AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY
IN KRAKOW, POLAND

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND ELECTRONICS

INSTITUTE OF COMPUTER SCIENCE

Optimization of Grid Application Execution

Master of Science Thesis

Joanna Kocot, Iwona Ryszka

Computer Science

Supervisor: Marian Bubak, PhD
Advice: Maciej Malawski, MSc

KRAKOW, JUNE 2007

Table of Contents

Acknowledgements *9*
Abstract *11*
Chapter 1 : Introduction *13*
1.1 Target Environment 13
1.1.1 ViroLab Rational Basis 13
1.1.2 ViroLab Mission 13
1.1.3 ViroLab Users Characteristics 14
1.1.4 General ViroLab Architecture 14
1.2 ViroLab Application Structure 17
1.2.1 Terminology Related to ViroLab Experiment 17
1.2.2 Relations Between the Specified Entities 18
1.3 Motivation for Optimization in ViroLab 18
1.4 The MSc Thesis Goals 19
1.4.1 Identification of Available Optimization Solutions in Grid Computing 19
1.4.2 Identification and Analysis of the Problem of Optimization in ViroLab 19
1.4.3 ViroLab Optimizer Design and Development 19
1.4.4 Proving the Usefulness of the Developed Optimizer for ViroLab 19
1.5 Summary 19
Chapter 2 : Issues of Optimization for Grid Computing – State of the Art *21*
2.1 Overview of the Grid Technology 21
2.2 Overview of the Grid Optimization Problem 22
2.2.1 Terminology Related to Optimization on Grid 22
2.2.2 Grid Optimization Problem Statement 22
2.2.3 Grid Optimization Process 22
2.2.4 Components in Grid Optimization Process 24
2.3 Overview of Optimization Algorithms for Grid Computing 25
2.3.1 Challenges of Optimization in Grid Computing 25
2.3.2 Hierarchical Taxonomy of Grid Optimization Algorithms 26
2.3.3 Extension to the Hierarchical Taxonomy 28
2.3.4 Optimization Algorithms Suitable for Optimization on Grid 28
2.4 Summary 31
Chapter 3 : Analysis of Optimization Issues in ViroLab Virtual Laboratory Runtime System *33*
3.1 Grid Optimization Problem in ViroLab 33
3.2 The Optimization in ViroLab Runtime System as an Example of Optimization Process for Grid Computing 33
3.2.1 Optimization Results Consumer 34
3.2.2 Optimization Process Phases 34
3.2.3 Communication Channels to Other ViroLab Components 35
3.3 Requirements for ViroLab Optimizer 35
3.3.1 Functional Requirements 35
3.3.2 Non-functional Requirements 36
3.4 ViroLab Optimizer Optimization Model 36
3.4.1 Constraints Imposed by ViroLab Environment 36
3.4.2 ViroLab Optimizer Optimization Type 36
3.4.3 Task Dependencies in Optimization Process in ViroLab 37
3.5 Analysis of Existing Optimization Algorithms Regarding the ViroLab Optimizer Requirements 37
3.6 Summary 38
Chapter 4 : ViroLab Optimizer Design *39*
4.1 GrAppO Use Cases 39
4.2 GrAppO Architecture 40
4.3 Data Provided by External Components 41
4.3.1 Grid Resource Registry 41
4.3.2 Runtime Library 42
4.3.3 Monitoring System 42
4.3.4 Provenance Tracking System (PROToS) 42
4.4 Alternative to the Connection to Provenance Tracking System 43

4.5 GrAppO Optimization Algorithm Plug-ins 43
4.6 GrAppO State During Execution of Applications 44
4.6.1 Short-Sighted and Medium-Sighted Optimization Modes 44
4.6.2 Far-Sighted Optimization Mode 44
4.7 Control Flow Inside GrAppO 46
4.7.1 Short-Sighted Optimization Mode 46
4.7.2 Medium-Sighted Optimization Mode 48
4.7.3 Far-Sighted Optimization Mode 50
4.8 Summary 52
Chapter 5 : GridSpace Application Optimizer Implementation Details 53
5.1 Current GrAppO Implementation Scope 53
5.2 GrAppO Configuration 53
5.2.1 Optimization Policy Usage Schema 53
5.2.2 Properties Configured with Optimization Policy 54
5.2.3 Service Access Configuration Usage Schema 55
5.2.4 Properties Configured with Service Access Configuration 55
5.3 Using GrAppO 55
5.3.1 Constructors 55
5.3.2 Initialization Methods 56
5.3.3 Optimization Requests 56
5.3.4 Getters and Setters 56
5.3.5 Using Configuration Classes 57
5.3.6 Using Configuration Files 57
5.4 Tools Used for GrAppO Design and Development 58
5.4.1 Design 58
5.4.2 Development 58
5.4.3 Other 58
5.5 Summary 58
Chapter 6 : Tests of GridSpace Application Optimizer 59
6.1 Introduction 59
6.2 GrAppO Unit Tests 59
6.2.1 GrAppO Manager Tests 60
6.2.2 Optimization Engine Tests 60
6.2.3 Performance Predictor Tests 61
6.2.4 Resource Condition Data Analyzer and Historical Data Analyzer Tests 61
6.2.5 Test Reports 61
6.3 GrAppO Integration Tests 62
6.4 GrAppO Quality Tests 66
6.4.1 Test Environment 66
6.4.2 Tests Objective Function 67
6.4.3 Comparison of Effectiveness of Different Optimization Modes 68
6.4.4 Comparison of Effectiveness of Different Optimization Algorithms 70
6.4.5 Impact of Quality and Availability of Information from External Components 70
6.5 Conclusions 71
Chapter 7 : Conclusions and Future Work 73
7.1 Conclusions 73
7.2 Future Work 73
Appendix A: Using GrAppO as External Library 75
Appendix B: GrAppO Class Diagrams 77
Appendix C: ViroLab Experiments Information 89
Appendix D: GrAppO Configuration Files Format 93
Appendix E: GrAppO Performance Tests Environment Details 95
References 99

List of Figures

Figure 1: ViroLab Virtual Laboratory general architecture 15
Figure 2: Detailed architecture of the ViroLab Virtual Laboratory 16
Figure 3: Dependencies between ViroLab entities – Grid Object Class, Grid Object Implementations, Grid Object Instances and Grid Resources (WS Containers and H2O Kernels represent containers on Grid Resources) 18
Figure 4: A hierarchical taxonomy of optimization algorithms in distributed systems [20] 26
Figure 5: Data flow between the optimizer and its data sources and consumers 35
Figure 6: The GridSpace Application Optimizer use cases diagram 40
Figure 7: Component diagram, showing GrAppO decomposition and dependencies between its components and external ViroLab components 41
Figure 8: Structure that is an alternative to the connection to Provenance Tracking System 43
Figure 9: A relation between an optimization algorithm and its different kinds 43
Figure 10: A state diagram of the optimizer performing short- or medium-sighted optimization 44
Figure 11: A state diagram of the optimizer running in far-sighted optimization mode 45
Figure 12: A sequence diagram illustrating the flow of control in GrAppO during short-sighted optimization 47
Figure 13: Part of a sequence diagram that illustrates the flow of control in GrAppO during medium-sighted optimization 49
Figure 14: A sequence diagram illustrating the flow of control in GrAppO during the formation of solution mapping (first phase) in far-sighted optimization 51
Figure 15: A sequence diagram illustrating the flow of control in GrAppO while answering the requests from GOI (second phase) in far-sighted optimization 52
Figure 16: Optimization Policy as an inner GrAppO class with optimization algorithm configured 54
Figure 17: Optimization Policy read from an external configuration file with optimization algorithm configured 54
Figure 18: Service Access Configuration as an inner GrAppO class 55
Figure 19: Service Access Configuration read from an external configuration file 55
Figure 20: A part of GrAppO tests report 62
Figure 21: A test coverage reports for GrAppO main part 62
Figure 22: A screen-shot of a ViroLab experiment run from MS Windows command line – Grid Operation Invoker debug logs show a result of the call to GrAppO optimization method (marked with the red frame) 63
Figure 23: A screen-shot of another ViroLab experiment – this time Grid Operation Invoker calls the GrAppO optimization method twice (again the request and result log parts are marked with red frame) 64
Figure 24: A screen-shot of an execution of the experiment that was also presented on Figure 22; the experiment was run using the Experiment Planning Environment – a Graphical User Interface for the ViroLab Runtime 65
Figure 25: A screen-shot of an execution of the experiment that was also presented on Figure 23; as with Figure 24, Experiment Planning Environment was used for the experiment execution 66
Figure 26: General view on GrAppO classes and connections 77
Figure 27: A class diagram concerning the output GrAppO produces 78
Figure 28: GrAppO configuration and Grid Resource Registry service access 79
Figure 29: A diagram of classes representing Grid Object entities, their aggregation and ranking 81
Figure 30: Classes concerning the optimization algorithms execution 82
Figure 31: Performance Predictor classes 84
Figure 32: Resource Condition Analyzer classes 85
Figure 33: Historical Data Analyzer classes 86
Figure 34: Detailed view on classes concerning GrAppO optimization policy configuration reading 87
Figure 35: Util class for reading configuration files 88

List of Tables

Table 1: GrAppO Manager main unit test information *60*
Table 2: Additional tests for GrAppO Manager *60*
Table 3: Optimization Engine main unit test information *60*
Table 4: Implemented optimization algorithms tests *61*
Table 5: Performance Predictor unit test information *61*
Table 6: A summary of Resource Condition Data Analyzer and Historical Data Analyzer tests *61*
Table 7: Test result for optimization of 10 Grid Object Classes *69*
Table 8: Test result for optimization of 20 Grid Object Classes *69*
Table 9: Test result for optimization of 50 Grid Object Classes *69*
Table 10: Influence of the availability of information on the makespan *71*

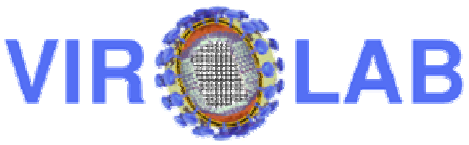
List of Code Snippets

Code Snippet 1: Configuring Cyfronet Repository as Maven2 repository 75
Code Snippet 2: Configuring GrAppO as Maven2 dependency 75
Code Snippet 3: Source code of the `aligh_experiment.rb` *script* 90
Code Snippet 4: Source code of the `weka_experiment.rb` 92
Code Snippet 5: Example XML GrAppO configuration file 93
Code Snippet 6: A sample entry in the text GrAppO configuration file 93
Code Snippet 7: Sample structure of data generated as obtained from GRR 95
Code Snippet 8: Sample structure of data generated as obtained from Monitoring System 96
Code Snippet 9: Sample structure of data generated as obtained from PROToS 97

Acknowledgements

We would like to express our thanks to our supervisor – Marian Bubak and Maciej Malawski for the support we received during the development of the present thesis. Especially, to Maciej Malawski for the commitment with the project.
Also the help of all the ViroLab Virtual Laboratory team members from ACC Cyfronet AGH cannot be overestimated.

This work was made possible owing to the ViroLab Project – EU project number: IST-027446. The official ViroLab website can be viewed under http://www.virolab.org. Please visit also the website dedicated to ViroLab Virtual Laboratory and created by the ACC Cyfronet AGH team at http://virolab.cyfronet.pl/. Some parts of the present thesis will be also published there.



Abstract

Despite the existence of many Grid optimization technologies and algorithms, there are some environments to which a standard optimization techniques cannot be applied. For them, new solutions have to be invented and developed.
This thesis is intended to build an optimization engine for one of such environments – ViroLab Virtual Laboratory Runtime. Its specific model – invoking operations on special objects which reside on Grid resources – imposes a new approach to optimization. Such an approach is presented in the shape of the GridSpace Application Optimizer (GrAppO) – an engine for optimizing Grid applications execution, designed especially for the purposes of the ViroLab Runtime.
GridSpace Application Optimizer has to face the challenges imposed by the Grid environment as well as specific only to ViroLab. They are e.g. dynamic nature of the environment, distributed sources of information, difficulty in defining suitable criteria. Nevertheless, it is able to significantly increase the quality of the ViroLab Runtime performance, by providing it with the most suitable objects to invoke operations on.
This thesis follows the complete process of the GrAppO development. From the problem definition and analysis of existing solutions in the matter, through the design of the final system, to its implementation and tests which results proved GridSpace Application Optimizer to be a valuable element.

The thesis is organized as follows:
A detailed description of the environment in which GridSpace Application Optimizer is going to operate is presented in sections 1.1 and 1.2. Basing on these information, the motivation and goals of the thesis are specified in sections 1.3 and 1.4, respectively.
Chapter 2 the general problem of optimization in Grid Computing is briefly introduced to identify main challenges regarding this issue.
As a conclusion derived from Chapter 1 and Chapter 2, the optimizer specification is introduced in Chapter 3 – including: comparison to a generic grid optimization (3.2), requirements for it (3.3), optimization type identification (3.4), analysis of the techniques presented in Chapter 2 as a possible solutions for the optimizer (3.5).
A detailed architecture, designed on basis of the specification (Chapter 3) is presented in Chapter 4, and its implementation details – in Chapter 5.
Chapter 6 provides a description and results of the tests performed on the ready GridSpace Application Optimizer prototype.
The conclusions derived from this thesis and some new issues worthy further exploration are proposed in Chapter 7.

Keywords: Grid Computing, Optimization on Grid, Runtime Optimization, Optimization Algorithms, Optimization Heuristics, ViroLab, Grid Object.

Chapter 1: Introduction

The subject of this thesis is an entity – optimizer that would be able to perform optimization of grid applications under certain circumstances. This chapter introduces the environment in which this optimizer will be working – ViroLab Virtual Laboratory Runtime, its basis, architecture and very specific conditions. The conditions and main challenges of the optimization will be described along with the goals that have to be achieved by the present thesis to provide a comprehensive solution.

1.1 Target Environment

The ViroLab Virtual Laboratory runtime, which is the target environment for the subject of the present thesis, is a part of ***ViroLab*** project. ViroLab, according to [1]: "is a Specific Targeted Research Project of the EU 6th Framework Programme for Research and Technological Development in the area of integrated biomedical information for better health". In this section, it will be presented in more detail from the point of view of this thesis.

1.1.1 ViroLab Rational Basis[1]

Throughout the world large, high quality databases store information related to various scientific subjects. Since they are not integrated, nor the tools that enable sharing and processing their data are available, their use is limited to only local applications.
An extensible Grid-based system could enable integration of these data and publishing it in a secure way through a number of specialized services.

1.1.2 ViroLab Mission

Nowadays the computer technology has a great impact on the development of medicine. In the area of the computer science recently has emerged some projects that are dedicated to support the process of the treatment of patients. One of them is ViroLab, which mission is to provide researchers and medical doctors in Europe with a virtual laboratory for infectious diseases. It is intended to facilitate transformation and computation concerning viral genetic information in order to develop new, more efficient treatments.
By offering an environment to develop, publish and use services that could retrieve data from distributed clinical databases and perform the complex computation on it, the researchers are given a comprehensive and integrated solution to the problems they so far solved by hand or with their own applications. ViroLab provides them with an ability to automate and conduct the complicated and long-lasting computations on the Grid. It also facilitates collaboration among the researchers.

[1] Sections 1.1.1 and 1.1.2 were based on [1].

1.1.3 ViroLab Users Characteristics[2]

The most common ViroLab users may be divided into three categories, characterized in this section. They are:

a. **Clinical Virologist** (other names: *medical user*, *healthcare provider*)

It is a person that works for a hospital and helps the medical doctor to get more information regarding the certain infection case. This user's main objective is to provide an appropriate treatment for patients. The main stress will be put onto HIV virus infections and the medical help for the people with such infections. Such user provides the system with domain expertise with respect to specific virus mutations and drug characteristics, while they are not assumed to have any computer-related knowledge.
From the application point of view, it is the person who provides it with all required data.

b. **Experiment Developer**

An experiment developer is usually a scientific programmer that works in a viral diseases research institute. The objective of this user's work is to provide viral diseases researchers (see: Experiment User) with proper tools that help them conduct experiments and to find useful results regarding their field of expertise. The knowledge this type of user is expected to have concerns both the domain field of viral disease research and non-basic skills regarding programming and computing technologies.
From the application point of view, it is the person who implements it.

c. **Experiment User**

This person archetype constitutes a scientist that work for a research institute and that put the vast expertise in the domain of viral diseases to use. The objective of this class of user in scope of the ViroLab is to combine the virtual laboratory platform with the experiments planned by experiment developers in order to acquire interesting findings. The ViroLab project enables them to execute experiments and gather, share and store scientific results.
From the application point of view, it is the person who launches it and makes use of the data it produces.

1.1.4 General ViroLab Architecture[3]

On Figure 1 an overview on the ViroLab virtual laboratory architecture is presented. It identifies main virtual laboratory subsystems and interactions between them, preserving a general level of abstraction.

The top layer of this architecture is Presentation. It enables ViroLab users (see section 1.1.3 for the users description) to interact with the system. The presentation layer consists of ViroLab Portal – an interface for Clinical Virologists (1.1.3a) and Experiment Users (1.1.3c), and Experiment Planning Environment – designed for Experiment Developers (1.1.3b). Both of these components are supported by Collaboration Tools. The presentation layer uses Experiment Repository for storing experiments which can be executed or redeveloped, and for keeping tracks of changes to them. It also interacts with two entities responsible for holding information about Grid Objects (see section 1.2.1 for explanation of the terminology related to Grid Objects): Domain Ontology Store – which links Grid Object Classes with regard to domain knowledge and Grid Resources Registry – containing specific Grid Object Classes properties (including their existing implementations and instances).

[2] Some parts of section 1.1.3 come from the ViroLab design document [4]

[3] The section is based on the general architecture description provided by the ViroLab design document [4].

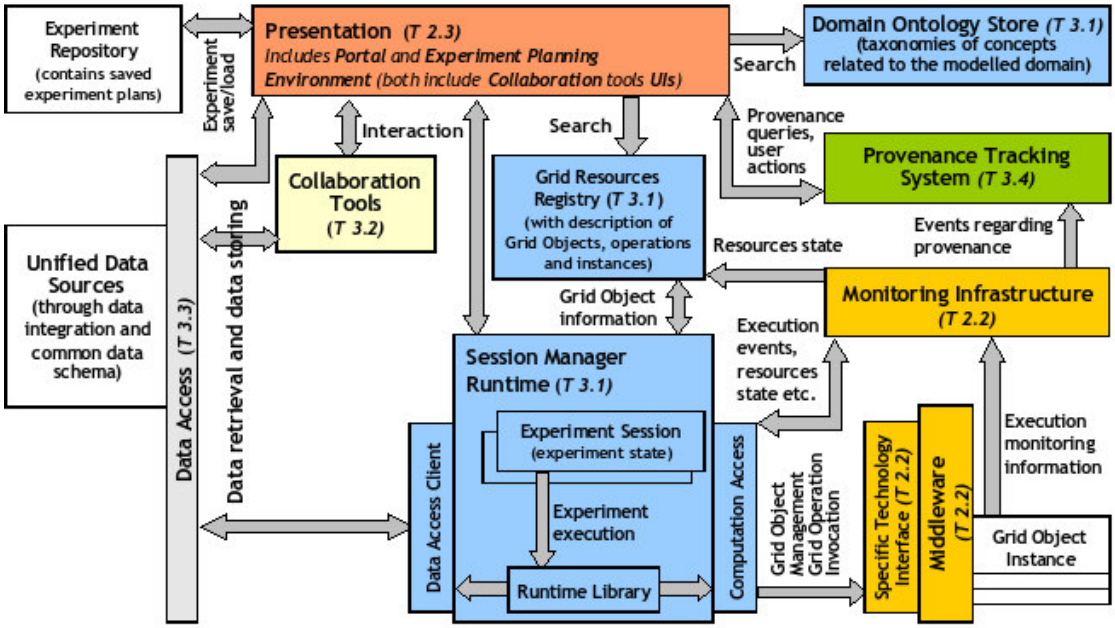


Figure 1: ViroLab Virtual Laboratory general architecture[4]

The interpretation and execution of an experiment script is performed by ViroLab Runtime, which does it in a model of invoking operations on either local or remote services. For this purpose it uses the components: Data Access Client – for retrieving remote data (through Data Access layer) and Computation Access – for optimization and invocation of Grid Operations contained in the experiment script. The runtime system is also responsible for experiment session maintenance.
All the events connected both to experiment execution and behaviour of the virtual laboratory components are intercepted by Monitoring Infrastructure and directed to Provenance Tracking System, which stores and publishes information about them.

A more detailed view on dependencies between ViroLab Virtual Laboratory subsystems and the external resources they use is presented on a component diagram – Figure 2. On this diagram the architecture was presented in a form of an UML component diagram, emphasizing the general order of its layers. The diagram also shows relations between the subsystems and the purpose of their connections in a more precise way.

[4] The figure comes from the ViroLab design document [4].

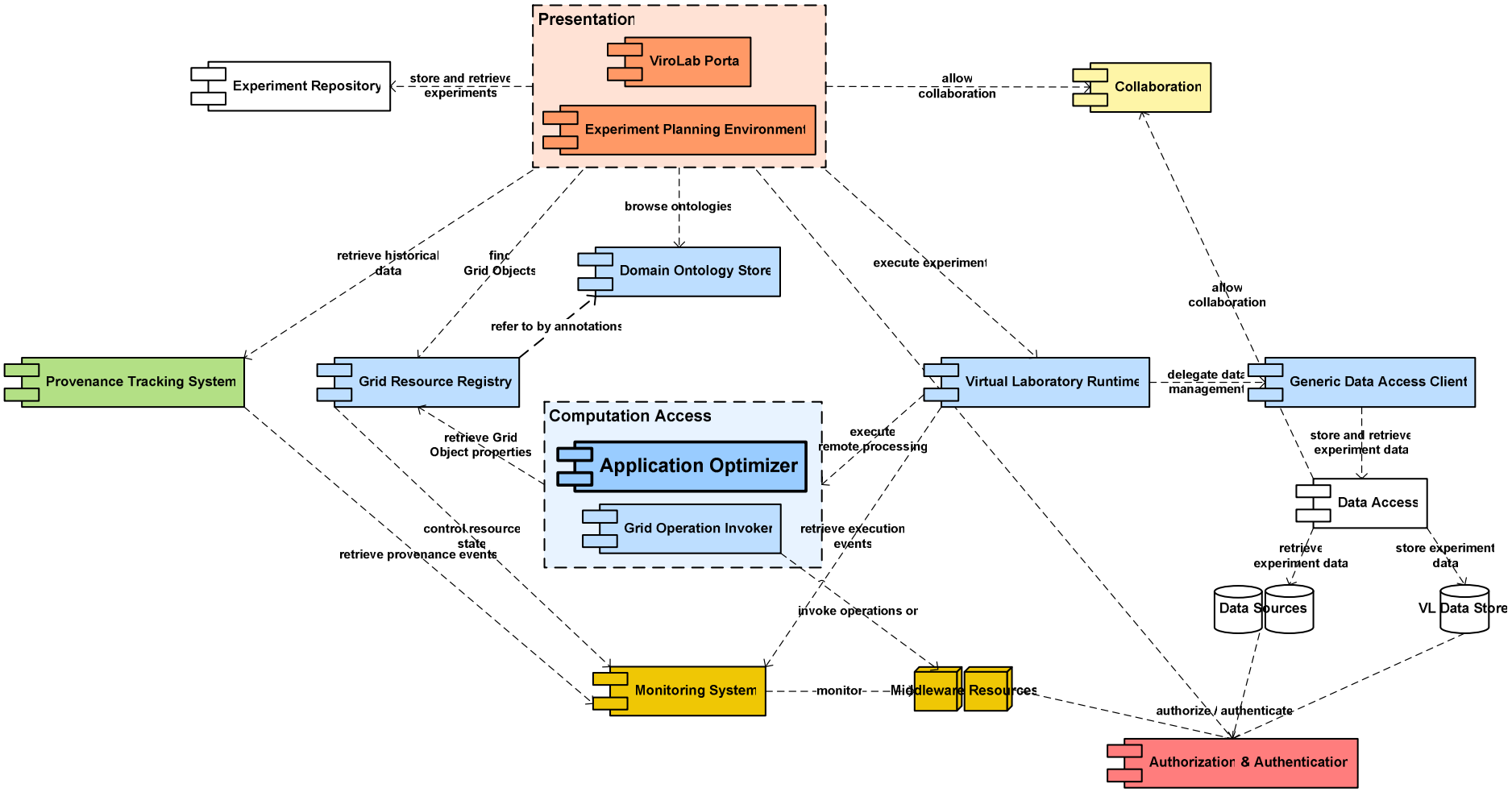


Figure 2: Detailed architecture of the ViroLab Virtual Laboratory

1.2 ViroLab Application Structure

The optimization problem in ViroLab descends from the model of the entities on which the executed application's operations are invoked. These entities are defined in this section, and relations between them are specified.

1.2.1 Terminology Related to ViroLab Experiment

The definition of the main entities related to an experiment executed by the ViroLab Runtime is provided below. The terms were first introduced in the ViroLab design document [4].

a. **Grid Operation (GOp)** (also *Grid Object Operation*, *Grid Object Method*)

Grid Operation is very similar to an *abstract method* in terms of Object-Oriented programming paradigm. It binds methods, offering the same functionality, and being abstract – cannot be invoked (only its implementations can).
Grid Operation is identified not only by its signature, but by a description specific for Grid and generally: remote computation.
One or more Grid Operation constitutes a Grid Object Class (see 1.2.1b).

b. **Grid Object Class (GOb)** (or simply *Grid Object*)

A Grid Object Class is – again, using the terminology of Object-Oriented programming paradigm – a kind of interface. It is an abstract of the same general functionality offered by different implementations and thus cannot be instantiated, nor invoked. It is a set of Grid Operations, which implementations have to be provided by the class's implementation. The implementation of a Grid Object Class is Grid Object Implementation (see 1.2.1c).

c. **Grid Object Implementation (GObImpl)**

Grid Object Implementation implements functionality of its Grid Object Class. It can only be bound to one Grid Object Class. The implementations of the same class may differ only in non-functional properties, since they have to offer the same Grid Operations (declared in Grid Object Class).
Grid Object Implementation is only a static entity, which has to be instantiated (deployed into a resource) to allow its operations invocation. A Grid Object Implementation deployed into and run on certain resource is called Grid Object Instance (see 1.2.1d).

d. **Grid Object Instance (GObInst)**

A Grid Object Instance is an instance (also understood in terms of Object-Oriented programming paradigm) of a certain Grid Object Class. It is bound to concrete Grid Object Implementation since it executes its code. A Grid Object Instance can be contacted to perform some computation. The instances which use the same implementation may (but do not have to) differ only in resource they are deployed into – not in the code base.

e. **Grid Resource**

Every resource in Grid which is able to host a Grid Object Instance – i.e. owns a running container into which Grid Object Implementation of a certain kind (implementation technology) could be / is deployed.

f. **ViroLab Experiment (Application)**

ViroLab Experiment[5] is a code in form of script composed of calls creating Grid Object Instances and invoking their Grid Operations. As the script interpreter is based on the Ruby language interpreter, Ruby commands may also be placed in the experiment code.

[5] Further in this thesis the term "application" is more commonly used.

1.2.2 Relations Between the Specified Entities

The dependencies between the entities described in paragraphs 1.2.1a-e can be summarized in the following way:

- Each Grid Object Class can be implemented by a number of various Grid Object Implementations – of the same functionality, but e.g. representing different technologies (Web Service, WSRF, MOCCA Component).
- Grid Object Implementation in turn, not being able to be invoked, is represented by numerous Grid Object Instances which are ready to perform processing.
- To create a Grid Object Instance, a Grid Object Implementation has to be deployed in a dedicated container, residing on a certain Grid Resource.

A sample dependency between these entities is shown on Figure 3.

