AGH University of Science and Technology
in Krakow, Poland
Faculty of Electrical Engineering, Automatics, Computer Science
and Electronics
Institute of Computer Science

Master of Science Thesis

Recording application executions enriched with domain semantics of computations and data

Michał Pelczar

Major: Computer Science
Specialization: Computer Systems and Databases Engineering

Supervisor
Marian Bubak, Ph.D.
Consultancy
Bartosz Baliś, M.Sc.

Matricula: 120576

Cracow 2008

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Akademia Górniczo-Hutnicza
im. Stanisława Staszica
w Krakowie
Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki
Katedra Informatyki

Praca magisterska

Zapisywanie przebiegu wykonania aplikacji wzbogacone o semantykę dziedzinową obliczeń i danych

Michał Pelczar

Kierunek: Informatyka
Specjalność: Inżynieria systemów komputerowych i baz danych

Promotor
dr inż. Marian Bubak
Konsultacja
mgr inż. Bartosz Baliś

Nr albumu: 120576

Kraków 2008

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Abstract

Provenance, perceived as derivation history of data or recorded application execution, is considered to be a critical part of all modern e-Science infrastructures. It serves as guarantee of data reliability and quality, regulatory mechanism of data protection and mean of efficiency optimization. On the basis of properly represented and collected provenance, reproducibility of scientific results is provided.

This thesis introduces a provenance ontology model covering workflow execution tracking, data items dependency, resources availability, performance issues and semantics of medical tools. The work is focused on a managable process of information building, in which provenance ontology is created based on low level monitoring events and data sets from distributed repositories. Semantically valuable representation, good adaptivity to the evolving ontologies and schemas as well as support for different levels of computation and data semantics are the main issues examined in this work. Within the scope of this thesis also a user-oriented querying tool, dedicated for virologists and clinicians, is presented. QUaTRO enables intuitive mining over both provenance information and medical data by the means of abstract language and mapping ontologies. Presented approach is validated on geno2drs application supporting HIV virus treatment and integrated into the ViroLab virtual laboratory.

Contents of this thesis is organized as follows. Chapter 1 gives motivation and problem survey. Chapter 2 contains overview of basic aspects of Semantic Web, scientific workflows, provenance and ViroLab. System requirements and overall solution are investigated in chapter 3. Chapter 4 introduces the provenance information model and the monitoring data model. The process of information building is discussed in chapter 5, while the technical aspects of how the process is implemented are explained in chapter 6. In chapter 7 the approach is validated against geno2drs application. The practical usage of created information is presented in chapter 8. Finally, conclusion and outcomes are described in chapter 9.

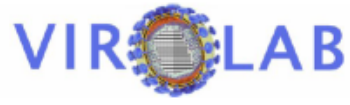
Keywords: e-Science, Semantic Web, ontologies, provenance, Grid, monitoring, scientific workflows, virtual laboratories, ViroLab.

Acknowledgements

I wish to express my deepest gratitude and thanks to my supervisor Marian Bubak, Ph.D., for his encouregment, expert guidance and invaluable commitment during the work of this thesis.

I am also most grateful to Bartosz Baliś, M.Sc., for his time, meaningful criticisms and very helpful collaboration.

This work was made possible thanks to the ViroLab project (`http://www.virolab.org`).



Contents

Chapter 1. Introduction . . . 5
1.1. Motivation . . . 5
1.2. Objectives . . . 7
1.3. Organization of this thesis . . . 8
Chapter 2. Background: Semantic Web, provenance and applications 9
2.1. Semantic Web . . . 9
2.2. Scientific workflows . . . 12
2.3. Provenance . . . 13
2.4. ViroLab virtual laboratory . . . 14
Chapter 3. Concept of system for provenance recording . . . 18
3.1. Overall requirements . . . 18
3.2. From requirements management to provenance mining . . . 20
Chapter 4. Provenance model . . . 23
4.1. Monitoring data model . . . 23
4.1.1. Generic monitoring events . . . 23
4.1.2. Domain monitoring events . . . 26
4.2. Provenance ontology . . . 27
Chapter 5. Semantic Event Aggregator . . . 30
5.1. Main idea behind Semantic Event Aggregator . . . 30
5.2. Monitoring events correlation . . . 32
5.3. Ontology Extension . . . 35
5.3.1. Derivation Concepts . . . 35
5.3.2. Concept of Delegates . . . 38
5.3.3. Aggregation Rules . . . 44
5.4. Experiment transaction support . . . 47

5.5. Semantic associations discovery . . . 52
5.5.1. Hashing individuals naming . . . 52
5.5.2. Knowledge history tracking . . . 54
5.5.3. Context association . . . 57
Chapter 6. Design and implementation of Aggregator . . . 61
6.1. Aggregator architecture . . . 61
6.2. Aggregator deployment . . . 66
Chapter 7. Proof of concept: A drug resistance case study . . . 70
7.1. Geno2drs scientific landscape . . . 70
7.2. Geno2drs ontology . . . 71
7.3. Geno2drs information building . . . 72
Chapter 8. Querying over provenance . . . 75
8.1. User-oriented querying approach . . . 75
8.2. Abstract query language . . . 76
8.3. Query processing . . . 79
8.3.1. Ontological queries . . . 82
8.3.2. Data base queries . . . 85
8.3.3. Relational queries . . . 85
8.3.4. Transient queries . . . 86
8.3.5. Inverse transient queries . . . 87
8.3.6. WebDAV queries . . . 87
Chapter 9. Summary and future work . . . 88
9.1. Outcomes . . . 88
9.2. Research outlook . . . 89
Appendix 1. Creation of monitoring events . . . 91
Appendix 2. Logging of monitoring events . . . 94
Appendix 3. GEMINI monitoring system . . . 96
Bibliography . . . 103
List of Figures . . . 106
List of Tables . . . 108
Publications . . . 109

Chapter 1

Introduction

This chapter briefly presents the scientific context of provenance. Idea of semantic description of information building process is presented together with main issues motivating this approach. Finally, organization of this thesis is given.

1.1. Motivation

It is commonly agreed that the computer systems has a great impact on todays biological science. It must also be emphasized that the scientific discoveries pace heavily rely on the capabilities offered by modern computational infrastructures. In the past, computer support was limited to the appliance of desktop applications supporting the creation of scientific models through preparing some mathematic and statistic computations on data sets collected in local base. However, this situation has changed dramatically with the rapid evolution of computers and internet. Development of a vision of infrastructure providing the integration of numberous data repositories, computational resources and communication channels as well as the support of collaboration among scientists would dramatically change the nature of science. Ideally, the researcher would be able to design a scientific research at a high level by describing the data sets he wishes to work with and the relationships he wishes to traverse by using a graphical tool or a high level description language [10]. This leads to the idea of *e-Science*.

The main goal of science is to posses knowledge explaining natural phenomena through the means of experiments. Similarly, the essential aspect of e-Science are sophisticated virtual experiments, which integrate scientific code at a high modelling level and may be executed in a distributed way, both in terms of computation and data distribution. In such an approach, a scienfitic task is virtualized as workflow describing the data processing stages and the data dependencies between these stages [9]. Generally, the activities applied by workflow management system leads to the achievement of two possible outcomes: solution of a particular problem or definition of a new service [8]. From the other side, the pure results obtained through the workflow execution are insufficient in scientific environment.

In e-Scientists community must exist a possibility to reproduce the scientific analysis in order to evaluate its validity [9]. Because of this, each piece of data must be associated with its *provenance*, describing how this data was produced, in what services and when it was trasformed, by who and it what context [7]. Moreover, the provenance tracking system may record the entire experiment execution. The data mining of workflows lifecycle history gives the possibilities to their optimizations [9], what is especially valuable because high performance is essential in large-scale computing.

As for now, provenance was analyzed in many terms: what is its purpose and scope, what is the suitable model, in what way it should be stored and what are the requirements for provenance accessing components. However, the aspect of how provenance information is extracted is still on the research state. Therefore, it is reasonable to define an *experiment information building process that is semantically described and conveniently manageable*. This is justified for several reasons.

Firstly, in order to enable the provenance dissimination, there must be agreed a common provenance understanding. That would be enable through the incorporation of provenance into Semantic Web, in which all accessible information is given a well defined meaning. This can be achieved by the development of provenance ontology [5]. However, the ontology may be unstable so the building process must be easily adaptive to the evaluating information model. In a desirable situation, the scientist who augments the provenance ontology should be able to effortlessly redefine the building process. He is expected to be familiarized with information modelling, so the ontological description of building principles would be most convenient in management.

What is more, the provenance information is built on the basis of data sets accessible somewhere in the computational infrastructure. The most suitable infrastructures for e-Science, providing data scaling, high performance computing and specialized scientific instrumentation are grid systems [4]. The studies of essential grid components leads to the conclusion, that the only existing source of data created transparently

for the end-user and accurately reflecting the experiment course is monitoring infrastructure [3]. Therefore, there is a need to provide generic mapping between low level monitoring data and high level provenance information. Generic design provides the reusability of information building infrastructure, because it may be adapted to another grid system differing in monitoring data model and provenance ontology.

Moreover, the building process should be easily extendable by the means of queries adressing remote data sources. In such an approach, the high level ontology might be extracted not only from low level monitoring data, but also from data bases and file systems content. Translation of monitoring data model combined with another data schemas to information would be a sophisticated and complex process. In order to enable querying of distinct, distributed data storages implemented in different technologies and based on different models, there should exists possibility to integrate into the building process middleware additional software mining the remote repositores. Principles of this software usage would be described semantically.

Another aspect of provenance tracking is that not all computational resources are incorporated into Semantic Web. Many services are not described semantically, however, it should not be a reason to depreciate their magnitude in scientific workflows. The same refers to the data sets with undefined meaning. The information building component must be able to integrate, in a single provenance record, information about data as well as computations modelled at different levels of semantic. That would also serve as encouragement for the appliance of presented approach in grid systems where semantics enrichment is at immature stage of development.

1.2. Objectives

The problem described in section 1.1 may be specified as the reconciliation of abstract modules related to provenance. Main objective is to design and implement a *Provenance Creation Process*, while the *Provenance Model* ontologically describes its meaning. The provenance information is created from low level data delivered from *Data Producers* by the means of *Monitoring Infrastructure*. Created information should be recorded in *Provenance Storage*, which enables easy way of exploration by *Provenance Access* component. Furthermore, this process may be augmented by the usage of separate *Data Access* integrating distributed *Data Bases*. Requirements and activities related with provenance system implementation are specified in chapter 3. In comparison to the existing provenance systems, this project presents more flexible approach. Principles of provenance extraction are manageable by the means of ontologies. What is more, this approach is truly semantic - ontologies are not only used to annotate the provenance concepts expressed on lower level, but are directly and explicitly built through their individuals. Moreover, provenance is well integrated

with data, what enables more expressive and meaningful queries. Last but not least, provenance is not built in grid middleware, but by a separate, dedicated service.

1.3. Organization of this thesis

Contents of this thesis is organized as follows. Chapter 1 gives motivation and problem survey. Chapter 2 contains overview of basic aspects of Semantic Web, scientific workflows, provenance and ViroLab. System requirements and overall solution are investigated in chapter 3. Chapter 4 introduces the provenance information model and the monitoring data model. The process of information building is discussed in chapter 5, while the technical aspects of how the process is implemented are explained in chapter 6. In chapter 7 the approach is validated against geno2drs application. The practical usage of created information is presented in chapter 8. Finally, conclusion and outcomes are described in chapter 9.

Chapter 2

Background: Semantic Web, provenance and applications

This chapter characterizes more precisely the context of application execution recording. Two aspects of e-Science are discussed: Semantic Web, together with its basic elements – XML, RDF and OWL languages, and grid computing dedicated for workflow execution. OWL features are illustrated by examples from ViroLab provenance ontology. There is also given explanation of why provenance is so important for scientists. Finally, the architecture of ViroLab virtual laboratory, in which scope this work is realized, is briefly described.

2.1. Semantic Web

Semantic Web is an extension of the current one, in which data is given well-defined meaning [5]. Thanks to the formal description of data types, particular data items accessible in the Web are understand in the same way by people exchanging the Web resources. This is crucial because of interoperability issue. In order to share something in a distributed way, for example a virus gene, there must be agreement about what a gene is, how genes are named and what pieces of data can be attached to a gene [10]. The data associated with taxonomy describing its semantics is called *information*. Besides inter-community coooperation, semantic information also enables the cooperation between people and computers. Data items present in computer systems are not abstract for human, as *row from a data base table* or *file*, but are named with a

taxonomy comprises concepts coming from the real world surrounding people in their everyday life.

Many technologies has been developed in order to meet the Semantic Web challenge [46]. Most of them are regulated and recommended by *The World Wide Web Consortium* (W3C) [45].

XML (*eXtensible Markup Language*) [31] enables arbitraly structural organization of documents, however, without defining their meaning. XML data model is specified by tree-based *XML Schema* [18, 17]. Basically, XML serves as serialization language for languages being at higher level of abstraction, as RDF.

RDF (*Resource Description Framework*) [48] is dedicated for description of Web resources metadata in a form of statements about them. Statement is represented as RDF triple comprises two Web resources: the subject and the predicate, uniquely identified by URI (*Universal Resource Identifier*), and the predicate specifying relation between them. In such approach, particular Web resources have properties with certain values. RDF data model is specified by graph-based *RDF schema* (RDFS) [47].

Applying of taxonomy to RDF resources provides a commonly agreed understanding of RDF assertions. OWL (*Ontology Web Language*) [19, 20, 21], built upon RDFS, models the Web resources description vocabulary trough *ontologies*. Ontology may be perceived as *specification of a conceptualization* [1], it serves as Semantic Web inter-lingua.

