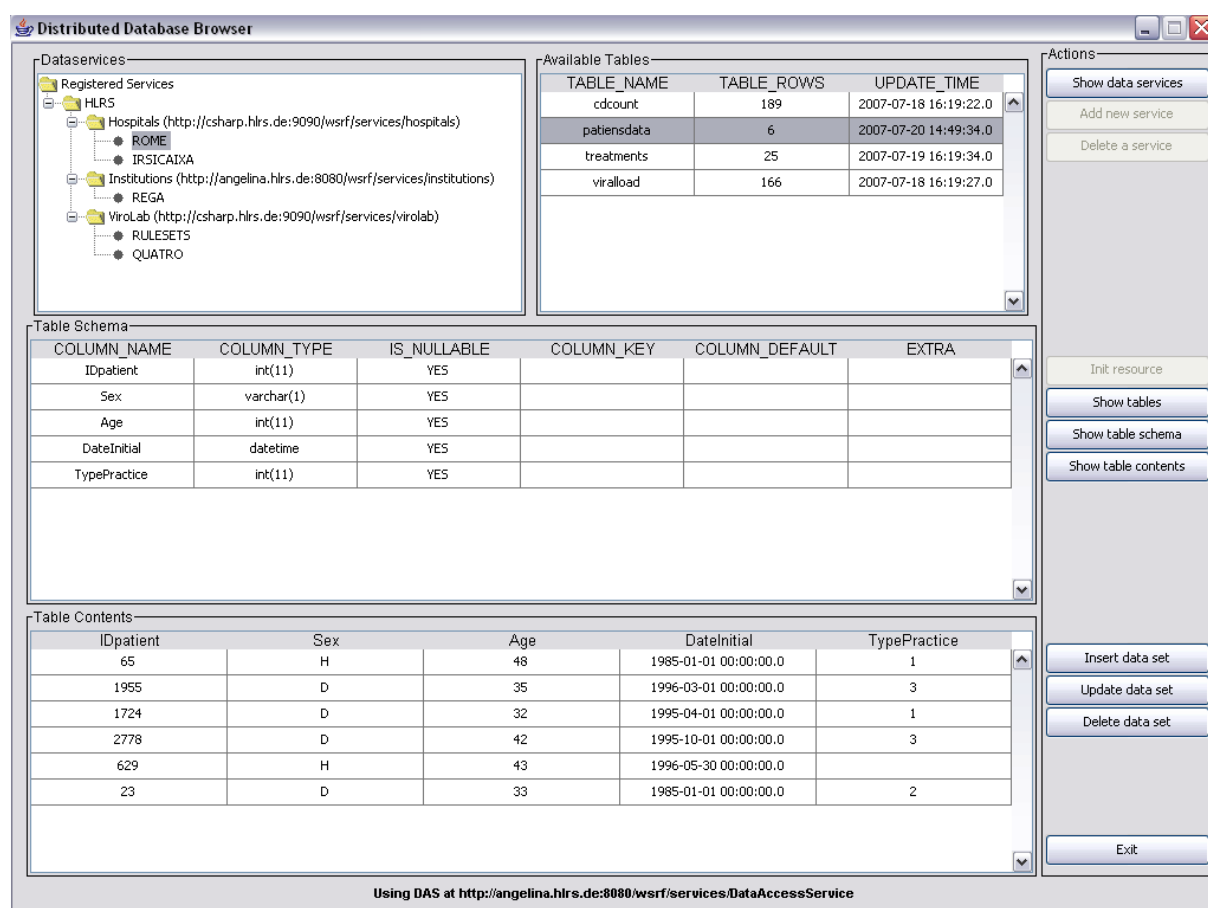


Figure -47: The main ‘Distributed Database Browser’ window

In order to facilitate the work with the application, the main functionalities shall be roughly described:

- *Show data services*: Visualizes all available data services and their corresponding resources based on the information in the central data service repository (see upper left part)
- *Add new service*: Registers a new data service in the central repository used by the DAS – see Figure -48
- *Delete a service*: Deletes a data service from the central repository