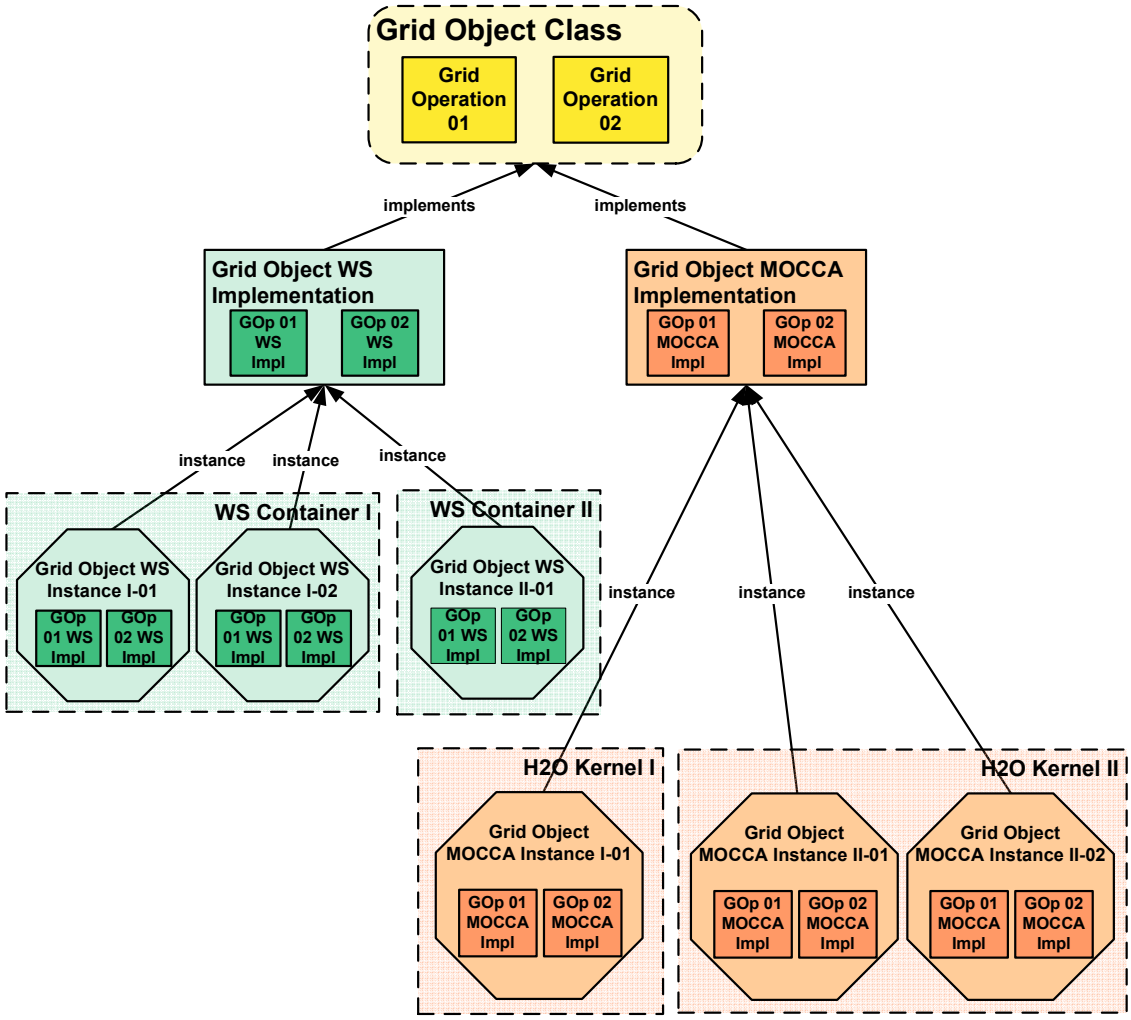


Figure 3: Dependencies between ViroLab entities – Grid Object Class, Grid Object Implementations, Grid Object Instances and Grid Resources (WS Containers and H2O Kernels represent containers on Grid Resources)

1.3 Motivation for Optimization in ViroLab

The core functionality of ViroLab Runtime System offers a possibility of executing a ViroLab Experiment (see section 1.2.1f). However, the source code of the experiment provides only the information on Grid Object Classes which instances have to be used to invoke certain operations on them. The choice of the instance, considering the structure of relations between the entities described in section 1.2 is not trivial and must be performed by making the following decisions:

- which Grid Object Implementation will be the most suitable to perform the processing
- which ready Grid Object Instance of this Grid Object Implementation will be the most suitable to perform the processing
- whether the Grid Object Instance should be chosen or a new one is to be deployed
- where (on which Grid Resource) a new Grid Object Instance should be created

The ViroLab Runtime System itself is not able to make these decisions, while it has to provide the answer to its invoker service. Therefore a need for a dedicated component, that would analyze all the available data and provide the optimum solution (according to certain criteria) to the mentioned problems emerges.
In traditional Grid environments a dedicated component – a scheduler of jobs or a resource broker – is introduced to make decisions similar to the aforementioned. However, the responsibilities of such component is greater than solving only these issues. Due to this fact, the term of "scheduling" in case of ViroLab Runtime is replaced with the term "optimization" (and similarly, the component name "scheduler" with "optimizer") as the latter term more accurately describes the core aim of the component.

1.4 The MSc Thesis Goals

The main goal of the thesis is to analyze, design and develop a system that will be able to rise to the challenges of optimization issues in ViroLab (see 1.3). In order to achieve the aim, the following sub-goals should be realized:

1.4.1 Identification of Available Optimization Solutions in Grid Computing

Performing a research of already available optimization solutions for Grid computing could bring a wider view on the problem. Some of the examined artefacts could also appear applicable to the solution developed within this thesis.

1.4.2 Identification and Analysis of the Problem of Optimization in ViroLab

The analysis of the possibilities and constraints imposed by the target environment (see section 1.1) – ViroLab should lead to a precise statement of the problem which this thesis will attempt to solve. All the project requirements have to be identified in order to develop a comprehensive solution.

1.4.3 ViroLab Optimizer Design and Development

The main product of this thesis is going to be an optimization engine for ViroLab Runtime (see 1.1.4). It will have to be designed from scratch and then implemented. It not only has to suit into the runtime system, but also be a robust and comprehensive solution to the stated problem.

1.4.4 Proving the Usefulness of the Developed Optimizer for ViroLab

The product developed with the thesis have to pass several kinds of tests – starting from unit tests for each implemented part, through integration tests with other ViroLab components to performance tests that show its quality. The design of implementation and performance tests itself is a challenge, as the tests should prove the products value in an unquestionable way.

1.5 Summary

The chapter presented environment to which the subject of this thesis is targeted, introducing its purpose and structure. By defining the entities that build a ViroLab application, the optimization subject was identified and the need for optimization stated.

Chapter 2: Issues of Optimization for Grid Computing – State of the Art

The ViroLab Project (compare 1.1) is developed using a Grid-based service oriented architecture, therefore the optimization it uses has to be designed with regard to the characteristics of the Grid. The following chapter presents the main concepts concerning Grid computing as a model of distributed computing. Basing on these information an issue of the optimization of the applications execution on the Grid is provided. The state of current research on optimization algorithms that comply with the Grid specific conditions will be presented in detail.

2.1 Overview of the Grid Technology

Grid computing is based on the distributed computing concept. The latter term refers to a model of computer processing that assumes the simultaneous execution of separate parts of program on a collection of individual computing devices. Hence the units in a distributed system do not operate in a uniform processing environment – they must communicate by protocol stacks that are less reliable than direct code invocation and shared memory [12]. Some characteristics of these systems are: resource sharing, openness, concurrency, scalability, fault tolerance, transparency.
The most important types of distributed computing systems are: multiprocessor systems, multicomputer systems, computing taxonomies, computer cluster, grid computing. In the following subsections the grid computing technology will be described in detail.

In the early 1990s significant improvements were introduced in the area of computing. The availability of the Internet and high performance computing gave the possibility to execute large-scale computation and to use data intensive computing applications in the area of science, engineering, industry and commerce. This idea led to the emergence of the concept of the Grid computing.
The term of "the Grid" was originated in the middle 1990s to describe a collection of resources geographically distributed that can solve large-scale problems. In the foundational paper "The Anatomy of the Grid. Enabling Scalable Virtual Organizations" [10] Ian Foster, Carl Kesselman and Steve Tuecke introduced the paradigm of the Grid and its main features. According to them the Grid concept can be regarded as coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations [10].
The Grid integrates and provides seamless access to a wide variety of geographically distributed from different administrative areas computational resources (such as supercomputers, storage systems, instruments, data, service, people) and presents them as a unified resource. The sharing of resources is restricted and highly controlled, with resource providers and consumers defining clearly and carefully their sharing rules. A dynamic

collection of individuals, multiple groups or institutions defined by such restrictions and sharing the computing resources of the Grid for a common purpose is called a Virtual Organization (VO) [10].
Moreover, in the Grid environment standard, open general-purpose protocols and interfaces should be used. The use of open standards provides interoperability and integration facilities. These standards must be applied for resource discovery, resource access and resource coordination [12].
Another basic requirement of a Grid Computing system is the ability to provide the quality of service (QoS) requirements necessary for the end-user community. The Grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, such as response time measures, aggregated performance, security fulfilment, resource scalability, availability, autonomic features e.g. event correlation, configuration management, and partial fail over mechanisms [11].

2.2 Overview of the Grid Optimization Problem

Among many challenges regarding Grid Computing, the problem of the optimization a Grid application execution is one of the most important. In this section the problem will be defined and the generic control flow during the application execution on the Grid presented.

2.2.1 Terminology Related to Optimization on Grid

In the remainder of this chapter, the following assumptions about the terminology will be made:

- **Resource** – an entity which is used by a process to execute a job. Examples of resources are storage systems, supercomputers, data sources or people.
- **Job** – a single, atomic unit of the application that can be independently assigned to a resource (an appropriate point on the Grid) to be executed on it.
- **Application** (Grid application) – a collection of atomic jobs which execution is requested by a user and can be performed on the Grid. These jobs are usually designed to be executed in parallel on different machines of the Grid. Additionally, specific dependencies between jobs may be introduced – e.g. a job can define a collection of other jobs that have to complete before the given job can be executed. Finally, the results of all of the jobs must be collected and appropriately assembled to produce the ultimate output for the application [8].

2.2.2 Grid Optimization Problem Statement

The Grid system is responsible for sending a job to a best suitable resource to be executed. In large systems it is very cumbersome for an individual user to select these resources manually [19]. Therefore a dedicated component – an optimizer, which acts as localized resource broker is available in the Grid system. Its scope of responsibilities includes mapping Grid jobs to resources over multiple administrative domains.
Basing on the aforementioned information, the generic problem of the optimization of the Grid application execution can be stated as follows:

The process of discovering of the best combination of a job and resources in such a way that the user and application requirements are fulfilled, in terms of overall execution time (throughput) and cost of the resources utilized [16]. This mapping is performed by taking static restrictions and dynamic parameters of jobs and resources into consideration.

2.2.3 Grid Optimization Process

As it was stated in section 2.2.2 the success of the optimization of Grid application depends mainly on appropriate mapping results. Hence, in this section the general process of the

optimization of Grid application execution will be described in detail. The process can be divided into three phases: resource discovery, system selection and job execution. This division was introduced by Jennifer M. Schopf in [15] – the description of the phases is based on this paper.

Phase 1: Resource Discovery
The main purpose of this stage is to generate a list of potential resources which are available to utilize at a given moment. The resource discovery phase involves determining a collection of resources to be investigated in more detail in Phase 2, information gathering. At the end of this phase, this collection will contain a set of resources that have passed a minimal feasibility requirements. This phase can be divided into the following steps:

Step 1: Authorization Filtering

This step determines a collection of resources to which an authorized access can be obtained. The most common solution is to store a list of resources with all relevant information such as account names or passwords. Although the necessary data can be obtained in a simple way, this method does not assure fault tolerance and scalability.

Step 2: Application Requirements Definition

The step is responsible for the specification of minimal job requirements for resources. The collection of possible job requirements can be divided into static and dynamic ones. The former involves for instance the type of the operating system or the specific architecture best suitable for the given code. The latter subgroup represents the requirements such as minimum amount of RAM available, a load of central processing unit or connectivity needed. The requirements should be specified in as much detail as possible in order to gain a better result of general process of optimizing.
The process of the specification of the application requirements is rather complicated, therefore gathering requirements and their storage is not automated.

Step 3: Minimum Requirement Filtering

Basing on information gathered in previous two steps the process of filtering resources is executed – the resources to which an access is not granted or that not fulfil the job requirements are discarded. The collection of valid resources will be investigated in more detail in the next phase.

Phase 2: System Selection
During this phase a single resource (or a resource collection) must be selected with regard to which is suitable for a given job. The selection is performed depending on available data. The process can be divided into two subphases (steps):

Step 1: Dynamic Information Gathering

Apart from the generic and static information about the resources, detailed dynamic data about the current condition of them is required to perform a better job-resource match. Taking into consideration such kind of information is important, since it may vary with respect to the application being requested to execute and the resources being examined. In general on the Grid, a dedicated component is introduced which a main task is to provide information about a current resource condition.
Another significant issue in this area is a problem of scalability and consistency. The larger the system is, the more queries must be performed that can be a time-consuming activity. Moreover, the system should be able to cope with an unavailability of a resource for a period of time and a request for the data from it. Currently, the situation is generally avoided by assuming that the shortage of the dynamic data from a resource leads to ignoring it in an optimization process.

Step 2: Final Decision

Having obtained all relevant, detailed information, the next step is to make a decision which resource (or set of resource) to use. Various approaches are possible according to a selected algorithm and a policy. Some of them are described in the section 2.2.4.

Phase 3: Job Execution

The last phase is running a job on a selected resource (or a selected collection of resources). The process consists of the following activities (steps):

Step 1: Advance Reservation [optional]

This step introduces a possibility to reserve a resource in advance in order to make the best use of a given system. The complexity of the process is varied and it can be performed with both mechanical and human means. Currently, such system is not introduced on many resources, but thanks to the emergence of service level agreements paradigm, this activity will have more impact on Grid application execution in the future.

Step 2: Job Submission

After the selection of resources, the application can be submitted to the resources. Job submission may be as easy as running a single command or as complicated as running a series of scripts and may or may not include setup or staging. In general, the lack of standards for job submission is observed.

Step 3: Preparation Tasks

Preparation Tasks

The step may involve setup, staging, claiming a reservation and other actions that may be necessary to prepare the resource to run the application. In a Grid environment, authorization and scalability issues can complicate the process. The common solution is to use scp, ftp or a large file transfer protocol (e.g. GridFTP) to ensure that the data files needed are in places.

Step 4: Monitoring Process

During the execution of the application, the status of the job can be monitored and regarding to its progress it can be rescheduled to another resource. Today, such monitoring is typically done by repetitively querying the resource for status information. In case of the insufficient progress of the execution, the job may be rescheduled. Sometimes this process can become complicated because of the lack of control over resources – other jobs may be scheduled causing the one of concern to be terminated, possibly without any warning or notification.

Step 5: Job Completion

After completion of the job, a notification should be issued to the user. The common solution is to include an e-mail notification parameter in a submission scripts for parallel machines. Receiving the information about the completion state is important because of the fault-tolerance reasons.

Step 6: Cleanup Tasks

The last step involves a retrieval of files from the resource in order to perform data analysis on the results or a temporary removal settings. The process can be executed by hand or by including clean-up information in the job submission scripts.

2.2.4 Components in Grid Optimization Process

During the Grid optimization process the optimizer has to interact with other components of Grid Computing environment.

First of all, to make a proper assignment in the heterogeneous and dynamic Grid environment the information about the current state and condition of available resources is needed. In general, Grid optimizers receive data from a general Grid Information System (GIS). GIS is responsible for collecting and predicting the resource state information (e.g. CPU capacities,

network bandwidth) and can answer queries for resource information or notify subscribers about the new data [15].
Another useful information for the process of optimization are application properties (such as memory and storage requirements, sub-jobs dependency in the application) and the performance of resource for different application species. Such data will simplify the computation of the cost estimation for a schedule candidate [5].
Next component is the Launching and Monitoring module. Its main task is to, according to the given schedule, submit applications to resources, staging input data or executables and monitoring the execution of the applications [5].

2.3 Overview of Optimization Algorithms for Grid Computing

Regarding to the optimization of the Grid application execution, the most important point in the generic Grid optimization process is System Selection (see section 2.2.3). The optimization can be performed owing to the possibility of utilization of different selection strategies, by introducing appropriate algorithms. In this section challenges regarding the optimization process in the Grid Computing and classification of optimization algorithms will be described.

2.3.1 Challenges of Optimization in Grid Computing

The optimization of the application execution on the Grid is significantly different from its counterparts in traditional parallel and distributed systems. Requirements of Grid Computing make many optimization algorithms that were suitable for systems with a parallel and distributed architecture useless in this new computing paradigm. In this section some of Grid characteristics will be investigated to prove the necessity of introduction of new optimizing algorithms [5]. These are:

- **Heterogeneity**. As explained in section 2.1, in the Grid environment resources are distributed among multiple domains in a computer network. Both the computational, storage nodes and the underlying networks connecting are heterogeneous. This fact influences directly the complexity of optimization process by creating different capabilities for job processing and data access. An optimizer has to cope with system boundaries and resources dependency on external restrictions.
- **Autonomy**. In contrast to traditional parallel and distributed systems, a Grid optimizer usually cannot control Grid resources directly hence it has no ownership of them. The optimizer does not have full information about resources because of their autonomy in the environment. This lack of control is one of the challenges that must be addressed.
  The optimizer cannot violate local policies of resources, which makes it hard for the Grid optimizer to estimate the exact cost of executing a job on different resources – in contrast to the traditional systems where the behaviours of resources are predictable and the process of mapping jobs to resources according to certain performance requirements does not cause any difficulties. In the Grid environment, the most common solution to this issue is to introduce adaptive optimization (see 2.3.3).
- **Performance Dynamism**. This challenge results directly from the autonomy paradigm of the Grid. The successful mapping jobs-to-resources depends on the estimate of the performance that the available resources can provide. However, taking into account the resource autonomy the exact performance cannot be calculated. This is because a resource has to comply to a local policy and cannot guarantee a fixed execution time. The same problem applies to networks that connect Grid resources – the available bandwidth can be heavily affected by network traffic flows which are non-relevant to Grid jobs. A optimizing algorithms should take into consideration this performance dynamism.
- **Computation-Data Separation**. In the Grid environment there is a large number of heterogeneous computing and storage sites connected via wide area networks. The communication bandwidth of the underlying network is limited and therefore the cost for data staging cannot be neglected by optimizing algorithms.

Considering the mentioned properties, a good optimization system on the Grid should have the following features [11]:

- adaptability,
- scalability in managing resources and jobs,
- ability to predict and estimate performance,
- ability to take the cost of resources into account when optimizing,
- ability to take user preferences and site policies into consideration.

2.3.2 Hierarchical Taxonomy of Grid Optimization Algorithms

In the paper [20] Casavant and Kuhl introduced a hierarchical taxonomy for optimization algorithms in generic parallel and distributed computing systems. The Grid optimization process may be classified using this taxonomy (it is visualized on Figure 4) as well.

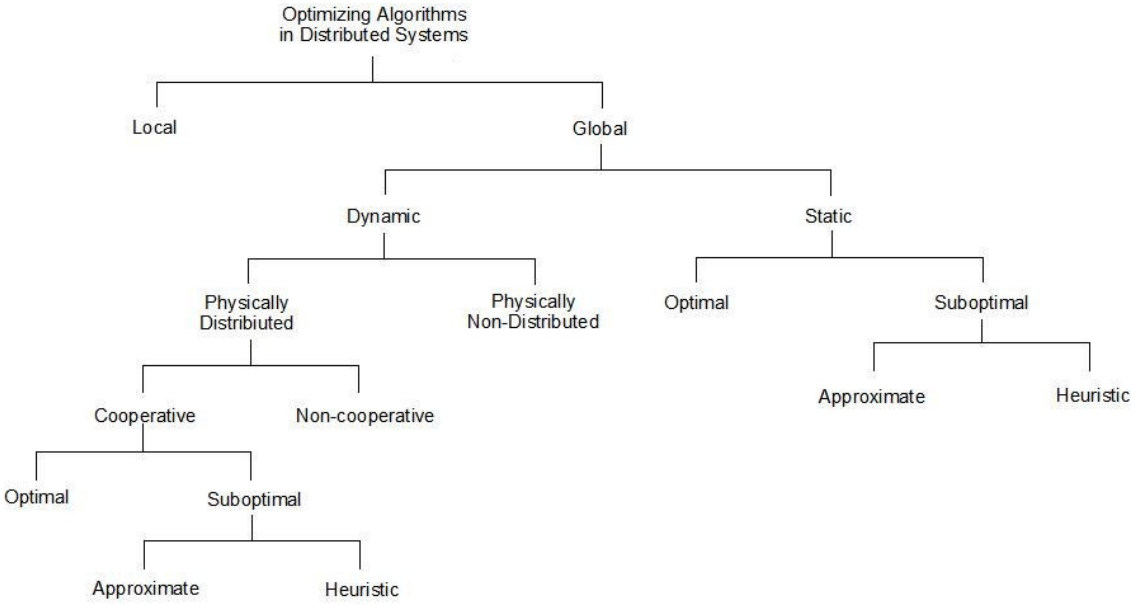


Figure 4: A hierarchical taxonomy of optimization algorithms in distributed systems [20]

The hierarchy can be built using the following criteria:

Number of processors. The criterion divides optimization techniques into local and global one. The former refers to the policy managing processes on a single CPU. In case of global optimization policy, information about the system is used to allocate processes to multiple processors to optimize a system-wide performance objective.
The generic Grid optimization process can be classified as a global one. Thus the other criteria described further in this section will concern the global optimizing policy.

Moment of making the assignment decision. At this level of the hierarchy, a distinction is drawn with regard to the time of making the optimization decisions, and results in division into static and dynamic subgroups.
In the static mode, information about both: all resources in the Grid and jobs in an application is assumed to be available before the optimization of the application. Thus, a firm estimate of the cost of the computation can be made in advance, and the process of the optimization can be simplified. However, such policy can become insufficient in dynamic environments where resources or jobs properties change dynamically.
In the dynamic mode, the possibility of the change of the topology of the system is taken into consideration. Hence, the optimizer may generate a new assignment of jobs to resources during execution of the application. This mode is useful when it is difficult to estimate the cost of

applications or jobs a priori. Dynamic optimization for a job execution has two major components:

- System state estimation – collecting current state information throughout the Grid and constructing an estimate,
- Decision making – using the estimate, assignment of a job to a resource is performed.

The advantage of the dynamic mode is load balancing of resources. It should be introduced when maximizing resource utilization is preferred to minimizing run time for individual jobs. Both static and dynamic optimization policies can be suitable for Grid environments.

Optimality of an algorithm. The optimum solution can be chosen only if all information regarding the state of resources and the tasks are available. However, taking into consideration the NP-complete nature of optimization algorithms and the fact that obtaining the information can be computationally infeasible, the sub-optimum solutions are usually sufficient.
In the Grid environment, the sub-optimum solutions are efficient enough and current research concerns them.

Type of the sub-optimum algorithm. The space of sub-optimum solutions can be divided further into two general categories. First represents approximate algorithms which aim at finding a solution that can be assumed as a good one according to a given metric. Instead of searching entire solution space for an optimum solution. Casavant in the paper [20] suggests metrics such as the time required to evaluate a solution or availability of a mechanism for intelligently pruning the solution space. The second category includes heuristic algorithms which assume having knowledge concerning process and system loading characteristics in advance. The evaluation of this kind of solution is usually based on experiments in the real world or on simulation.
From the Grid optimizer point of view, heuristic algorithms are appropriate hence they can be easily adapted to dynamic nature of the Grid.

Number of optimizers. Three optimization paradigms can be introduced regarding the responsibility for making global optimization decisions [19]:

- **Centralized optimization** – the strategy assumes existence of a single optimizer responsible for optimizing the execution of jobs on all surrounding nodes that are part of the environment. Although the centralized optimizer is easy to implement and can produce better optimization decisions, as it has all relevant information about available resources, it can suffer from the lack of scalability and fault tolerance.
- **Distributed optimization** – this scenario assumes the responsibility of making global optimization decisions is shared by multiple distributed optimizers. Distributed optimization overcomes the problem of scalability and can offer better tolerance and reliability. However, the lack of all the necessary information on available resource, usually can result in sub-optimum optimization decisions.
- **Hierarchical optimization** – in hierarchical optimization, a centralized optimizer dispatches local optimizers a job submission. This type also suffers from problem of the scalability and communication bottlenecks. Nevertheless, its main advantage is that the global optimizer and local optimizer can have different policies in optimizing jobs.

Cooperation between optimizers. This criterion concerns distributed optimization policy. In case of non-cooperative local optimizers, each of them acts as autonomous entity and makes an optimization decision regarding only its own optimum objects independently of the effects of the decision on the rest of the system.
The second mode assumes cooperation between local optimizers. Each optimizer is responsible for its own jobs, but all optimizers are working in order to achieve a common system-wide goal. The local policy of each one pays attention not only local to performance of a particular job, but also takes into consideration the contribution to achieving the global aim.

2.3.3 Extension to the Hierarchical Taxonomy

The hierarchical taxonomy presented in section 2.3.2 does not include all distinguishing characteristics which optimization systems may have. The characteristics do not fit uniquely under any particular branch of that tree-structured taxonomy and can be considered as a flat extension to the scheme presented before. These aspects include [5]:

Objective Functions. This criterion determines the object of the execution of the application. In Grid computing, the functions can be classified into two subcategories [5]:

- **Application centric –** this objective function particularly pays attention to the performance of an application, for example the total cost to run it or a makespan (a period of time between the initialization of the first job and the termination the last job).
- **Resource centric** – such scenario aims to optimize the performance of the resources. Resource-centric objectives are usually related to resource utilization, for example: throughput, which is the ability of a resource to process a certain number of jobs in a given period; utilization, which is the percentage of time a resource is busy.

Adaptive Optimization. An adaptive solution to the optimization problem is the one in which the algorithms and parameters used to make optimization decisions change dynamically according to the previous, current and / or future resource status [5].
In Grid computing, the process of the adaptation may concern:

- **Resources**. Taking into account resource heterogeneity and application diversity, discovering available resources and selecting an appropriate collection of resources can impact on high performance and reduce the cost.
- **Dynamic Performance**. Such kind of adaptation can be regarded as changing the optimizing policy or rescheduling, or workload distributing according to application-specific performance models, or finding a proper number of resources to be used.
- **Application**. In order to achieve the high performance, an optimizer dedicated and specific to a given application can be introduced. The common solution in such a situation is to divide the optimizer into two components. First of them is then responsible for the core process of the optimization, whereas the second is application-specific (e.g. performance models) and platform-specific (e.g. resource information collection). The advantage of such solution is the fact that a core of the optimizer can be general-purpose, when dedicated application-specific components with well-defined interfaces will be responsible for the communication.

Dependencies Between Jobs in the Application. In the section 2.2.1 a Grid application was defined as a collection of jobs. In addition, the jobs can be dependent or independent of each other within one application. Usually, the dependency means there are precedence orders existing for jobs. Taking into account this issue is crucial hence different and more effective optimization techniques can be used according to related jobs. The jobs can be presented in the form of a directed acyclic graph (DAG), in which a node represents a job and a directed edge denotes the precedence orders between its two vertices.

2.3.4 Optimization Algorithms Suitable for Optimization on Grid

In this section the most common and suitable for the Grid Computing optimization algorithms will be described. These techniques are used during the second step of phase "System Selection" in the Grid optimization process (see section 2.2.3). Each of them will be classified according to the described in sections 2.3.2 and 2.3.3 taxonomies. Description of these algorithms was created on the base of the paper [5].

2.3.4.1 Simple heuristics for independent jobs

In case of a collection of independent jobs, some static heuristics algorithms based on execution cost time can be applied. Most common metrics are [22]:

- **Expected execution time** – the amount of time to execute the given job assuming that the selected resource has no load when a job is assigned.
- **Expected completion time** – the wall-clock time at which the selected resource completes the execution of the given job (after having finished any previously assigned jobs).

Using this terminology several simple heuristics for optimizing the execution of independent jobs can be introduced [17]: *Min-min*, *Max-min*, *Suffrage* and *XSuffrage*. These heuristics in an iterative way assign jobs to resources by considering jobs not yet optimized and computing their expected Minimum Completion Time (MCTs) on each of the available resource and finding the minimum completion time over all the resources. For each job, a metric is computed using their MCTs and the job with the best metric is assigned to the resource that let it achieve its MCT. The process is then repeated until all tasks have been optimized [22]. The distinction between the heuristics is defined by the function that determines the metric:

- **Min-min** – the metric is the lowest MCT. The advantage of this method is load-balancing of the resources.
- **Max-min** – on the contrary, the job with the maximum minimum completion time is assigned to the corresponding resource. The heuristic attempts to minimize the penalties incurred from performing tasks with longer execution times [5].
- **Suffrage** – a heuristic based on the idea that better schedules can be found by assigning a machine to a task that would "suffer" most in terms of expected completion time if that particular machine is not assigned to it. For each job, its suffrage value is defined as the difference between its best MCT and its second-best MCT [5].
- **XSuffrage** – an improved Suffrage heuristic proposed by Casanova et al [9], which instead of computing the suffrage between nodes this metric gives a cluster level suffrage value to each job.

These general and simple optimization algorithms does not consider QoS, which can affect their performance in a general Grid environment. Moreover, in heterogeneous systems, the effectiveness these algorithms is also affected by the rate of heterogeneity of the jobs and the resources as well as the consistency of the job estimated completion time on different machines [5].

2.3.4.2 Dependent Jobs Optimization

In case of dependent jobs optimization, the precedence order of jobs is required in advance. Hence, these techniques are kinds of static optimization. The simple way to present the dependence of jobs is to construct a directed acyclic graph (DAG), in which nodes represent jobs and edges represent the data dependencies among the jobs..

In dependent jobs optimization the dichotomy between maximum parallelism and minimum communication appears. High level of parallelism means dispatching more jobs simultaneously to different resources, thus increasing the communication cost, especially when the communication delay is significant [5].