There exist some similarities between OWL ontology model and typical object model. Essential OWL features are depicted below:

- *Class* – a general concept, abstract set of individuals which share the same properties, but can differ in these properties values.
- *Individual* – a concrete instance of a given class.
- *Sub-class* relation – introduces an element of taxonomy, indicates that one class derives all properties from another one, namely, one class is a more specific kind of another class.
- *Object Property* – defines relationship between classes, the individuals may be associated with each other by concrete object propety values. Each object property has particular *range* class and *domain* class.
- *Datatype Property* – defines some data attributes of a class. Each datatype property has particular *domain* class and *range* data type.
- *Sub-property* – introduces relationship between properties, indicates that one property is a more specific kind of another property.

A simple example of OWL specification of a virtual experiment concept is presented

below. The class is modeled through RDF resource. Typically, all names within a concrete ontology are augmented with its name space:

```
<owl:Class rdf:about="http://www.virolab.org/onto/exp-protos/Experiment">
```

Class properties are defined as RDF triples:

```
<owl:DatatypeProperty
rdf:about="http://www.virolab.org/onto/exp-protos/name">
  <rdfs:domain
   rdf:resource="http://www.virolab.org/onto/exp-protos/Experiment"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

The next piece of code presents how to introduce a new concept – *ExecutionStage*, which is an abstract part of the experiment, and define, through object property, new relationship:

```
<owl:Class
rdf:about="http://www.virolab.org/onto/exp-protos/ExecutionStage"/>

<owl:ObjectProperty
rdf:about="http://www.virolab.org/onto/exp-protos/executedIn">
  <rdfs:range
   rdf:resource="http://www.virolab.org/onto/exp-protos/Experiment"/>
  <rdfs:domain
   rdf:resource="http://www.virolab.org/onto/exp-protos/ExecutionStage"/>
</owl:ObjectProperty>
```

The relation of generalization is introduced in a ditinct RDF triple:

```
<owl:Class
rdf:about="http://www.virolab.org/onto/exp-protos/Computation">
  <rdfs:subClassOf>
    <owl:Class
     rdf:about="http://www.virolab.org/onto/exp-protos/ExecutionStage"/>
  </rdfs:subClassOf>
</owl:Class>
```

Besides meta-data, ontology may also include conrete data item, which is called ontology *individual*. This is equivalent to classes and its objects in the object model. Individuals are uniquely identified by URIs:

```
<exp-ns:Experiment rdf:ID="http://www.virolab.org/onto/exp-protos/Exp1">
  <exp-ns:name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    geno2drs
  </exp-ns:name>
</ext-ns: Experiment >
```

It is a good practice to put some restraints on the allowed OWL constructions, similar to contraints in the data bases or object models, what forces the ontology author to create individuals making more sense, and therefore more accurately representing the modeled domain. A sample restraint is construction which states that a concrete property must be functional:

```
<owl:ObjectProperty
rdf:about="http://www.virolab.org/onto/exp-protos/executedIn">
  <rdf:type
   rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</owl:ObjectProperty>
```

Another OWL contraints as well as another, more advanced, OWL features applied in the designed ontologies are described in the next chapters.

W3C distinguishes three dialects of OWL language:

- *OWL Lite* has the lowest formal complexity and supports only generalization hierarchy with the simplest constraints.
- *OWL DL* is more expressive, it enables almost all language constructs, however, the reasoning over OWL DL is more complex.
- *OWL Full* is the most expressive dialect, in which all OWL features are supported, but the conclusions based on ontologies are not guaranteed to be computable.

Due to performance reasons, OWL Lite is used, cause both the Semantic Web frameworks, as *Jena* [34], as well as the reasoning software, as *Pellet* [35], are still at a very immature state of development.

There is no commonly used standard of ontology visualization. Therefore, a representation similar to the UML model is used, extended with circular coloured shapes indicating the ontology affiliation, as in Fig. 4.3.

Ontologies support reasoning, which is based on *sub-class* and *sub-property* relations and reflects basic rules of deduction realized by human brain in the perception of surrounding concepts. In the preceding example, reasoning software is able to presume that a *Computation* is an *ExecutionStage*. On the basis of this deduction, some more sophisticated reasoning may be applied. Ontological information together with reasoning rules may be perceived as *knowledge*.

2.2. Scientific workflows

Scientific experiments are often very complicated – they include various steps of analysis and are based on huge amounts of data. In order to support the scientists, there were introduced a workflow – a virtualization of experiment, which cover all steps

of data analysis. The steps representing processes and computations are linked according to the data flow and dependencies among them [9]. The workflow management systems must provide automatic execution, covenient management and provenance recording. Morever, it must enable high-performance computing and large-ditributed databases. These requirements are met in Grid technologies [3]:

- Thanks to the collaboration between different institutions in virtual organizations (VO), the resources used in workflows may be shared and reused. Grid systems include mechanisms providing management and usage of well-ditributed computational units, data storages, network resources and code repositories.
- Workflows optimization is supported by software discovery services, which select the best sotfware implementation and execution platform on the basis of some particular workflow task parameters.
- Data access realized by workflows is efficient thanks to the data replication services, what refers to many performance metrics such as response time, reliability and cost.
- Monitoring services control resources availability and deliver events about workflow enactment status. Monitoring events may reflect the experiment course, informing about Web Services invocations as well as queries adressing data storages and transformations applied to the fetched data sets.

The reconciliation of grid computing, which provides *computation scalability* with Semantic Web, providing high level *data scalability*, constitutes the idea of *Semantic Grid*, essential part of *e-Science*.

2.3. Provenance

Provenance may be defined as *metadata that pertains to the derivation history of a data product starting from its original sources* [6].

As provenance refers to many aspects of metadata (*the seven W's*: *Who, What, Where, Why, When, Which, hoW*), there is a need for taxonomy defining provenance scope. Basically, four types of provenance may be distinguished [7]:

- **Process provenance** Unambiguous workflow execution trac specyfing what services were invoked respecting their orchestration.
- **Data provenance** Graph of data items, describing how concere data objects depend on other data sets, from the input parameters and partial results to final workflow results.
- **Contextual provenance** Context of enactment. Information of who is the workflow executor, in what project the experiment is developed and what is the hypothesis being validated.
- **Organization provenance** Circumstances in which the contextual information were created and how they evolved.

This complete data lineage serves in many fields of data usage:

- Incorporation of provenance into Semantic Web, constituting the idea of *provenance web* [7], significantly extends the provenance information interoperability. That enables a meaningful collaboration within the research community. Publicated scientific results are possible to be validated through the reproducing of conditions, in which a concrete piece of data was obtained, in similar environment.
- Scientists are able to reuse the results obtained by others and draw their own hypothesis as well as utilize the results in their own experiments. In this cases, credibility of data is crucial. It is especially important due to the scale of obtained results quantity. A researcher is interested only in data stored in trusted repositories and published by trusted people, being sure of its reliability and quality.
- Provenance may also serve as a regulatory mechanism of sensitive data protection. For example, having a convenient insight into how medical data is used in experiments, by who and in what services, data managers from clinics and hospitals might be more willing to share their data with another virtual laboratory participants.

2.4. ViroLab virtual laboratory

ViroLab virtual laboratory (VLvl) [23, 2, 24] constitutes an idea of unifying the medical scientific community, computer science community and healthcare professionals in the activity on the field of infectious diseases treatment. Its goal is to integrate data bases storing information about patients from European hospitals, provide a modern environment for development and execution of medical experiments within a grid infrastructure and expose decision support systems that would be valuable in treatment process.

From the practical point of view, three main classes of VLvl users may be identified:

- **Experiment Developers** People who write the experiments using the development environments offered by VLvl. They are expected to posses strong programming skills as well as basic medical knowledge sufficient to understand the meaning of virological data and the requirements of clinic researchers.
- **Scientists** People doing some research in the field of infectious diseases. They execute the experiments prepared by the developers. Their responsibility is to design scenarios of how to execute concrete experiments as well as to decide what data the experiments should be parameterized with and how to use their results in other experiments. Additionally they prepare, in cooperation with developers, prototypes of new experiments.
- **Clinical Virologists** People who search through all data available within the VLvl, including the experiment results. They apply some data mining to the

integrated data in order to support their decisions during a treatment process. From their point of view, the most valuable aspects of VLvl are the integration of the information from many hospital data bases into a unified schema accessible in a single point and mixing of the original data with the experiment results.

It seems clear that the Experiment Developers will be interested mostly in programming tools, while the Scientists in experiments enactment environments and the Clinical Virologist in data and results exploration tools. It also has to be emphasized, that, since all their activities are somehow related to data, all of them will be somehow involved in data provenance. It is also strongly believed, that the ontologies are a great mean to serve as inter-lingua between these three kinds of specialist, because a high level of abstraction provide the understandability by all of them.

Main layers of VLvl grid infrastructure are conceptualized in Fig. 2.1.

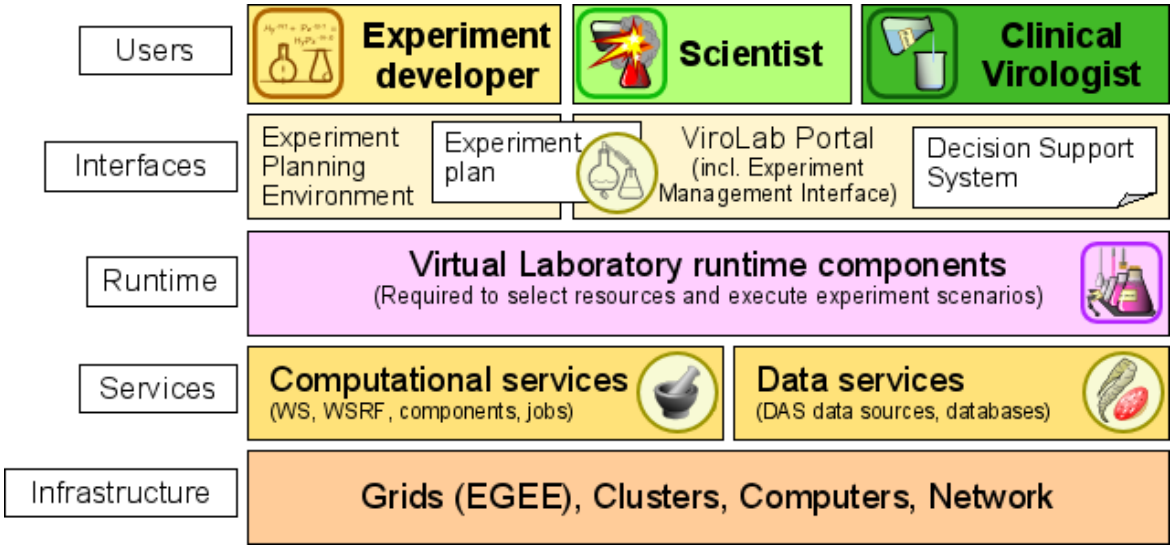


Figure 2.1: VLvl layers. Laboratory enables the cooperation between three kinds of users. Figure source: `http://virolab.cyfronet.pl`.

Overall system architecture is presented in Fig. 2.2.

Vlvl components listed below are directly related with provenane. Please note, that understanding of the purpose of these components and of how they function within VLvl is crucial for the following studies.

- ***Grid Resources Registry*** **(GRR)** A registry storing information about services accessible from experiments. All computational units are virtualized as so-called ***Grid Object*** **(GO)** and must be registered in GRR before farther usage. These resources are classified in a triple hierarchy. A Grid Object is defined only by the methods specification. It has one or more ***Grid Object Implementations***

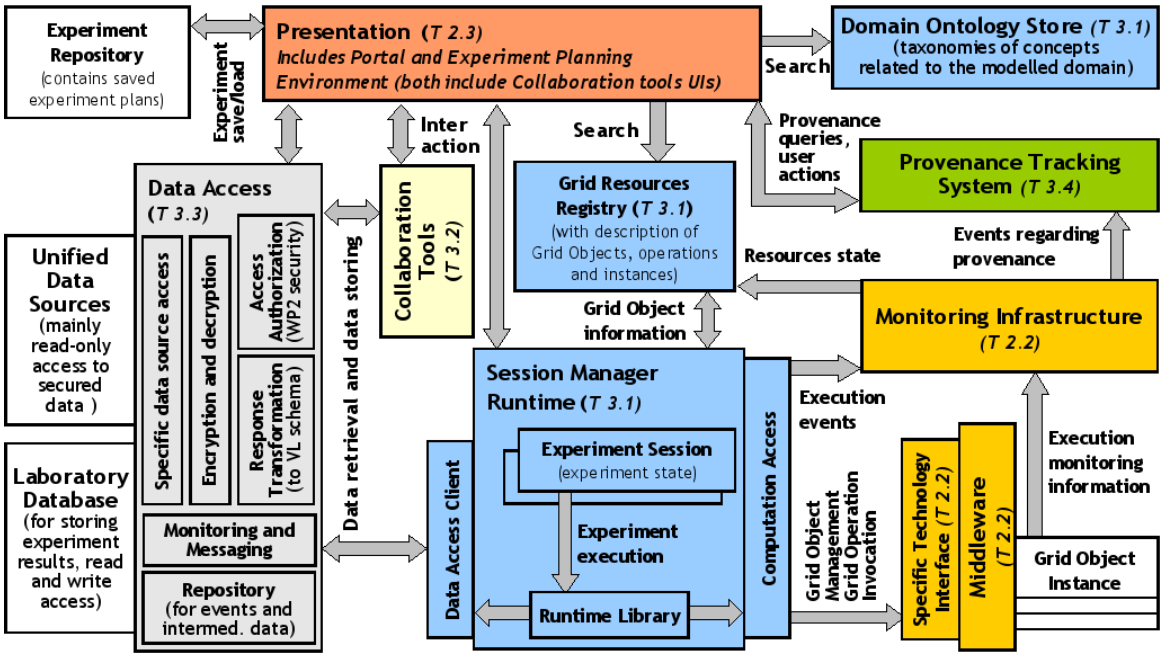


Figure 2.2: VLvl grid architecture. Provenance tracking is realized by a separate component. Figure source: `http://virolab.cyfronet.pl`.

(GOI), which differ in the implementating technology. GRR supports the wrapping of Web Services, MOCCA components, WSRFs and local jobs. A particular implementation has one or more ***Grid Object Instances***, which are concrete, deployed services with a known and accessible endpoint.

- ***Experiment Planning Environment*** **(EPE)** A development environment based on *Eclipse*. The experiments are developed in script *JRuby* language [40]. By the usage of EPE platform, it is possible to write scripts, synchronize them with SVN repository, develop new services and deploy the services as Grid Objects. The functionality of EPE can by easily extended by writing new plugins, dedicated, for example, for ontology browsing.
- ***Experiment Management Interface*** **(EMI)** An environment devoted for the experiments management, versioning and execution. It also display the experiment results and gain feedback from the user. EMI is integrated in a web portal.
- ***Grid Application Optimizer*** **(GrAppO)** A component responsible for the most optimal selection of concrete Grid Object Implementation and Grid Object Instance, on the basis of some historial and current performance data, in order to provide the highest possible experiment evaluation efficiency. It is especially important when dealing with medical services, which offer long term computations.
- ***Grid Space Engine*** **(GSEngine)** An enactment engine executing the experi-

ment scripts. In can be installed on a local machine, or, optionally, experiments may be executed remotely on a server. The main important, from the provenance point of view, GSEngine parts, are Runtime component which directly interprets the script and Invoker which executes the grid operations on Grid Object Instances selected by GrAppO.

- ***Data Access Client*** **(DAC)** A VLvl client for the more generic Data Access Service (DAS). It integrates all the data bases accessible within the VLvl in a single, accessible point. DAC is independent from the underlying technologies and supports basic SQL constructions. One of the most important VLvl challange is the migration of all clinical data bases into the unified schema.
- ***Provenance Tracking System*** **(PROToS)** A provenance XML data base. It stores ontological information in an optimized, distributed way and provide a number of algorithms guarantying a high performance of ontological queries processing. It may be perceived as an event-driven component, because the pieces of information are delivered to PROToS in a form of events passed by Web Services.
- ***Query Translation Tool*** **(QUaTRO)** A provenance access component. It exposes a graphical interface for the construction of queries accessing both PROToS and DAC components, providing the ability of mining both over provenance information and data. Some aspects of the QUaTRO implementation are explained in details in chapter 8.

Chapter 3

Concept of system for provenance recording

In this chapter, provenance infrastructure is conceptualized on high abstraction level. Overall requirements addressing provenance recording and querying are specified. There is also presented solution overview.

3.1. Overall requirements

Provenance system architecture is conceptualized in Fig. 3.1. Components directly included in the scope of this thesis are marked with yellow color.

Implementation of the provenance system includes following activities:

- Design information model for provenance,
- Design data model for monitoring system,
- Adapt existing monitoring infrastructures to the provenance requirements,
- Define ontology creation process,
- Design and implement component realizing the process,
- Incorporate the component into system grid infrastructure,
- Design and implement provenance access component.

Both the information model as well as the data model are expected to:

- Be understandable by the human user, especially by a non-IT specialist,

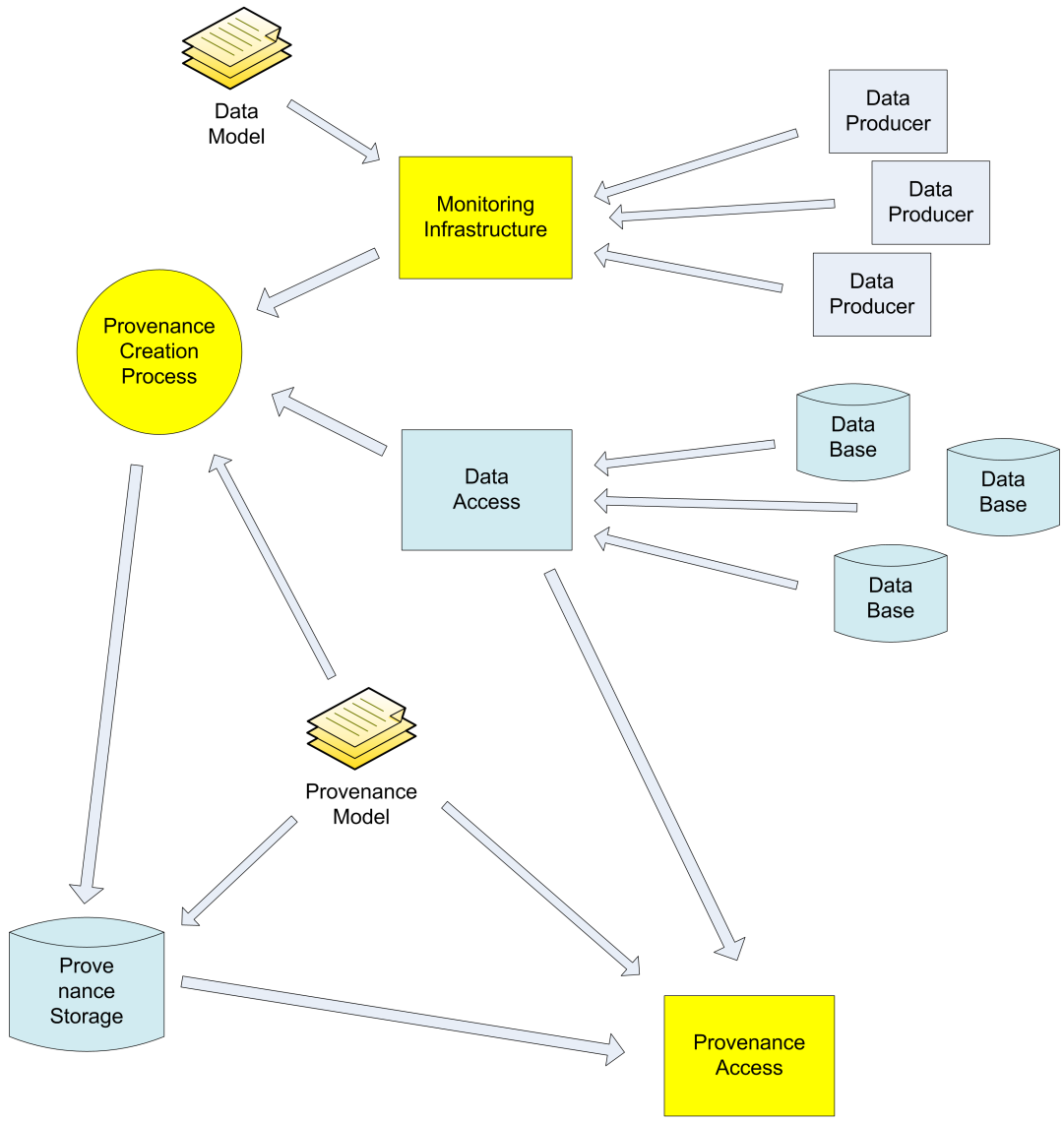


Figure 3.1: Abstract provenance architecture. Provenance information model influences the creation process, defines the semantics of stored provenance data and regulates the expresiveness of provenance queries.

- Captures possibly many details of modelling domain,
- Introduces series of contraints in order to provide data integrity,
- Avoids data redundancy,
- Uses expressive modeling relationships like generalization, association and aggregation,
- Be easy to evaluate in a sense of fast storing and querying.

It is justified, that the ontology creation process should meet following requirements:

- Be provenance-model-independent, i.e. well adaptive to the evaluating provenance ontologies,
- Be data-model-independent, i.e. well adaptive to the evaluating monitoring data model,
- Be reconfigurable,
- Be augmentable trough accessing additional information from distributed data bases,
- Be easy to integrate with end-user's pieces of software, regardless of implementation technology,
- Provides integrity and coherency of created information,
- Presents high level of performance,
- Supports different levels of data granularity,
- Supports different levels of information semantics.

The expectations addressing the provenance accessing component are as follows:

- Be convenient to use by non-IT specialists,
- Be extendable, through providing the ability to increase the provenance queries expressiveness,
- Be independent from the underlying querying technologies,
- Provides optimizations of query evaluation,
- Provides the transparency of data sources distribution,
- Enables accessing both provenance storage and data bases, as well as another data repositories,
- Be easy to integrate in Web portal.

3.2. From requirements management to provenance mining

There were identified all activities in ViroLab, both the ones existing earlier and the ones newly intruduced, that are somehow related to the undertaken problem. The 13 abstract steps constitute the whole process leading from monitoring data to knowledge mining. Some of these steps, not yet mentioned, are explained in details in the next chapters. All of them, integrated with each other, provide the solution for the problem presented in chapter 1. The steps are presented in Fig. 3.2

1. **Requirements Management** Some information from the end-users, the clinicians and virologists, is collected, in order the get to know their exeptations addressing the provenance querying. On the basis of the outcomes, after the feasibility study, some requirements for QUaTRO are specified.

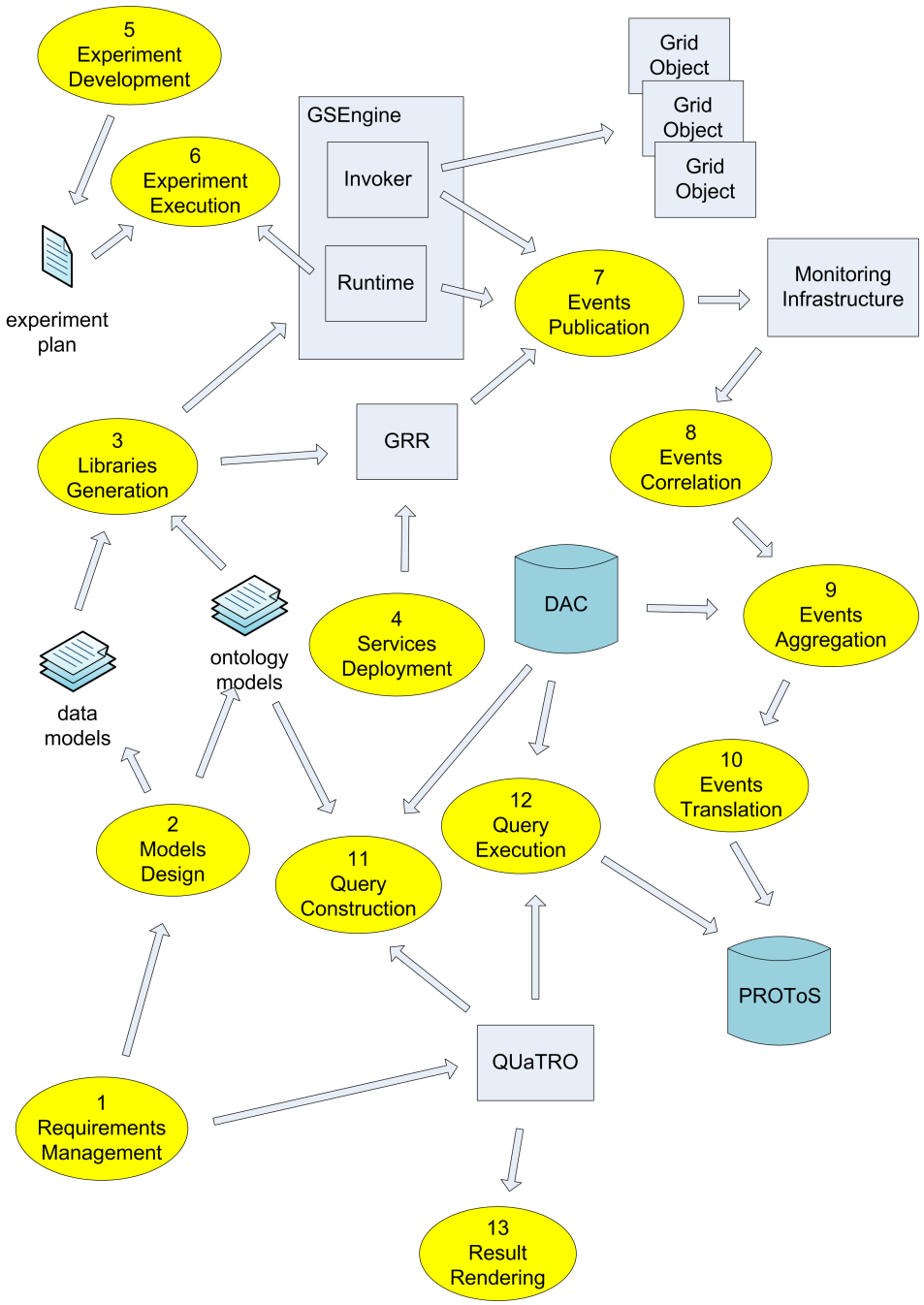


Figure 3.2: 13 abstract steps leading from provenance requirements management to provenance mining. Most of them are included in this thesis scope.

2. **Models Design** A proper XSD model for monitoring system and OWL models for data, experiment and medical scenarios are designed.
3. **Libraries Generation** On the basis of the XSD model, there are generated, either automatically or *by hand*, libraries enabling convenient interaction with monitoring infrastructure.

4. **Services Deployment** Services used in experiments are developed, deployed and registered as Grid Objects. During this, GRR publishes monitoring events referring to resources availability.
5. **Experiment Development** Experiments supporting concrete scenarios are implemented. The author of the script may publish additional monitoring events in order to augment the provenance semantics.
6. **Experiment Execution** The experiment is executed within the VLvl enactment engine.
7. **Events Publication** During the experiment execution, some monitoring events are published by Runtime, Invoker and, possibly, another components.
8. **Events Correlation** All the events referring to the same activities, components or types of data are correlated.
9. **Events Aggregation** The correlated events are aggregated, that means, basing on the low-level monitoring events containing pure data, they are generated high-level, semantically valuable ontological events.
10. **Events Translation** The aggregated ontological events are translated into PROToS-specific events that can be delivered directly by its Web Services.
11. **Query Construction** A query addressing PROToS as well the relational data bases behind the DAC is constructed with the support of Graphical User Interface.
12. **Query Execution** The constructed query is effectively executed by a sophisticated processing engine.
13. **Result Rendering** The obtained results are rendered in a way understandable by the user.

Chapter 4

Provenance model

In this chapter models for monitoring and provenance are presented. XSD schema of monitoring events is explained, including generic events comprises pure XML data and domain events describing OWL individuals. Also the requirements addressing provenance ontology are specified. Finally, ontology model meeting these requirements is visualized and commented.

4.1. Monitoring data model

4.1.1. Generic monitoring events

Several events that may occur in VLvl were identified. Events describing the experiment course and the appearance of new available services were incorporated into the monitoring data model:

- *ApplicationStarted*, *ApplicationFinished* refer to experiment enactment
- *GridOperationInvoking*, *GridOperationInvoked* refer to computations
- *DataAccessQuerying*, *DataAccessQuered* refer to Data Access calls
- *GridObjectRegistered*, *GridObjectImplementationRegistered*, *GridObjectInstanceRegistered* refer to resources availability

All events are enclosed in abstract event of type *MonitoringData*. This type is associated with comprised event type and resource in which a concrete event was generated:

```
<xsd:complexType name="MonitoringData">
  <xsd:sequence>
    <xsd:element name="applicationStarted" type="ApplicationStarted"
     minOccurs="1" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="dataTypeID" type="xsd:NMTOKEN"
   fixed="events.application-started"/>
  <xsd:attribute name="resourceID" type="xsd:string"/>
</xsd:complexType>
```

Monitoring schema includes also *Application Correlation Identifier* (ACID), described in details in section 5.2. XSD conceptual model is presented in Fig. 4.1.

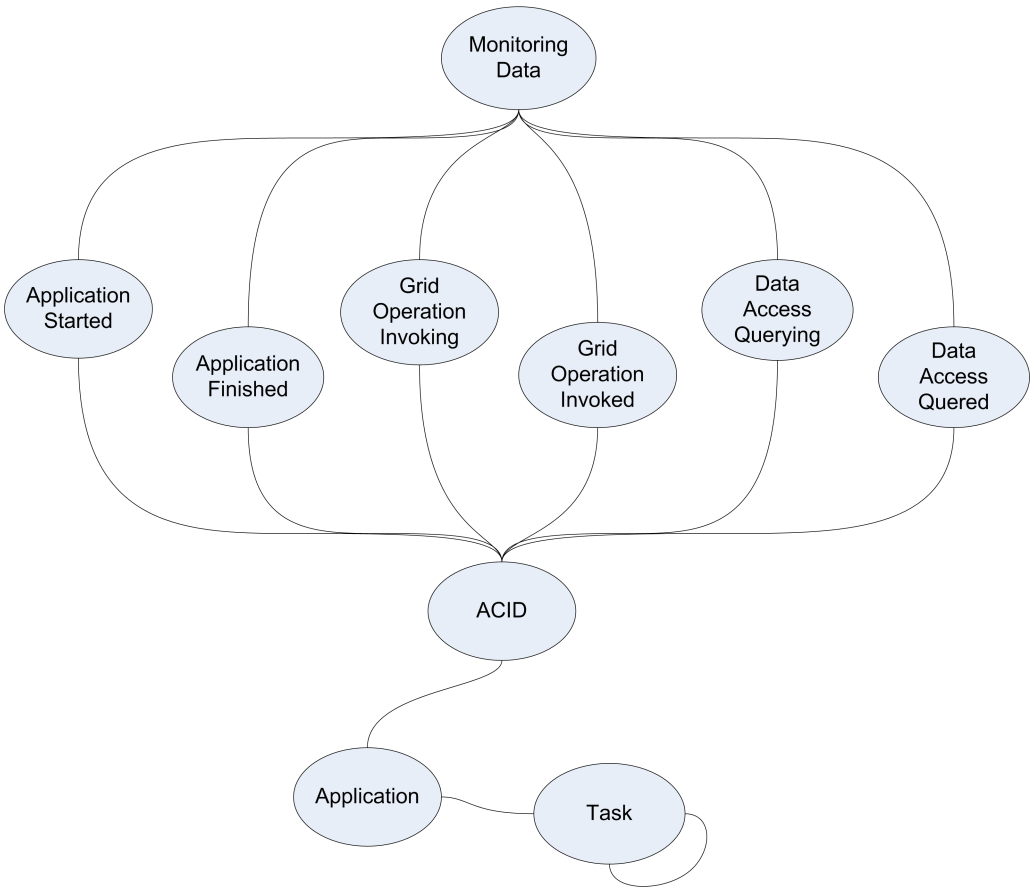


Figure 4.1: Monitoring XSD schema. The events reflect basic experiment activities. ACID identifier enables the correlation of events sharing application context.

Also the computational resources: Grid Objects, Grid Object Implementations and Grid Object Instances are represented on the XSD-schema level, as in Fig. 4.2.

As for the events describing the beginning and the end moments of some activities, to avoid the data redundancy, most of the data entities is collected in the beginning

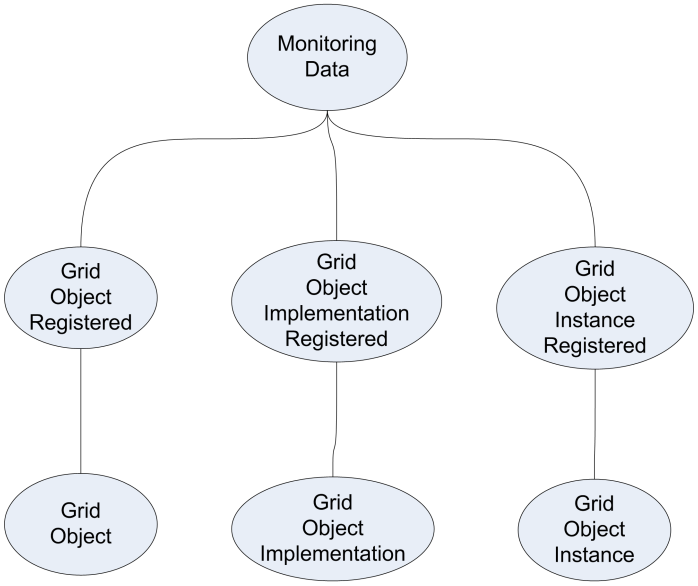


Figure 4.2: XSD schema for events informing about resources availability. Resources removal is not included because of the provenance information immutability.

event, while the finish event stores only its time and the ACID needed to correlation, as in following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<MonitoringData dataTypeID="events.application-started">
  <applicationStarted
  time="1216142825504"
  executedBy="JohnDoe"
  sourceFile="repo1/geno2drs"
  version="4.1"
  name="geno2drs">
    <acid>
      <application id="app1"/>
    </acid>
  </applicationStarted>
</MonitoringData>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<MonitoringData dataTypeID="events.application-finished">
  <applicationFinished
  time="1216142855505">
  <acid>
    <application id="app2"/>
  </acid>
  </applicationFinished>
</MonitoringData>
```

4.1.2. Domain monitoring events

Besides the generic events, the model includes also the events related to a concrete medical domain. The main difference between the generic event and the domain ones is that the domain event directly reflects the domain ontology and therefore it can be mapped almost 1:1 to ontology individual. This approach is motivated with the fact that the domain ontology exactly specified the type of event that may be published in monitoring system.

There does not exist a method of transferring the OWL individuals. In fact, the individuals usually reside frozen in an immutable ontology storage. This is typical for OWL model structure, in which individuals may be recorded only in the context describing their domain ontology. However, there is a need for a method of transfering only small pieces of ontology represented by single individuals. Therefore, the individual specification is enclosed in the existing generic events model. All properties should be specified explicitly in the MonitoringData object. Thanks to this, the domain individual description may be augmented with its time and the ACID number, what is justified for several reasons, described in the following chapters. There were introduced dedicated tags, *class* and *property* containing the ontological class URI as well as the properties URIs with the associated values.

Sample domain event structure is presented below. The event reflects domain ontology class that models invocation of a particular medical service. Functional properties are recorded explicitly. Object properties, which refer to data ontology individuals describing data sets that were used or obtained, are recorded implicitly. That means, instead of the individual identifiers, the identifiers specifying localization of reference objects in Data Access are recorded. Thanks to this, the reponsibility of data individuals instantiation is shifted from the component which publishes monitoring event to the component aggregating this event.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <MonitoringData dataTypeID="events.domain.newDrugRanking">
3    <NewDrugRanking time="1216165279568">
4      <acid>
5        <application id="appID"/>
6      </acid>
7      <class name="http://www.virolab.org/onto/drs-protos/NewDrugRanking"/>
8      <property name="http://www.virolab.org/onto/drs-protos/resultRanking"
9       value="324623"/>
10     <property
11     name="http://www.virolab.org/onto/drs-protos/testedMutation"
```