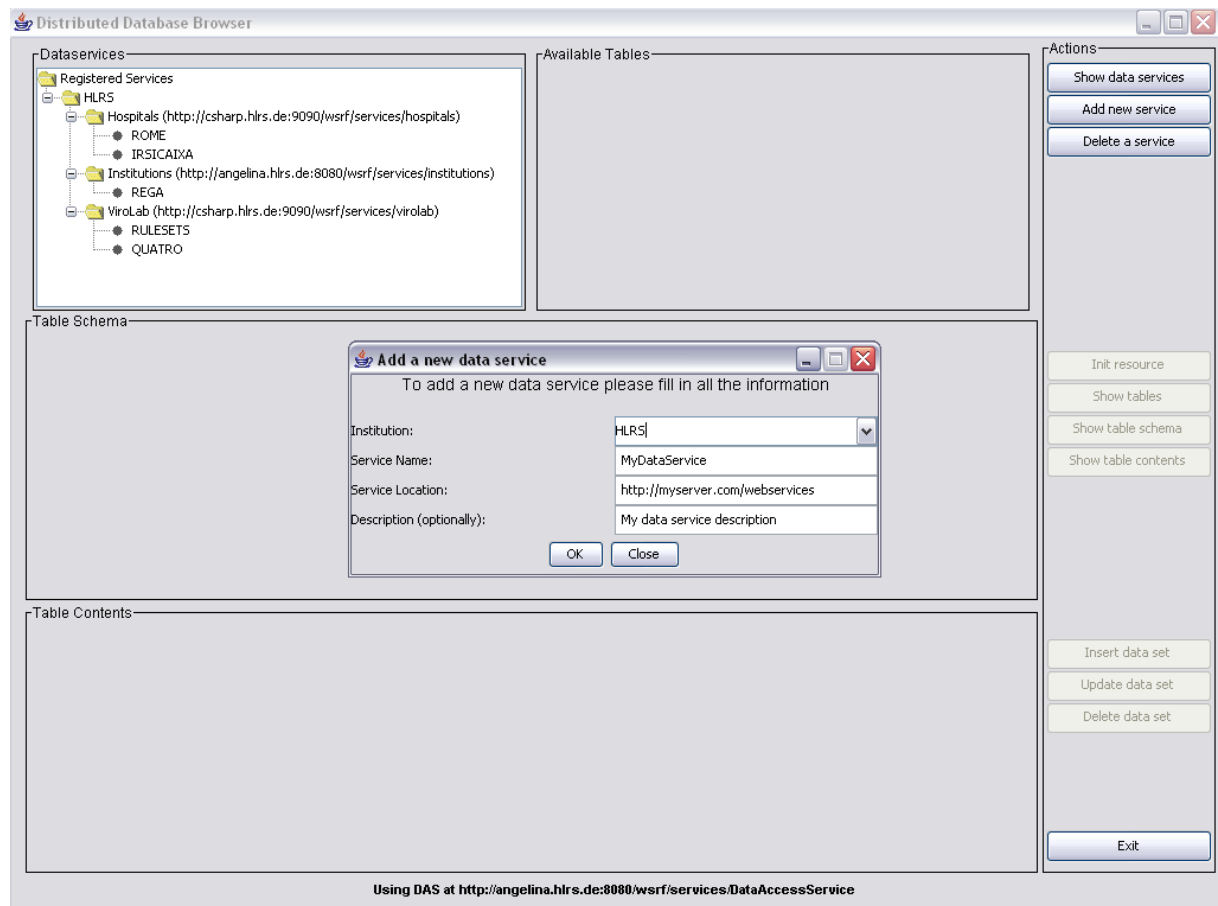


Figure -48: Pop-up window for adding a new data service instance

- *Init resource*: Initializes the selected resource according to the service URL provided
- *Show tables*: Displays the available tables including the number of rows and the latest update time (upper right part)
- *Show table schema*: Displays the table schema (central part).
- *Show table contents*: Displays the table contents based on the query provided (lower central part)
- *Insert data set*: Adds a new data set to the current table (if permitted)
- *Update data set*: Updates a selected data set (if permitted) – see Figure -49
- *Delete data set*: Deletes a selected data set (if permitted)
- *Exit*: Closes the application

The application is currently in an early stage of development so that not all planned functionalities of the DAS have been implemented so far. A new prototype is considered for the end of 2007, which then will include also the capabilities of submitting queries to multiple resources at the same time.

In parallel, a second user interface is developed that almost provides the same functionalities but implemented as a portlet for the project portal, which is based

on the Google Web Toolkit (GWT) instead of being a stand-alone Java application.