In the following section a different approaches to the problem will be presented with some example algorithms.

a. List Heuristics

The common idea of list optimization heuristics is to make a list of jobs and execute these from the front of the list. Thus it consists of two phases: a job prioritizing phase and a resource selection phase. During the first, one priority of each job is computed to make a ready ordered list. Then the most appropriate resource is selected for the current highest priority job during the resource selection phase [13].

Two important issues regarding this heuristic appear: how to compute a job priority and how to define the cost function. Considering the job priority, it has to be assumed that the priority of the job must be set before any mapping decision is made. To solve this problem, approximations of the job node weight (it represents the computation cost) and the edge weight (representing the communication cost) are used. Two attributes are introduced in order to compute the priority of the job: t-level (top level) and b-level (bottom level). The former is the length of the longest path reaching the job, the latter being the length of the longest path beginning with the job [21]. The most common heuristics of resource selection phase are:

- Earliest-finish-time-first.
- Assigning critical-path job to a resource.

Both heuristics were proposed by Topcuoglu et al. [18].

Their realization are:

- *Heterogeneous Earliest Finish Time* – the algorithm sets the weight of a job as the average execution cost of the node among all available resources. Similarly, the weight of the edge is the average communication cost among all links connecting all possible resource pair combinations. The priority of a job node is the b-level attribute of that node, which is based on mean computation and mean communication costs. Hence, during the resource selection phase the job with the highest priority is picked for allocation and a resource which ensures the earliest completion time is selected. A disadvantage of this method is the possibility of falling into local optima like a greedy method in case determining the earliest finish time [13].
- *Dynamical Critical Path (DCP)* – the algorithm was proposed by Kwok et a. [14]. It does not maintain a static optimization list, but selects the next job node to be optimized dynamically. At each step of the optimization, it computes the dynamic critical path (DCP), that can be defined as a critical path in job graph on which the sum of computation cost and communication costs is maximized. In order to reduce the value of DCP, the node selected for optimization is the one that has no unoptimized parent on the DCP [21]. While determining the resource, the algorithm does not examine all resources, but considers only the ones that hold the nodes communicating with the one in question. The resource with a job on the critical path has an earliest start time is selected.

Some other approaches such as Mapping Heuristics or Fast Critical Path (FCP) exist [21]. In the case of Grid Computing, it is important to pay attention to the fact that algorithms should consider the heterogeneity of resources, jobs and communication links as well.

b. Clustering Based Heuristics

The main idea of these heuristics is to group heavily communicating jobs to the same cluster and then assign jobs in a cluster to the same resource. The clusters can be classified as a linear or as a nonlinear one. A cluster is called nonlinear if two independent jobs are mapped in the same cluster, otherwise it is called linear [21].
The general algorithm can be divided into two phases [5]:

- job clustering phase – it is responsible for partitioning the original job graph into clusters,
- post-clustering phase – it can refine the clusters produced in the previous phase and get the final job-to-resource map.

Some most common particular algorithms using this heuristic is Dominant Sequence Clustering and CASS-II [5].

c. Duplication Based Algorithms

The main idea behind duplication based optimizing is to use resources' idle time to duplicate predecessor jobs. In this way, some of the more critical jobs of a parallel program are duplicated on more than one resource. Thereby it can potentially reduce the start times of waiting tasks and reduce the communication cost.

The difference between particular algorithms of the heuristic is the strategy for selecting the job for the duplication. The examples of this method are TDS (Task Duplication-based Scheduling Algorithm) and TANH (Task duplication-based scheduling Algorithm for Network of Heterogeneous systems) [5].

2.4 Summary

The aim of the chapter was to introduce the current state of the research into the optimization issue in Grid Computing. After presenting the core of the optimization problem, the taxonomy of the optimization techniques was shown. Thanks to the taxonomy, the issue can be considered from different points of views and can simplify the process of the design of the optimizer for the ViroLab Runtime System. In Chapter 3 the taxonomy will be used in order to specify the optimization model offered by the ViroLab Runtime.
The algorithms described in section 2.3.4 will be analyzed from the point of view of their usefulness in the developed system in Chapter 3.

Chapter 3: Analysis of Optimization Issues in ViroLab Virtual Laboratory Runtime System

In this chapter, basing on the information provided in Chapter 2 and Chapter 1, a detailed specification of the optimization problem in ViroLab Virtual Laboratory Runtime System was provided. The problem is discussed with regard to the general optimization issue in Grid environment that was introduced in section 2.2, including a comparison of the general optimization process on Grid (2.2.3) with the optimization specific to ViroLab. Afterwards, the requirements for the optimizer and its model are identified. Finally, the common optimization techniques are discussed with concerning their possible utility in the optimizer with regard to the specified requirements.

3.1 Grid Optimization Problem in ViroLab

Although the component inside ViroLab Runtime System that executes subsequent commands of an experiment (Grid Operation Invoker) uses Grid Object Instances (for the Grid Object-related vocabulary see section 1.2.1) to invoke Grid Operations of a given Grid Object Class, it is not able to decide which class's instance would be the best to perform the operation. This is because it neither acquires sufficient information to make such a decision, nor is it able to process it.
Therefore, in order to improve the ViroLab Runtime System performance, a mechanism that chooses an optimum Grid Object Instance or requests creating it (passing all required data) has to be developed. One of the goals of this thesis (compare 1.4) is to design and develop this mechanism in the form of an optimizing engine – later called ViroLab Optimizer or simply optimizer, that maintains communication channels to other ViroLab components.

3.2 The Optimization in ViroLab Runtime System as an Example of Optimization Process for Grid Computing

The problem of the optimization in ViroLab can be regarded as a special case of the generic optimization process in Grid Computing, described in section 2.2. The vocabulary of ViroLab environment (section 1.2.1) can be mapped into the general Grid terminology (section 2.2.1) in the following way:

- a Grid Object Instance – a resource
- a Grid Operation – a job
- a ViroLab experiment – a Grid application.

A core optimizing process has to be present during executing an experiment inside ViroLab Runtime System.

Thanks to the component architecture of ViroLab Virtual Laboratory Runtime (see section 1.1.4), the ViroLab optimizer will not have to be responsible for all steps during the optimization process. It will be able to interact with other components which are a part of the runtime and have access to the information gathered by components of the Middleware layer. For the description and visualization of ViroLab components please consult section 1.1.4, Figure 1 and Figure 2.
In the following sections the grid optimization process will be presented from the ViroLab environment point of view.

3.2.1 Optimization Results Consumer

The only consumer of optimization results is ViroLab Invoker – Grid Operation Invoker (GOI). It is responsible for creating Grid Object Instances or contacting existing ones, and invoking Grid Operations on them. The Invoker provides an identification of class of the Grid Object (Grid Object Class) it is going to use, and obtains a Grid Object Instance or all the data necessary to create it.

3.2.2 Optimization Process Phases

In section 2.2.3 a general division of the Grid optimization process into several phases was introduced. In this section these phases will be described from the perspective of the optimization process in ViroLab Runtime System. The following phases should be compared to the presented in section 2.2.3.

Phase 1: Resource Discovery
In this phase a list of potential resources which are available to utilize at this moment should be generated. In a case of ViroLab Optimizer all appropriate Grid Object Implementations, including their running instances for the given by the invoker Grid Object Class should be discovered.
The source of information which are used during this phase is ViroLab Registry that contains information, including generic and static data, related to Grid Objects – Grid Resource Registry (GRR), which is located in ViroLab Runtime. Grid Resource Registry is the only required data source for the optimizer. The step of Authorization Filtering can be neglected because the Grid Resource Registry is assumed to provide information only about the resources to which the access is granted.

Phase 2: System Selection
The result of this phase is finding the best combination of a resource and a given job. Taking into consideration ViroLab Runtime System, it means determining the best Grid Object Instance on which the given Grid Object Operation could be performed.
The phase involves also gathering dynamic information about resources. In case of ViroLab Runtime System the source of such data is ViroLab Monitoring System, which is located in the Middleware layer. The system provides information about the state and condition of the resources on which a Grid Object Instance is running or can be created.
An additional component from the Middleware layer which can be used during this phase is ViroLab Provenance Tracking System – PROToS. PROToS can be a source of data related to Grid Object Instances performance in their previous invocations. Utilization of this component can improve the result of the optimization process.
However, when the far-sighted optimization (see section 3.4.3) is performed an additional data about the structure of executed application is required. The source of such information is the manager of ViroLab Runtime – Runtime Library.
The core step of the system selection is performed using the obtained information and according to the specified policy.

Phase 3: Job Execution
In the generic Grid optimizing process this phase is responsible for running a job on a selected resource, while in ViroLab Runtime System this task is assigned to the Grid Object Invoker component. Hence the role of ViroLab Optimizer is only significant during the first and the second phase of the optimizing process. The phase of Job Execution is out of space of the ViroLab Optimizer's activity.

3.2.3 Communication Channels to Other ViroLab Components

The data flow diagram on Figure 5 summarizes the interactions the ViroLab Optimizer with other ViroLab Components during the optimization process.

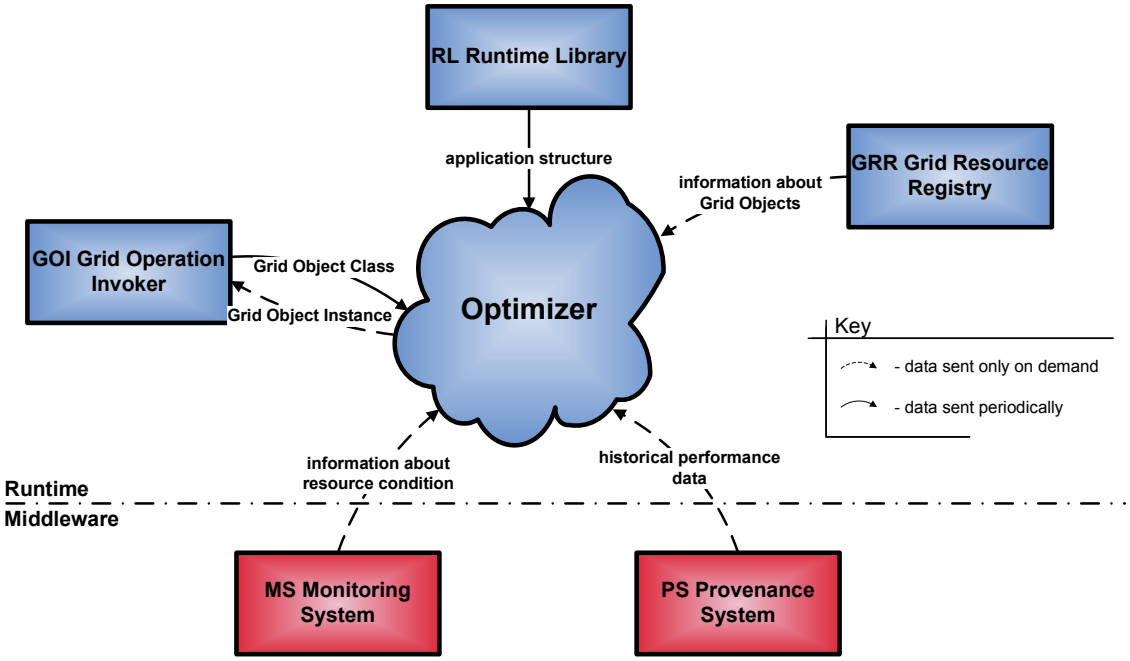


Figure 5: Data flow between the optimizer and its data sources and consumers

To work correctly, ViroLab Optimizer should require only the connection to Grid Resource Registry – in order to retrieve the information it has to pass to the Grid Operation Invoker (and of course to the invoker itself). Such solution is not optimum, since the information stored in the registry is insufficient to make any accurate prediction of performance. Therefore, interactions with other sources of information significantly improve the quality of solution the optimizer chooses

3.3 Requirements for ViroLab Optimizer

This section describes the basic functional (3.3.1) and non-functional (3.3.2) requirements that ViroLab optimizer should satisfy.

3.3.1 Functional Requirements

To be useful in the ViroLab environment, the optimizer has to meet the basic requirement:

- Provide a possibly accurate answer (or report an appropriate error) to each question of type: which Grid Object Instance (ready or only to be created) of the Given Grid Object Class is most suitable to perform an operation on.

3.3.2 Non-functional Requirements

The most important non-functional requirement for ViroLab Optimizer is that its computation processes cannot bring significant latency in responses to the requests from invoker.
What is more, the optimizer design should comply with the following requirements:

- Clear interface with an appropriate exception handling.
- Easily adjustable to environment – it should be able to adjust to different conditions of its environment – i.e. if the existing sources of information do not provide sufficient data, the optimizer should try to create and utilize new information channels.
- Easily extendible – it should offer a simple way to extend its functionality with new features, more advanced algorithms, analysis of data from new information channels.
- Easily configurable – the optimizer should offer a possibility to configure it externally – this configuration includes: optimization algorithms to be used, utilized data channels, properties constituting a policy of optimization.

3.4 ViroLab Optimizer Optimization Model

The following subsections contain a study of the ViroLab Optimizer characteristics which enables to fully recognize its optimization model. The constraints it has to comply with are identified. On this basis, regarding the classifications of Grid optimization solutions (sections 2.3.2 and 2.3.3) and the characteristics of the optimization problem in ViroLab (3.2), the model of optimization that will be performed is identified.

3.4.1 Constraints Imposed by ViroLab Environment

The ViroLab environment (see section 1.1) puts some restraints on its optimizer. These limitations occur due to the fact that the optimizer does not own the resources it uses. Therefore, it does not have an authorization to administrate them.
The constraints are as follows:

- No direct control over resources – the optimizer can only act as a broker; it cannot guarantee that the task it schedules will be executed on the resource it chooses.
- No exclusive access to resources – the information about resources gathered by the optimizer can be imprecise or out-of-date. Furthermore, the optimizer cannot guarantee that the quality of scheduled jobs will not be aggravated by some tasks executed by the local scheduler.
- No queue – the optimizer is not able to manage the tasks after they are submitted to resources.

3.4.2 ViroLab Optimizer Optimization Type

Basing on the classification introduced in sections 2.3.2 and 2.3.3, the following characteristics of the ViroLab Optimizer can be distinguished:

- Global – it uses information about the system to allocate processes to multiple processors and optimizes a system-wide performance objective.
- A hybrid solution between static and dynamic optimization – the basic functionality offers choosing the optimum resources at runtime (based on information from other ViroLab components). However, this functionality can be extended by providing a plan of resource choices for more than one application's tasks at a time.
- Centralised – it is not distributed.
- Optimum in its basic functionality – it finds the solution that is optimum from the point of view of the basic knowledge it gains; suboptimum – when it is extended, the amount of optimization prerequisites increases and the optimum solution based on the extended knowledge becomes impossible to find. Still, the suboptimum solution obtained with the extended functionality may occur to be better than the optimum solution computed using only the basic mode.

- Application centric – the optimization process pays attention to the performance of the application.
- Adaptive – the process of the optimization can be dynamically adapted to changing situation in the ViroLab environment thanks to optimization policy and algorithms configuration.

3.4.3 Task Dependencies in Optimization Process in ViroLab

The optimizer can be used in one of the following modes – depending on whether the dependencies between subsequent tasks are taken into consideration:

- Short-sighted optimization mode – it implements the basic optimizer's functionality, which is choosing an optimum solution for one task at a time. The other tasks are neglected.
- Medium-sighted optimization mode – the tasks can be submitted to the optimizer in groups – for each of the tasks a resource is chosen regarding the previous choices. The tasks are not reordered, nor are they arranged in queues – the group of tasks is mapped to the group of their solutions (resources).
- Far-sighted optimization mode – a suboptimum solution, includes choosing resources for all tasks of the executed application as well as optimization by ordering the tasks.

The mentioned far-sighted optimization features should introduce significant improvement to the solution. What is more, running in this mode, the optimizer will able to search for optimum solution, regarding the actual Grid Operation[6]. In the other scenarios only the Grid Object Classes are known to the optimizer, and thus only collective performance of all the classes operations can be considered. Nevertheless, such optimization scarcely ever brings solutions for all the requested tasks (the short-sighted optimization must be used to later assign solution to them), thus, in some rare cases it brings only unnecessary computation.

3.5 Analysis of Existing Optimization Algorithms Regarding the ViroLab Optimizer Requirements

In section 2.3 the overview of optimizing algorithms for the Grid Computing were introduced. In this section they will be discussed from the ViroLab Optimizer point of view with respect to its requirements (section 3.3).

In order to perform the optimization process using any algorithm, some sources of information must be available. In case of the ViroLab Optimizer, other ViroLab components will provide necessary data (see 3.2). Simple heuristics use mostly information about the execution cost. Thanks to the availability of information from ViroLab Monitoring System and ViroLab Provenance Tracking System such computations can be performed. However, the issue of the heterogeneous environment can have a significant impact on such estimations. The performance of these algorithms can be affected by the rate of heterogeneity of the jobs and the resources. Unfortunately, this problem concern all simple heuristics for independent jobs. The solution to this problem can be adaptive optimization techniques. Simple heuristics should give results well enough for short-sighted optimization mode (see 3.4.3).

Some more challenges may appear when far-sighted optimization mode is taken into consideration. In these case the problem of optimizing dependent jobs must be analyzed. An additional difficulty is the issue of mapping a ViroLab application script to DAG. Because of the dynamic character of the script, the process of creating the DAG for the application can be complicated. Nevertheless, executing the script in parallel, if possible, should give some improvement in the optimization regarding to short-sighted optimization mode. The heuristics

[6] Choosing the solution with regard to a specific Grid Operation of Grid Object Class could be implemented also as an extension to the previous (short- and medium- sighted modes).

for dependent jobs optimization should be suitable for this mode. Probably, changing the optimization of the whole script at a time into an iterative process of optimizing its parts should also bring a positive impact on the optimization process. In addition, it would make the method less complicated.

Medium-sighted optimization mode can be regarded as a simplified version of the far-sighted optimization mode. Neglecting the dependencies between jobs and setting the appropriate priorities should improve the optimization result in comparison to the short-sighted optimization mode. The heuristics which are the most suitable for this kind of optimization should be adapted from the heuristics for the independent jobs optimization.

The possibility of introducing adaptive optimization is an important issue concerning the ViroLab Optimizer. A suitable configuration and selection of different algorithms is the best solution for the optimization process in this heterogeneous environment.

3.6 Summary

The target of this chapter was to present the analysis of optimization issue in a certain environment – ViroLab Virtual Laboratory Runtime system. It was proved that the issue is a particular case of the general optimization problem in Grid Computing. The identification of the specific requirements for the ViroLab Optimizer enables to state the characteristics of the optimization model suitable for it. These characteristics in turn will be used to determine the optimizer's architecture. In this chapter it was also shown that the requirements determine the collection of possible optimization techniques that can be used in a ViroLab Optimizer.

Chapter 4: ViroLab Optimizer Design

Having the precise ViroLab Optimizer characteristic, a design model for its implementation can be built. This chapter specifies and describes the structure and behaviour of the implementation of ViroLab Optimizer specification provided by Chapter 3 – ***GridSpace Application Optimizer****, in short:* ***GrAppO****.*

Note: Some ideas concerning the GridSpace Application Optimizer design, developed for the purpose of this thesis were also included in the ViroLab design document [4].

4.1 GrAppO Use Cases

On the basis of optimization mode analysis (section 3.4) the following optimizer use cases can be identified:

- **Plan application execution** – corresponds to the first stage of far-sighted optimization mode. It includes creation of the application graph from the experiment code (if the code only is available), ordering the Grid Operation calls in the experiment – such ordering may significantly improve the optimization quality, and assignment of resources for invocation of each Grid Operation in the application (some assignments may not be available at this time). This process introduces significant delay to the application execution, however, in most cases it also minimizes the delay to responding the Grid Operation Invoker requests.
- **Assign resource to Grid Object** – this use case represents the activity performed on Grid Operation Invoker request – either in short-sighted or in the last phase of far-sighted optimization mode (in these two cases the activities of the optimizer may differ).
- **Assign resources to Grid Objects** – an equivalent of medium-sighted optimization mode – the optimizer assigns resources to all the given Grid Objects at a time.

The described use cases are illustrated by Figure 6.

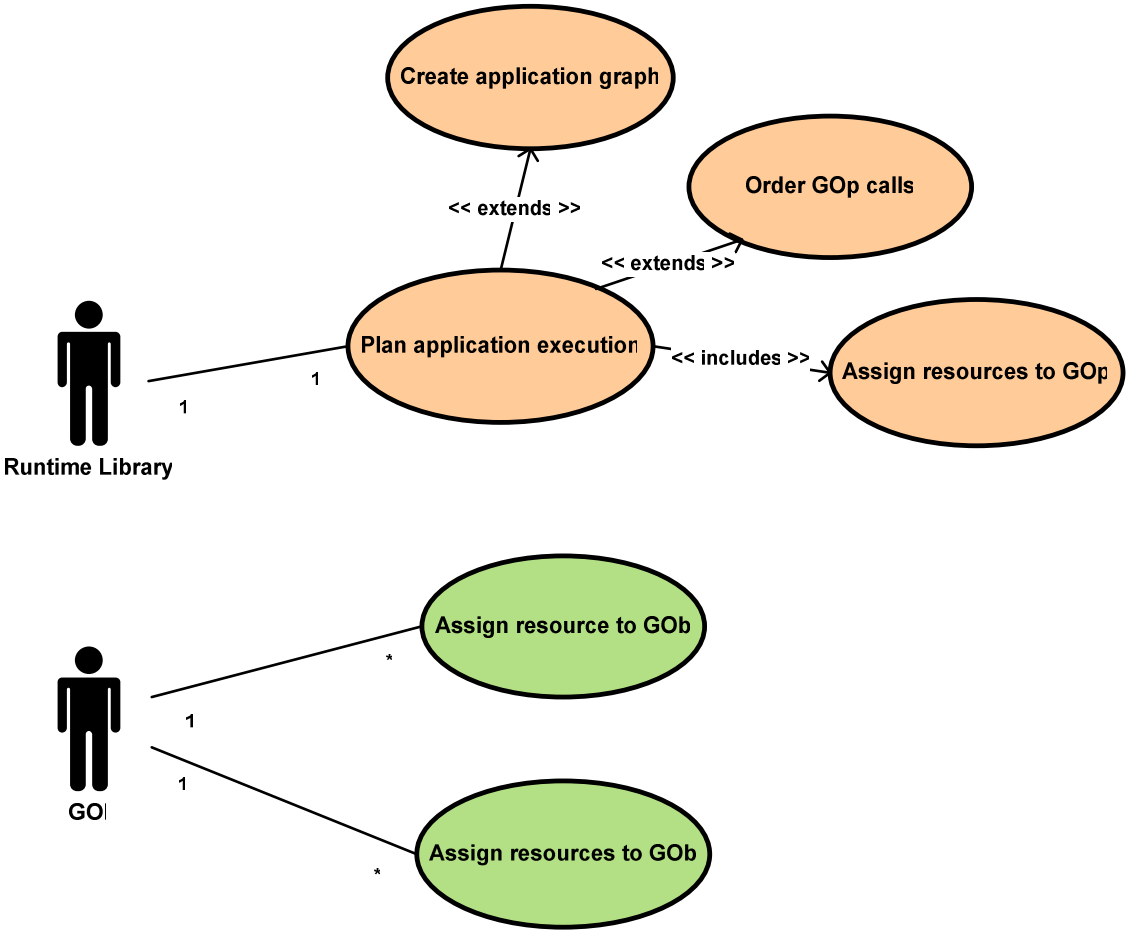


Figure 6: The GridSpace Application Optimizer use cases diagram

4.2 GrAppO Architecture

In the GridSpace Application Optimizer architecture the following components can be distinguished:

- **GrAppO Manger** – a component responsible for coordination of the work of other components. It also contacts the most important external to the optimizer component – Grid Resource Registry and receives calls from Grid Operation Invoker.
- **Optimization Engine** – a heart of the optimization. It executes dedicated optimization algorithms. The optimization in this part can be based on the estimation of each solution[7] performance prepared by the *Performance Predictor*.
- **Performance Predictor** – a component used for making predictions of Grid Object Instances performance – including execution time and resource consumption. It bases on the analysis from *Historical Data Analyzer* and *Resource Condition Data Analyzer*.
- **Historical Data Analyzer** – contacts the external Provenance Tracking System (PROToS) to obtain information about previous executions, analyses it, and prepares for the performance estimation.
- **Resource Condition Data Analyzer** – contacts the external Monitoring System to obtain information about condition of resources that can be used, analyses it, and prepares for the performance estimation.

[7] Further in this document the term 'solution' means 'Grid Object Instance or Grid Object Implementation with link to resource where it can be created'. In some cases also the sole 'Grid Object Instance' can be used in this context.

- **Application Structure Analyzer** – a component used in far-sighted optimization mode (see section 3.4.3 for the explanation of this term). Provided with the application graph or code defines the task assignment to Grid Object Instances and order of execution.

The component diagram on Figure 7 shows the dependencies between components of GridSpace Application Optimizer.

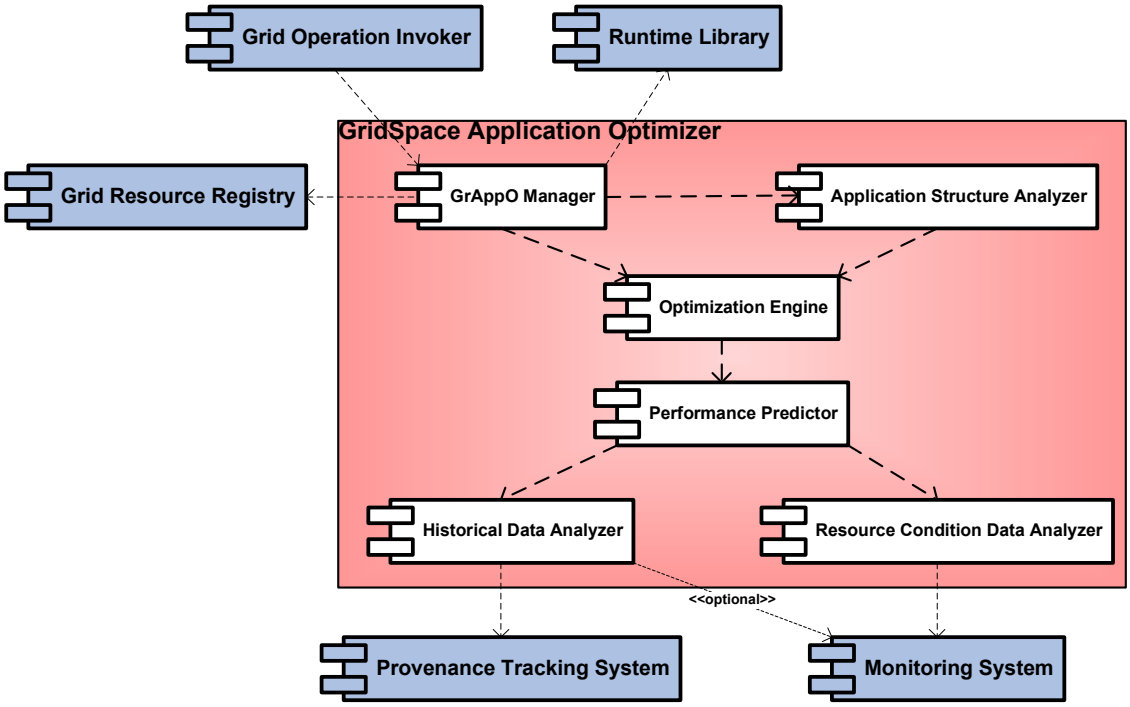


Figure 7: Component diagram, showing GrAppO decomposition and dependencies between its components and external ViroLab components

A detailed description of the data obtained from external components for analysis is provided in section 4.3.
Owing to its modular architecture, any new sources and their data analyzers could be connected to GridSpace Application Optimizer. The only required change is to modify the Performance Predictor to tell it that a new data analyzer should be used.

4.3 Data Provided by External Components

The data connections between the optimizer and other ViroLab components were identified in section 3.2. In this section a detailed study on what information will be passed through these connections is presented.

4.3.1 Grid Resource Registry

Grid Resource Registry is the most important and the only required source of information for the optimizer. The registry lists its contents applicable to the given input (Grid Object Class). That is:

- All the Grid Object Implementations of the given Grid Object Class. Each implementation is assigned an information about:
  - id – identification in the registry, that is passed to the Grid Object Invoker, so that it can obtain information needed for creating its instance
  - service implementation technology – e.g. Web Service, MOCCA Component, WSRF
  - type – e.g. stateful/stateless, shared/not shared

    - constraints – e.g. it can be deployed only on some resources
    - resource consumption – e.g. it requires a certain amount of memory
    - running instances (Grid Object Instances) of the implementation – see the next bullet
- All the Grid Object Instances of the listed Grid Object Implementation or a given implementation. The information attached to each instance includes:
    - id – identification in the registry, that is passed to the Grid Object Invoker, so that it can obtain information needed for contacting the instance
    - location – a link to the resources where the Grid Object Instance is deployed; this information is used to query the resource monitoring service
- All the resources, where Grid Object Implementations could be deployed to create a Grid Object Instance – e.g. H2O kernels for MOCCA Components. The only required information about them is location. As in case of Grid Object Instance it is a link to the resource, used while querying the monitoring service

As the Grid Resource Registry cooperates with Monitoring Service, it controls the state of its contents, and thus never returns any object that is not functioning – i.e. neither Grid Object Instances that were destroyed, nor service containers that are not running. However, it does not control whether the instances or containers work correctly.