```
12      value="138421" />
13      <property name="http://www.virolab.org/onto/drs-protos/region"
14      value="RT" />
15      <property name="http://www.virolab.org/onto/drs-protos/usedRuleSet"
16      value="ANRS" />
17    </NewDrugRanking>
18 </MonitoringData>
```

4.2. Provenance ontology

The experiment provenance ontology, visualized in Fig. 4.3, was designed in order to meet the exceptations of all components and users and cover several aspects of VLvl, with respect to requirements desribed in section 1.2.

- **Experiment re-execution** The experiment is virtualized as a sequence of abstract execution stages. Currently, two kinds of stages are realized in experiments: invocation of a grid operation of a particular grid objects or a query addressing the DAC. When the experiment is being executed, its next stages are continously recorded. Each stage is situated in a concrete moment of time – thanks to this, they can be ordered and there appears a complete, unequivocal *experiment trace*. Thanks to that trace, it is possible to execute the experiment once again, the whole experiment, in a case it has failed or only some parts of it in more complicated use cases.
- **Data dependencies** Each execution stage is associated with its input and output data. The input data is usually read from DAC or created in one of the earlier stages. The output data is a newly created piece of information. After the association of output data from some stages with input data of another stages it is possible to determine a complete *provenance graph*, expressing exactly how a concrete piece of data was obtained, from what information, in which operations, by usage of what resources, when, by who, in context on what experiment. It is a complete provenance information, which is called a *fine-grained provenance*.
- **Results management** Besides the fine-grained provenance, also a so-called *coarse-grained provenance* is recorded. It does not refer to the results of concrete stages, but rather to the results of complete experiments. An experiment result is, unlike a computation result, a more complicated entity, saved in one of the separate data storages, such as as *WebDAV* [37], and augmented with more detailed technical and security-related information.
- **Performance** The ontology includes historical performance information. For each computation, some technical data is recorded, such the duration, the processor usage and the memory usage. This information may be used in the algorithms offered

by GrAppO for the selection of the most efficient Grid Object Implementation and Grid Object Instance.

- **Resources availability** The ontology directly reflects the triple computational resources hierarchy, including Grid Object, Grid Object Implementation and Grid Object Instance. GRR is responsible for providing the current information about newly registered Grid Objects and Grid Object Implementations, as well as newly deployed Grid Object Instances.

Principles of ontology experiment building are described in chapter 5. Experiment ontology is linked with a series of *domain ontologies*, which describe the contexts of particular medical workflows. There is an assumption, that there exists exactly one domain ontology per one medical scenario. The ontology describes the semantics of computations realized in this scenario. Domain ontologies are linked with a *data ontology*, which describes data sets analyzed or produced by medical services.

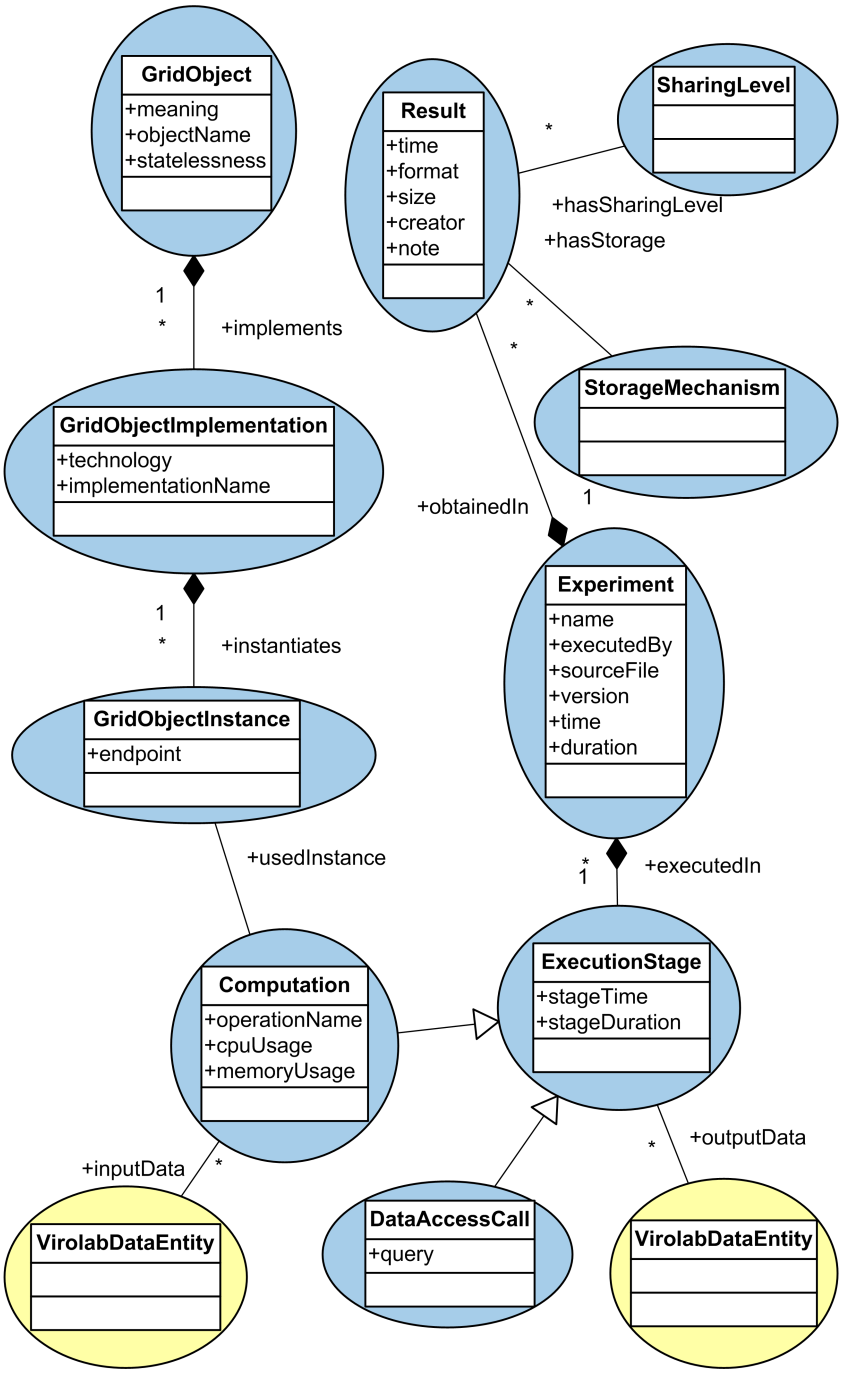


Figure 4.3: Experiment ontology reflects 5 aspects of provenance in ViroLab: experiment trace, results metadata, data dependencies, performance optimization and services availability.

Chapter 5

Semantic Event Aggregator

Detailed description of provenance information building is given in this chapter. The purpose of component Semantic Event Aggregator is presented. It is explained, why correlation of monitoring events is important. Idea of ontology extension, which annotates the experiment ontology, is introduced, together with its main elements – concepts describing the derivation of information, delegates incorporating separate pieces of code and aggregation rules. Next, experiment transaction is defined. Also the principles of associating of created individuals are depicted.

5.1. Main idea behind Semantic Event Aggregator

As described in the preceding chapters, in VLvl exist: PROToS reponsible for the ontology storage, the infrastructure responsible for the generation and transferring of the monitoring data, the ontologies describing metadata and the data producers. There is high need for the one missing component – the one responsible for the building of ontologies from the monitoring data. This component should satisfy the following requirements:

- **Ontology Independence** Because ontology is a model of a specific real-world domain, it is expected to be constantly extended and modified, so that it would reflect the modeled domain in a better way. At the mature state of an ontology life cycle it is expected to contain more detailed information and more associations

addressing another related ontologies. After the refinement of an ontology, the new component should be able to create the new information details.

- **XSD Schema Independence** The continuously changes applied to laboratory components make them able to provide more specific technical information. So the XSD schema may be also extended, similar like the ontologies. After the refinement of the XML events model, the new component should be able to make use of the additional available data.
- **Configurable** The information building process should be extendible and easily reconfigurable. As far it is possible, the building manners should be described in a semantically valuable way.

The proposed component is Semantic Event Aggregator, serving as a mediator between monitoring infrastructure and the ontology storage, as in Fig. 5.1

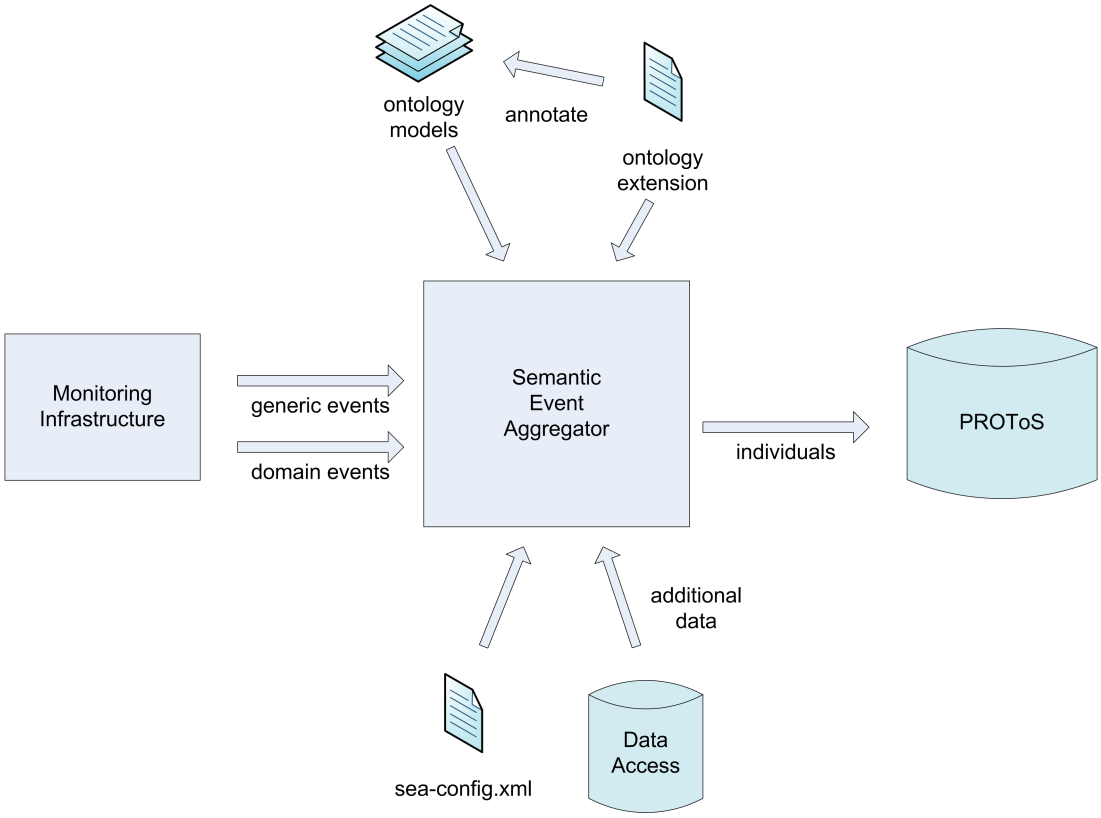


Figure 5.1: Semantic Event Aggregator builds the provenance information on the basis of monitoring events. It is configurable by ontology describing the semantics of aggregation principles. External data sources may be queries in order to augment information expressiveness.

5.2. Monitoring events correlation

The monitoring events hierarchy, with respect to the origin granularity, comprises:

- *upper-level* events that correspond to concrete application
- *middle-level* events that correspond to concrete stage of a particular application
- *low-level* events that correspond to concrete sub-stage of a particular stage within a particular application

A *stage* is an abstract concept. In general, all experiments may be decomposed into many stages, in most cases Grid Object Calls and Data Access Queries. However, the stages may be also constituted by some computations defined explicite in the scripts, for example concrete regions of experiment code.

A *sub-stage* is also an abstract concept. The current event model does not include low-level events, however, the extension of granularity is possible to implement. For example, a Grid Object Call might be decomposed into computations executed on another, *transparently called* grid objects (what constitutes a typical workflow approach) or regions of code implementing the Web Service logic layer.

Tab. 5.1 presents a sample VLvl experiment execution context:

Application	*App1*										*App2*									
Stage	1 GO Call			2 DA Query					3 GO Call		1 GO Call	2 DA Query			3 DA Query					
Region	1	2	3	1	2	3	4	5	1	2	1	1	2	3	1	2	3	4	5	6

Table 5.1: Sample experiment execution context.

In this example two applications, *App1* and *App2* are executed in parallel. Both of them have three stages, executed sequentially. As in this example, a monitoring event origin may be localized as point in a multidimensional, hierarchical space (one of the rectangular areas within the table). What is more, the monitoring events come from different VLvl components and occurred in different moments of time.

Because of the described issues, there is a need to provide the *correlation* of the monitoring data. This correlation must be organized in a hierarchical way to enable the association of different pieces of monitoring data at their different origin levels. This is provided with Application Correlation Identifier (ACID). In order to provide a convenient extension of XML monitoring events, ACID is also organized as an XML tag:

```
<xsd:complexType name="ACID">
  <xsd:sequence>
    <xsd:element name="application" type="Application" minOccurs="1"
    maxOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>
```

At the top level of the ACID structure, there exists *Application* tag, identified with a unique string value:

```
<xsd:complexType name="Application">
  <xsd:sequence>
    <xsd:element name="task" type="Task" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>
```

In this approach ACID is designed in a generic way. It enables the decomposition of some experiment stages into more granular sub-stages continuously, introducing as many granularity levels as needed. Sub-stages are defined in a recursive way:

```
<xsd:complexType name="Task">
  <xsd:sequence>
    <xsd:element name="subtask" type="Task" minOccurs="0" maxOccurs="1"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>
```

That enables farther decomposition, as in the Fig. 5.2

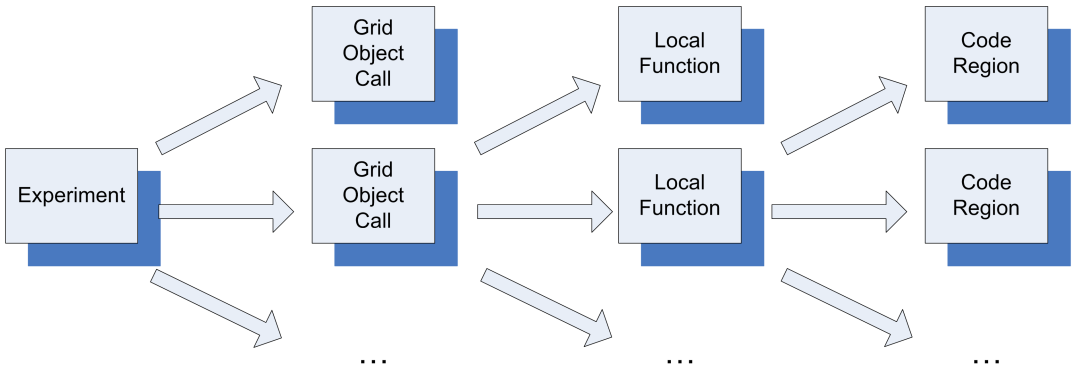


Figure 5.2: Experiment decomposition. Monitoring event may be published in the context of experiment stage, local invocation or code region.

All event producers are responsible for the augmentation of the created events with ACID. It can be easily done by incorporation of ACID tag into event XML structure, as in following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<MonitoringData dataTypeID="events.grid-operation invoking">
  <gridOperationInvoking
  time="1215338465487"
  name="drs"
  endpoint="http://virolab.cyfronet.pl:8080">
  <acid>
    <application id="app1">
      <task id="tsk1"/>
    </application>
  </acid>
  </gridOperationInvoking>
</MonitoringData>
```

All components participating in experiment execution generate their own parts of ACID on a proper granularity level. GSEngine generates the application identifier while Invoker generates unique identifiers for all Grid Object Calls. In such an approach all workflow components are aware of the ACID temporal structure and each component is responsible for the passing of already augmented ACID to sub-components, as in Fig. 5.3

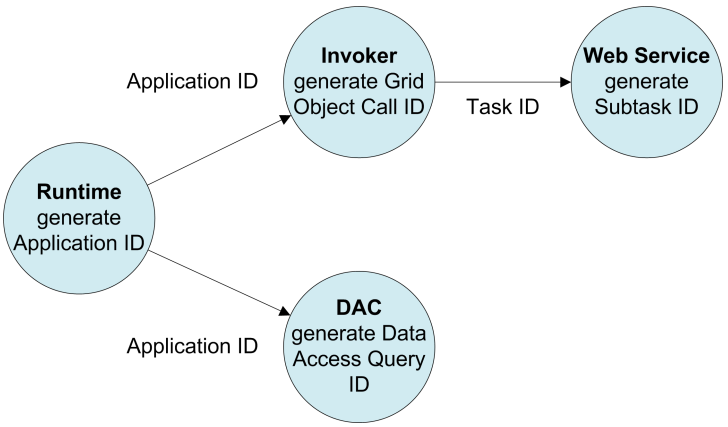


Figure 5.3: Passing of ACID between VLvl components. At each execution level, ACID is augmented with newly generated identifier.

The technical problem of how to pass ACID parts to the Web Service context (see arrow with *Task ID* in the above figure) remains unsolved , however, it is necessary to enable the correlation of events that would be published directly in Web Services.

The described hierarchical ACID structure additional benefit is that it, assuming appropriate monitoring infrastructure support, enables sophisticated, structural events subscriptions, as in following use cases:

- *subscribe for all upper-level events*
- *subscribe for all events concerning a concrete application*
- *subscribe for all upper-events concerning a concrete application*
- *subscribe for all middle-level events*
- *subscribe for all middle-level events concerning a concrete application*
- *subscribe for all middle-level events concerning a concrete application and a concrete Grid Object Call context*
- *subscribe for all middle-level events concerning a concrete Grid Object Call context*

5.3. Ontology Extension

The significant problem to be solved is how to transform the collected and correlated raw XML data into ontological information. There should be provided a well-defined and convenient mapping between XML data and OWL data. What is more, also the data describing the mapping principles should be represented and stored. Three approaches to that problem were considered:

- Enclose mapping information in ontology
- Enclose mapping information on XML data
- Enclose mapping information in a distinct representation

The first solution was choosen to be applied, so that the mapping principles would have a well-defined semantic and remains understandable by a human being. In fact, in such approach, the Aggregator is configurable by an ontology.

Therefore, an ontology extension was defined – a dedicated ontology which describes how to build another ontology from the correlated XML data. There is an assumption that there exists exactly one ontology extension per one ontology built be the Aggregator. Moreover, the extension must not influence the ontology itself.

The extension is designed to consist of three kinds of content – *aggregation rules*, *derivations concepts* and *semantic annotations*.