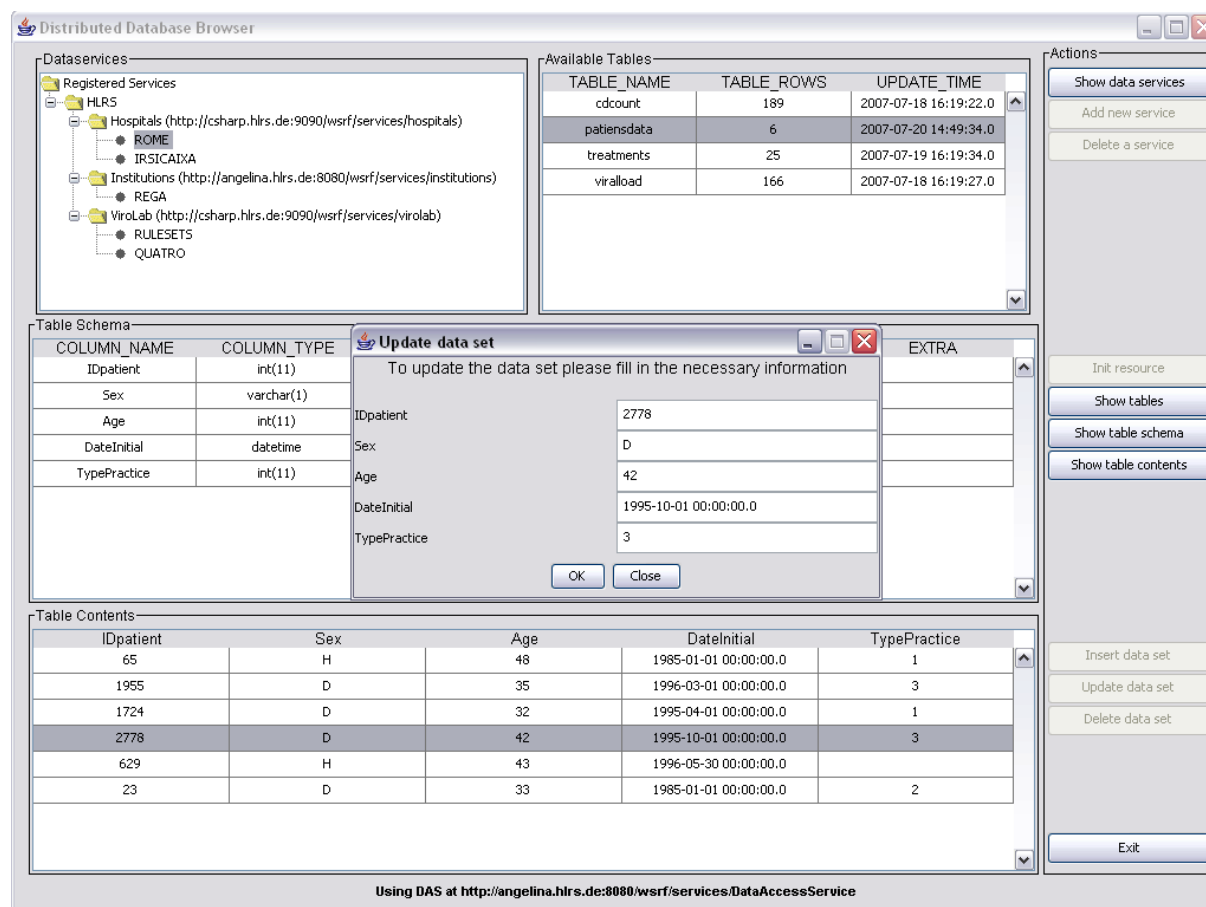


Figure -49: Pop-up window for updating a selected data set

7.4. TROUBLESHOOTING Q&A

Q: Upon submitting a DAS query, the system responds with an error saying that it is unable to connect to the DAS (*ConnectException – connection refused*).

A: The DAS may be down for maintenance or you may be experiencing network problems. If the situation persists, please contact *Matthias Assel* (assel@hls.de).

Q: The Axis engine reports an error that it could not find the target service to invoke.

A: Check the service URL you are using. This error typically indicates a malformed or incorrect URL.

Q: The DAS complains with an exception that the '*Passed argument cannot be null*' (*IllegalArgumentException*).

A: While calling any of the interfaces provided, please make sure that none of the arguments passed is null.

Q: The DAS complains with an exception that the '*Passed argument cannot be an empty string*' (*IllegalArgumentException*).

A: While calling any of the interfaces provided, please make sure that none of the string arguments passed is empty.

Q: The DAS answers with the exception '*You do not have the permission to perform this action on the resource*' (*DASException*).

A: You are actually not authorized to perform the chosen action on the corresponding resource. This may have two reasons. Firstly, the data provider has denied access to his resources for particular users due to some personal decisions (-> contact the responsible data provider), or secondly your institution is not well prepared for the ViroLab security infrastructure (-> contact your local responsibility).

Q: While executing a query on a particular resource, the DAS throws an exception saying that it is '*Unable to perform the query on the resource*' (*DASException*).

A: This error might occur basically if a query is incorrect or malformed and cannot be interpreted by the corresponding data resource technology. Please check your statement carefully and try to resubmit the query. In case your queries are correct but you are still getting the error, please contact *Matthias Assel* (assel@hlrs.de) who will be able to check the logs for further information.

7.5. IMPLEMENTATION STRUCTURE

7.5.1. Product Use Cases

The DAS is designed as a set of services that virtualize different underlying resources so that users and applications can access them in a transparent way. Figure -50 depicts multiple requesters using different services provided by the DAS in order to deal with several databases as if they were one large single data resource.

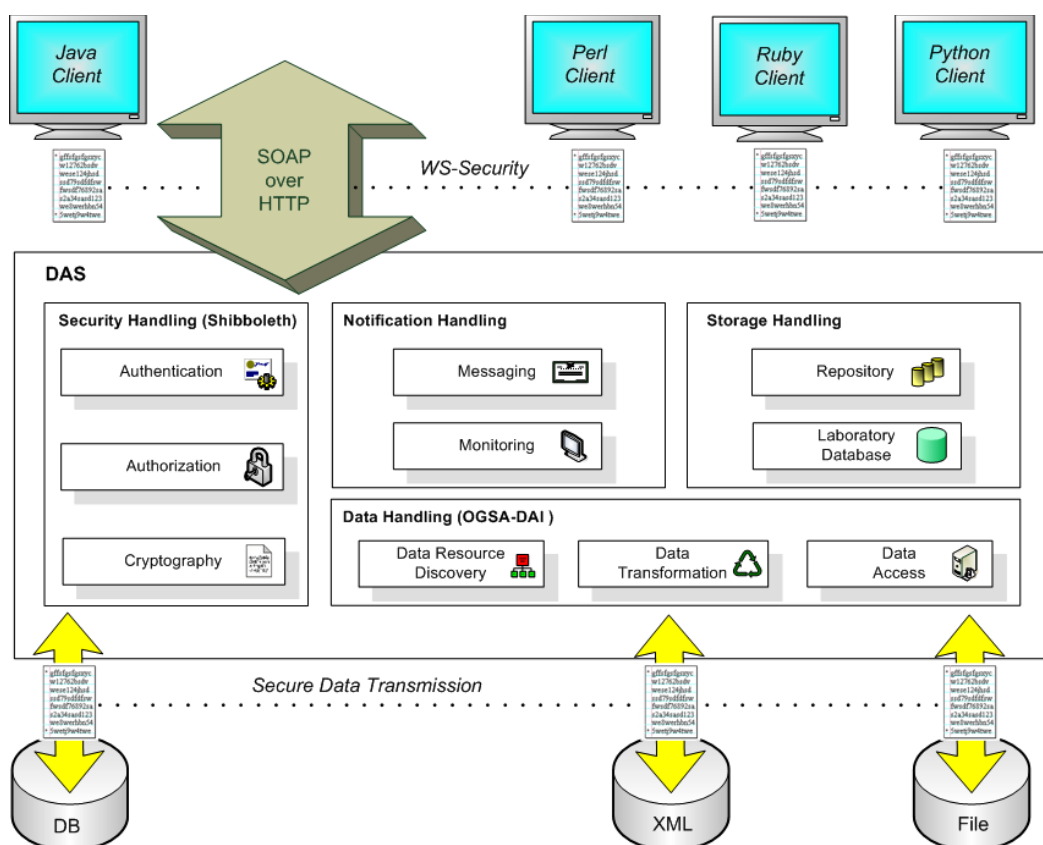


Figure -50: General architectural overview of the DAS

A typical use case explaining the core functionalities of the services is shown in Figure -51. Each single step - starting with the user's request up to the response sent back by the DAS - is highlighted within this chain by one specific block. Basically, all the components play a particular role within this workflow and for each of them different interfaces need to be provided and implemented. For pure data access, the functionalities provided by the OGSA-DAI toolkit are sufficient, but additional effort on security, mapping of user statements into resource-dependent statements as well as transforming query results into application-readable statements, is needed.