4.3.2 Runtime Library

A communication channel to this component is not required and is used only when the optimizer works in far-sighted optimization mode (see 3.4.3). The data provided by Runtime Library concerns:

- Application graph or code. If it is the code – the optimizer has to build its graph.
- Application context – the identification and configuration of the experiment that is to be executed. This information is used when the invoker requests the optimization – the request have to be associated with the experiment the invoker is analyzing.

4.3.3 Monitoring System

The information obtained from the monitoring helps to choose the resource that is in optimum condition to handle Grid Operations execution. It includes:

- Information about the hardware of the node on which the resource is operating (e.g. CPU, RAM)
- Information about state and condition of the node – e.g. its load, available memory

Another connection to monitoring system can be created if no provenance implementation is available. Through this connection then, notifications of events concerning the lifecycle and executions of Grid Object Instances is sent in a publisher/subscriber mode. This data, gathered by the optimizer, could be interpreted to compute the Instances performance (e.g. execution time, resource consumption) – in this way replacing information that in other circumstances would have been obtained from Provenance Tracking System (4.3.4).

4.3.4 Provenance Tracking System (PROToS)

Analysis of the information from Provenance Tracking System together with monitoring data can lead to a very accurate time and resource consumption estimation. From the provenance data, information about how a certain Grid Object Instance performed on each resource, is extracted. This information includes:

- Resource consumption – e.g. how much RAM or CPU it used
- Duration of the executed Grid Operations of given Grid Object Instance – time that was needed for the operation to be completed.

4.4 Alternative to the Connection to Provenance Tracking System

When no implementation of Provenance Tracking System exists in GrAppO environment, the connection to that system can be replaced by *Performance Data Database*. It stores the same information relevant to Grid Object Instances lifecycle as Provenance Tracking System. These data are obtained through an additional information channel to Monitoring Service, based on the publisher/subscriber communication mode. The information from Monitoring Service are gathered by a Historical Data Analyzer component subclass: *Monitoring Notification Listener*. The listener awaits notifications from the monitoring, and if such occurs, interprets it and stores in proper format in Performance Data DB.
A schema of this solution can be viewed on Figure 8.

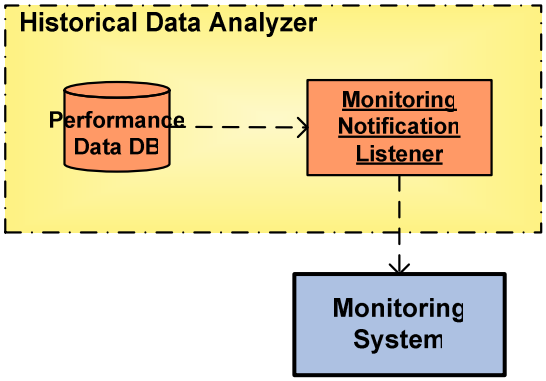


Figure 8: Structure that is an alternative to the connection to Provenance Tracking System

Such solution is a way to adapt to an environment where some of the information sources are missing, still acquiring sufficient data to perform optimization, in this way fulfilling the requirement of the optimizer being adjustable to the environment (compare 3.3.2).

4.5 GrAppO Optimization Algorithm Plug-ins

The algorithms GrAppO uses for optimization are pluggable as each of them has to implement the same interface. Which of the algorithms will be used for optimization is a matter of external configuration. However, there always is a default optimization algorithm available.
Such solutions On Figure 9 a sample relation between the general optimization algorithm and its different implementations is shown.

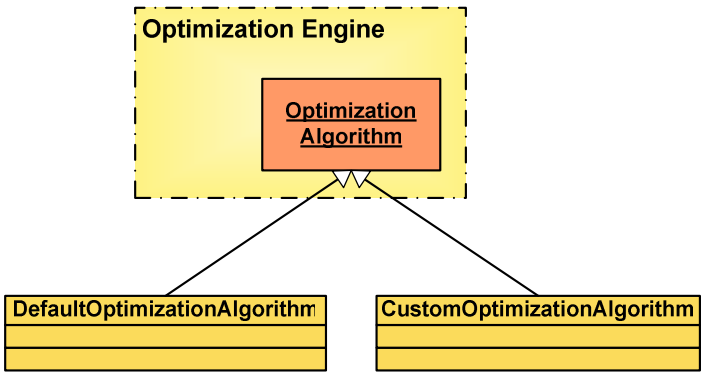


Figure 9: A relation between an optimization algorithm and its different kinds

Obviously, instead of the *CustomOptimizationAlgorithm* that appeared on Figure 9 any optimization algorithm implementations can be plugged. The default optimization algorithm is required, while its implementation is unrestricted.

4.6 GrAppO State During Execution of Applications

Running in the basic optimization modes (short-sighted and medium-sighted optimization – see section 3.4.3), GrAppO acts like a stateless service. After its creation, each call to it from Grid Object Invoker is independent of previous calls.
In case of far-sighted optimization mode, the optimizer state has to be maintained in order to identify the application that was optimized, in the call from the invoker. This is required since the optimization is done before the Grid Object Invoker requests it, and the invoker obtains previously prepared data. This section presents all possible GrAppO states and transitions between them concerning the three possible optimization modes.

4.6.1 Short-Sighted and Medium-Sighted Optimization Modes

The analysis of GrAppO state changes during its lifecycle in short- or medium- sighted optimization mode is depicted on Figure 10 (in both modes the states and transitions are the same). Its detailed description can be found further in this subsection.

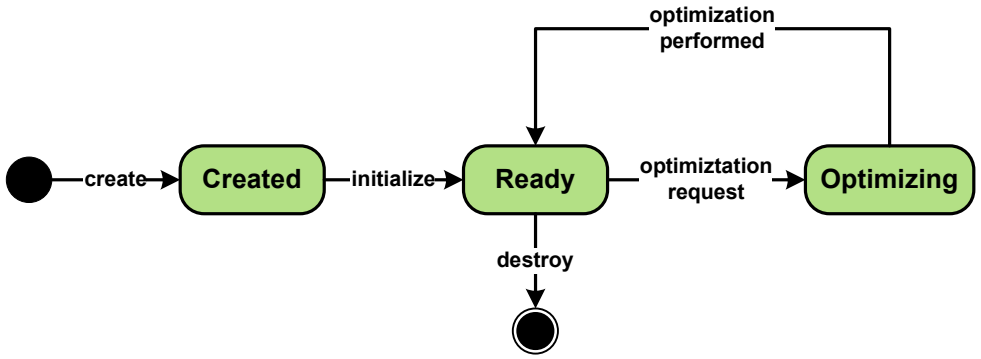


Figure 10: A state diagram of the optimizer performing short- or medium-sighted optimization

The allowed states of the optimizer performing short- or medium- sighted optimization are as follows:

- **Created** – a state in which the optimizer is after invocation of its constructor. In this state GrAppO is not yet initialized, therefore it cannot perform any optimization.
- **Ready** – initialized optimizer waits in this state for incoming optimization requests from the invoker.
- **Optimizing** – GrAppO performs optimization (on request of Grid Operation Invoker).

In this scenario, the following transitions between states are allowed:

- **Null→Created** – this transition is triggered by a call to GrAppO constructor, during the transition the optimizer is being created.
- **Created→Ready** – initialization of the optimizer.
- **Ready→Optimizing** – a transition triggered by an optimization request from the invoker, GrAppO prepares to the optimization then.
- **Optimizing→Ready** – after performing optimization (state: Optimization), results are returned to Grid Object Invoker and GrAppO is ready for the next request.
- **Ready→Null** – destroying the optimizer instance, could be done automatically by a garbage collection mechanism.

4.6.2 Far-Sighted Optimization Mode

Figure 11 presents the process of state changes in the far-sighted optimization scenario. Please see the next paragraphs for its explanation.

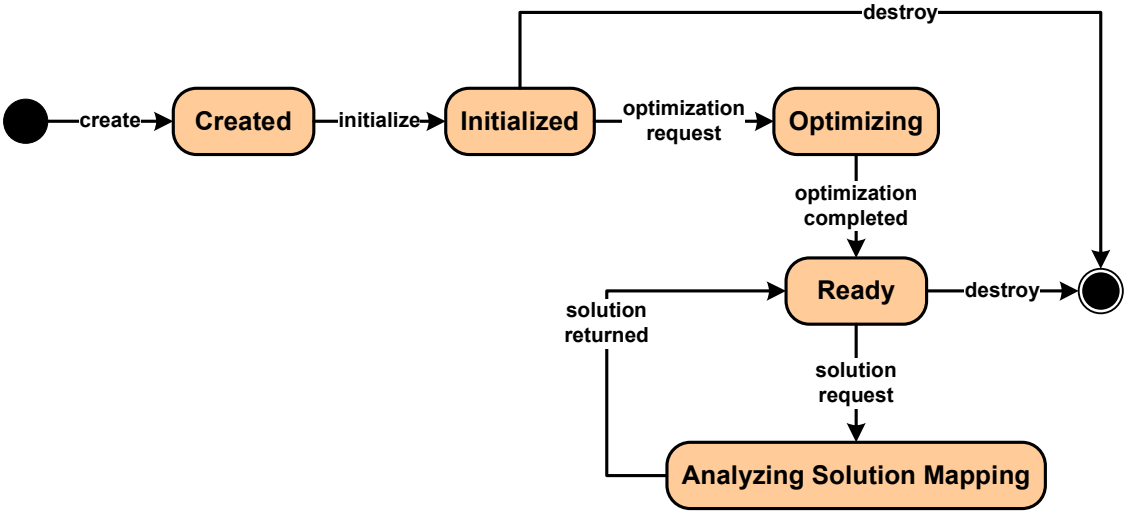


Figure 11: A state diagram of the optimizer running in far-sighted optimization mode

During far-sighted optimization, the optimizer can reach the following states:

- **Created** – a state in which the optimizer is after invocation of its constructor. In this state GrAppO is not yet initialized, therefore it cannot perform any optimization.
- **Initialized** – initialized optimizer waits in this state for optimization of the whole application request from Runtime Library.
- **Optimizing** – GrAppO performs the application optimization (on request of Runtime Library). A mapping Grid Object Class : solution is created.
- **Ready** – at this stage the optimization is completed and the optimizer is waiting for incoming solution assignment requests from Grid Object Optimizer.
- **Analyzing Solution Mapping** – GrAppO analyzes the solution mapping created during the optimization and, on that basis, chooses solution for the Grid Object passed by the invoker. In some cases an additional optimization (similar to the one in short-sighted mode) is needed.

The following transitions are allowed:

- **Null→Created** – this transition is triggered by a call to GrAppO constructor, during the transition the optimizer is being created.
- **Created→Initialized** – initialization of the optimizer.
- **Initialized→Optimizing** – a transition triggered by an optimization request from Runtime Library.
- **Optimizing→Ready** – the optimization is being performed and the results stored as a list of Grid Object Class : solution mapping.
- **Ready→ Analyzing Solution Mapping** – a transition triggered by a solution assignment request from Grid Operation Invoker. For invoker the request could be indistinguishable from the GOI's optimization request in previous scenario. However, in this case, the request must carry an application identifier.
- **Analyzing Solution Mapping→Ready** – after analysis of the solution mapping that that takes place in the Analyzing Solution Mapping state, the results are returned to Grid Operation Invoker.
- **Ready→Null**, **Initialized→Null** – destroying the optimizer instance – could be done automatically by a garbage collection mechanism.

4.7 Control Flow Inside GrAppO

The subsections 4.7.1, 4.7.2, and 4.7.3 describe the general sequence of calls between GridSpace Application Optimizer components and external subsystems during optimization in short-, medium-, and far- sighted optimization scenarios, respectively.

4.7.1 Short-Sighted Optimization Mode

The flow of control in GridSpace Application Optimizer while performing short-sighted optimization is illustrated by the sequence diagram on Figure 12. The detailed description of each method call is provided further in this subsection.

The interpretation of the calls presented on the diagram is as follows:

1: Grid Operation Invoker requests optimization – an assignment of Grid Object Instance or data necessary to create it to the given Grid Object Class. It is also possible to pass a Grid Object Implementation name to GrAppO (namely, its façade – GrAppO Manager) – only the solutions including this implementation will be chosen then.

2: GrAppO Manger queries the Grid Resource Registry for any relevant information about the obtained Grid Object Class (or Grid Object Implementation). These information was specified in section 4.3.1.

3: GrAppO Manager provides the main optimization engine (Optimization Engine) with the data obtained from registry and requests for the solution.

4: Performance Predictor is called to estimate invocation time and resource consumption for each solution. It is provided with the data from the registry.

5a: Request for analysis of the resources condition to a dedicated component – Resource Condition Data Analyzer. URLs of the resources are passed as an argument of the call.

6a: Resource Condition Data Analyzer queries the Monitoring System for resource state information with the obtained URLs.

7a: The results of the analysis are returned to Performance Predictor.

5b: Request for analysis of the historical data concerning previous Grid Object Instances performance. It is sent to a dedicated component – Historical Data Analyzer. The call passes a list of Grid Object Implementations and a list of URLs to resources.

6b: Historical Data Analyzer queries the Provenance Tracking System for the information with the obtained Grid Object Implementations and resource URLs. This two arguments allow the Provenance Tracking System to find: for all Grid Object Implementations and for all URLs: all the Grid Object Instances that were built from the given implementation on the given URL. When no Provenance Tracking System implementation is available, the call is replaced by a query to Performance Data Database (see section 4.4 for a detailed description of this solution).

7b: The results of the analysis are returned to Performance Predictor.

8: On the basis of the analysis results, the estimation of execution costs for each solution is calculated.

9: The results of the estimation are returned to Optimizer Engine.

10: Basing on the estimation made by Performance Predictor and using certain optimization algorithm, Optimizer Engine chooses the most optimum solution.

11: The solution is returned to GrAppO manager and then

12: to Grid Operation Invoker.

Note: Operations 5a-7a can be performed in parallel with 5b-7b.

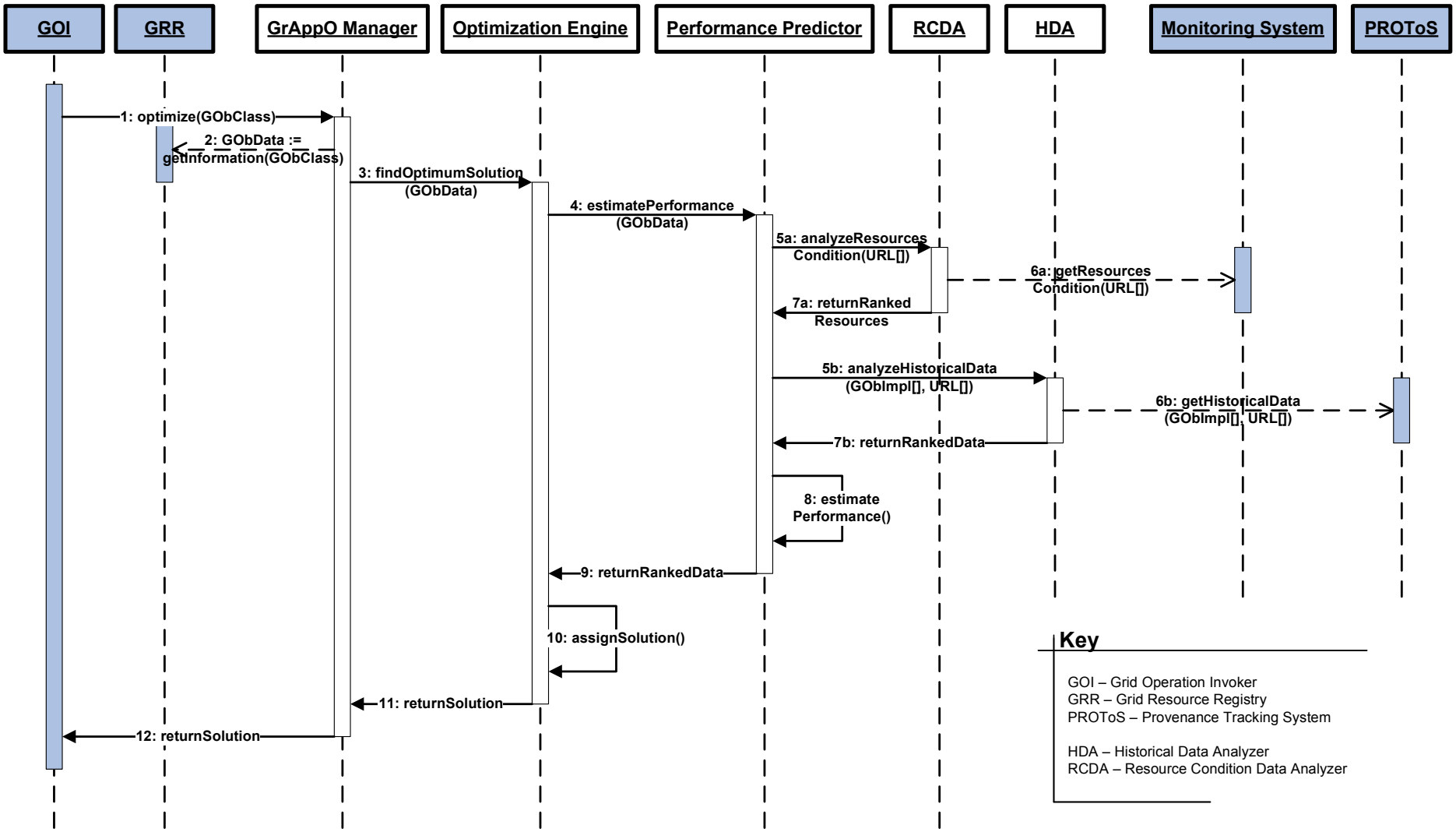


Figure 12: A sequence diagram illustrating the flow of control in GrAppO during short-sighted optimization

4.7.2 Medium-Sighted Optimization Mode

The sequence diagram (Figure 13) in this scenario is very similar to the one in case of short-sighted optimization (Figure 12). The main difference can be seen in method calls, which carry a list of arguments (or contain a list of solutions in the resulting value) instead of just one. The main scheme of the optimization is the same as in the previous case.

In comparison to the control flow in short-sighted optimization mode, the significant difference appears in the first calls. In this scenario, the request from Grid Operation Invoker (**1**) contains a list of Grid Object Classes instead of just one. Similarly, the query to Grid Resource Registry (**2**) includes a list of classes.
Certainly, the implementation of subsequent operations has to support the case of multiple Grid Object Classes, but the methods **5**-**8** can be exactly the same. The value returned to the invoker also has to carry solutions for all the requested Grid Object Classes (in the form of Grid Object Class : solution mapping).

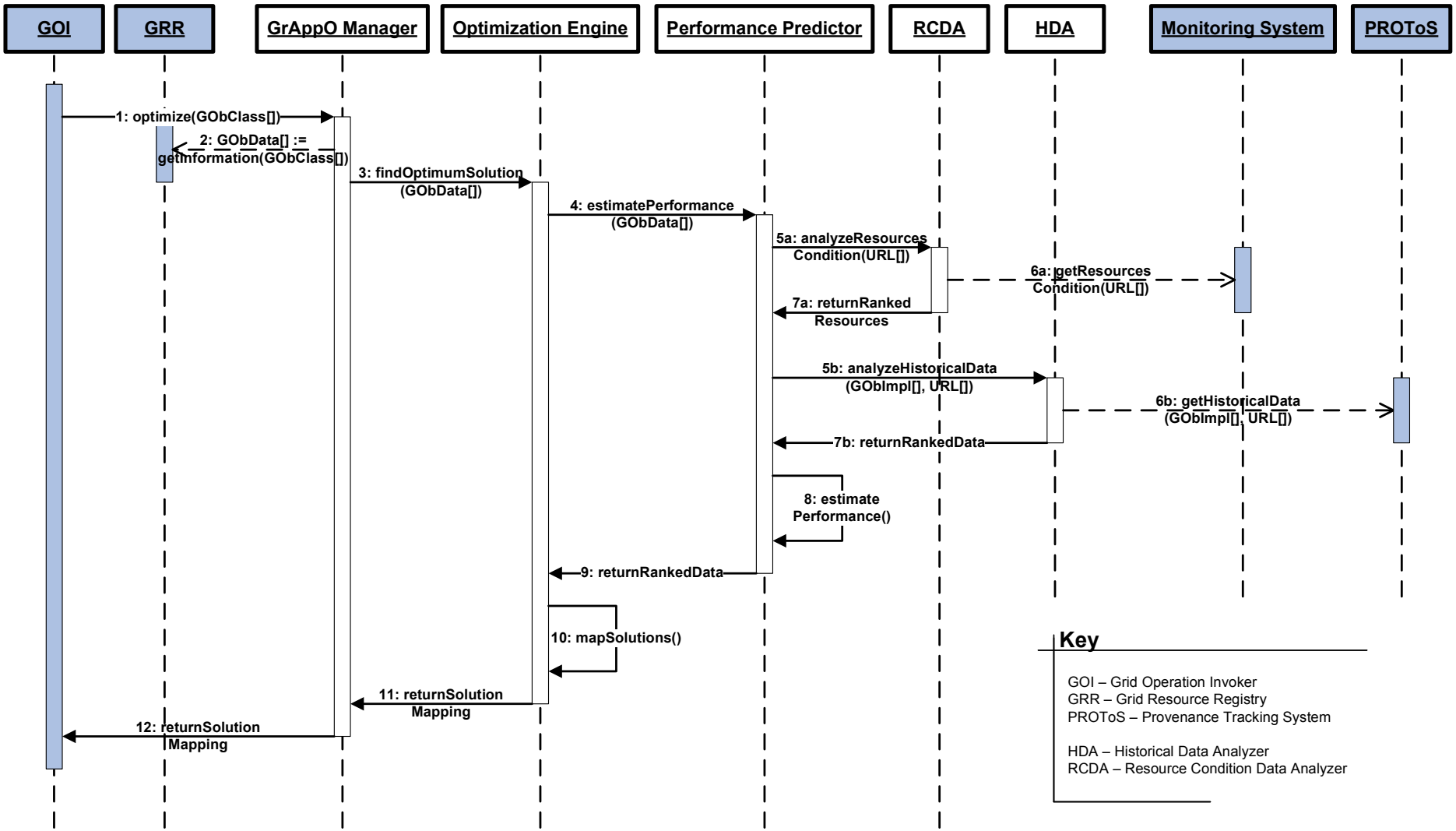


Figure 13: Part of a sequence diagram that illustrates the flow of control in GrAppO during medium-sighted optimization

4.7.3 Far-Sighted Optimization Mode

The control flow in far-sighted optimization can be divided into two phases:

1. The solution mapping is being created – whole application optimization is being performed. The mapping form is similar to the one used in case of medium-sighted optimization.
2. The optimizer is answering the requests from Grid Operation Invoker with already assigned solution.

GridSpace Application Optimizer has to maintain its state throughout the two phases and between them.
The sequence diagrams for the two phases are displayed on Figure 14 and Figure 15, respectively.

The first phase is very similar to medium-sighted optimization. However, in this case, it is Runtime Library that requests the optimization (**1**) – not Grid Operation Invoker. The request parameter is different as well, since here it has to be the whole application. What is more, to query Grid Resource Registry, the Grid Object Classes first have to be extracted from the application code or graph (**2**). During this process also an application graph can be built and the Grid Operation calls reordered.
The final solution mapping (**11**) should contain solutions for all the application classes. Some of them may not be possible to compute at this stage though. In such case, for some of the classes, an additional optimization (search for solutions) have to be performed in the second phase.

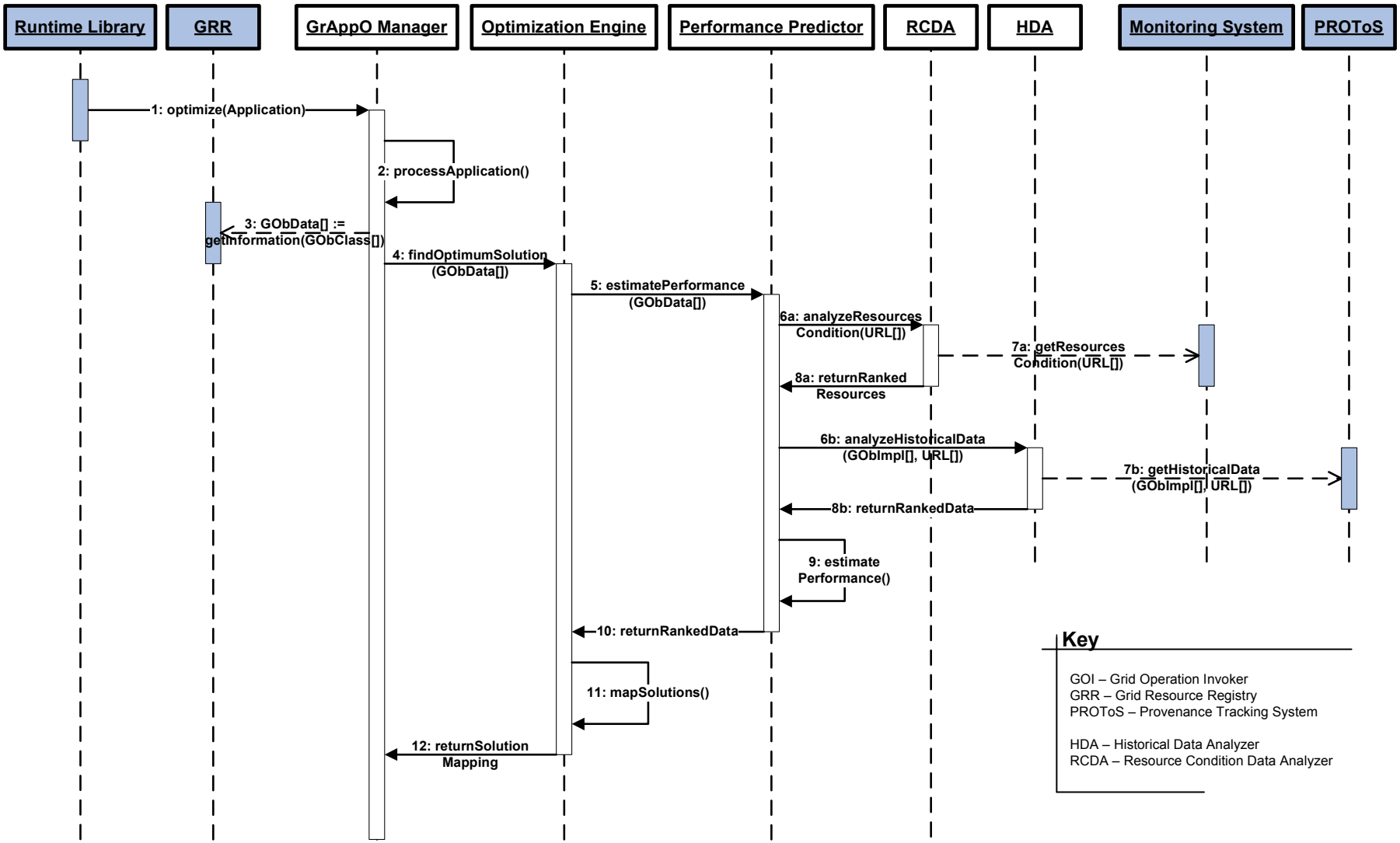


Figure 14: A sequence diagram illustrating the flow of control in GrAppO during the formation of solution mapping (first phase) in far-sighted optimization

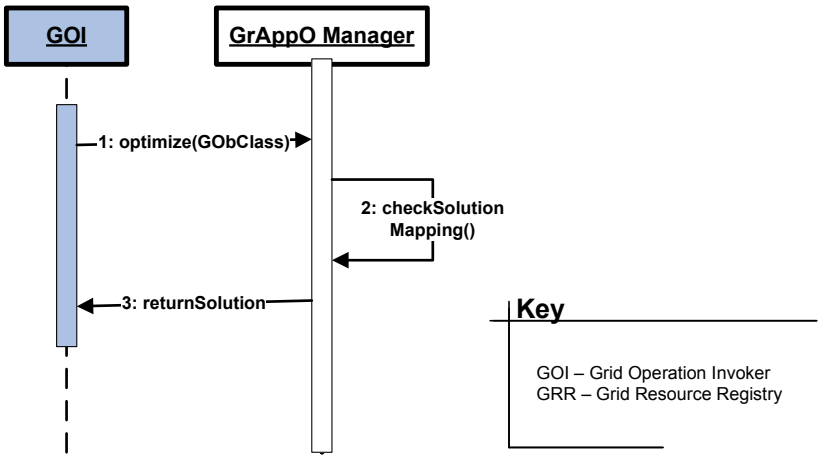


Figure 15: A sequence diagram illustrating the flow of control in GrAppO while answering the requests from GOI (second phase) in far-sighted optimization

In the second phase GrAppO awaits for requests from Grid Operation Invoker, keeping the session. When a request occurs (**1**), the optimizer analyzes the solution mapping (**2**) to find a solution for the requested Grid Object Class, which is then passed to the invoker (**3**). If no solution was provided, the optimization like in short-sighted mode is performed.