5.3.1. Derivation Concepts

It is of high importance then, when dealing with a well-defined semantic, all kinds of information should be classified in an ontological concepts hierarchy. It also refers to the information describing how to create the ontology. At a high level of abstraction, *derivation concept* is proposed. This is an individual describing how a concrete onto-

logical property *derives* from collected XML data – namely, it describes the *derivation* of an ontological property. This part of ontology extension is presented in the Fig. 5.4

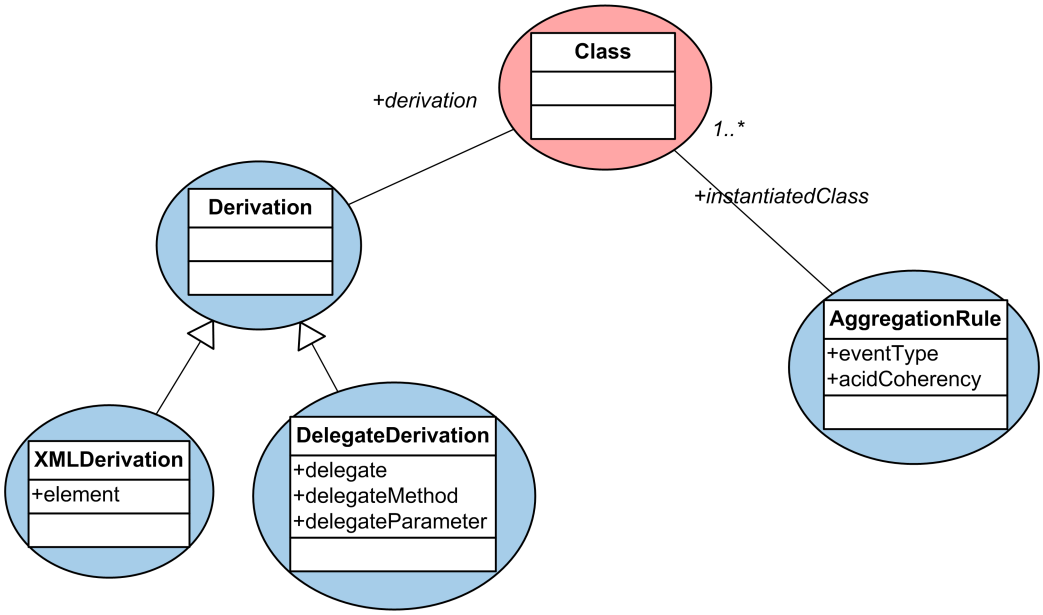


Figure 5.4: Ontology extension is a separate ontology annotating the ontology being built. It comprises derivation objects, which specify how values of properties should be established and aggregaton rules describing the principles of monitoring events processing.

In a trivial case, an XML element is mapped directly to a functional property. A concept describing such a mapping was called an *XMLDerivation*. This concept has only one functional property called *element*:

```
<owl:Class rdf:ID="XMLDerivation"/>
  <rdfs:subClassOf rdf:resource="#Derivation"/>
</owl:Class>
```

```
<owl:DatatypeProperty rdf:ID="element">
  <rdfs:domain rdf:resource="#XMLDerivation"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

The element property defines how a concrete value is placed within the XML document, what is recorded in XPath [22] format.

All the derivation concepts are associated with the ontological properties by OWL *AnnotationProperty* structure. The annotations are defined in ontology extension. The example in the next figure presents the Experiment class and some of its annotations, as in Fig. 5.5

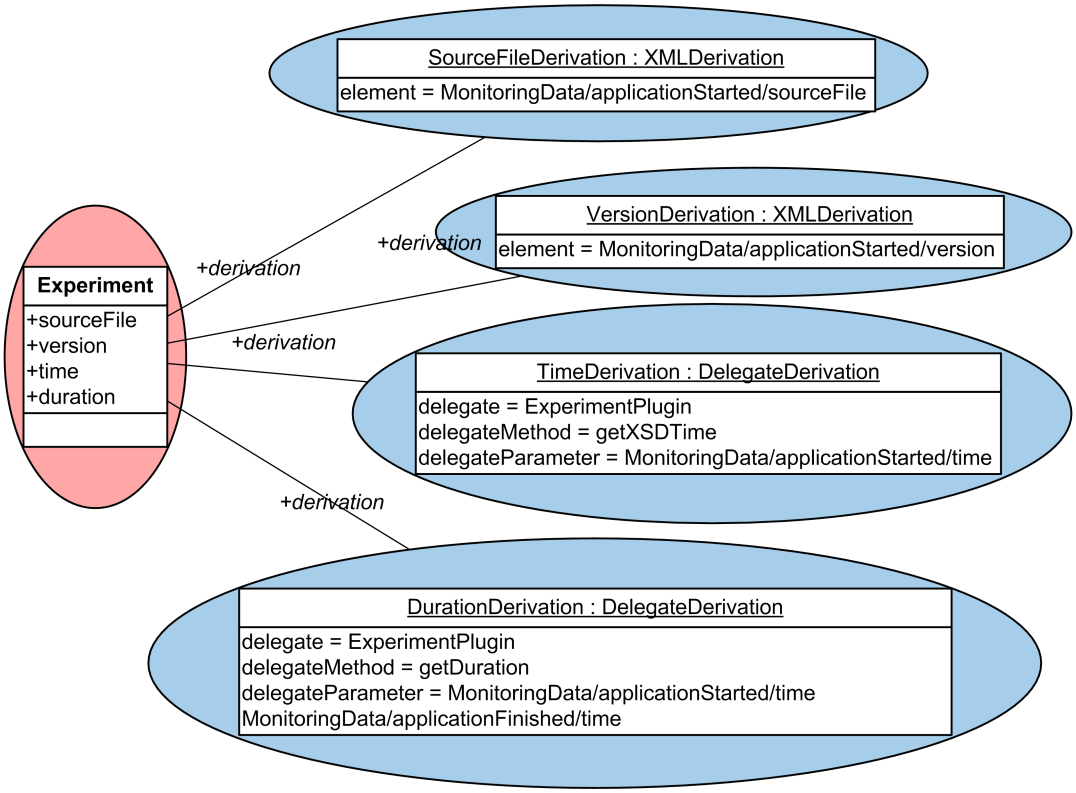


Figure 5.5: Each property of newly created individual is annotated. Each annotation describes value localization in XML file or invocation of a separate piece of code.

There is an assumption that all the annotated properties are established in Aggregator. If some annotations are missed, the created information may be incomplete. A sample definition of a derivation concept is presented below:

```
<rdf:Description
rdf:about="http://www.virolab.org/onto/expprotos/ownerLogin">
  <ext-ns:derivation>
    <ext-ns:Derivation>
      <ext-ns:element
       rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
       MonitoringData/applicationStarted/ownerLogin
      </ext-ns:element>
    </ext-ns:Derivation>
  </ext-ns:derivation>
</rdf:Description>
```

It indicated that the value of *ownerLogin* experiment property should be copied from the *MonitoringData/applicationStarted/ownerLogin* localization in XML file.

5.3.2. Concept of Delegates

During the further, advanced studies it appeared that the simple mapping between XML data and OWL individuals is insufficient and that the knowledge collected in this way is not valuable enough to apply the desirable data mining.

The knowledge creation process was organized in a most generic and extendible way. The suitable solution is to provide the ontology extension author with the ability to design and implement his own computational units that would be utilized during the processing of collected data. In such an approach, there are no restrictions on how the information creation complicated would be, it depends only on the ontology extension author's development skills.

A concept of delegates is proposed. *Delegate* is a distinct, independent computational unit whose usage may be defined in ontology extension. Thanks to this, the creation of information may be delegated from Aggregator to a separate component. The delegate is identified by its unique name and offers a number of methods accessible within the Aggregator context.

Thanks to such an approach, many crucial problems regarding the transformation between XML data and OWL data may be solved. Delegates would enable the following functionality:

- **Transformation between data formats** Some data entities included in XML events may be represented in a format that is not suitable in OWL language. The data types built in XML Schema recommended for use with OWL are *xsd:string, xsd:long, xsd:double, xsd:time, xsd:date, xsd:dateTime, xsd:boolean*. A dedicated delegate is responsible for the conversion to the above formats.
- **Aggregation of the collected data** The differences between ontology models and XML models are present also in data granularity. Some pieces of XML datums are not mapped directly to the ontological pieces of information, therefore, they should be aggregated. In this case, a single piece of information is derived from several pieces of data. This may be an implication of some events-related issues. To exemplify this, let us put into consideration events referring to the beginning and to the end of a concrete activity. The collected data determines only the moments of beginning and the end of this activity. But the valuable information about this activity might be only the duration time. In this case, the duration must be calculated so that the information about the beginning and the end moments is no longer need to be stored. All the time moments are represented as number of in milliseconds between the events and midnight, January 1, 1970 UTC. The converted time information is represented in one of the *xsd:dateTime* correct formats:

<Year>-<Month>-<Day>T<Hours>:<Minutes>:<Seconds>

Duration information is represented also as a *xsd:duration* data type.

- **Generation of individuals identifiers** All individuals created in the ontology are identified in an unique way. This is provided by *rdf:ID* tag. Instead of leaving the identifier generation to the semantic framework it is more reasonable to take control of the identifiers, and through this have an unambiguous access to all of the created individuals. The motivation for such an approach, as well as practical implementation and usage is described in following chapters.
- **Querying PROToS** Delegate may search through the concepts instantiated in the past in order to associate them with already instantiated individual.
- **Querying Data Access** Delegate may extract information, that augment XML data, for example data origin, from data bases.

It is of high importance that the implementation of delegates should be as simple as possible so it would be convenient to develop them for the ontology extension author. The ontology extension designer is expected to be a specialist in ontologies, XML and OWL languages but, furthermore, he should be also familiarized with delegates implementation technology. For that reason, the best approach would be to design the delegates in technology-independent way. However, it is extremely hard to achieve, mostly from the technological issues. In this situation, the ontological description of the delegates usage should be *language-independent* and through this it will be easily extendable, but the support for some additional implementation languages will be provided in the future.

To provide the language-independence an abstract delegate specification is introduced. A delegate can by unequivocally identified by its name. It should be a fully-qualified name, containing package names, because the localization of a computational unit name inside a concrete name space is a commonly used approach, present in many modern programming languages.

A delegate is constituted by a set of methods. To access a concrete method, it is necessary to specify the method name and its parameters, which are restrained to *string* types. Also, the method return value should be of *string* type, for two reasons. Firstly, it is better not to support some advanced data types, because some data formats may be unavailable in some programming languages, so such an approach leads to the loss of language-independence. Secondly, the return values are expected to be translated directly to OWL property values, the one that is annotated with this concrete delegate. As for *string* type, this transformation is fast and simple, it is only the way of placing the return value into the XML tag.

The invocation of delegate methods is described semantically in ontology:

```
<owl:Class rdf:ID="DelegateDerivation"/>
  <rdfs:subClassOf rdf:resource="#Derivation"/>
</owl:Class>
```

```
<owl:FunctionalProperty rdf:ID="delegate">
  <rdf:type
   rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Derivation"/>
</owl:FunctionalProperty>
```

```
<owl:FunctionalProperty rdf:ID="delegateMethod">
  <rdf:type
   rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Derivation"/>
</owl:FunctionalProperty>
```

The conceptual problem is how to semantically describe the parameters passed to the called delegate. OWL language does not support a structural primitive data types, while, in this case, there is a need to specify a *list* of parameters. The OWL data type properties are only conceptual associations – they are defined and analyzed regardless of their order. In fact, there is no semantic description of order-sensitive list data type, what is justified, because a list is a technical programming concept, not a conceptual value present in a real world, hence should not be represented in ontology.

However, the order of delegate parameters should be somehow represented. Five approaches to that problem were examined:

1. **RDF list notation** [15] RDF language supports the list properties. To describe list-like structure, one should use *rdf:List* tag, which defines a special instance of *rdf:Class*. The properties used to specify the position of a concrete item on the list are *rdf:first* and *rdf:rest*. This is done in a notation of three kinds of triples:
   L rdf:first I indicates that item I is the first resource on list L
   L rdf:rest L2 indicates that the rest elements of list L (omitting the first item) are placed on list L2
   L rdf:rest rdf:Nil indicates that list L has only one item, it functions as terminator
   In this notation, the specification of three sample method invocation parameters *param1, param2, param3* is relatively complicated:

```
<rdf:List
rdf:about="http://www.virolab.org/onto/extension/DelegateParameters">
  <rdf:first
   rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
```

```
    param1
  </rdf:first>
  <rdf:rest>
    <rdf:List>
      <rdf:first
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        param2
      </rdf:first>
      <rdf:rest>
        <rdf:List>
          <rdf:first
            rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            param3
          </rdf:first>
          <rdf:rest rdf:resource="&rdf;#nil"/>
        </rdf:List>
      </rdf:rest>
    </rdf:List>
  </rdf:rest>
</rdf:List>
```

2. **RTF shorted list notation, numbered** [16] In a shorten notation, the list items may be specified in tags *rdf:_n*, where n is the position of the node element on the list. The method parameters might be specified be the definition of RDF sequence, as in the following example:

```
<rdf:Seq
rdf:about="http://www.virolab.org/onto/extension/DelegateParameters">
  <rdf:_1 rdf:resource="param1"/>
  <rdf:_2 rdf:resource="param2"/>
  <rdf:_3 rdf:resource="param3"/>
</rdf:Seq>
```

3. **RDF shorten list notation, unnumbered** [16] RDF defines also a property *rdf:li* that is equivalent to *rdf:_n*. The difference is that the list element ordered number must not be specified, however, the order that this properties appear in XML documents is relevant. A corresponding RDF sequence is presented below:

```
<rdf:Seq
rdf:about="http://www.virolab.org/onto/extension/DelegateParameters">
  <rdf:li rdf:resource="param1"/>
  <rdf:li rdf:resource="param2"/>
  <rdf:li rdf:resource="param3"/>
</rdf:Seq>
```

4. **OWL list** OWL does not support ordering, however, it would be possible to model a list-liked structure in a new ontology or adopt an existing ontology containing such model. The conceptual model of list class would be similar like in the approach adapted in RDF, as in Fig. 5.6

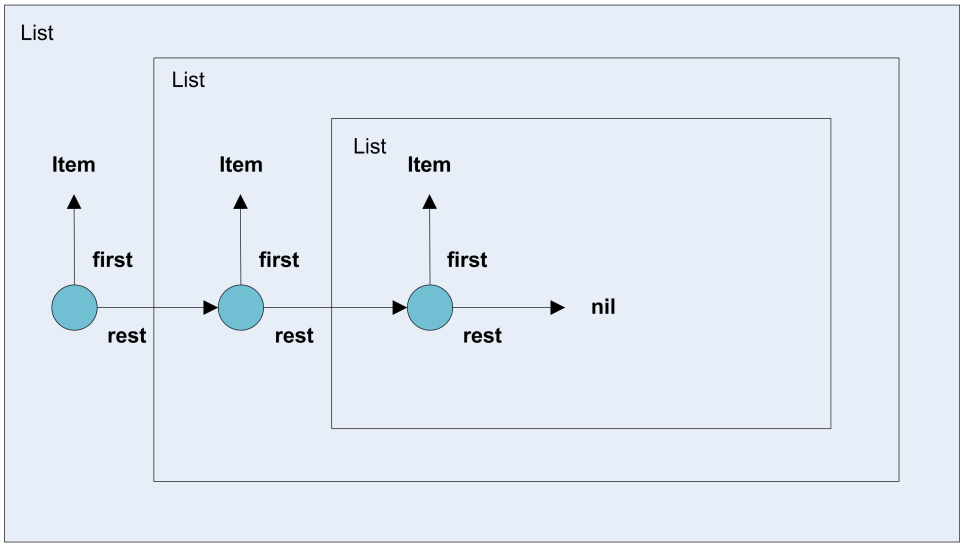


Figure 5.6: List conceptual model. Its structure is defined recursively.

There should be defined a *List* class, possibly derived from more generic *Collection* concept. The connectors between list items should be modeled as ontological properties, as in Fig. 5.7
hasFirst – object, functional, non-transitive property
hasRest – object, functional, non-transitive property

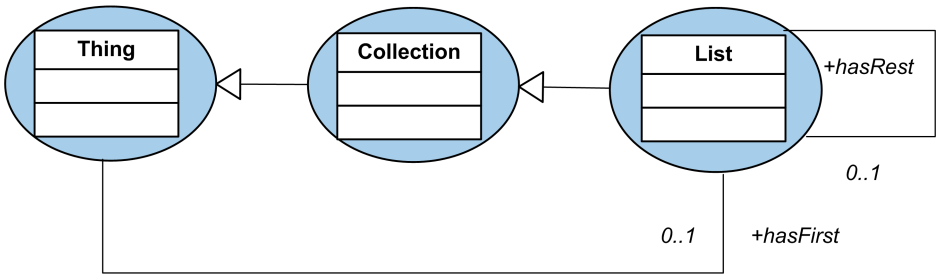


Figure 5.7: List structure description expressed in OWL language.

Some cardinality restrictions should be imposed on these properties. Actually, OWL supports such restrictions – their specification should be enclosed in separated tags:

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasFirst"/>
  <owl:maxCardinality
   rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
</owl:Restriction>
```

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasRest"/>
  <owl:maxCardinality
   rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
</owl:Restriction>
```

5. **OWL properties not supported semantically** The last considered solution is to specify the parameters as simple functional properties, as in the following piece of code:

```
<owl:FunctionalProperty rdf:ID="delegateParameter">
  <rdf:type
   rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Derivation"/>
</owl:FunctionalProperty>
```

This approach is not order-sensitive. It means that the parameters order is not described semantically, so, if some reasoning is applied to this ontology, the order is transparent for the reasoner and therefore cannot be checked. However, it is reasonable to make assumption that the ontology parser built in Jena framework will always parse the properties in the specified order. That means the information about the order is lost during the ontology processing, as in Fig. 5.8

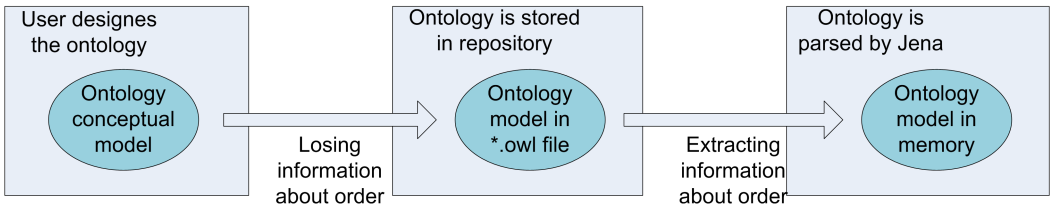


Figure 5.8: Parameters order information incoherency. OWL model is parameters order insensitive.

This may lead to the information inconsistency. In order to apply this solution, two assumptions have to be taken:

- (*) The ontology storage does not apply changes in OWL file structure so that it remains unchanged from the moment it was designed by user

- (**) The Semantic Web Framework used in ontology parsing parse the functional properties in the order they are specified in OWL file.

The solutions (1), (2), (3) have beed dismissed, as RDF triples are not semantically valuable, so there are no advantages in comparison with the (5). The (4) approach is too complicated and may cause that the ontologies will not be understandable by the user. The (5) is not ideal solution, however, the requirements (*), (**) are possible to be satisfied. The ontologies after their designing are deployed as OWL files on HTTP service and do not undergo farther changes. Furthermore, the only used Semantic Web Framework is Jena, which parser is *properties order sensitive*.

At the current stage of development, the delegates are implemented in Java technology. A sample delegate usage refers to the experiment duration. It can be computed by the invocation of method *getDuration* exposed by *ExperimentPlugin* delegate:

```
<rdf:Description
rdf:about="http://www.virolab.org/onto/exp-protos/duration">
  <ext-ns:derivation>
    <ext-ns:Derivation rdf:ID="DurationDerivation">
      <ext-ns:delegate
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        cyfronet.gridspace.mring.aggregator.delegates.ExperimentPlugin
      </ext-ns:delegate>
      <ext-ns:delegateMethod
      rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
        getDuration
      </ext-ns:delegateMethod>
      <ext-ns:delegateParameter>
        MonitoringData/applicationStarted/time
      </ext-ns:delegateParameter>
      <ext-ns:delegateParameter>
        MonitoringData/applicationFinished/time
      </ext-ns:delegateParameter>
    </ext-ns:Derivation>
  </ext-ns:derivation>
</rdf:Description>
```

The list of currently used delegates and their responsibilities is presented in Tab. 5.2.

5.3.3. Aggregation Rules

While the derivation concepts describe how to create ontological individual from correlated XML data, the aggregation rules define when the process of aggregation should be initiated, what subset of gathered XML data should be used and what

Delegate	Method	Result
ExperimentPlugin	getExperimentID	Experiment individual identifier
	getGOID	GridObject individual identifier
	getGOIID	GridObjectInstance individual identifier
	getGOpInvocationID	Computation individual identifier
	getXSDTime	time converted to xsd:dateTime format
	getDuration	the duration of experiment
		the duration of computation
GOIPlugin	findGOIByEndpoint	GridObjectInstance individual identifier

Table 5.2: Delegates.

ontological class should be instantiated. Like the derivation concepts, the aggregation rule is modeled semantically as an ontological concept.

An aggregation rule comprises following information:

- **What event type should be correlated** The aggregated event types typically refer to the same activity. When all the related events type are registered in Aggregator, the information creation process may be triggered, because all the monitoring data related to this concrete aspect of VLvl is collected. Typical correlated events pair refers to the beginning and to the end of some activity – for example *ApplicationStarted* and *ApplicationFinished*.
- **The ACID number correlation level** The existence of two events of correlated types is not enough to trigger the aggregation process, because these events may differ in the origin context – for example, ApplicationStarted and ApplicationFinished events may come from different applications. Therefore, aggregation rule contains also information of how the correlated events ACID identifiers should be related. If the correlation level is 1, it means that the events must have the same application identifiers thus must come from the same application. If the correlation level is 2, they additionally must have the same task identifier thus must come from the same application stage. The correlation level 3 indicates the same sub-stage. An exception is the correlation level set to 0. Events of this type are not correlated with each other but are processed directly after the appearance in aggregator. The examples of such directly-aggregated events are the registration of a new Grid Object, Grid Object Implementation or Grid Object Instance.
- **What ontological classes should be instantiated when an aggregation rule is satisfied** This is the implementation of *declarative programming approach*. When the aggregation process begins and the related XML data is provided, the Aggregator engine creates an individual of a particular class and then studies all

of the annotations of its properties. Proper values of these properties are established on the basis of derivation concepts and the available XML data. Therefore, the shape of created information depends on how many ontological properties are annotated and how valuable is the XML data from already correlated monitoring events. Also, the created information may be extended by the querying the Data Access for additional data, what should be implemented in a delegate dedicated for the accessing of data bases.

The rule definition in OWL language is presented below:

```
<owl:Class rdf:ID="AggregationRule"/>

<owl:DatatypeProperty rdf:ID="eventType">
  <rdfs:domain rdf:resource="#AggregationRule"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="instantiatedClass">
  <rdfs:domain rdf:resource="#AggregationRule"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>

<owl:FunctionalProperty rdf:ID="acidCoherency">
  <rdf:type
   rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  <rdfs:domain rdf:resource="#AggregationRule"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:FunctionalProperty>
```

The aggregation rules are defined on the basis of the above schema, as in following example:

```
<ext-ns:AggregationRule rdf:ID="ExperimentAggregation">
  <ext-ns:eventType
   rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     ApplicationStarted
  </ext-ns:eventType>
  <ext-ns:eventType
   rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     ApplicationFinished
  </ext-ns:eventType>
  <ext-ns:instantiatedClass
     rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
     http://www.virolab.org/onto/exp-protos/Experiment
  </ext-ns:instantiatedClass>
  <ext-ns:acidCoherency
```