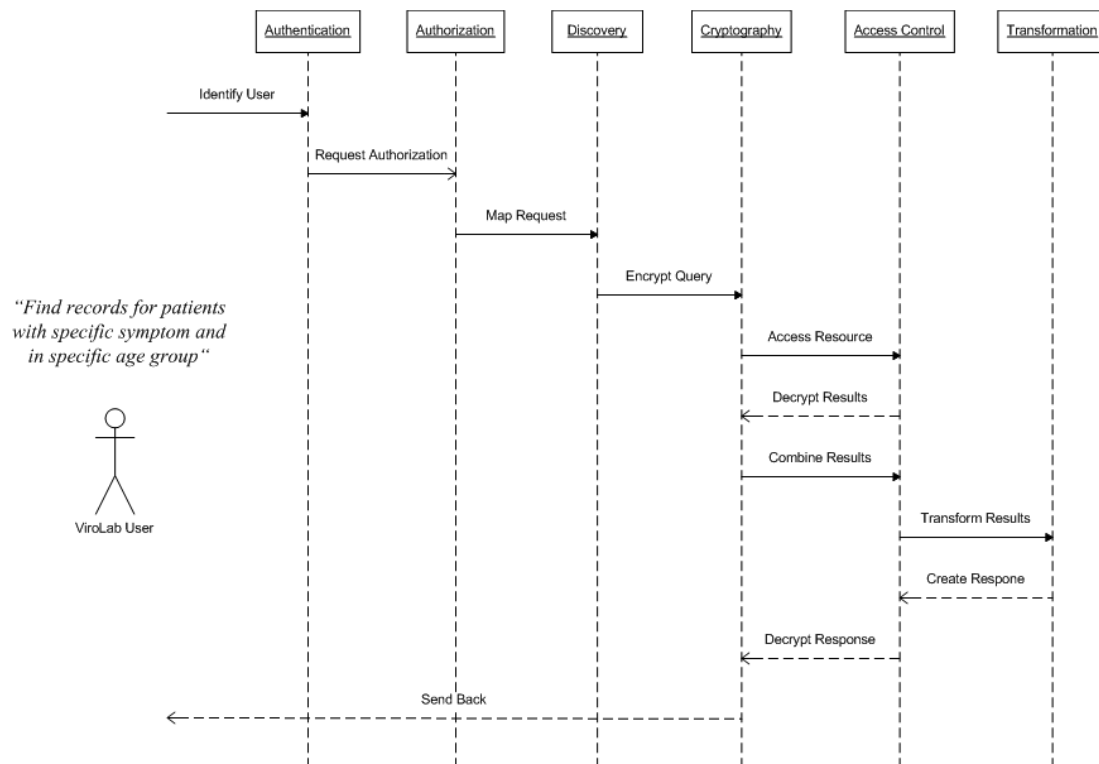


Figure -51: A typical use case within the ViroLab scenario

7.5.2.Product Component Model

An overview on the main components of the DAS and their dependencies among each other is visualized in Figure -52.