4.8 Summary

In this chapter the design of the implementation of ViroLab Optimizer – GridSpace Application Optimizer (GrAppO) was provided. Its architecture was prepared regarding the requirements specified in section 3.3 and for each of the optimization modes described in 3.4.3 an evaluation scenario was presented – including the analysis of GrAppO states and the control flow among its components.

The chapter provided a complete design of the requested optimization system – GridSpace Application Optimizer. However, considering the progression of work on its environment, some of the designed features are currently impossible to implement (the scope of the implementation is specified in section 5.1).

Chapter 5: GridSpace Application Optimizer Implementation Details

In the following chapter implementation of the most important modules of GridSpace Application Optimizer is described, beginning with the scope of the implementation. The way of using GrAppO is then presented, including its configuration details and API description. The class diagrams can be viewed in Appendix B. In the last section the tools used during GrAppO design and development process are listed.

5.1 Current GrAppO Implementation Scope

For the purpose of this thesis only short- and medium- sighted optimization was implemented. This is because the far-sighted optimization introduces a great amount of new problems to be solved e.g. building the application graph[8], not bringing any significant performance improvements in comparison to medium-sighted optimization. What is more, the Runtime Library component is not yet adapted to provide the application in any form to GrAppO.
Although the connections to Monitoring and Provenance Tracking Systems are not yet available (the systems are not configured to receive and answer requests), the analysis of data obtained from the Monitoring and Provenance Tracking Systems is implemented in GrAppO – using mock components that produce test data. The listener and database gathering provenance data from Monitoring infrastructure (the solution is described in 4.4) were not implemented.

5.2 GrAppO Configuration

One of the required features (see section 3.3) of GridSpace Application Optimizer is to be able to use different, pluggable optimization algorithms and policies. As a realization of this mechanism, an Optimization Policy was introduced.
Since the optimization quality improves when it interacts with other ViroLab components (see section 4.3) a way of configuration access to them is provided in GrAppO as well. By using this option, default links to services of these components can be replaced with links to their other instances offering a wider and more accurate set of information and/or a better connection quality.

5.2.1 Optimization Policy Usage Schema

The Optimization Policy can be configured externally, in a static way or passed from an application execution service. Therefore, the configuration is implemented in two ways: it

[8] This functionality is planned to be implemented as an external module, placed in ViroLab Virtual Laboratory Runtime, as other ViroLab subsystems may need to use the graph as well.

could be specified either by an instance of inner GrAppO class, or with an external XML or properties text configuration file. A visualization of the two possible solutions is presented on Figure 16 and Figure 17, respectively. A sample property (optimization algorithm) configuration is also illustrated there.

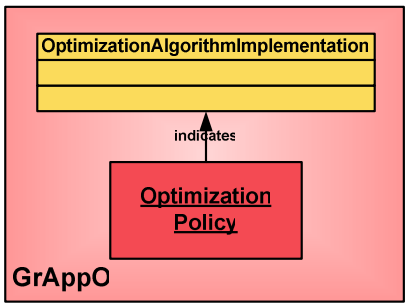


Figure 16: Optimization Policy as an inner GrAppO class with optimization algorithm configured

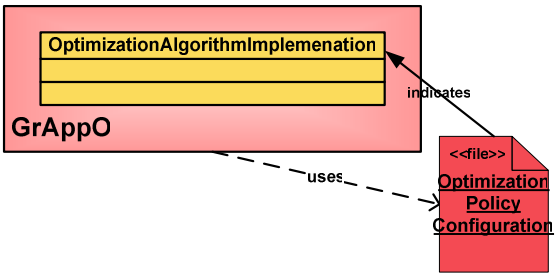


Figure 17: Optimization Policy read from an external configuration file with optimization algorithm configured

5.2.2 *Properties Configured with Optimization Policy*

The properties which can be defined with Optimization Policy include:

- Optimization algorithm – the algorithm implementation that should be used to perform optimization (the simplest one could be choosing random Grid Object Instance from the available). Certainly, the optimization algorithm implementation does not have to be placed inside GridSpace Application Optimizer as shown on Figure 16 and Figure 17 – it can be any external implementation of the optimization algorithm interface specified by GrAppO.
- Preferred implementation type – this feature defines which service implementation technology should be preferred to others (e.g. the preference could be given to services implemented as MOCCA components). It also says whether other than preferred implementation technologies will be allowed.
- Whether the preference should be given to the services which use less RAM / CPU rather than to others. If both options are used, GrAppO will try to choose the solution that keeps balance between utilization of these resources.
- Whether Monitoring / Provenance Tracking System should be contacted to obtain useful optimization data or the connection shouldn't be allowed.
- Algorithms that should be used to process and analyze the data retrieved from Monitoring / Provenance Tracking System. These, like optimization algorithm are pluggable.

Should no Optimization Policy be specified, a default configuration is used.

5.2.3 Service Access Configuration Usage Schema

The ways of implementation of Service Access Configuration are analogous to these used in case of Optimization Policy (see Figure 18 and Figure 19). However, these the two configurations are fully independent.

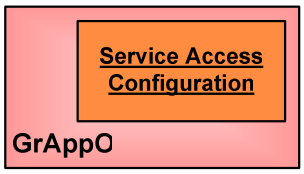


Figure 18: Service Access Configuration as an inner GrAppO class

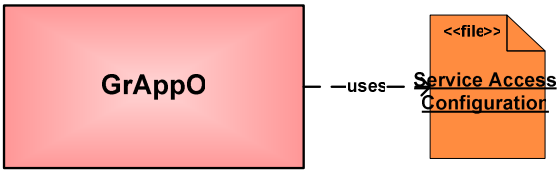


Figure 19: Service Access Configuration read from an external configuration file

5.2.4 Properties Configured with Service Access Configuration

Service Access Configuration consists of three properties:

- Grid Resource Registry service URL – this property is mandatory as GrAppO cannot be used without a connection to the registry.
- Monitoring System service URL – optional.
- Provenance Tracking System service URL – optional.

5.3 Using GrAppO

Grid Application Optimizer is distributed as a java library (`grappo-<version>.jar`). It can be downloaded from http://virolab.cyfronet.pl/maven2/cyfronet/gridspace/grappo/grappo/ or defined as Maven2 dependency (for instructions how to do it see Appendix A). When the library is added to a project classpath, methods which the optimizer publishes can be invoked.
The methods published for Grid Operation Invoker are a part of to the main GrAppO class: `cyfronet.gridspace.grappo.GridspaceApplicationOptimizer` (it is an equivalent of the GrAppO Manager component façade). This section contains the method signatures and description, which are also available in GrAppO javadoc documentation at https://zeus45.cyf-kr.edu.pl/~asia/grappo/apidocs/index.html.

5.3.1 Constructors

After invocation of these methods, GrAppO state changes to *created* (see section 4.6 for analysis of the optimizer's states):

- `GridspaceApplicationOptimizer()` – creates GrAppO instance with default configuration.
- `GridspaceApplicationOptimizer(OptimizationPolicy)` – creates GrAppO instance with the given Optimization Policy (the policy class is a member of the `cyfronet.gridspace.grappo.configuration` package).
- `GridspaceApplicationOptimizer(File)` – creates GrAppO instance with Optimization Policy parsed from the specified configuration file. This method may throw `GridspaceApplicationOptimizerInitializationException` in case when the configuration file is improper.

5.3.2 Initialization Methods

A successful invocation of one of the following methods causes GrAppO to change to the *ready* state (see section 4.6):

- `init(String)` – initializes the GrAppO instance, configuring it to use URL passed as an argument to establish connection to Grid Resource Registry service.
- `init(ServiceAccessConfiguration)` – initializes the GrAppO instance, configuring it to use the provided Service Access Configuration to establish connections to Grid Resource Registry, Monitoring System, and Provenance Tracking System services. The service configuration class is a member of the `cyfronet.gridspace.grappo.configuration` package)
- `init(File)` – initializes the GrAppO instance, configuring it to use the Service Access Configuration read from the specified file to establish connections to Grid Resource Registry, Monitoring System, and Provenance Tracking System services.

These methods throw a `GridspaceApplicationOptimizerInitializationException` if the initialization fails due to inability to find the specified services or improper service configuration.

5.3.3 Optimization Requests

A call to one of the `optimize(…)` methods triggers optimization, transferring the optimizer to the *optimizing* state (see section 4.6). Application Context passed as their last argument represents an identifier of the application for which the optimization was requested. It contains also credentials of the user who executed it and is later used to authenticate in the services GrAppO uses.
The optimization request can have one of the following syntax:

- `optimize(String, ApplicationContext)` – triggers the optimization for Grid Object Class provided as the first argument.
- `optimize(String, String, ApplicationContext)` – triggers the optimization for Grid Object Class provided as the first argument using only instances of Grid Object Implementation passed as the second argument. If the implementation is `null`, the method acts like the previous one i.e. considering all instances of the class.
- `optimize(String[], ApplicationContext)` – triggers the optimization for Grid Object Classes provided as the first argument.
- `optimize(String[], String[], ApplicationContext)` – triggers the optimization for Grid Object Classes provided as the first argument using only instances of Grid Object Implementations passed as the second argument. If some of the implementations are `null`, all instances of the class will be considered.

These methods throw either a `NoAvailableResourcesException` if no resources can be found in the registry for the provided Grid Object Class / Implementation, or a `GridspaceApplicationOptimizerException` if an internal GrAppO error occurs.

5.3.4 Getters and Setters

A value of arguments passed in constructors and initializers can later be changed by calling an appropriate setter method and obtained with proper getter method. The parameters that can be modified and viewed are:

- Optimization Policy – methods: `getOptimizationPolicy()`, `setOptimizationPolicy(OptimizationPolicy)`;
- Grid Resource Registry service URL – methods: `getRegistryServiceURL()`, `setRegistryServiceURL(String)`;
- Service Access Configuration – methods: `getServiceAccessConfiguration()`, `setServiceAccessConfiguration(ServiceAccessConfiguration)`.

Application Context is read-only – it can be obtained by calling an appropriate getter method (`getApplicationContext()`).

5.3.5 Using Configuration Classes

As stated in section 5.2, GridSpace Application Optimizer can be configured either with a proper instance of a configuration class or with a configuration file.
In case of the optimization policy (class `OptimizationPolicy`), if configuration with a class is requested, the following variables (in accordance to the list in section 5.2.2) has to be set (with respective getter and setter methods):

- `optimizationAlgorithm` – instance of the `OptimizationAlgorithm` class (package: `cyfronet.gridspace.grappo.engine`). Default: `cyfronet.gridspace.grappo.sample.RandomOptimizationAlgorithm`.
- `preferredType` – one of the types enlisted in an enumeration object from the library provided with Grid Resource Registry service – `cyfronet.gridspace.grr.GridObjectType`. Default: `null` (meaning any).
- `strictTypePreferrence` – a `boolean` value determining whether other than preferred implementations types will be used `(false)` or not `(true)`. Default: `false`.
- `useMonitoring` – a `boolean` value indicating whether the Monitoring System service will be used `(true)` or not `(false)`. Default: `false`.
- `useProvenance` – similarly to the `useMonitoring` variable, decides about using Provenance Tracking System service. Default: `false`.

The values of the properties contained in `ServiceAccessConfiguration` class are of `java.lang.String` type and can be read or written with the methods:

- `get/setRregistryServiceURL` – the URL to the Grid Resource Registry service. Cannot be `null`.
- `get/setMonitoringServiceURL` – the URL to the Monitoring System service. Default: empty.
- `get/setProvenanceServiceURL` – the URL to the Provenance Tracking System service. Default: empty.

5.3.6 Using Configuration Files

In section 5.3.5 a description of GrAppO configuration with a class was provided. In this section the configuration using configuration files will be discussed. The configuration files for both the optimization policy and service access configuration must be in one of the formats presented in Appendix D and contain the described further properties.
If optimization policy configuration file misses one of the properties, the default values of `OptimizationPolicy` class are used (these default values were provided with the enumeration in section 5.3.5). The key-value properties that can be placed in the file are (they are listed in the same order as the corresponding variables in section 5.3.5):

- `optimization.algorithm` – the value has to be a fully qualified name of the class that implements the `OptimizationAlgorithm` interface.
- `preferred.service.type` – one of the values: `WS`, `MOCCA`, `WSRF`, `JOB`.
- `preferred.strict` – `true` or `false` value.
- `use.monitoring` – `true` or `false` value.
- `use.provenance` – `true` or `false` value.

In case of service access configuration, the key-value properties that can be placed in the configuration file are:

- `registry.url` – URL to Grid Resource Registry service.

- `monitoring.url` – URL to Monitoring System service.
- `provenance.url` – URL to Provenance Tracking System service.

The `registry.url` property value has to be provided – otherwise the configuration file is invalid.

5.4 Tools Used for GrAppO Design and Development

For the purpose of design and development of the present project, the following tools were used:

5.4.1 Design

Microsoft Visio – a member of Microsoft Office package, used for creating all of the diagrams contained in this thesis (except for Figure 1, which was copied from the ViroLab design document [4]).

5.4.2 Development

Eclipse SDK (http://www.eclipse.org) – a programming environment, which was used for implementation of all the project's classes. Version: 3.2.2
Subversion (http://subversion.tigris.org/) – a version control system, that facilitated maintaining consistency of the implementation code.
Maven (http://maven.apache.org/) version:2.0.6, and Ant (http://ant.apache.org/) version: 1.7.0 – for project code, releases, and homepage ([25]) management.
XFire (http://xfire.codehaus.org/) library – required to implement connection to Grid Resource Registry. Version: 1.2.6.
JUnit (http://www.junit.org/) – testing framework, used in the project for writing and performing unit tests as well as some integration tests. Version: 3.8.1.
Cobertura (http://cobertura.sourceforge.net/) – used as maven plug-in for generating test coverage reports.
Enterprise Archtect (http://www.sparxsystems.com.au/) – a tool used for class diagrams generation using reverse engineering.

5.4.3 Other

For collaboration with other ViroLab Virtual Laboratory GForge system (http://gforge.org/) containing GrAppO project [24], and developer's Wiki pages [8] were used.

5.5 Summary

During the implementation of GrAppO 85 Java classes were created. The total number of source code lines is about 9500. Apart from plain Java classes, dedicated XML files with configuration were created. The source code can be viewed under http://gforge.cyfronet.pl/viewvc/trunk/?root=grappo or checked out from Subversion repository https://gforge.cyfronet.pl/svn/grappo/trunk.
GrAppO as a final product is distributed as a Java archive library, which size (without dependencies) is 50kB. With all dependent libraries the size of the distribution is 2.5MB.

Chapter 6: Tests of GridSpace Application Optimizer

This chapter contains a description of test methods and techniques used to control GridSpace Application Optimizer execution and performance, along with the results of the performed tests.

6.1 Introduction

The process of testing involved measurement of the optimization quality from different points of view. Unit tests were performed to prove the correctness and completeness of individual units of source code of GrAppO. Integration tests were used to validate the connections to other components[9] of ViroLab Virtual Laboratory Runtime – Grid Resource Registry and Grid Operation Invoker (the other modules – Runtime Library and components of the Middleware Layer – Monitoring System and Provenance Tracking System were not yet implemented). Finally, quality tests were executed to verify the usefulness of GrAppO operations in a given environment.

6.2 GrAppO Unit Tests

Unit tests were implemented[10] for all GrAppO components during or before their development. They were intended to show that the optimizer works correctly with all kinds of input data – including the proper behaviour in case of incorrect input. To release the project, all the unit tests had to be passed.
The information about all the performed unit tests were assembled in the sections: 6.2.1-6.2.4. The most detailed tests were performed for the GrAppO Manager (6.2.1) component, since it is the interface (façade), GridSpace Application Optimizer exposes for requests from other ViroLab components (external to GrAppO). This is why it has to be tested against various, also erroneous input – to develop proper error handling and resistance.

Note: For the explanation of the roles of modules under test please see section 4.2.

[9] See section 3.2.3 for the summary of communication channels to other ViroLab components and section 1.1.4 for the situation of these components in ViroLab Virtual Laboratory architecture.

[10] All unit tests were performed using the JUnit framework (see also 5.4.2).

6.2.1 GrAppO Manager Tests

`GridspaceApplicationOptimizerTest`[11] - tests the functionality of the main component – GrAppO Manager. The detailed description of the methods of this test can be found in Table 1.

Test method name	Description
`testConstructors`	Runs each of the GridSpace Application Optimizer constructors (see also 5.3.1). Optimization policy configuration files were tested both for XML and .properties format.
`testInit`	Tests all the initialization methods using the Grid Resource Registry URL and service access configuration files in both formats. The init method were also called with files containing errors – to catch the proper exception.
`testOptimize`	Checks whether the optimization methods work correctly – including proper Grid Object Class names and the ones that are not present in the repository. An attempt to query GRR, when its service URL is improper was also performed.
`testTypePreference`	Tests whether GrAppO, configured to choose only one type of Grid Object Implementation does not choose another type.

Table 1: GrAppO Manager main unit test information

Additional GrAppO Manager tests included tests of the classes that perform parsing of the configuration files: `ConfigurationFileReader` and `OptimizationPolicyBuilder`.

Test name	Description
`ConfigurationFileReaderTest`	Tests the two static methods of the class that parse: (first) optimization policy or (second) service access configuration file. The files were given in XML and .properties format. Improper file paths were also used to obtain a proper exception.
`OptimizationPolicyBuilderTest`	Tests building an optimization policy class from the configuration read from configuration file.

Table 2: Additional tests for GrAppO Manager

6.2.2 Optimization Engine Tests

Tests of the Optimization Engine component include the test of the component interface – `OptimizationEngine` class along with test of all the implemented optimization algorithms. The test methods of the `OptimizationEngineTest` were presented in Table 3.

Test method name	Description
`testConstructor`	Runs the Optimization Engine constructor with an optimization policy class or `null` – to check the exception handling.
`testOptimize`	Checks whether the optimization methods work correctly – for both optimization of single Grid Object (short-sighted) and a group of Grid Objects (medium-sighted).
`testOptimizeList`	

Table 3: Optimization Engine main unit test information

[11] the test names consist of the class name and the suffix '`Test`' – e.g. the test for the class '`ClassTest`' is named '`ClassTest`'

The various implemented optimization algorithms were tested with `OptimizationAlgorithmsTest` test class methods, enumerated in Table 4.

Test method name	Description
`testXXXAlgorithm`	Tests each implemented optimization algorithm (`XXX` means any algorithm that was implemented in the system – e.g. Random, Min-min, Max-min, Suffrage). Each algorithm implementation is provided with correct, incomplete, empty and `null` data set to monitor its behaviour in case of proper and improper input. The output is also compared to the expected.

Table 4: Implemented optimization algorithms tests

6.2.3 Performance Predictor Tests

The Performance Predictor component test performed calls to all the component's main methods – constructor and prediction methods. The tests of prediction used various combinations of generated data. A short summary of the test is provided in Table 5.

Test method name	Description
`testConstructor`	Runs the Performance Predictor constructor with monitoring and provenance service URL. Tests also empty URLs, which should not cause an error.
`testPredictMonitoring`	Checks whether the predictor obtains data from monitoring service and transforms it correctly to a performance estimation.
`testPredictProvenance`	Checks whether the predictor obtains data from provenance service and transforms it correctly to a performance estimation.
`testPredictMonitoringProvenance`	Checks whether the predictor obtains data both from provenance and monitoring services and transforms it correctly to a performance estimation
`testPredictList`	Checks whether the predictor obtains data from provenance and monitoring services and whether the estimated values are correct – for a group of Grid Objects (medium-sighted optimization).

Table 5: Performance Predictor unit test information

6.2.4 Resource Condition Data Analyzer and Historical Data Analyzer Tests

The two components were tested by generating various sets of data for analysis – simulating the results of queries to Monitoring System (in case of Resource Condition Data Analyzer) and to Provenance Tracking System (in case of Historical Data Analyzer). These included also corrupted data. Table 6 summarizes these components' test information.

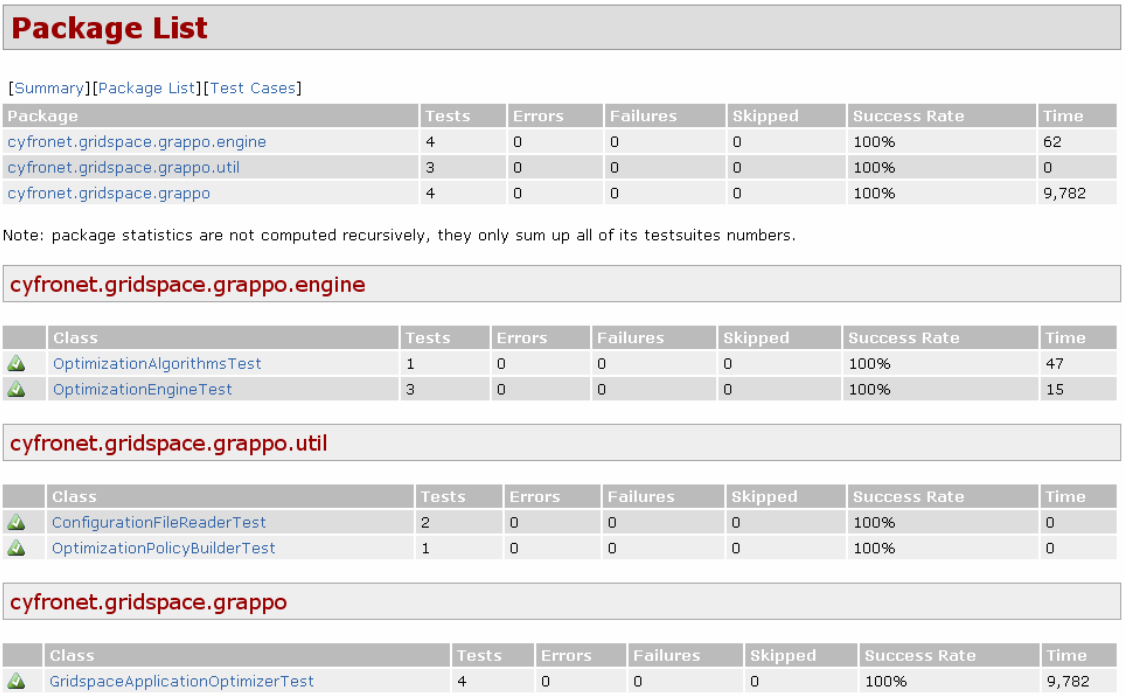
Test name	Description
`ResourceConditionDataAnalyzerTest`	Both contain a `testAnalyze()` method, which performs the component's analysis on generated sets of data.
`HistoricalDataAnalyzerTest`	

Table 6: A summary of Resource Condition Data Analyzer and Historical Data Analyzer tests

6.2.5 Test Reports

The tests described in sections 6.2.1 and 6.2.2 concern the GridSpace Application Optimizer part that was released and integrated into ViroLab Runtime release (see also 6.3). They are therefore summarized by test reports, which can be viewed at the GrAppO homepage [25]

under *Maven Surefire Report* in *Project Reports* section. A part of this report is also displayed on Figure 20.



Package List

[Summary][Package List][Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
cyfronet.gridspace.grappo.engine	4	0	0	0	100%	62
cyfronet.gridspace.grappo.util	3	0	0	0	100%	0
cyfronet.gridspace.grappo	4	0	0	0	100%	9,782

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

cyfronet.gridspace.grappo.engine

Class	Tests	Errors	Failures	Skipped	Success Rate	Time
OptimizationAlgorithmsTest	1	0	0	0	100%	47
OptimizationEngineTest	3	0	0	0	100%	15

cyfronet.gridspace.grappo.util

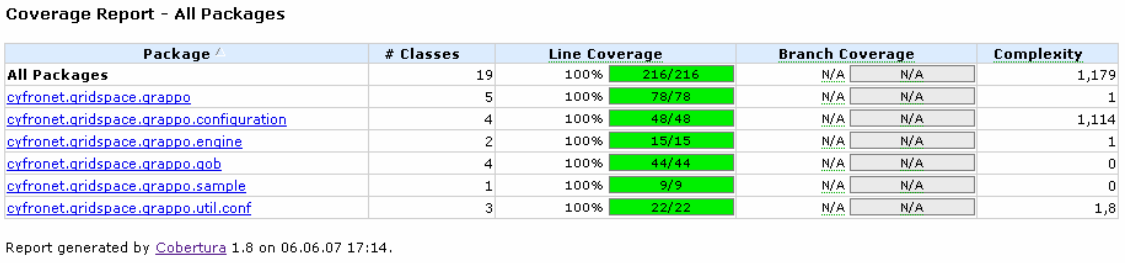
Class	Tests	Errors	Failures	Skipped	Success Rate	Time
ConfigurationFileReaderTest	2	0	0	0	100%	0
OptimizationPolicyBuilderTest	1	0	0	0	100%	0

cyfronet.gridspace.grappo

Class	Tests	Errors	Failures	Skipped	Success Rate	Time
GridspaceApplicationOptimizerTest	4	0	0	0	100%	9,782

Figure 20: A part of GrAppO tests report

The released part GridSpace Application Optimizer (see earlier in this section) was also investigated with a test coverage reporting tool – Cobertura (http://cobertura.sourceforge.net/). Its report indicates the percent of project code covered by unit tests. A part of it is presented on Figure 21. A more detailed view on it is available at the GrAppO homepage [25] under *Cobertura Test Coverage* link in *Project Reports* section.



Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	19	100% 216/216	N/A N/A	1,179
cyfronet.gridspace.grappo	5	100% 78/78	N/A N/A	1
cyfronet.gridspace.grappo.configuration	4	100% 48/48	N/A N/A	1,114
cyfronet.gridspace.grappo.engine	2	100% 15/15	N/A N/A	1
cyfronet.gridspace.grappo.gob	4	100% 44/44	N/A N/A	0
cyfronet.gridspace.grappo.sample	1	100% 9/9	N/A N/A	0
cyfronet.gridspace.grappo.util.conf	3	100% 22/22	N/A N/A	1,8

Report generated by Cobertura 1.8 on 06.06.07 17:14.

Figure 21: A test coverage reports for GrAppO main part

The report shows that all the main GrAppO classes were fully covered with unit tests.

6.3 GrAppO Integration Tests

GridSpace Application Optimizer was successfully integrated with Grid Resource Registry and Grid Operation Invoker. Both of this connections were tested during the successive ViroLab Runtime prototype releases. Additionally, the connection to the registry is being checked with unit tests (see 6.1).

The ViroLab Virtual Laboratory Runtime releases are the best integration tests for the optimizer. Therefore, in this section screen-shots showing also execution logs of ViroLab experiments[12] runs are shown. The experiments are ready ViroLab applications, designed also

[12] The scripts are distributed under a MIT License which can be found in Appendix C along with the

for the purpose of ViroLab Runtime integration testing – one of them is used for an alignment of a viral DNA strain, the other one uses data mining for a weather prediction. Their sources can be found in Appendix C.

The figures: Figure 22 and Figure 23 show screen-shots of command line execution of two of the ViroLab experiments. The logs of Grid Operation Invoker and GridSpace Application Optimizer (set to the DEBUG mode) indicate the following steps important from the point of view of integration between components:

- request for optimization (GOI),
- query to the Grid Resource Registry (GrAppO),
- GRR response (GrAppO),
- optimization – choosing and returning to GOI the ID of one of the obtained instances (GrAppO),
- receiving response from GrAppO (GOI)

These steps are marked with red frames on both screen-shots.

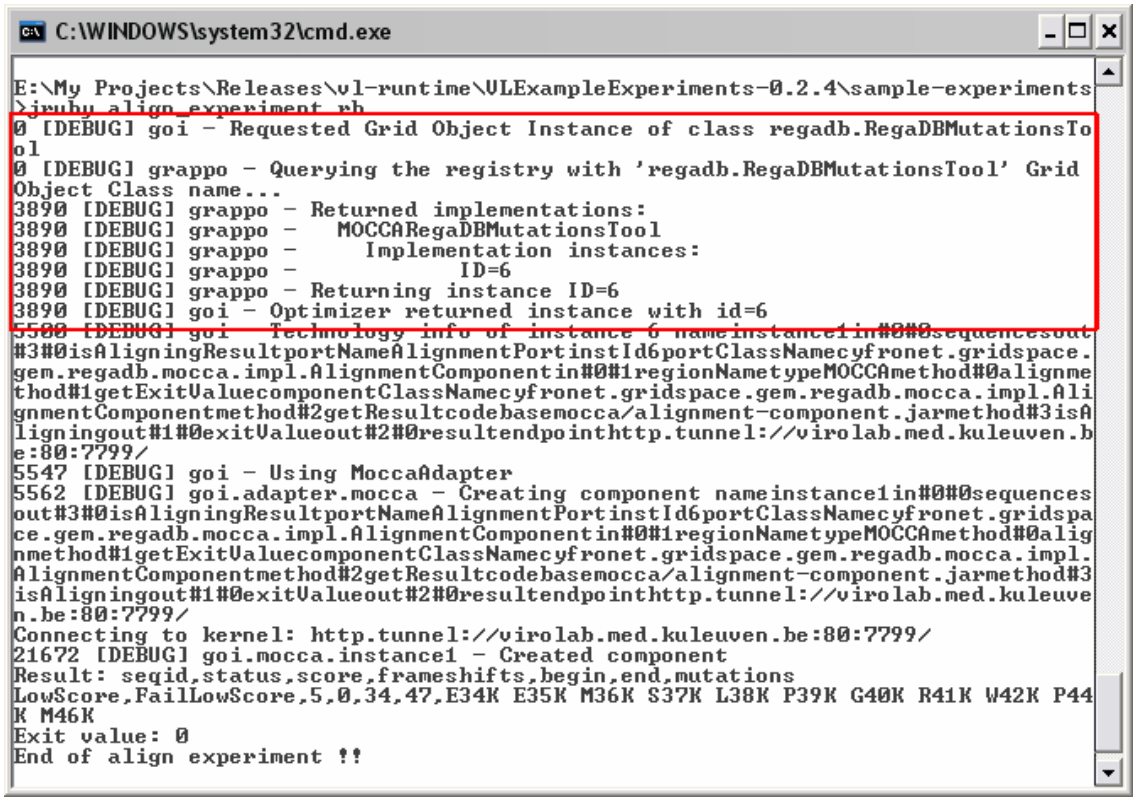


Figure 22: A screen-shot of a ViroLab experiment run from MS Windows command line – Grid Operation Invoker debug logs show a result of the call to GrAppO optimization method (marked with the red frame)

experiments contents and basic information about them.

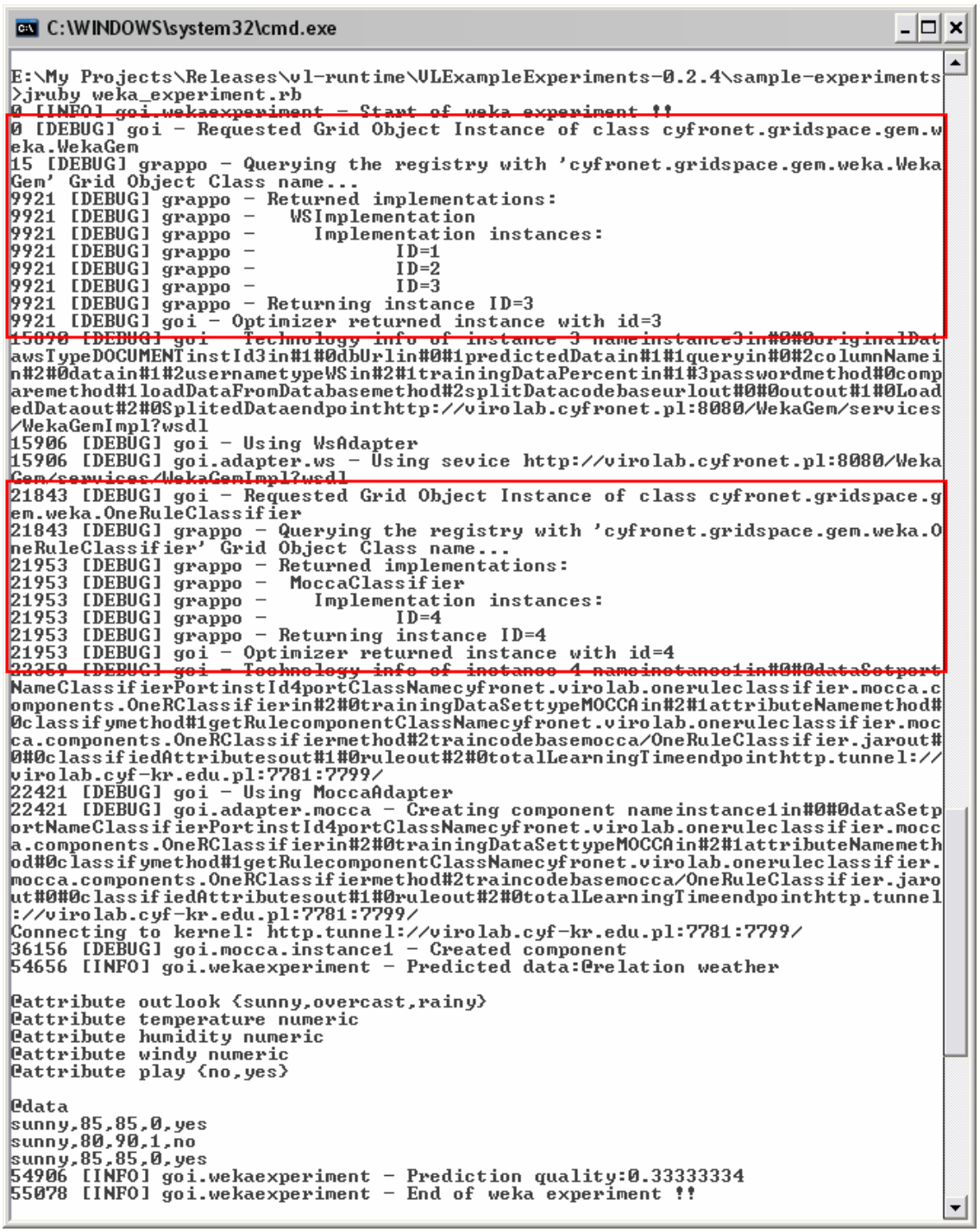


Figure 23: A screen-shot of another ViroLab experiment – this time Grid Operation Invoker calls the GrAppO optimization method twice (again the request and result log parts are marked with red frame)

It is easy to see that the optimizer works in either short- or far- sighted[13] optimization mode here. Especially the Figure 23 shows that – since two calls for single Grid Object Instance took place there.

Figure 24 and Figure 25 show the usage of ViroLab Virtual Laboratory Runtime with GrAppO through a Graphical User Interface for Experiment Developers (for ViroLab users see 1.1.3) –

[13] In fact it was the short-sighted optimization, since far-sighted optimization was not implemented for the present thesis (see 5.1)

the Experiment Planning Environment. The scripts presented on the screen-shots were the same as the ones used for preparing the visualizations from Figure 22 and Figure 23.

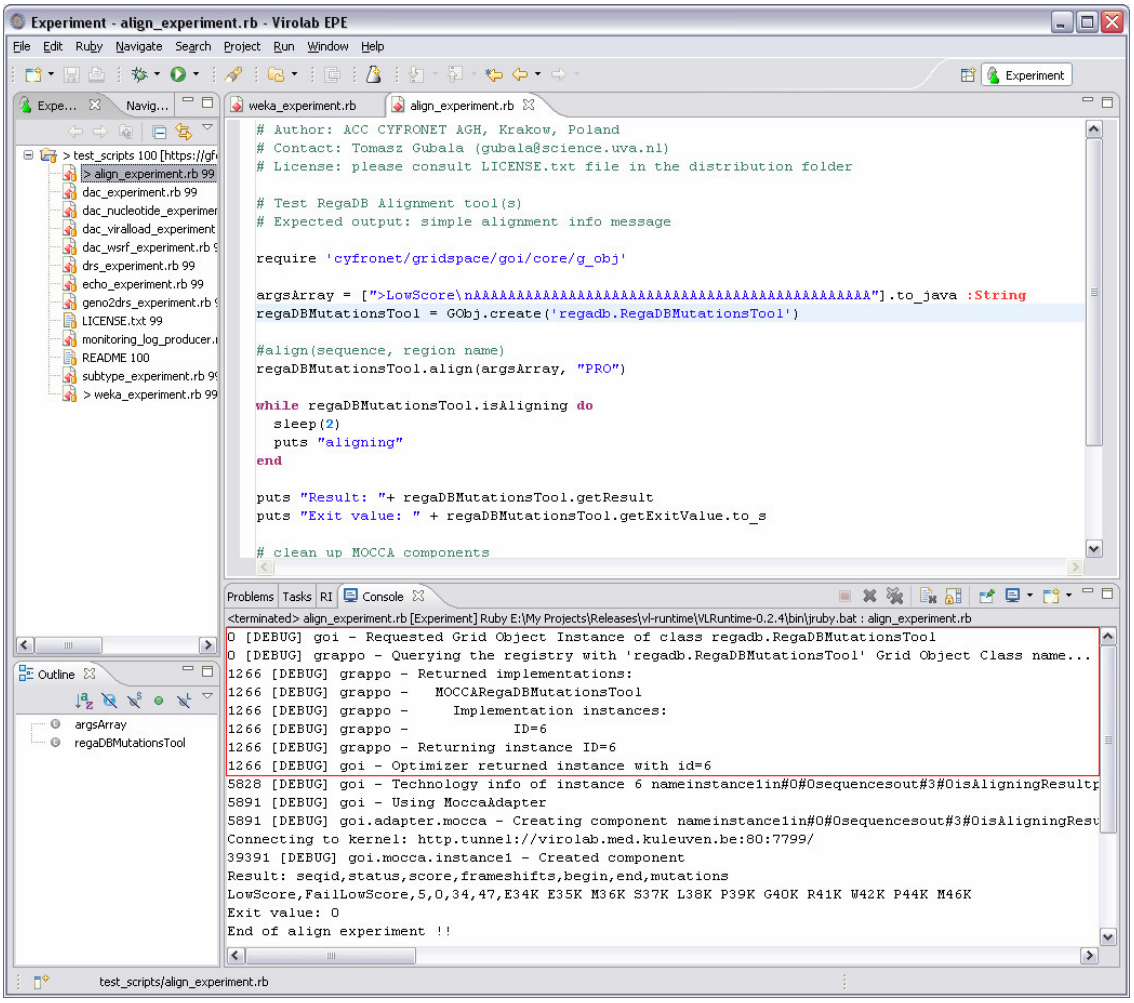


Figure 24: A screen-shot of an execution of the experiment that was also presented on Figure 22; the experiment was run using the Experiment Planning Environment – a Graphical User Interface for the ViroLab Runtime

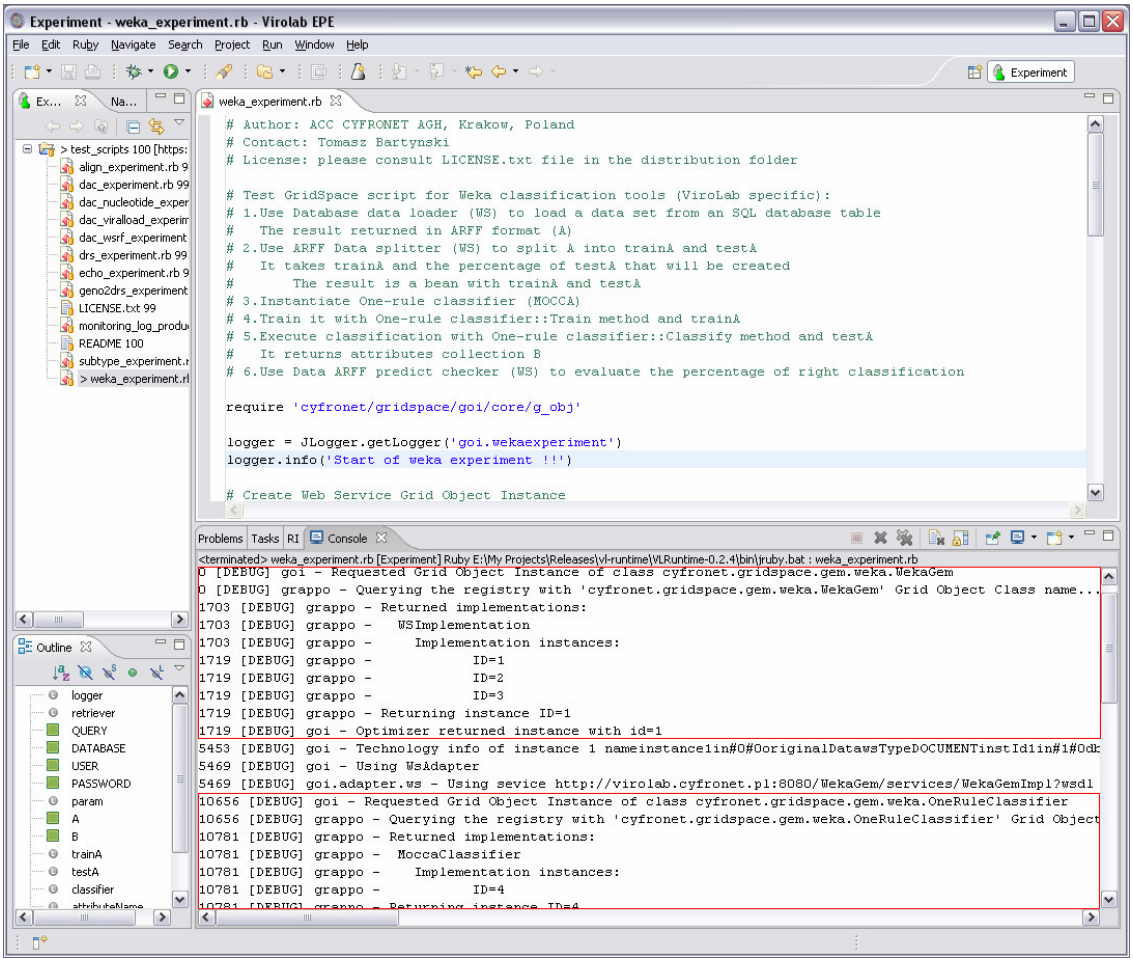


Figure 25: A screen-shot of an execution of the experiment that was also presented on Figure 23; as with Figure 24, Experiment Planning Environment was used for the experiment execution

6.4 GrAppO Quality Tests

The section presents information about the methods and techniques that were used to prove the utility of GridSpace Application Optimizer in ViroLab Runtime System and the quality of the information that can be obtained from it depending on the environment.

The aim of the quality tests was to show general trends of the optimization quality and its dependence on different parameters of the environment. The test results presented in this section (subsections 6.4.3-6.4.5) were obtained for the test environment described in section 6.4.1 and on a single machine. Therefore, although the tests were repeated may times, the results may slightly differ depending on the environment condition and their repeatability cannot be guaranteed.

6.4.1 Test Environment

At the current stage of development of the ViroLab Runtime System, not all its components are available. The sources of information for GridSpace Application Optimizer are limited as Monitoring System and Provenance Tracking System services have not been completed yet and the Grid Resource Registry contains only the small amount of the sample data. Taking this into consideration, the performance tests of GrAppO appeared to be infeasible to perform in the target environment.

To solve this problem, a simulated ViroLab Runtime environment had to be developed. The Grid Resource Registry, Monitoring System and Provenance Tracking System services were implemented as mock components that offer only the functionality required by GridSpace Application Optimizer. These are:

- `GRRService` – a component that provide the following functionality of Grid Resource Registry (compare: `GRRService` interface on Figure 28):
  - `getOptimizationInfo(String gridObjectFullName, String gridObjectImplementationName, GridObjectType gridObjectType)` – the method that can be used to obtain information about the Grid Object Implementations of the given Grid Object Class and available resources.
  - `getOptimizationInfo(List<String> gridObjectFullNames, List<String> gridObjectImplementationNames, GridObjectType gridObjectType)` – a bulk version of the previous operation.
- `MonitoringService` – a component that provide the following functionality of Monitoring System (compare: `MonitoringService` class on Figure 32):
  - `getResourcesInformation(Set<String> locations)` – the method enables to obtain information regarding the current state of resources in the given locations
- `ProvenanceService` – a component that can be regarded as a substitution of PROToS service with the following functionality (compare: `MonitoringService` class on Figure 33):
  - `getProvenanceInfo(Set<String> implNames, Set<String> locations)` – the method is delegated to provide historical data about the previous execution of operations on the given Grid Object Implementations in the given locations

These components cannot communicate with the original sources of information they have to obtain the required information in other way. What is more, since the data are only necessary for the testing purposes, they can be randomly generated if only they meet general constraints regarding the information from these components.
In order to make the test environment more generic, the components implementation use the information from previously generated XML files. These files contain data specified to each component that are serialized to a format of XML. They can be automatically generated by a provided generator or created manually. Thanks to such a solution the process of testing becomes more precise. An example of such XML file with generated data can be viewed in Appendix E.

Every execution of a test in the environment has to define files that will be the source of needed data. In this way different optimization techniques can be performed using the same data in order to compare their achievements. Therefore the performance of a single technique can be verified according to the varied amount of data.
The amount of generated test data can be configured a priori by:

- number of available resources,
- maximum number of Grid Object Class definitions available in Grid Resource Registry,
- maximum number of Grid Object Implementations per one Grid Object Class,
- maximum number of Grid Object Instances per one Grid Object Implementation
- maximum number of kernels per one Grid Object Class.

The generators produce various numbers of these elements (up to the maximum number) which simulate the actual situation in the Grid Resource Registry.

6.4.2 Tests Objective Function

Optimizers in Grid environments can use a variety of optimization criteria, such as maximization of resource utilization, minimization of the cost to the user or minimization the makespan. For the purpose of these testing the latest criterion was selected as GridSpace Application Optimizer is an application centric optimizer (see 3.4.2). The following paragraphs define additional terms related to this criterion. The definitions were based on [22].

Expected completion and execution time:

- **Expected completion time (CT_{jr})** of the job *j* on the resource *r* is defined as a moment of time when the resource *r* completes the job *j* (after finishing execution of all previously planned jobs and executing the job *j*).
- **Expected execution time (ET_{jr})** of a job *j* on a resource *r* can be regarded as an amount of time needed by the resource *r* to execute the job *j* assuming that the resource *r* has no load at the moment of the assignment the job *j* to it.

Let T_r be the time when the resource *r* is able to initialize the execution of any not assigned yet job. From the aforementioned definitions the expected completion time can be defined as: $CT_{jr} = T_r + ET_{jr}$.

Let *R* be the set containing resources available in an environment. Let *J* be the set of jobs and each job *j* is to be assigned to any resource *r* from the set *R*. Then, the **makespan** for the complete assignment is defined as:

$$\max_{j \in J} (CT_{jr})$$

The makespan is a measure of the throughput of the heterogeneous computing system. The objective function is to minimize the makespan.

In order to compute the metrics of the makespan in GridSpace Application Optimizer, additional sources of information have to be used. These are the mock components described in 6.4.1 – the data concerning expected completion time of Grid Object Instance (jobs) is obtained from `ProvenanceService` and the data about the time of availability of the resources (locations) is provided by `MonitoringService`.

6.4.3 Comparison of Effectiveness of Different Optimization Modes

In section 3.4.3 three optimization modes for GridSpace Application Optimizer were specified: short-, medium- and far- sighted optimization. As explained in 5.1, at the current stage of development GrAppO can be run in short- and medium- sighted optimization modes, while it does not offer far-sighted optimization. Therefore testing of GrAppO optimization effectiveness involves only the two available modes.

The tests assume availability of all necessary data produced by local implementations of `ProvenanceService` and `MonitoringService` (see section 6.4.1). During the execution of the tests random data from the services were generated with the constraint that the maximum expected execution time of Grid Operation is 500 seconds and the maximum time waiting for the resource to become available is set to 1500 seconds. The data generated by `GRRService` implementation followed the following constraints:

- maximum number of Grid Object Class definitions available in Grid Resource Registry is set to 100 ,
- maximum number of Grid Object Implementations per one Grid Object Class is set to 10,
- maximum number of Grid Object Instances per one Grid Object Implementation is set to 10
- maximum number of kernels per one Grid Object Class is set to 7.

The variable parameter in the test is a number of available resources and the number of Grid Object Classes for the optimization. For each combination of a given number of resources and a given number of Grid Object Classes a list with names of Grid Object Classes that have to be optimized was randomly generated and passed to GridSpace Application Optimizer. The elements in the list can be repeated. For each list the short-sighted optimization and the medium-sighted optimization were performed. For a given combination of a number of resources and the number of Grid Object Classes a test was performed 1000 times.

The results of the tests were collected in Table 7, Table 8 and Table 9 for the test cases that use the list of Grid Object Classes containing 10, 20 and 50 elements, respectively.

Number of resources	Improvement of makespan	Percentage of improved results	Percentage of the same results	Average improvement
5	5.78%	57.1%	27.0%	12.58%
10	4.58%	43.7%	49.7%	11.69%
20	2.87%	26.4%	71.5%	10.74%

Table 7: Test result for optimization of 10 Grid Object Classes

Number of resources	Improvement of makespan	Percentage of improved results	Percentage of the same results	Average improvement
5	4.03%	61.7%	4.0%	11.70%
10	5.90%	61.1%	19.6%	12.23%
20	5.47%	51.0%	45.8%	11.23%
50	2.00%	20.2%	79.1%	10.28%

Table 8: Test result for optimization of 20 Grid Object Classes

Number of resources	Improvement of makespan	Percentage of improved results	Percentage of the same results	Average improvement
5	2.95%	61.4%	0.0%	11.07%
20	5.59%	57.5%	19.4%	12.99%
50	4.05%	43.0%	53.1%	10.25%
100	2.21%	26.4%	72.6%	8.85%

Table 9: Test result for optimization of 50 Grid Object Classes

The meaning of the values in columns of the tables is as follows:

- **Improvement of makespan** – presents the average improvement of the makespan for all results while using the medium-sighted optimization in comparison to the short-sighted optimization mode.
- **Percentage of improved results** – shows the percentage of jobs that obtained a better result (makespan) thanks to the introduction of the medium-sighted optimization mode (in comparison to the short-sighted mode).
- **Percentage of the same results** – presents the percentage of jobs for which both optimization modes provided the same solution.
- **Average improvement** – represents the average improvement of the makespan when only the improved results are considered.

The improvement of the makespan depends on the number of available resources and the number of Grid Object Class that are requested for the optimization. If the number of resources is significantly greater than the number of the objects for the optimization, the improvement of the makespan is the smallest. It is caused by the fact that with random generated data available Grid Object Instances or Grid Object Implementations are distributed on random locations and the probability of mapping two or more Grid Object Class to the same resource is smaller than in the other cases. Therefore, the medium-sighted optimization suggests the solution very similar to the solution of short-sighted optimization, which is shown by the significant amount of the same solutions obtained from both optimization modes in such situation.

The best improvement of the makespan is observed for the situation when the number of available resources is smaller that the number of Grid Object Classes for the optimization. The rate of the improvement varies between 5.5% and 6.0% in the results of tests. However, if the number of resources is significantly smaller than the number of Grid Object Classes, the improvement is smaller (e.g. with 5 resources and 50 Grid Object Classes for the optimization in the test the improvement was only 2.95%).
With the increase of proportion of the number of Grid Object Classes to the number of available resources, the amount of improved results also increases. At the same time, the amount of the same results from both modes decreases.
Taking into consideration only these results for which the makespan was improved, the average rate of improvement is the best also for the situation when the number of resources is smaller than the number of Grid Object Classes for optimizing. The improvement overcomes the level of 12%.

6.4.4 Comparison of Effectiveness of Different Optimization Algorithms

In order to apply any reasonable optimization algorithm (for the description of different algorithm please see section 2.3.4), again, external sources of information are needed – represented by local implementations of `ProvenanceService` and `MonitoringService` (see section 6.4.1).
In situations, when no additional optimization data (other than the implementations list obtained from the Grid Resource Registry) is available, GridSpace Application Optimizer can use an algorithm for choosing a random solution from all possible. As it almost always brings a very week solution, it is discouraged. For the purpose of these tests, using algorithm that chooses random solution could be treated as if no optimization was performed. In such case, GrAppO if configured to use any other optimization algorithm, even in short-sighted optimization mode, brings on average 200% better solution (minimizing the makespan). In example tests GrAppO was requested to optimize 5 Grid Object Classes. Each test was performed 1000 times.

The most common algorithms used for the minimizing the expected completion time – Min-min and Max-min (for the algorithms description see section 2.3.4.1) – for random input data give very similar results. However, Max-min heuristic gives better solutions when, among the Grid Object Classes to optimize appears one or relatively small group of Grid Object Classes with significantly longer execution time than the others. The improvement that this heuristic brings in such a situation, as the tests proved, is at the rate 5.6% in comparison to Min-Min heuristic.

6.4.5 Impact of Quality and Availability of Information from External Components

The previously described optimization quality tests (sections 6.4.3 and 6.4.4) assume the GridSpace Application Optimizer sources of information provide a complete set of data – all the requested information is obtained. However, in some cases not all the data is available, therefore, the provided information is incomplete. The tests described in this sections were aimed to investigate how GrAppO results are influenced by the lack of some information.
Assuming that if the optimization metrics is impossible to compute for some Grid Object Classes a random solution is chosen, the tests were performed with regard to how much information is missing. While simulating such situations, the test parameters were set as follows:

- number of available resources is set to 10
- maximum number of Grid Object Class definitions available in Grid Resource Registry is set to 100 ,
- maximum number of Grid Object Implementations per one Grid Object Class is set to 10,

- maximum number of Grid Object Instances per one Grid Object Implementation is set to 10

A single step of the test involved generating the list with 20 names of Grid Object Classes. The elements in the list could be repeated. Afterwards, the request for the optimization of these Grid Object Classes was sent to the GrAppO. The optimization mode was set to medium-sighted and the algorithm used was the Max-min heuristics. First, the optimization was performed assuming all data from `ProvenanceService` and `MonitoringService` are available. After its completion, a given amount of data was removed from the set obtained from `MonitoringService` and the same request for the optimization was sent to the GrAppO again. The test considered removing the following amount of data: 10%, 20%, 30%, 40%, and 50%. For each of the values the test was executed 1000 times. The results of the tests are shown in Table 10.

Percentage of removed data	Deterioration of makespan
10%	9.81%
20%	20.90%
30%	42.93%
40%	88.12%
50%	124.26%

Table 10: Influence of the availability of information on the makespan

As Table 10 shows, the lack of information from external data sources has significant influence on the makespan. The more data are unavailable, the greater deterioration of the makespan is observed. The relation is non-linear. With 10% of lacking information the rate of aggravation of thee makespan is approximately at the same level – 9.81%, whereas, when half information is unreachable, the deterioration of the makespan is 124%! This effect is caused by the necessity of using of the algorithm that chooses random solution in for more Grid Object Classes (see section 6.4.4 for the comparison of this algorithm to more advanced heuristics).

6.5 Conclusions

The target of this chapter was to present the results of testing the Grid Application Optimizer.
The report from *Cobertura* reporting tool showed that all main GrAppO classes code was fully covered with unit tests, which, passed, suggest that GridSpace Application Optimizer responds properly to all kinds of requests.
The integration tests that concern Grid Resource Registry and Grid Operation Invoker were performed during preparing the ViroLab Virtual Laboratory Runtime releases and showed, by executing ViroLab experiments, that the existing communication channels between components are operating correctly.
The aim of the quality tests was to prove the utility of GrAppO in the ViroLab Runtime System and verify the quality of information provided by it in different environment configurations. Testing of two optimization modes (short-sight and medium-sight) showed that with a suitable rate of available resources to Grid Object Classes for optimizing the medium-sight optimization mode can achieve the improvement at the level 13% in the comparison to short-sighted mode. In test cases Min-min and Max-min heuristics were used and gave satisfactory results. However, for these heuristics additional data from external components are necessary. In a case when the data is incomplete and the necessity of using a random algorithm for the optimization appears, the results of the optimization worsen when the lack of information becomes more significant.

Chapter 7: Conclusions and Future Work

This chapter summarizes the thesis goals that were achieved during the work on it. It also identifies the new challenges that emerged during the work on the project.

7.1 Conclusions

The main goal of the thesis – providing an optimizer for ViroLab was successfully achieved. After the design process, the implementation and testing a correctly operating system can be released. It offers the functionality that was possible to implement at the time, which is described in 5.1.
During the design of GridSpace Application Optimizer a thorough research (Chapter 2) was performed on optimization technologies for Grid Computing. Some of the heuristics analyzed during the research and problem analysis (Chapter 3) were later implemented as GrAppO optimization algorithms, giving satisfactory results.
The GridSpace Application Optimizer design was done in compliance with the requirements identified in 3.3. Thanks to the possibility of using different optimization policies, service access configurations, and pluggable optimization algorithms mechanism, GridSpace Application Optimizer is an easily configurable and adaptive component. Its adjustability and reliability can be extended by implementing the spare connection to Monitoring System (future task: 0)
All the performed tests proved the completeness and correctness of the developed system. The results of the execution of quality tests showed the introduction of different optimization modes was beneficial. Thanks to adaptive techniques the results that GridSpace Application Optimizer produces are improved and can improve the performance of the whole ViroLab Runtime System. However, the optimizer is easily influenced by the quantity and the quality of information gathered from external data sources – dependencies to other components of ViroLab.