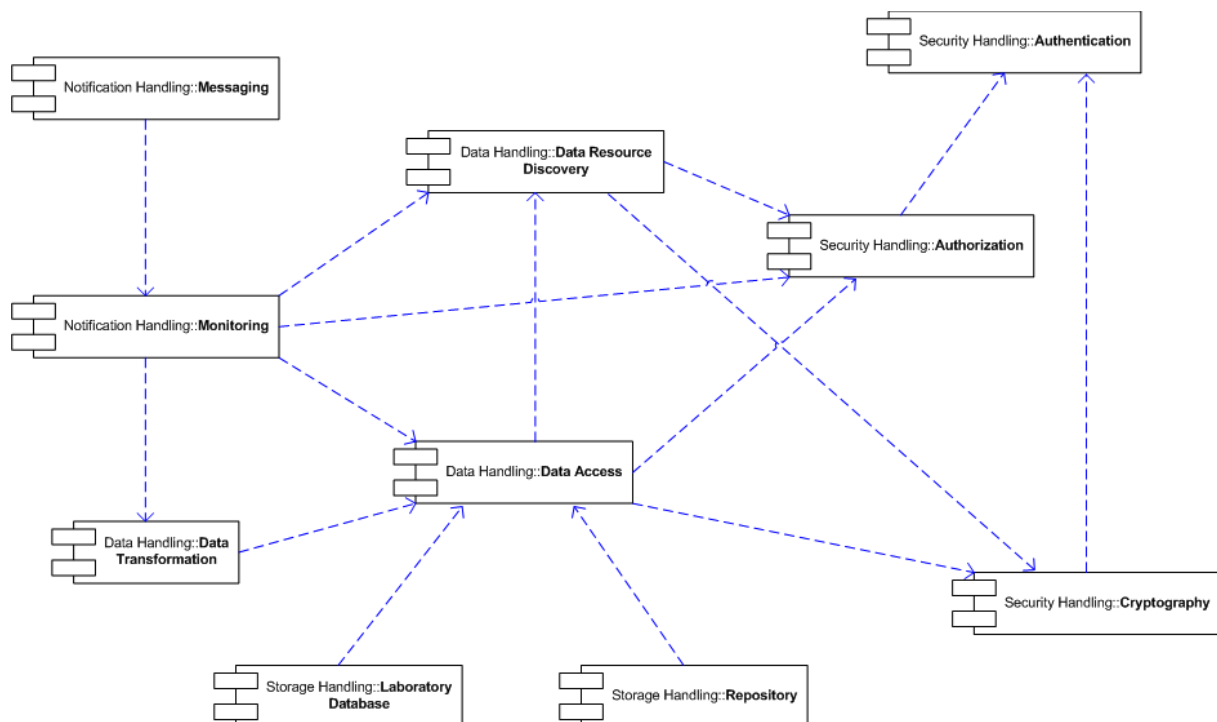


Figure -52: Main components of DAS

- *Authentication*: The authentication module is responsible for the identification of a user based on his credentials
- *Authorization*: The authorization interface decides whether a user is authorized to perform a certain task by mapping user attributes on data handling activities and resources
- *Cryptography*: The service provides capabilities for decrypting incoming and encrypting outgoing messages to ensure secure transmission between different endpoints
- *Data Resource Discovery*: The discovery service virtualizes the location of data resources by mapping common language terms onto data resource-dependent statements
- *Data Access*: The data access infrastructure is the most important part of the overall system. It provides interfaces to access different types of resources. Its main functionalities are based on the OGSA-DAI toolkit.
- *Data Transformation*: The transformation service provides methods for dynamically adding new transformation schemes in order to change the output format conceptual for a user application
- *Messaging*: The messaging subcomponent of the notification handling infrastructure contains mechanisms for publishing, subscribing to, and managing subscription to notifications about single events or families of interest
- *Monitoring*: The monitoring service is responsible for recording all transactions that occur inside the data access subsystem
- *Repository*: The repository will be used for storing any kind of intermediate data
- *Laboratory Database*: The laboratory database acts as a long time storage with a relatively short access time and can be used also by other components of the ViroLab infrastructure like for example the Provenance system

7.5.3.Detailed Implementation Model

Since the DAS is based on an service-oriented architecture and its functionalities are provided as standard web service interfaces, the focus for describing the implementation model lies on one specific service implementation class ('*DataAccessServiceImpl*' – see API description for further details), which includes almost two-thirds of the main interfaces. For more details on the design and implementation of single DAS components, please refer to deliverable D3.3.

The interfaces can be summarized into two main parts. The first part contains standard interfaces for querying remote databases including one specific interface for submitting distributed queries, whereas the second part includes particular methods for requesting publicly available rule sets and for storing application-dependent data.

Part 1: Connecting to and querying from remote databases

Most of the interfaces provided are directly connected with corresponding OGSA-DAI interfaces. Figure -53 depicts the specific use case where a user wants to invoke a query using the interfaces of the DAS. On the right side of the picture, the involved components of OGSA-DAI and their interactions are listed. Based on

the request, different activities are performed after an authorization mechanism has granted access to them.

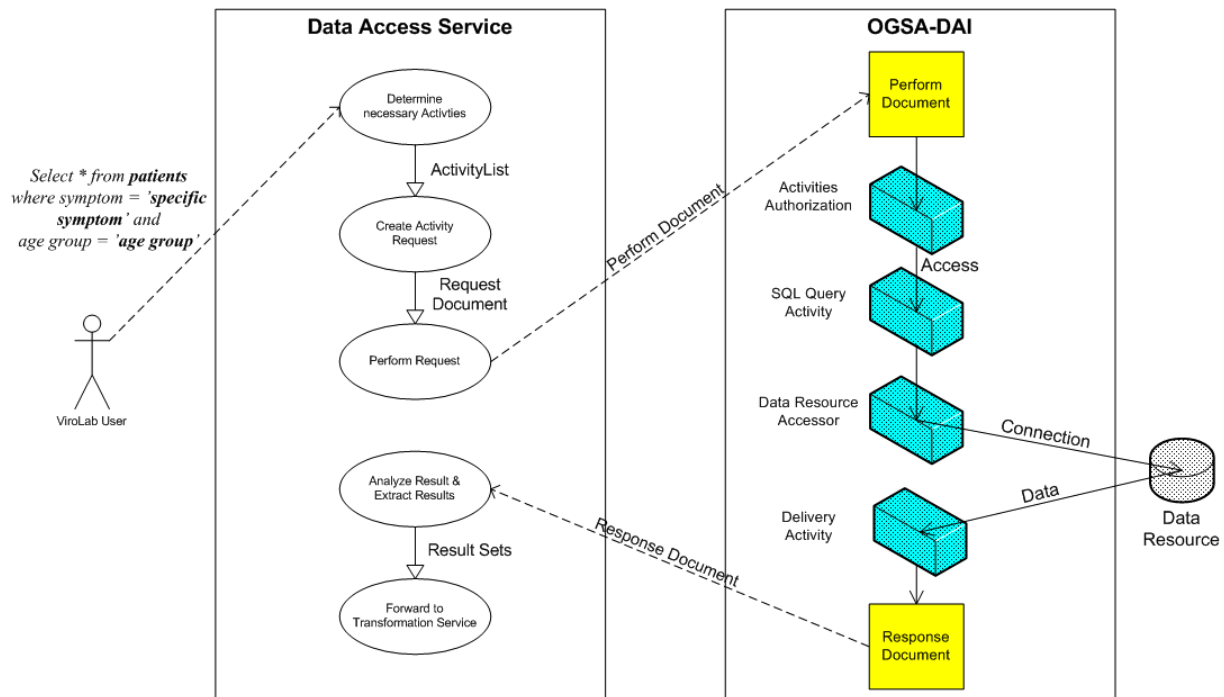


Figure -53: Use case showing a typical data access request and corresponding interactions with OGSA-DAI

They are typically connected with one data resource accessor, which uses a database-dependent driver to establish a connection with the underlying resource.

When dealing with multiple data providers, each of them usually has its own installation of a data access system including the data access service linked with an OGSA-DAI data service. The coordination of all these single systems requires one central entry point, which acts as the only “visible” and accessible data access system, and which hides all other data access systems from the users. In theory users should be unaware that they are using a federation rather than a single data resource. Currently, the DAS offers one specific functionality that handles a federated query. The main operations are similar to the one shown in Figure -53 with the difference that this has to be done many times.

Part 2:

- Requesting publicly available rule sets

The main usage of the *RequestRuleSets* method has already been described in section 7.2.2 so that only the different steps, which are transparent to the users, shall be explained in more detail. The following diagram schematically shows the control flow of the *RequestRuleSets* function in a sequential order.

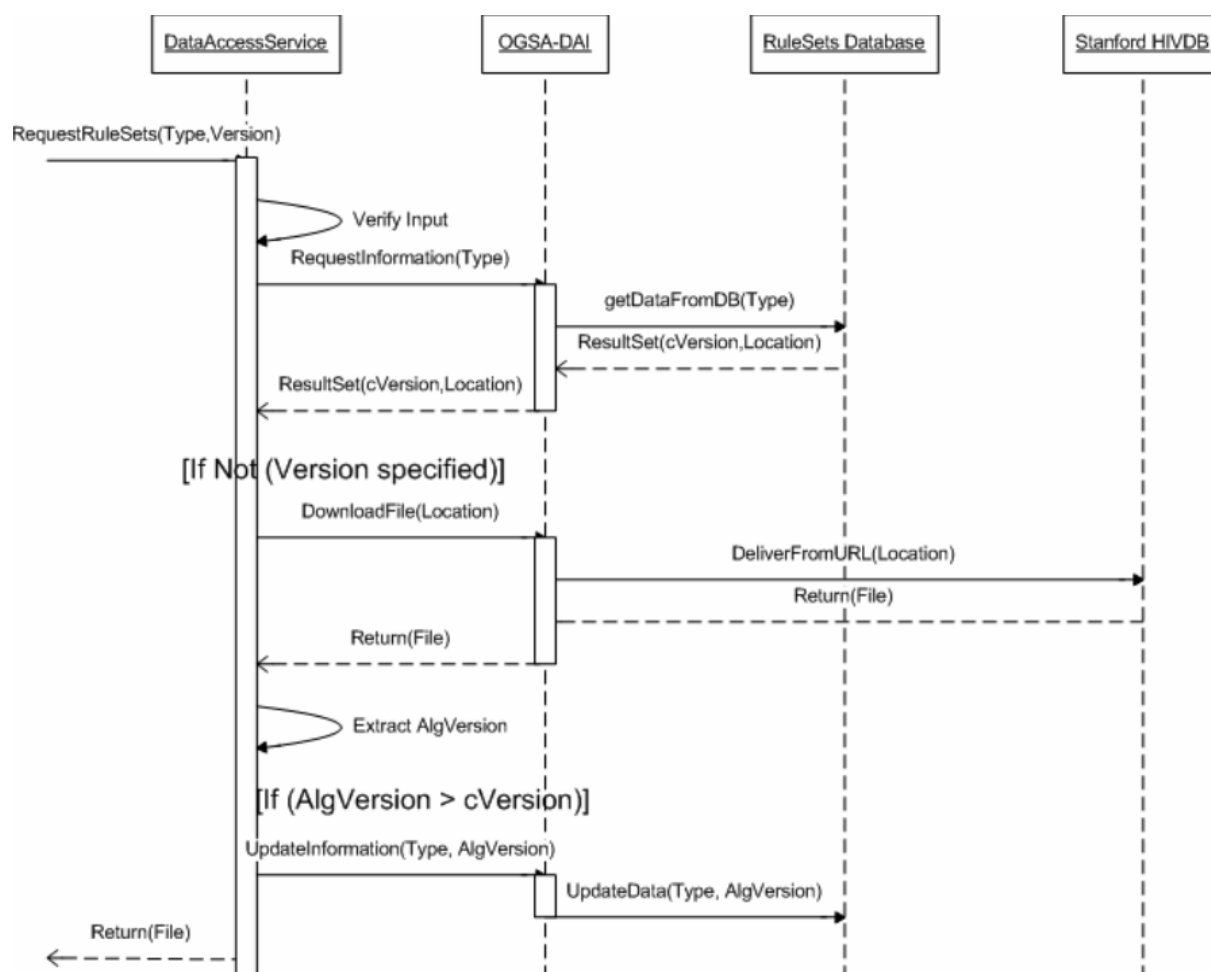


Figure -54: Control flow of the specific *RequestRuleSets* method

One can see that the function reverts to OGSA-DAI's data access capabilities, which are used to download files from an online data repository and also for storing relevant information in a local database. This local information is used to manage the current available and used versions of the rule sets. The diagram also illustrates the interactions between the DAS specific components/services and the corresponding interfaces offered by OGSA-DAI.

- Storing application-dependent data

Based on the general description in section 7.2.2, the following diagram shall explain the internal steps performed during the processing. Depending on the current application type, a particular internal application-related function is called, which performs relevant data transformation on the values provided. Once these data manipulations are finished, the OGSA-DAI functionalities for accessing distributed databases are used by the method in order to store the values in the corresponding database.