7.2 Future Work

In the future, the following work could be done to improve GrAppO optimization. Its evaluation could also become a valuable research topic. The tasks for the future, presented in the next subsections were ordered from the most to the least urgent.

Implementing Connections to Other ViroLab Components. It is the most urgent issue, as only the information gathered by Grid Resource Registry are insufficient for any efficient optimization (see test results in section 6.4.5). However, although GrAppO is prepared for receiving additional data, other components services were not yet published. Obtaining the communication channels to Monitoring System and Provenance Tracking System will also enable verifying implemented heuristics with real data.

Parallelizing the Queries to Monitoring System and PROToS. The queries to Monitoring System and Provenance Tracking System as pointed in 4.7 can be implemented as parallel operations, which can significantly decrease the delay the optimization brings to the application execution. This is because the queries are sent to remote services, and each takes long to complete. Unless they are executed simultaneously, the impact on the application optimization time they bring is a sum of the queries duration.

Optimization with Regard to Grid Operations. Introducing an optimization that concerns not only the Grid Object Class that will be used, but the actual Grid Operation to be invoked as well should bring a certain improvement to the optimization. This would require additional input from Grid Object Invoker, still it would narrow the analysis of historical information from Provenance Tracking System only to the required Grid Operation, making the performance estimation more accurate.

Testing New Optimization Algorithms. To find the most suitable for the optimization, more, various algorithms should be implemented and tested. Introducing new techniques such as genetic algorithms or simulated annealing and dynamic adjustment the algorithm to the input data can give better solutions. Furthermore, if external components were able to provide additional data concerning Grid Object Classes, implementing the heuristics that include Quality of Service guidance could be also an interesting issue.

Graphical Interface for GrAppO Configuration. To make GrAppO more user-friendly a graphical interface can be developed. For this purpose for example a JMX console could be used. User would gain an opportunity to observe the results of GrAppO operations on-line and, if necessary, easily change its configuration.

Implementing Far-Sighted Optimization. For now the Runtime Library, which should provide the application graph does not exist. In order to perform the far-sighted optimization, the GrAppO will have to obtain an application graph or at least the application structure – that brings also the challenge of defining the method of mapping the application code into a graph.
For the far-sighted optimization mode dedicated heuristics can be implemented. Thanks to introduction of this mode, the GrAppO should offer better results.

Implementing the Spare Connection to Monitoring System. Since the data obtained from Provenance Tracking System are crucial for GrAppO, introducing a spare connection to Monitoring System that would enable GrAppO to receive the information even if no Provenance System is present, should improve the reliability of the results provided by GrAppO. The solution that uses a dedicated database and connects to the Monitoring System in a model of notifications to listeners was described in 4.4.

Appendix A: Using GrAppO as External Library

The appendix specifies how to utilize GridSpace Application Optimizer functionality from within other projects built with Maven.

To use GridSpace Application Optimizer library as a dependency for a Maven2 project, two sections in the project `pom.xml` file have to be configured: the Maven2 repository location and dependency details.
The repository location can be configured by inserting the code presented on Code Snippet 1 into the `<repositories>` section (if this section is not present, it has to be created):

```
<repository>
      <releases>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
      </releases>
      <snapshots>
            <enabled>false</enabled>
      </snapshots>
      <id>cyfronet-repository</id>
      <name>Cyfronet Repository</name>
      <url>http://virolab.cyfronet.pl/maven2</url>
      <layout>default</layout>
</repository>
```

Code Snippet 1: Configuring Cyfronet Repository as Maven2 repository

Dependency details are specified in the `<dependencies>` section (as before, if it does not exist, it has to be created) with a code similar to one on Code Snippet 2:

```
<dependency>
      <groupId>cyfronet.virolab.grappo</groupId>
      <artifactId>grappo</artifactId>
      <version>0.2.2</version> <!-- use the latest version -->
</dependency>
```

Code Snippet 2: Configuring GrAppO as Maven2 dependency

Appendix B: GrAppO Class Diagrams

This appendix contains the class diagrams of the elements that served for the implemented GridSpace Application Optimizer functionality. The diagrams were created using reverse engineering.

The most general view on GridSpace Application Optimizer classes is presented on Figure 26. It is reflecting the GrAppO architecture – compare with section 4.2, Figure 7.

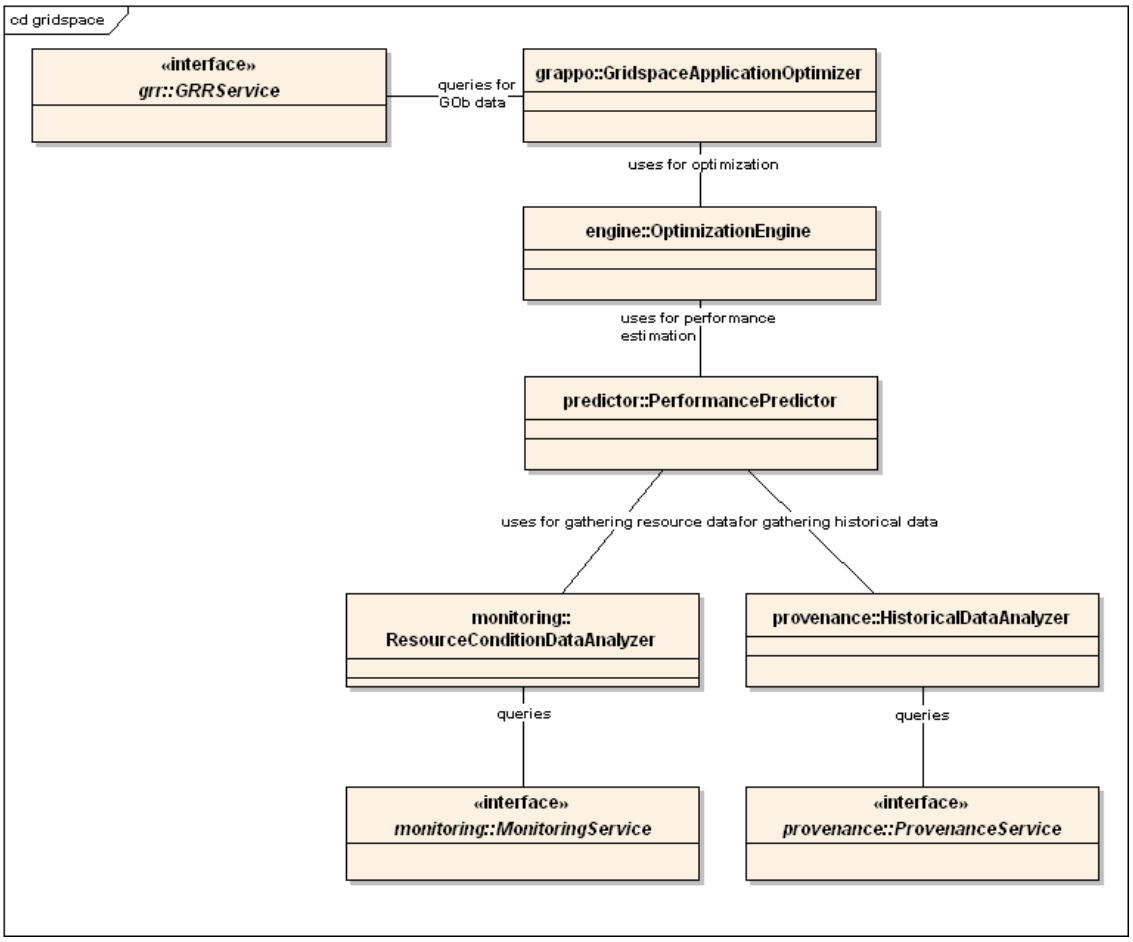


Figure 26: General view on GrAppO classes and connections

The main class – `GridspaceApplicationOptimizer`[14] corresponds to the GrAppO Manager component from section 4.2. The other classes bear the names of the components they represent. The `XService` interfaces stand for the services of other ViroLab components GrAppO contacts – the registry `(GRRService)`, monitoring `(MonitoringService)`, provenance `(ProvenanceService)`.

The subsequent diagrams will present a more detailed view on implemented GridSpace Application Optimizer classes.

Figure 27 presents the GrAppO façade – the main class `(GridspaceApplicationOptimizer)` along with the output it produces when called – either an instance of `OptimizationResult` or one of the exceptions.

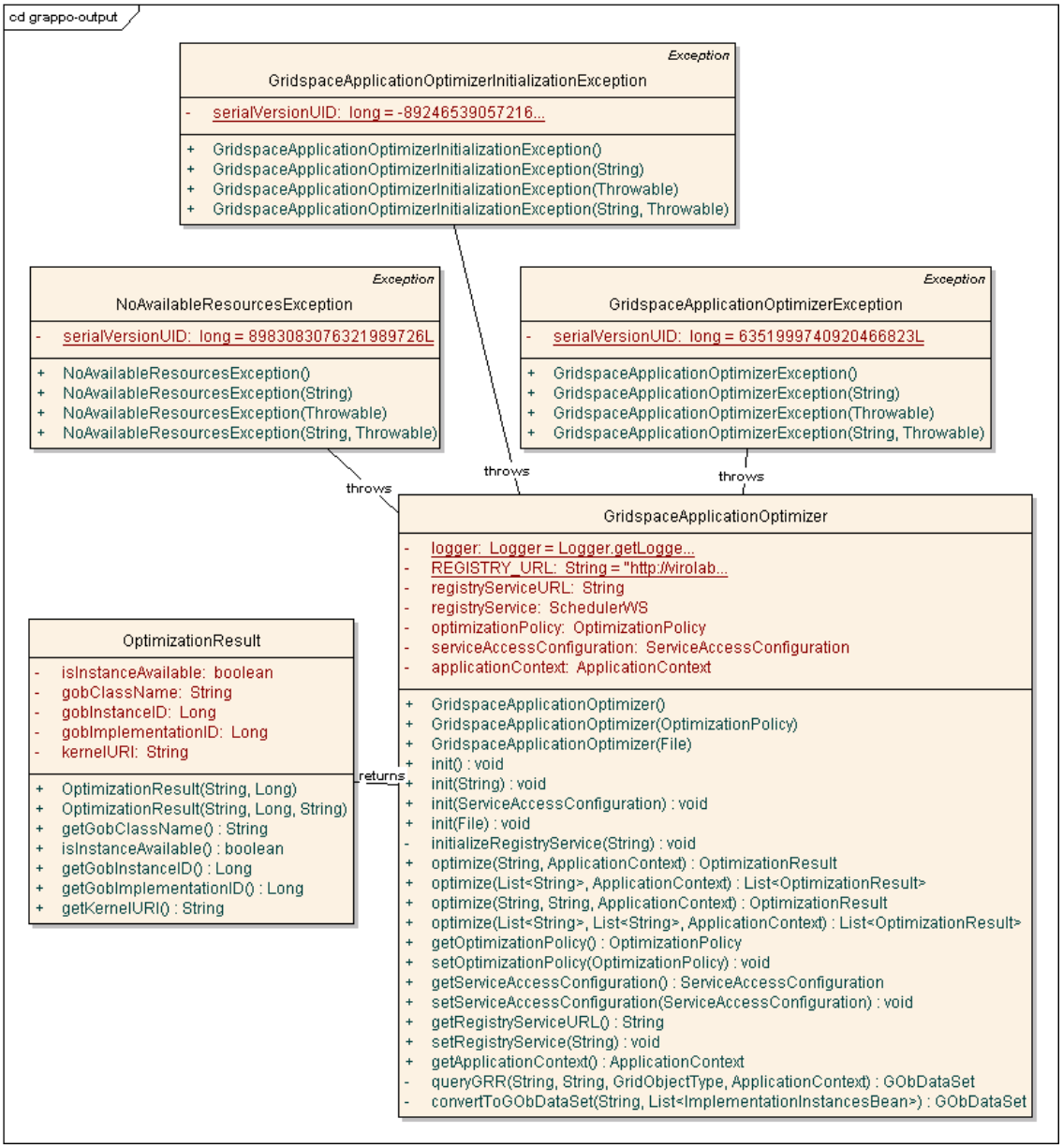


Figure 27: A class diagram concerning the output GrAppO produces

[14] Here and further in this appendix the provided classes and packages names are relative to the main GrAppO package: `cyfronet.gridspace.grappo`.

`OptimizationResult` is a class for encapsulating the result of optimization – the solution. As the solution can be either a ready Grid Object Instance or Grid Object Implementation with Grid Resource (for now – a H2O kernel) to deploy it on, the presented structure was proposed, to enable proper returning of the solution to Grid Object Invoker.

On Figure 28 GridSpace Application Optimizer configuration was presented – using classes:

- `configuration.ServiceAccessConfiguration` – for storing connection URLs to services which are GrAppO data sources,
- `configuration.OptimizationPolicy` – for configuring properties that are used during optimization process.

On the diagram also a Grid Resource Registry service interface is present, as the connection URL to the service is either provided directly by an initialization method or obtained from the service access configuration class `(configuration.ServiceAccessConfiguration)`.
The `ServiceAccessConfiguration` class extends the `java.util.Properties` class. Thanks to that it can be easily read from a text or XML file that use special format – described in Appendix D – using the method `Properties.load(java.io.InputStream)` or `Properties.loadFromXML(java.io.InputStream)`, respectively. The properties values were described in sections 5.3.5 (configuration using the provided classes) and 5.3.6 (configuration using files).
To facilitate serializing the `OptimizationPolicy` class to file, it implements the `java.io.Serializable` interface.

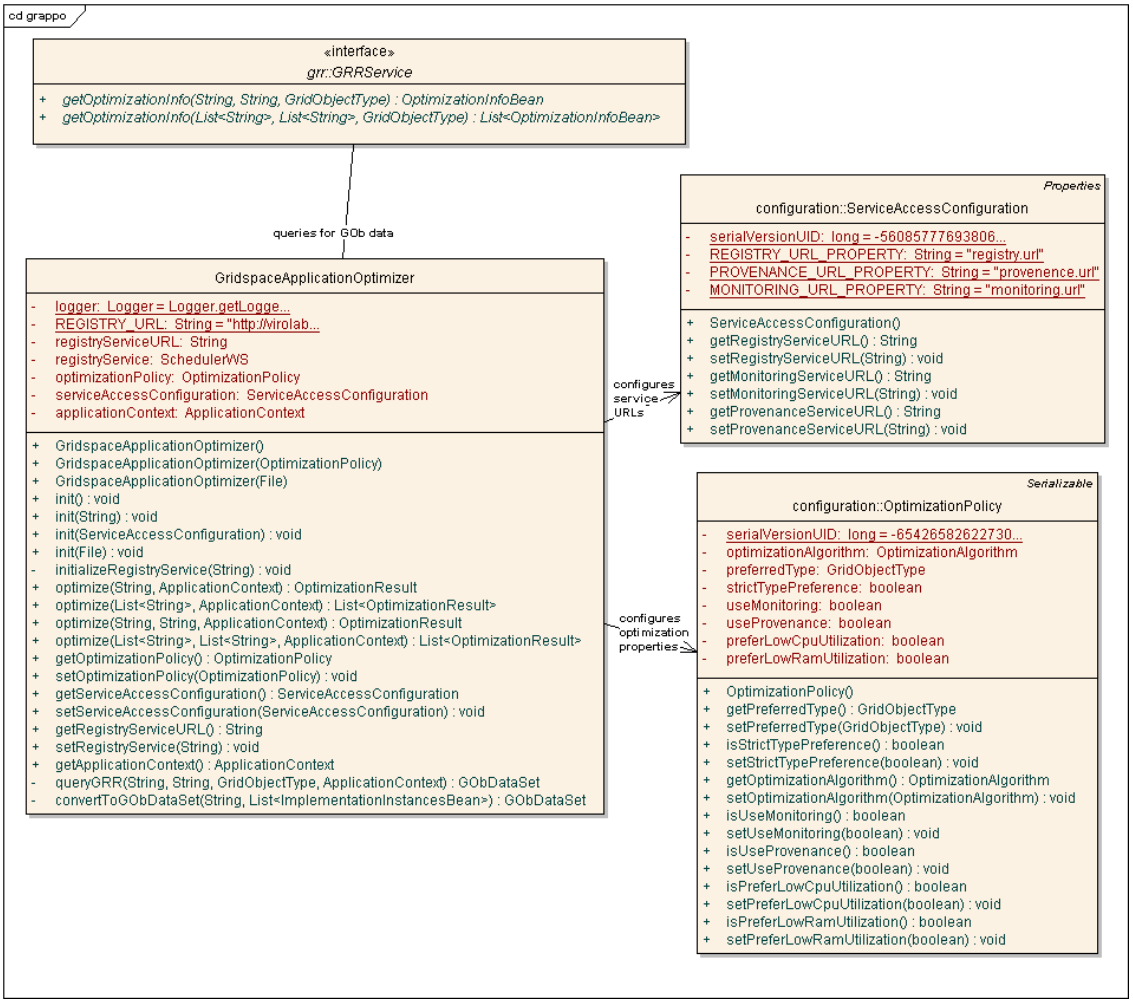


Figure 28: GrAppO configuration and Grid Resource Registry service access

A more detailed view on classes concerned with reading the configuration files is presented on Figure 34 and Figure 35.

The diagram presented on Figure 29 (next page) depicts the relations between classes that represent Grid Object entities (compare with section 1.2.1 and Figure 3).
The class `gob.GObDataSet` accumulates all the data concerning Grid Object Implementations and their instances (Grid Object Instances) and H2O kernels, that was obtained from Grid Resource Registry service. It contains collection of the objects representing Grid Object Implementations `(gob.GridObjectImplementation)` – which in turn aggregates objects representing all its instances `(gob.GridObjecInstance)` along with a collection of objects representing resources – namely H2O kernels `(gob.H2OKernel)`. `GObDataSet` also references an instance of `gob.GObRanking` which stores additional (to the performance estimation) rates concerning each entity in the data set (the rates can be higher e.g. for the implementations of type set as preferred in the optimization policy).

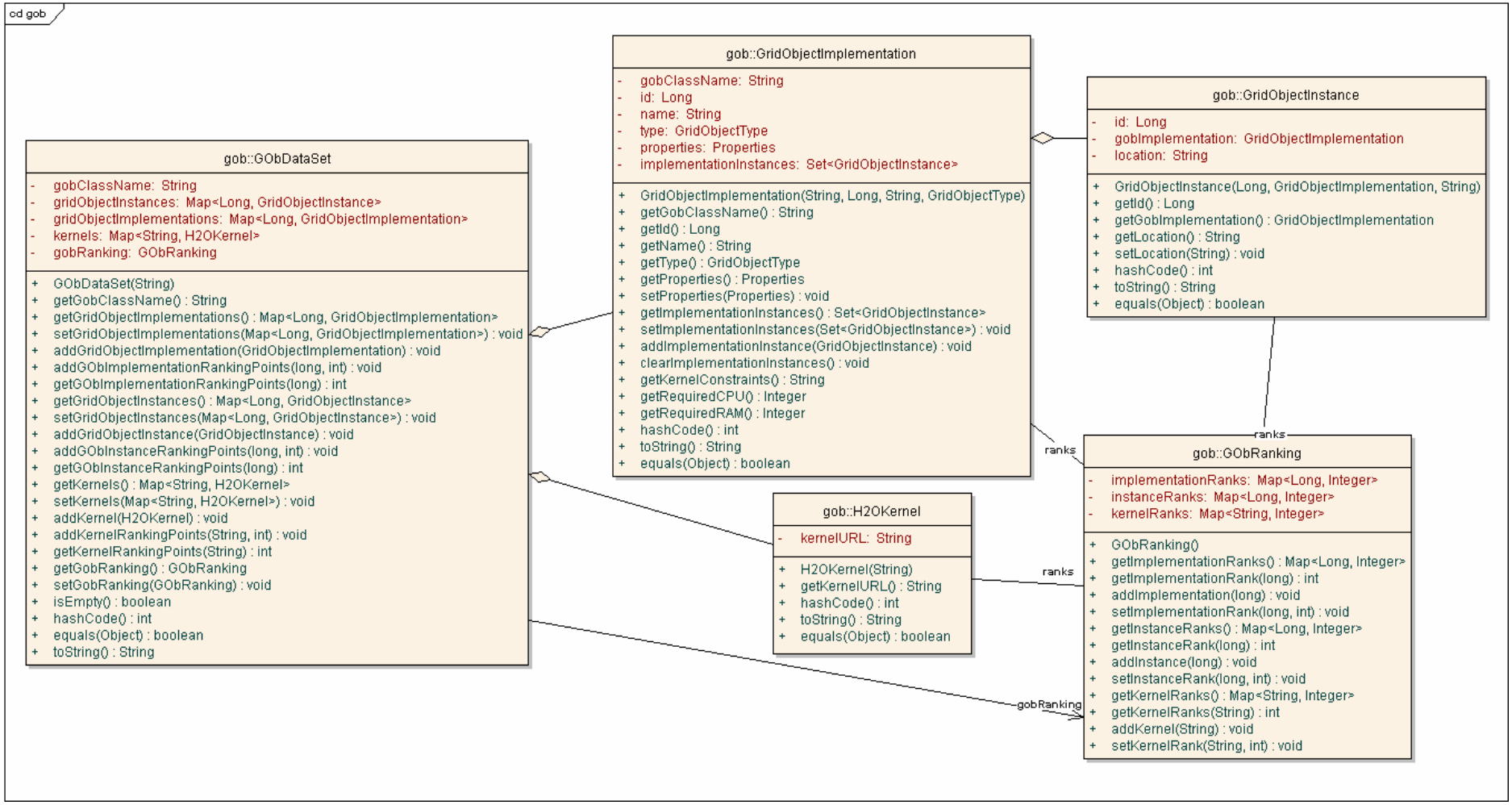


Figure 29: A diagram of classes representing Grid Object entities, their aggregation and ranking

The classes responsible for the optimization mechanism evaluation are depicted on Figure 30. The main class here is `engine.OptimizationEngine` class, which is a core of the Optimization Engine component – compare with GrAppO architecture (4.2). It uses the data obtained from the registry, the estimated performance data (gathered from other ViroLab services if available) – `predictor.PerformanceEstimation` – to evaluate the optimization algorithm, indicated by the optimization policy. The algorithm evaluated must implement of the `engine.OptimizationAlgorihtm` interface and its two methods – for single and group optimization (according to the optimization mode – short- or medium- sighted, respectively). The implementation can be simple (as `sample.RandomOptimzationAlgorithm`) or introduce advanced features, by extending the `algorithm.SimpleHeuristic` abstract class. In case of any error during the optimization process, the `engine.OptimizationException` is thrown.

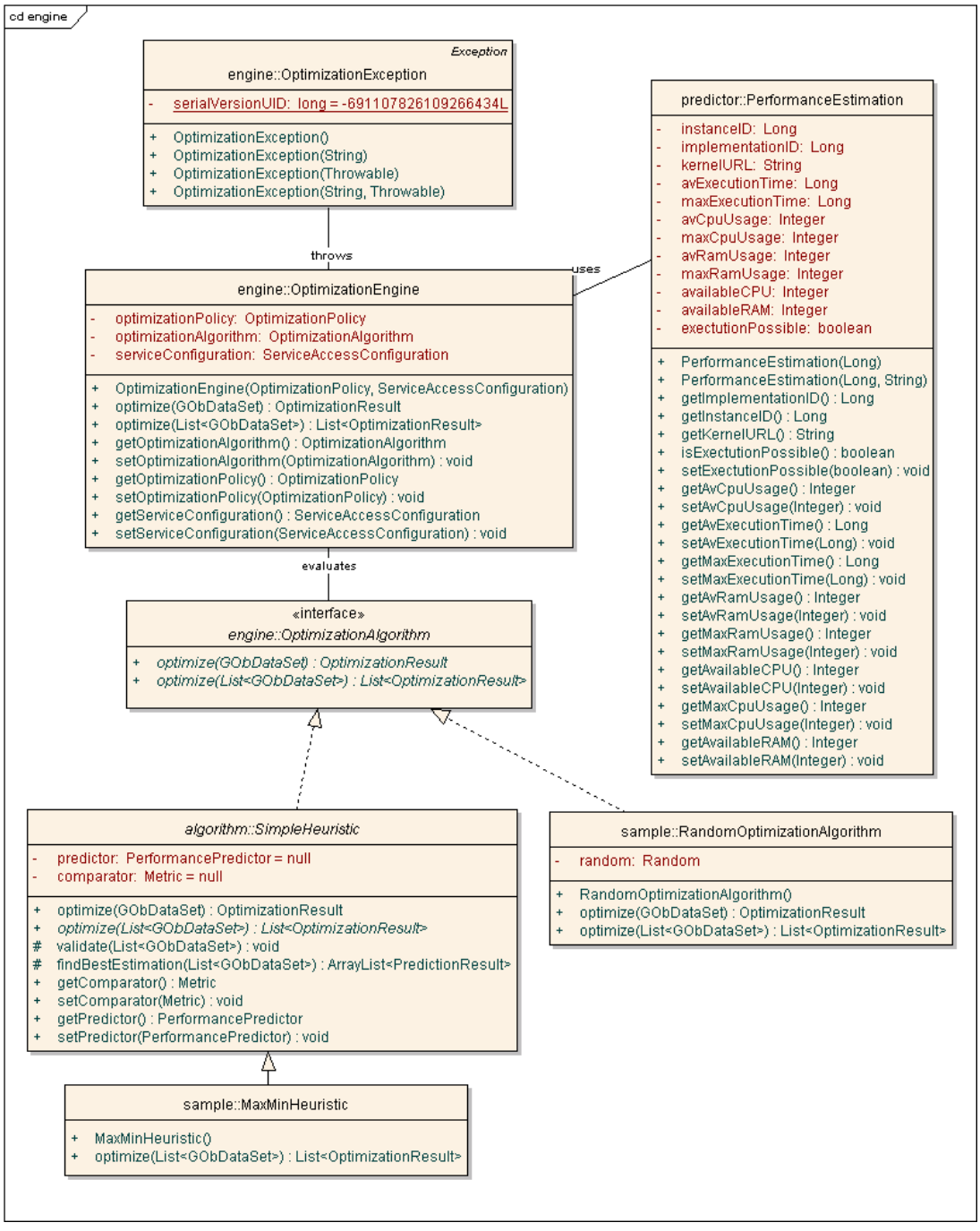


Figure 30: Classes concerning the optimization algorithms execution

The classes that take part in the estimation of each solution performance are presented on Figure 31 (next page). Their core class is `predictor.PerformancePredictor` – being also the engine of the whole GrAppO Predictor Performance component (please compare with GrAppO architecture – section 4.2).
The class uses the data gathered by Resource Condition Data Analyzer (see also Figure 32) in form of `monitoring.ResourceConditionData` and by Historical Data Analyzer (see also Figure 33) – `provenance.HistoricalImplementationPerformanceData`, which in turn aggregates a number of `provenance.HistoricalOperationPerformanceData` (the data structures will be further described with Figure 32 and Figure 33, respectively).
The output produced by `PerformancePredictor` is a list of performance estimations for the solutions for which any data relevant to performance were available. Each estimation is in form of `predictor.PerformanceEstimation` class instance. The estimated values it contains were derived from the data structures:

- `ResourceConditionData` – available CPU, RAM and availability time (the time in which the resource is going to be available)
- `HistoricalImplementationData` – average and maximum: execution time, RAM and CPU usage – the values are computed on the basis of the information of each implementation's operation executed, contained in `HistoricalOperationPerformanceData` objects.

In case of any error during computing the estimation, a `predictor.PerformancePredictionException` is thrown.