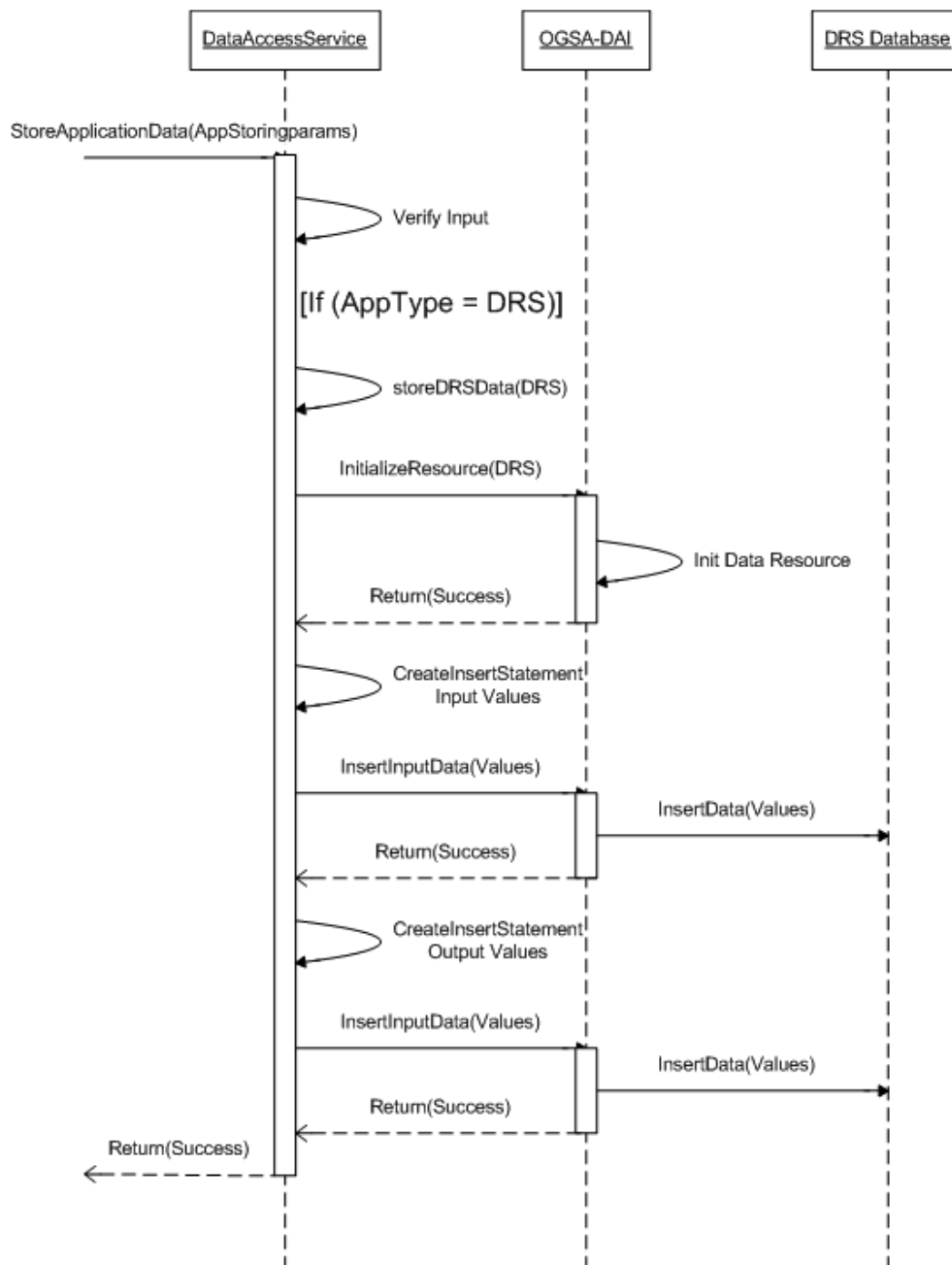


Figure -55: Internal flow of *StoreApplicationData* method

7.6. PRODUCT TESTING

According to Work Package 4 integration guidelines (specified in D4.2), each component itself should perform a unit testing procedure for its functions and methods. For verifying reliability, efficiency, compatibility, integrity, and usability of the DAS functionalities, a set of unit test cases were written and used to check the source code including different functions of the API:

- *TestServiceInitialization*: Initializes a particular service and its corresponding resource
- *TestSourcesAvailability*: Checks the availability of particular resources
- *TestDataResourceInformation*: Collects the meta-data of the resource

- *TestSQLStatements*: Performs different queries and verifies the data retrieved
- *TestRequestRulesets*: Checks whether different types of rule sets can be requested
- *TestSubmitDQ*: Tests the submission of queries to multiple data resources
- *TestStoreAppData*: Tests the storage of application data (Currently only for the DRS application)

The GUI of the JUnit toolkit, which is shown in Figure -56, can be used to visualize the testing procedure and to facilitate the testing process. One can simply load the test class and start the procedure by clicking the 'run' button. The screenshot below depicts the successful test of the above explained testing routines.