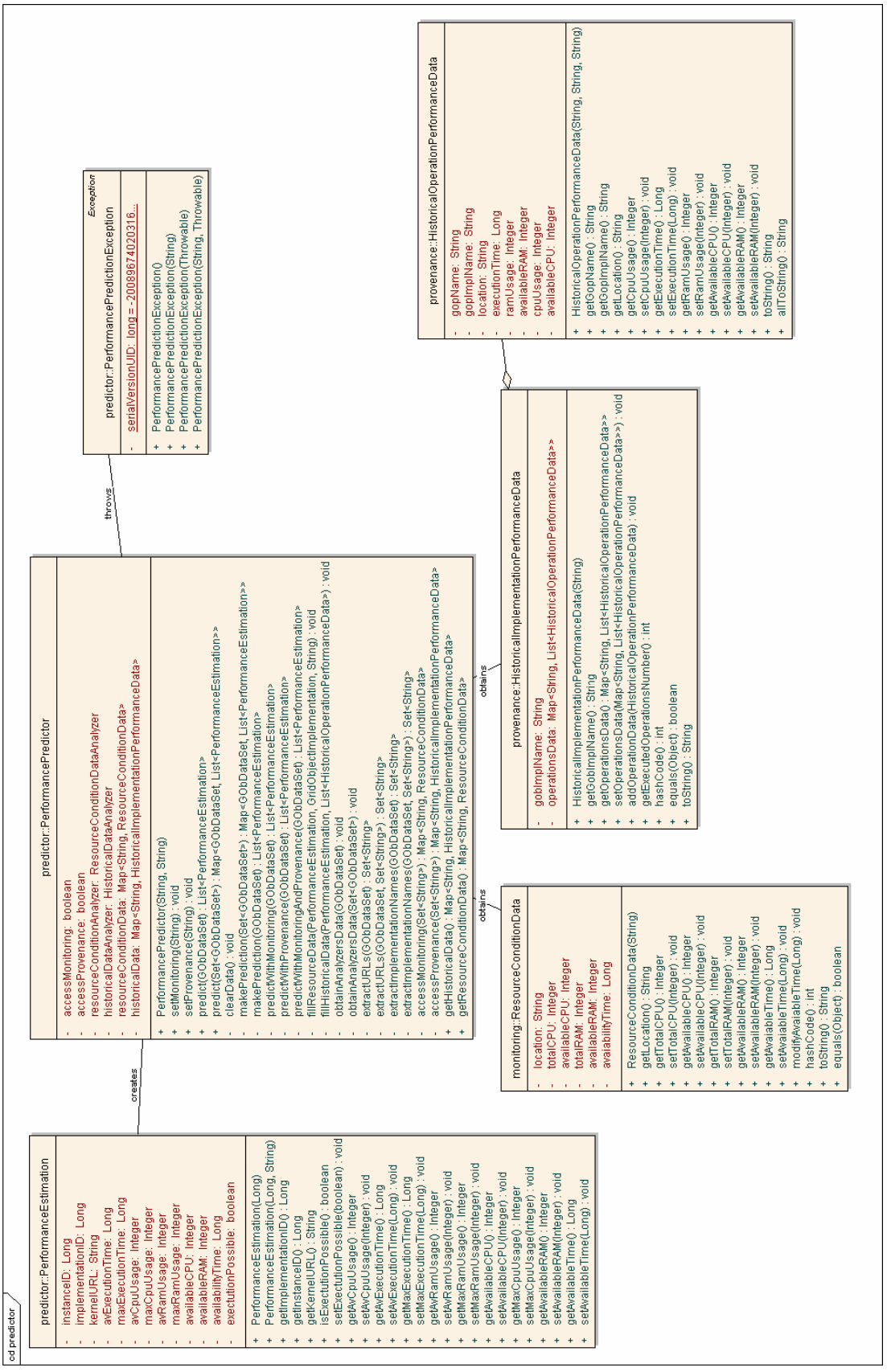


Figure 31: Performance Predictor classes

The main class responsible for contacting the Monitoring System service and analyzing data obtained from it is `monitoring.ResourceConditionDataAnalyzer` (see Figure 32). It is the heart of the Resource Condition Data Analyzer GrAppO component – compare with GrAppO architecture (section 4.2). On the diagram on Figure 32, the Monitoring System service is represented by an interface `monitoring.MonitoringService`. For the sake of performance tests (see section 6.4), it is now implemented as a local service producing test data.

The analysis performed by `ResourceConditionDataAnalyzer` class results in a `monitoring.ResourceConditionData` class instance, containing all the information that were be extracted from the information being result from the query to the Monitoring System. The query itself carries only the URLs of resources of interest as an argument.

In case any error during connection to monitoring or the analysis itself, the `monitoring.ResourceConditionDataAnalysisException` is thrown.

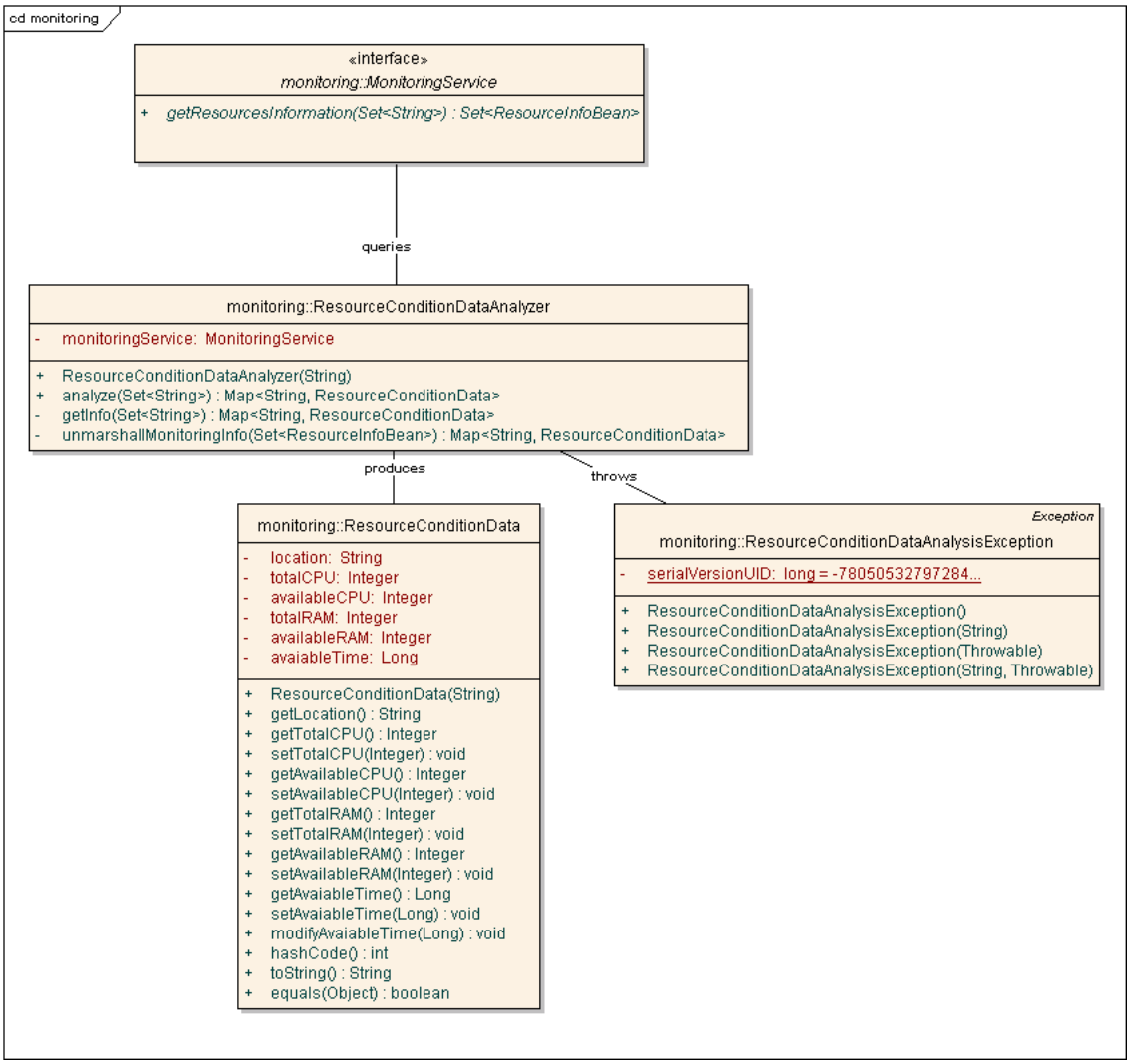


Figure 32: Resource Condition Analyzer classes

The construction of the Historical Data Analyzer component (for GrAppO architecture see section 4.2), presented on Figure 33, is very similar to Resource Condition Data Analyzer – compare with Figure 32.

The provenance service used for testing is, like in case of monitoring service, a local implementation of interface `provenance.ProvenanceService`, that is used for producing test data. The class `provenance.HistoricalDataAnalyzer` contacts the Provenance Tracking System service, and queries it with the names of Grid Object Implementations which previous performance it is interested in along with the URLs of resources which are currently available (not to obtain information about resources it cannot use). If the URLs set is empty, all resources are considered.

The result of the analysis performed by `HistoricalDataAnalyzer` is `provenance.HistoricalImplementationPerformanceData` – object representing all information that could be obtained about the performance of one Grid Object Implementation. It aggregates the data concerning all operations of this implementation that were executed in the past – in form of `provenance.HistoricalOperationPerformanceData`. This object is identified by the operation name, the implementation name and the resource that it was performed on – these data might not be unique, as the same operation could have been performed more than once. Other values, `HistoricalOperationPerformanceData` carries are information about the operation's execution time, the amount of RAM and CPU it consumed and that were available at the moment of execution.

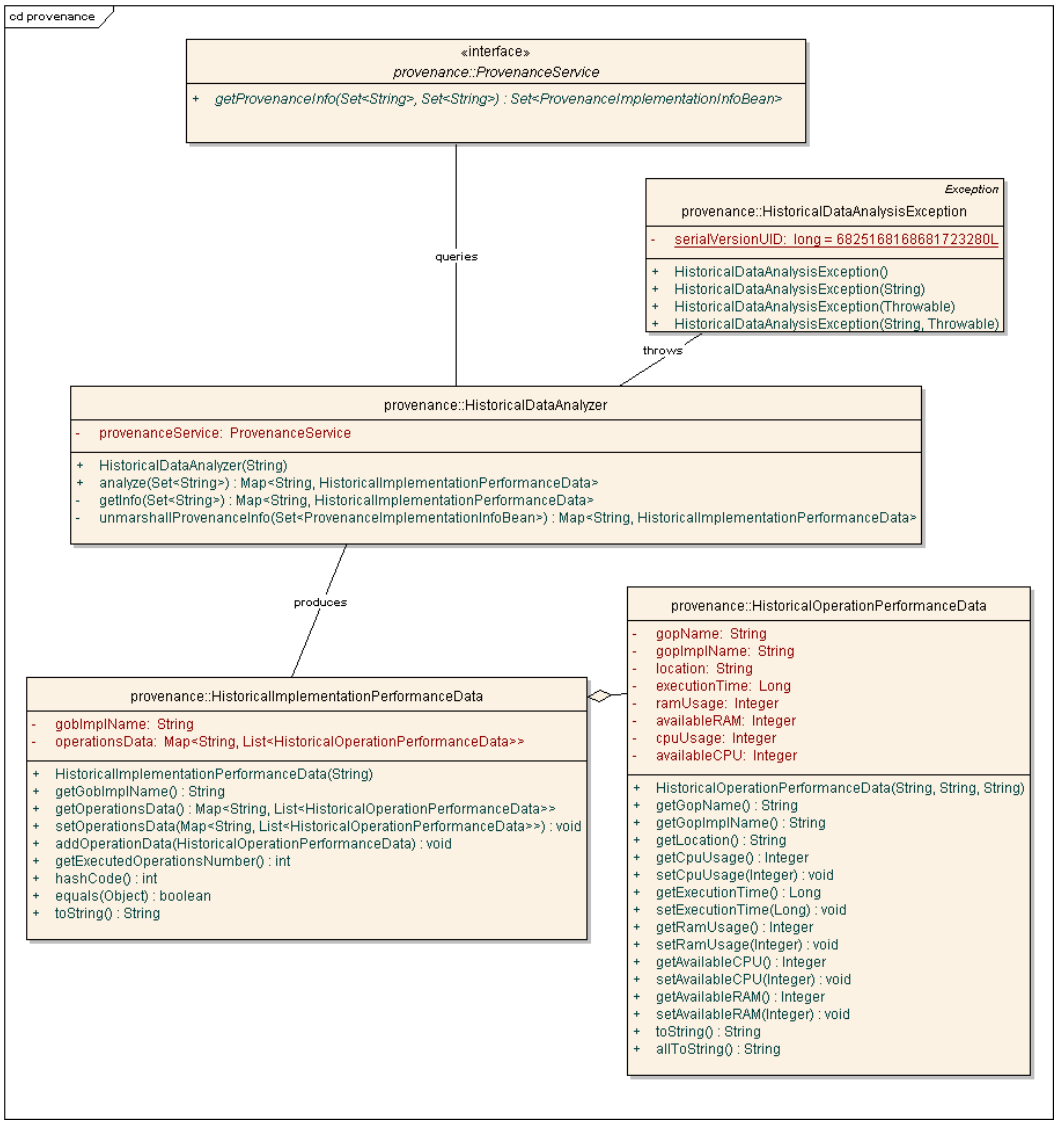


Figure 33: Historical Data Analyzer classes

The Figure 34 presents in detail the solution that was used when reading the optimization policy configuration from a file. From the contents of the configuration file, a `configuration.OptimizationPolicyConfiguration` is loaded, which contains only `String` variables. It is due to extending the `java.util.Properties` class. The `OptimizationPolicyConfiguration` class is afterwards transformed by a static method of `configuration.OptimizationBuilder` into a ready to use `configuration.OptimizationPolicy` class instance. It uses the final values stored in `OptimizationPolicyConfiguration` to determine the properties keys for the optimization policy when crating its instance.
The described solution was needed because the `OptimizationPolicy` class variables, unlike the variables of `ServiceAccessConfiguration` class, are instances of other than `String` java classes.

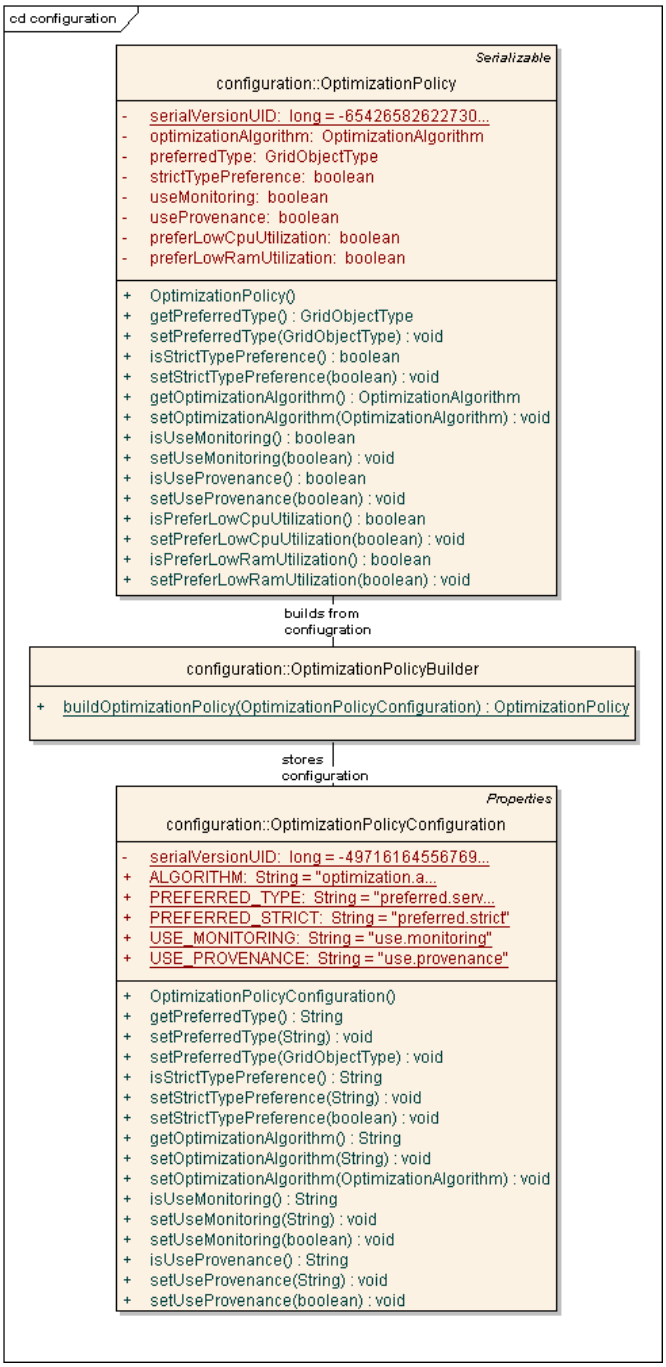


Figure 34: Detailed view on classes concerning GrAppO optimization policy configuration reading

The `util.ConfigurationFileReader` class, presented on Figure 35, is used for reading optimization policy and service access configuration files. It creates an `OptimizationPolicyConfiguration` class instance, which will be turned into an `OptimizationPolicy` instance (see Figure 34). The `ServiceAccessConfiguration` as well as `OptimizationPolicyConfiguration`, thanks to extending the `java.util.Properties` class, can be read directly from the file (see the description of Figure 28).
The exceptions can be thrown when the given file cannot be read `(util.ConfigurationFileReadingException)` or when the properties in the file are incorrect `(util.ConfigurationException)`.

Figure 35: Util class for reading configuration files

Appendix C: ViroLab Experiments Information

The appendix is intended to present information about the ViroLab experiments that were used in the tests described in section 6.3.

ViroLab Virtual Laboratory Experiments basic information:

```
Release version: 0.2.4
Release date: 18.05.2007
Authors: ACC CYFRONET AGH Krakow Poland (http://www.cyfronet.pl)
Project: EU IST ViroLab (http://www.virolab.org)
Virtual Lab website: http://virolab.cyfronet.pl
```

ViroLab Virtual Laboratory Experiments License:

```
The MIT License

Copyright (c) 2006-2007 ACC CYFRONET AGH Krakow Poland
(http://www.cyfronet.pl)

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

The following experiments were written in *GScript* – a scripting language for ViroLab based on the Ruby programming language [26].

Experiment sources of `align_experiment` (used for screen-shots on Figure 22 and Figure 24):

```
# Author: ACC CYFRONET AGH, Krakow, Poland
# Contact: Tomasz Gubala (gubala@science.uva.nl)
# License: please consult the MIT license provided previously in
#     this section

# Test RegaDB Alignment tool(s)
# Expected output: simple alignment info

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

argsArray =
[">LowScore\nAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"].to
_java :String

regaDBMutationsTool = GObj.create('regadb.RegaDBMutationsTool')

#align(sequence, region name)
regaDBMutationsTool.align(argsArray, "PRO")

while regaDBMutationsTool.isAligning do
  sleep(2)
  puts "aligning"
end

puts "Result: "+ regaDBMutationsTool.getResult
puts "Exit value: " + regaDBMutationsTool.getExitValue.to_s

# clean up MOCCA components
regaDBMutationsTool.undeploy
MoccaResource.cleanup
puts 'End of align experiment !!'
JSystem.exit(0)

#end
```

Code Snippet 3: Source code of the `aligh_experiment.rb` script

Experiment sources of `weka_experiment` (used for screen-shots on Figure 23 and Figure 25):

```
# Author: ACC CYFRONET AGH, Krakow, Poland
# Contact: Tomasz Bartynski
# License: please consult the MIT license provided previously in
#     this section

# Test Weka classification tools (ViroLab specific)
# Experiment:
# 1.Use Database data loader (WS) to load a data set from an SQL
#     database table. The result is returned in ARFF format (A)
# 2.Use ARFF Data splitter (WS) to split A into trainA and testA
#     It takes trainA and the percentage of testA that will be
#     created. The result is a bean with trainA and testA
# 3.Instantiate One-rule classifier (MOCCA)
# 4.Train it with One-rule classifier::Train method and trainA
# 5.Execute classification with One-rule classifier::Classify
#     method and testA. Returns attributes collection B
# 6.Use Data ARFF predict checker (WS) to evaluate the
```

```
#     percentage of right classification


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

logger = JLogger.getLogger('goi.wekaexperiment')
logger.info('Start of weka experiment !!')

# Create Web Service Grid Object Instance
retriever =  GObj.create('cyfronet.gridspace.gem.weka.WekaGem')
# Build the query

QUERY = 'select outlook, temperature, humidity, windy, play from
weather limit 100;'
DATABASE = "jdbc:mysql://127.0.0.1/test"
USER = 'testuser'
PASSWORD = ''

param = {
   :dbUrl => DATABASE,
   :query => QUERY,
   :username => USER,
   :password => PASSWORD
 }
# (1)
A = retriever.loadDataFromDatabase(DATABASE, QUERY, USER,
PASSWORD)

# (2)
# prepare args for WS splitting data
param = {
   :data => A,
   :trainingDataPercent => 20
 }
B = retriever.splitData(A, 20)
trainA = B.predictingData
testA = B.testingData

# (3)
classifier =
GObj.create('cyfronet.gridspace.gem.weka.OneRuleClassifier')

# Set the name of attribute that will be predicted
attributeName = 'play'
# (4)
classifier.train(trainA, attributeName)

# (5)
prediction = classifier.classify(testA)
logger.info('Predicted data:' + prediction.to_s)

# (6)
classificationPercentage = retriever.compare(testA, prediction,
attributeName)
# show results
logger.info('Prediction quality:' +
classificationPercentage.to_s)

# clean up MOCCA components
classifier.undeploy
logger.info('End of weka experiment !!')
```

```
MoccaResource.cleanup

JSystem.exit(0)
```

Code Snippet 4: Source code of the `weka_experiment.rb`

Appendix D: GrAppO Configuration Files Format

This appendix presents the format of files that can be used for GridSpace Application Optimizer configuration. The format is a standard for storing properties, that can be read either with `java.util.Properties.loadFromXML(java.io.InputStream)` – for an XML file or `java.util.Properties.load(java.io.InputStream)` – for an ordinary text file.

The XML configuration file must begin with the DTD schema declaration: http://java.sun.com/dtd/properties.dtd. Like on Code Snippet 5:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
       "http://java.sun.com/dtd/properties.dtd">
<properties>
      <entry key="property">value</entry>
</properties>
```

Code Snippet 5: Example XML GrAppO configuration file

A text file intended for storing the GrAppO configuration should be an ordinary file filled with key=value entires. For example:

```
property=value
```

Code Snippet 6: A sample entry in the text GrAppO configuration file

The actual properties names needed for GrAppO configuration were described in Appendix B.

Appendix E: GrAppO Performance Tests Environment Details

This appendix presents a snapshots of the XML files that were used for testing of the GridSpace Application Optimizer performance.

The extract on Code Snippet 7 presents the XML representation of example data that can be obtained from Grid Resource Registry service.

```
<?xml version="1.0" encoding="UTF-8"?>
<GridSpace>
  <OptimizationInfoBean className="classname0">
     <kernels>
        <kernel>address1</kernel>
        <kernel>address7</kernel>
     </kernels>
     <ImplementationInstances>
        <ImplementationInstance>
           <id>1000</id>
           <name>impl1000</name>
           <implementationType>WS</implementationType>
           <cpuUtilizationRate>47</cpuUtilizationRate>
           <ramUtilizationRate>584</ramUtilizationRate>
           <InstanceInfos>
              <InstanceInfo id="1000" location="address3" />
              <InstanceInfo id="1001" location="address4" />
           </InstanceInfos>
        </ImplementationInstance>
        <ImplementationInstance>
           <id>1001</id>
           <name>impl1001</name>
           <implementationType>MOCCA</implementationType>
           <cpuUtilizationRate>782</cpuUtilizationRate>
           <ramUtilizationRate>409</ramUtilizationRate>
           <InstanceInfos>
              <InstanceInfo id="1002" location="address2" />
           </InstanceInfos>
        </ImplementationInstance>
     </ImplementationInstances>
  </OptimizationInfoBean>
</GridSpace>
```

Code Snippet 7: Sample structure of data generated as obtained from GRR

Code Snippet 8 shows XML representation of sample data from Monitoring System service.

```
<?xml version="1.0" encoding="UTF-8"?>
<MonitoringInformation>
<resource>
     <location>address0</location>
     <cpu>4853</cpu>
     <cpuUsage>1717</cpuUsage>
     <ram>2761</ram>
     <ramUsage>800</ramUsage>
     <availableTime>389</availableTime>
</resource>
<resource>
     <location>address1</location>
     <cpu>2748</cpu>
     <cpuUsage>1617</cpuUsage>
     <ram>4360</ram>
     <ramUsage>947</ramUsage>
     <availableTime>248</availableTime>
</resource>
<resource>
     <location>address2</location>
     <cpu>4827</cpu>
     <cpuUsage>1635</cpuUsage>
     <ram>3555</ram>
     <ramUsage>59</ramUsage>
     <availableTime>458</availableTime>
</resource>
</MonitoringInformation>
```

Code Snippet 8: Sample structure of data generated as obtained from Monitoring System

The last extract (Code Snippet 9) presents the XML representation of example data that can be obtained from the Provenance Tracking System service.

```
<?xml version="1.0" encoding="UTF-8"?>
<HistoricalOperationPerformance>
  <GObImpl name="impl1000">
        <operation name="name1" location="address1">
           <availableCPU>2889</availableCPU>
           <cpuUsage>89</cpuUsage>
           <availableRAM>3823</availableRAM>
           <ramUsage>88</ramUsage>
           <executionTime>3</executionTime>
        </operation>
        <operation name="name1" location="address4">
           <availableCPU>4426</availableCPU>
           <cpuUsage>89</cpuUsage>
           <availableRAM>4418</availableRAM>
           <ramUsage>93</ramUsage>
           <executionTime>17</executionTime>
        </operation>
  </GObImpl>
  <GObImpl name="impl1001">
        <operation name="name1" location="address0">
           <availableCPU>3037</availableCPU>
           <cpuUsage>129</cpuUsage>
           <availableRAM>1660</availableRAM>
           <ramUsage>85</ramUsage>
           <executionTime>31</executionTime>
        </operation>
        <operation name="name1" location="address1">
```

```
            <availableCPU>3059</availableCPU>
            <cpuUsage>135</cpuUsage>
            <availableRAM>4326</availableRAM>
            <ramUsage>99</ramUsage>
            <executionTime>26</executionTime>
         </operation>
   </GObImpl>
</HistoricalOperationPerformance>
```

Code Snippet 9: Sample structure of data generated as obtained from PROToS

References

[1] Official ViroLab webpage: http://www.virolab.org

[2] 6th Framework Programme: http://ec.europa.eu/research/fp6/index_en.cfm?p=0, European Commission.

[3] ViroLab Virtual Laboratory homepage by ACC Cyfronet AGH team: http://virolab.cyfronet.pl

[4] ViroLab WP3 team: *"ViroLab Virtual Laboratory Design Document"*; December 2006.

[5] Fangpeng D., Selim G. Akl: *"Scheduling Algorithms for Grid Computing: State of the Art and Open Problems"*; School of Computing, Queen's University, Kingston, Ontario; January 2006.

[6] Fahringer T., Prodan R., Duan R., Nerieri F., Podlipnig S., Qin j., Siddiqui M., Truaong H., Villazón A., Wieczorek M.: *"ASKALON: A Grid Application Development and Computing Environment"*; Institute of Computer Science, University of Innsbruck.

[7] Ritchie G., and Levine J.: *"A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments"*.

[8] Jacob B.: *Introduction to Grid Computing*, IBM Redbooks, 2005.

[9] Casanova H., Legrand A., Zagorodnov D., and Berman F.: *Heuristics for Scheduling Parameter Sweep Applications in Grid Environments*, Proc. of the 9th hetero-geneous Computing Workshop 2000, 349—363.

[10] Foster I., Kesselman C., and Tuecke S.: *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, International Journal Supercomputer Applications 2001, 200—220

[11] Joseph J. and Fellenstein C.: *Grid Computing*, IBM Press, 2004.

[12] Joseph J., Ernest M., and Fellenstein C.: *Evolution of grid computing architecture and grid adoption models* (http://www.research.ibm.com/journal/sj/434/joseph.pdf).

[13] Kim J., Rho J. Lee J., Ko M.: Effective Static Task Scheduling for Realistic Heterogeneous Environment, 7th International Workshop on Distributed Computing, IWDC 2004, Khargpur, India, December 2005.

[14] Kwok Y.-K.and Ahmad I.: *Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors*. IEEE Trans. Parallel Distrib. Syst., 7(5):506–521, 1996.

[15] Nabrzyski J., Schopf J. M., and Weglarz J.: *Grid Resource Management. State of the Art and Future Trends, ch 2*. Ten Actions When Grid Scheduling, Kluwer Academic Publishers, 2003.

[16] Li M. and Baker M.: *The Grid: Core Technologies*, John Wiley & Sons, 2005.

[17] Maheswaran M., Ali S., Siegel H.J., Hensgen D., and Freund R.: *Dynamic mapping of a class of independent tasks onto heterogeneous computing systems*, IEEE Heterogeneous Computing Workshop 1999, 30—44.

[18] Topcuoglu H., Hariri S. and Wu M.Y., *Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing* , IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 3, pp. 260 - 274, 2002.

[19] Buyya R. and Baker M. (eds.), *Grid Computing - GRID 2000, ch. Evaluation of Job-Scheduling Strategies for Grid Computing*, Springer, 2000.

[20] Casavant T.L. and Kuhl J.G.: *A Taxonomy of Scheduling in General-Purpouse Distrbuted Computing Systems*, IEEE Trans. on software Engineering vol.14 (1998), no.2, 141—154.

[21] Shi Z.: *Scheduling Tasks with Precedence Constraints on Heterogeneous Distributed Computing Systems*, The University of Tennesse, Knoxville, December 2006.

[22] He X., Sun X., and Laszewski G.: *A QoS guided scheduling algorithm for grid computing*, Workshop on Grid and Cooperative Computing 2002.

[23] ViroLab Wiki pages for developer from ACC Cyfronet AGH: http://grid.cyfronet.pl/virolab/wiki/doku.php

[24] GridSpace Application Optimizer GForge project: http://gforge.cyfronet.pl/projects/grappo

[25] GridSpace Application Optimizer homepage: https://virolab.cyfronet.pl/~asia/grappo

[26] Ruby Programming Language: http://www.ruby-lang.org/en/