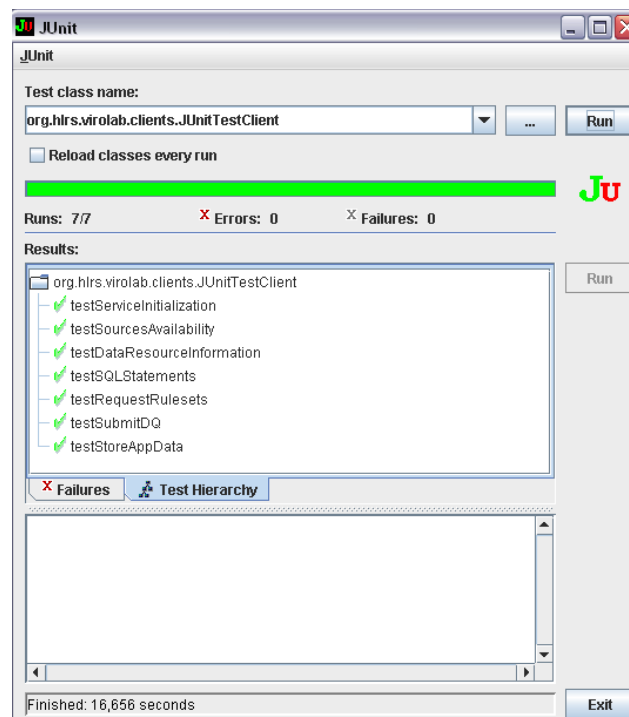


Figure -56: Visualization of DAS unit test cases

7.7. CONTACT INFORMATION AND CREDITS

For additional information, questions, errors, bugs etc. please contact the following author:

- Matthias Assel (assel@hirs.de)

The author also wants to thank all who contributed to this work, in particular:

- Aenne Löhden (USTUTT, Stuttgart, Germany)
- Bettina Krammer (USTUTT, Stuttgart, Germany)
- Stefan Wesner (USTUTT, Stuttgart, Germany)
- Piotr Nowakowski (ACK Cyfronet AGH, Kraków, Poland)

ABBREVIATIONS

Fix the list **[Tomasz Gubala]**

Abbreviation/Term	Explanation
AAS	Aminoacid Sequence
API	Application Programmer's Interface
ARID	Application Run Identifier
CA	Certificate Authority
CCA	Common Component Architecture
DAC	Data Access Client
DAS	Data Access Services
DB	Database
DEISA	Distributed European Infrastructure for Supercomputing
DGE	Data Gathering Engine
DO	Domain Ontology
DRAM	Drug Resistance Associated Mutations
DRE	Data Retrieval Engine
DRS	Drug Ranking System
DS	Distributed Storage
DSS	Decision Support System
EGEE	Enabling Grids for e-Science in Europe
EPL	Experiment Planning Language
FLOWR	For-Let-Where-Order by-Return
GOB	Grid Object Class
GOBI	Grid Object Instance
GOBID	Grid Object Identifier
GOBImpl	Grid Object Implementation
GOp	Grid Operation
GOI	Grid Operation Invoker
GPL	GNU General Public License
GrAppO	Grid Application Optimizer
GRR	Grid Resources Registry
GT	Globus Toolkit
GT4	Globus Toolkit 4.0
GUI	Graphical User Interface
HIV	Human Immunodeficiency Virus
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment

Abbreviation/Term	Explanation
IEEE	Institute of Electrical and Electronic Engineers
Jar	Java Archive
JMX	Java Management Extensions
JSR	Java Specification Request
JVMTI	Java Virtual Machine Tool Interface
LCG	LHC Computing Grid
LHC	Large Hadron Collider
LOB	Large Object
MQL	Meta Query Language
NS	Nucleotide Sequence
OGSA	Open Grid Services Architecture
OGSA-DAI	Open Grid Services Architecture – Data Access Integration
OGSA-DQP	Open Grid Services Architecture – Distributed Query Processing
OO	Object-Oriented
OR	Object-Relational
OWL	Web Ontology Language
PDP	Policy Decision Point
PROToS	Provenance Tracking System
RAD	Rapid Application Development
RBAC	Role-Based Access Content
RDF	Resource Description Framework
RDQL	Resource Description Framework Data Query Language
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SN	Storage Node
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Socket Layer
SSN	Storage Super Node
SSO	Single Sign-On
SVN	Subversion, a revision control system
TLS	Transport Level Security
UML	Unified Modeling Language
URI	United Resource Identifier
URL	Uniform (or Universal) Resource Locator
UTF8	8-bit Unicode Transformation Format

Abbreviation/Term	Explanation
ViroLab	A virtual laboratory for decision support in HIV treatment
VL	Virtual Laboratory
VM	Virtual Machine
VO	Virtual Organization
VPN	Virtual Private Network
WP	Work Package
WS	Web Service
WS-I	Web Services Integration
WSDL	Web Services Definition Language
WSRF	Web Services Resource Framework
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language

REFERENCES

- [D3.2] ViroLab Project Consortium: *Deliverable 3.2 – Design of the Virtual Laboratory*, 2007
- [D3.3] ViroLab Project Consortium: *Deliverable 3.3 - Session Manager and runtime system and data layer: installation, integration and usage; description of interfaces to WP2, WP4 and WP5 – report and demonstration*, August 2007
- [EGEE] *Enabling Grids for E-Science Project*,
<http://public.eu-egee.org/>
- [LCG2] Antonio Delgado Peris, Patricia Mendez Lorenzo, Flavia Donno, Andrea Sciaba, Simone Campana, Roberto Santinelli: *LCG-2 User Guide Manuals Series*, EGEE Project Consortium, August 2005
- [JRUBY] JRuby – Java powered Ruby implementation,
<http://jruby.codehaus.org>
- [MOCCATUT] Maciej Malawski: *The MOCCA Component Development Tutorial*, <http://mocca.icsr.agh.edu.pl>, ICS University of Science and Technology AGH, April 2007
- [RESBROWDEV] ViroLab Project Consortium: *On-line tutorial on Adding new context operation to Resources Browser*,
<http://virolab.cyfronet.pl/trac/epe/wiki/AddingContextActionToGrrBrowser>, August 2007
- [WTS] Pieter Libin, Bart De Deckere, Joris Van Santvoort: *Wts: a stateful web service infrastructure*,
<http://wts.sf.net/>