```
15    rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
16      1
17    </ext-ns:acidCoherency>
18  </ext-ns:AggregationRule>
```

This rule states that the events ApplicationStarted and ApplicationFinished should be aggregated with each other in a case they have the same application identifier. When such an aggregation does take place, an Experiment class should be instantiated on the basis of the XML data from these two events.

Another examples of aggregation rules addressing the experiment ontology are presented in Tab. 5.3

Event types	**ACID coherency level**	**Instantiation**
ApplicationStarted *ApplicationFinished*	1	*http://www.virolab.org/onto/exp-protos/ Experiment*
GridOperationInvoking *GridOperationInvoked*	2	*http://www.virolab.org/onto/exp-protos/ Computation*
DataAccessQuerying *DataAccessQuered*	2	*http://www.virolab.org/onto/exp-protos/ DataAccessCall*
GridObject InstanceRegistered	0	*http://www.virolab.org/onto/exp-protos/ GridObjectInstance*
GridObject ImplementationRegistered	0	*http://www.virolab.org/onto/exp-protos/ GridObjectImplementation*
GridObjectRegistered	0	*http://www.virolab.org/onto/exp-protos/ GridObject*

Table 5.3: Aggregation rules.

5.4. Experiment transaction support

In the presented events processing model, the ontology individuals may be classified, with respect to correlation time, into three groups:

- Individuals created immediately after the appearance of new event in Aggregator
- Individuals created after the appearance of two events, which appear one-by-one
- Individuals created after the appearance of two events, which are distant in time

There exists only one ontological class from the third group – Experiment, instantiated after the appearance of ApplicationStarted and ApplicationFinished events. In

the period of time between these two moments there appear many events related to this experiment, what results in instantiation of individuals describing experiments stages, despite the fact the Experiment individual does not yet exist. This is presented on the figure below. Above the time-line are situated the events delivered to Aggregator while under the time-line are situated the created individuals, as in Fig. 5.21

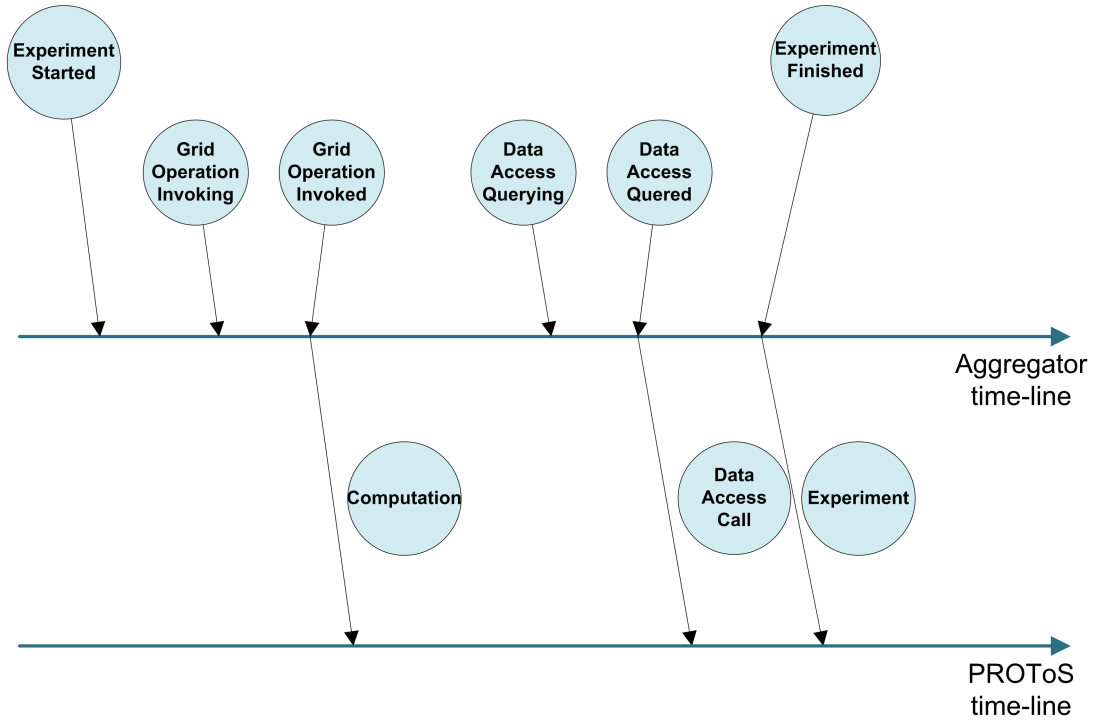


Figure 5.9: Time relation between events. Events describing individuals are delivered to PROToS directly after aggregation of corresponding monitoring events.

This may lead to the information-inconsistency, because, in a given moment in time, in the ontology storage there exist individuals *Computation* and *DataAccessCall* but there do not exists individual Experiment they are related to. The experiment stages without their context are semantically invaluable, therefore, the association between a stage and its experiment is obligatory. In a fact, there is no guarantee that the Experiment individual will be instantiated, because of two reasons:

- The experiment may fail
- Due to the monitoring infrastructure failure, the event *ExperimentFinished* may not be delivered

This problem was solved by the implementation of the experiment transaction support mechanism. An experiment may be perceived as a transaction in a sense that all the individuals related to this experiment are recorded, together with the

experiment individual, or no individual is recorded. In order to realize this idea, there should exists a third individual processing layer, a buffer storing all individuals related to concrete experiment. This conception is presented in the Fig. 5.10

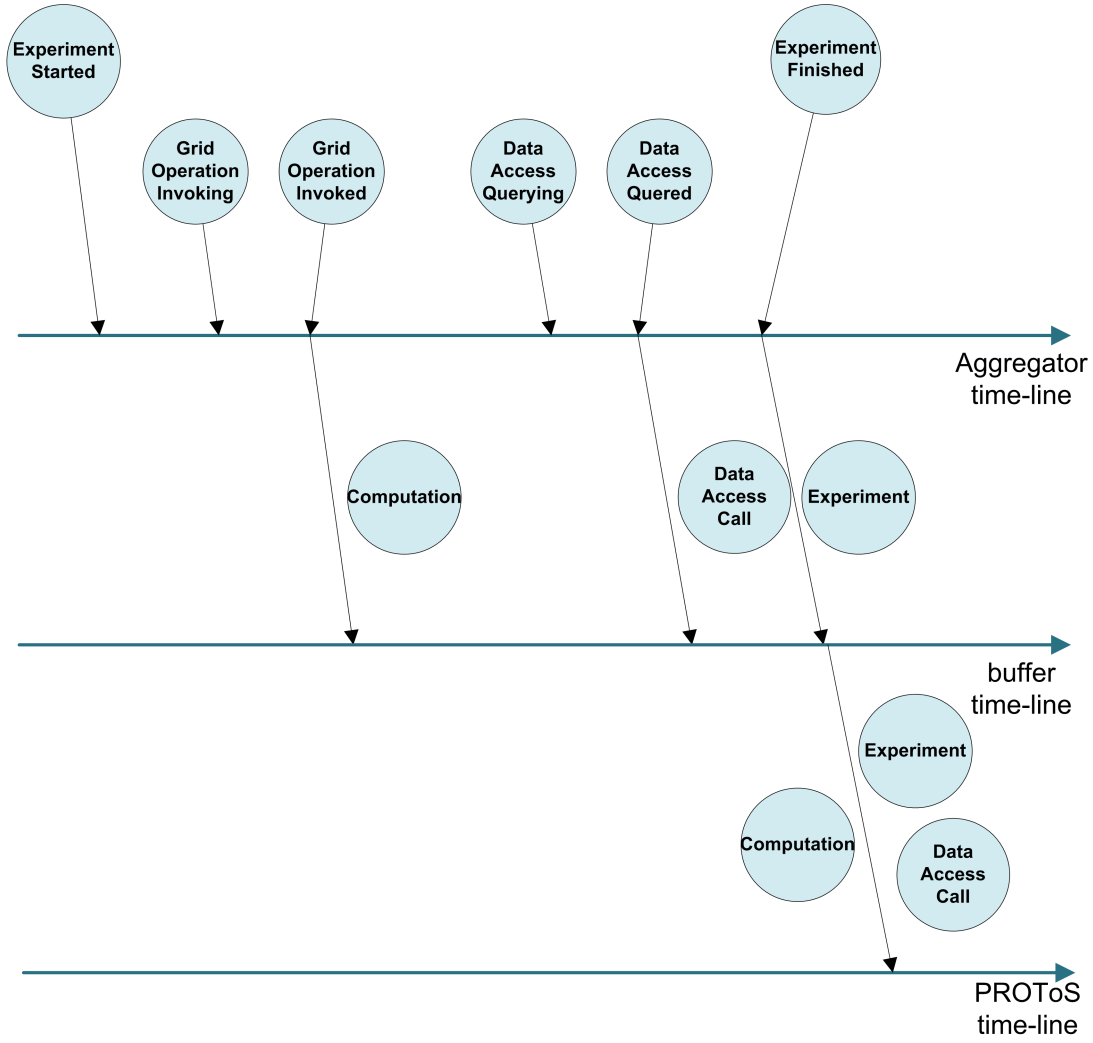


Figure 5.10: Time relation between events in transactional processing. Events describing individuals are temporarily buffered. They are delivered to PROToS when experiment is successfully finished.

It is possible to determine two classes of experiment failure:

- **High-level failure** In this case, one of the experiment stages fails and the execution cannot proceed because the next steps are dependent on the failure stage. It may refer to several kinds of situations:
  — Computation being realized by Grid Object fails
  — Grid Object is temporarily unavailable
  — Data Access querying is not successful

— Required user-feedback is not gained

In that case, the continuation of experiment is possible, but makes no sense.

- **Low-level failure** In this case, the experiment continuation is not possible at all. In high-level failure the continuation is impossible due to dependencies of workflow stages , while in low-level failure is impossible due to technical issues, such as:
  — Script enactment engine failure
  — Invoker failure

The experiment re-execution issue may be considered in the two described failure classes. It is of high importance especially when dealing with long-time computations – assuming that a concrete service is stateless (in a sense that the obtained result does not depend on the time of the service call but only on the input data), the output from this service may be recorded and used in the future.

The experiment re-execution depends on two important aspects:

- Stages linear execution order
- Stages dependency

In following simple example, an experiment consists of 5 stages executed one-by-one in the order *(Stage1, Stage2, Stage3, Stage4, Stage5)*, as in Fig. 5.11

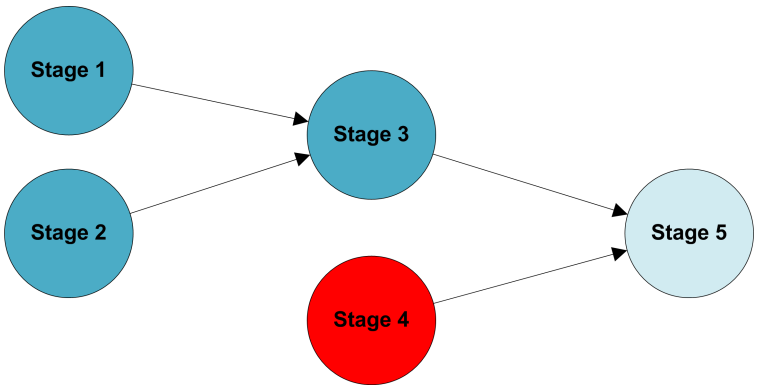


Figure 5.11: Dependencies between stages in sample experiment. All successfully executed stages are recorded.

After the failure of Stage4, the experiment cannot be continued because there exists a data-dependency between (Stage3, Stage4) and Stage5. However, based on the Stage3 output data (the outputs of Stage1, Stage2 storage is not necessary), in the experiment re-execution only the Stage4 and Stage5 should be performed in order to gained final results, as in Fig. 5.12.

As for technical issues concerning the implementation of described idea, there should be recorded information about experiment stages, even in a case the exper-

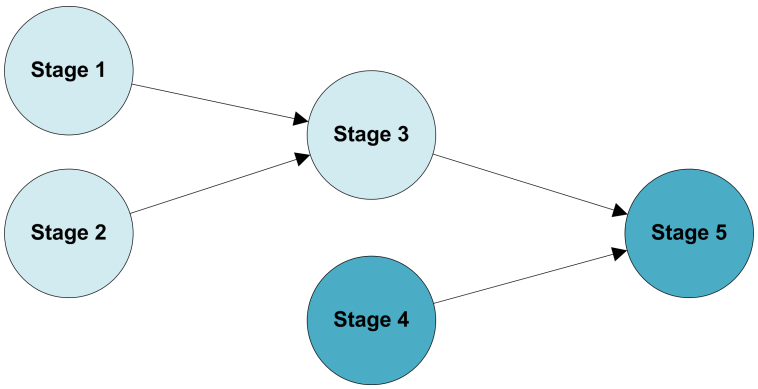


Figure 5.12: Experiment re-execution. Only failed stages and stages depending on them have to be executed.

iment does not finish successfully. It is reasonable that only the high-level failure will be supported in this way.

To implement this kind of re-execution, crucial question must be answered: *What is an experiment transaction from the aggregation point of view?*. Or, more precisely: *What are the moments of the experiment transaction beginning and the transaction commitment from the aggregation point of view? What is the transaction context?*.

The following definitions may be introduced:

- ***transaction beginning*** *is the appearance of event ApplicationStarted in Aggregator*
- ***transaction context*** *are all individuals that are associated with the application that is in progress*
- ***transaction commitment*** *is the appearance of event that aggregates with ApplicationStarted and has a suitable application identifier*

It is easy to extend the aggregation process to realize the support of transaction perceived in this way. An additional aggregation rule should be introduced for the Experiment class, which associates events (ApplicationStarted, ApplicationFailed). In this case, the instantiation of a particular class may be triggered after the correlation of two possible sets of events – ApplicationStarted may be correlated both with ApplicationFinished or ApplicationFailed.

It must be emphasized that the Aggregator component responsible for the aggregation rules monitoring is *transaction-transparent*. It means that the transaction is not described semantically, what is justified, because transaction processing is a pure technical issue, as in Fig. 5.13

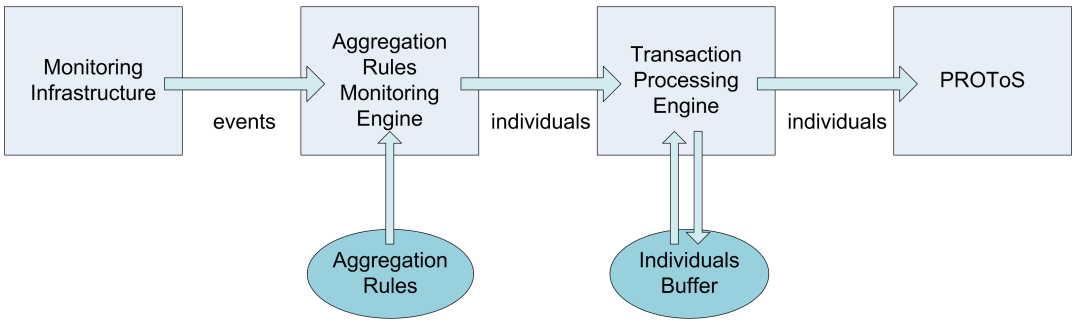


Figure 5.13: Events aggregation conceptual model. Already gathered XML events contents is being monitored in terms of aggregation rules fulfilment. Already created individuals are being monitored in terms of experiment transaction commitment.

5.5. Semantic associations discovery

The methodology described in section 5.3 is well-adapted to the datatype ontological properties. However, while the creation of a concrete individual and filling its functional properties is a simple process, the establishment of the object properties is significantly more complicated.

In OWL language, the association between individuals is defined by providing a value of object property. The class of associated object and individual identifier:

```
<Experiment rdf:about="http://www.virolab.org/onto/exp-protos/Exp1">
  <hasStage>
    <Computation
     rdf:about="http://www.virolab.org/onto/exp-protos/Cmp1" />
  </hasStage>
</Experiment>
```

In order to discover the associated individual identifiers, several approaches were undertaken. They are described in the following sections.

5.5.1. Hashing individuals naming

The first approach was inspired by the observation of events structures. It leads to the conclusion that the contexts of created individuals, constituted by the pieces of data from monitoring events, may share the same kind of information. For example, the pieces of information associated with Computation and Experiment share the experiment id, as in Fig. 5.14.

In this situation, assuming that the experiment id is unique for all the Experiment instances, the Experiment individual identifier may be computed as a result of hashing function applied to the experiment id, as in Fig. 5.15.

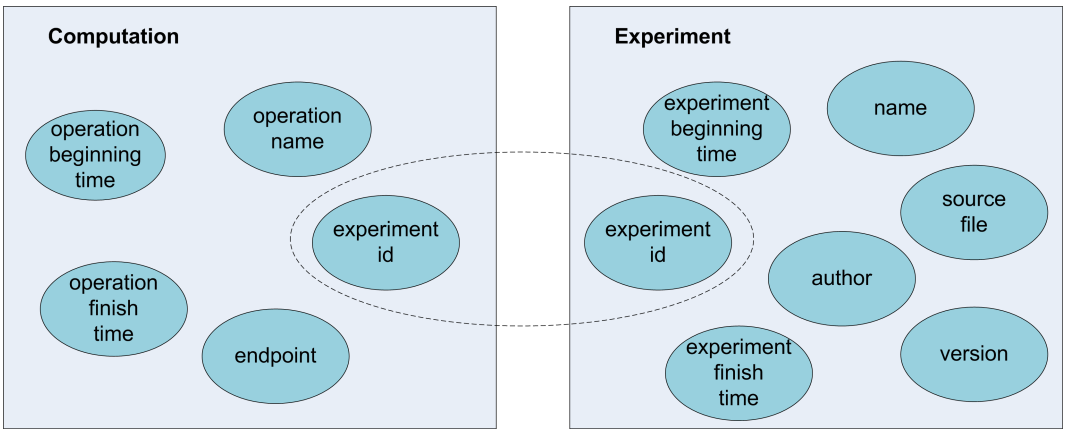


Figure 5.14: Sets of XML data items correlated before aggregation to *Computation* individual and before aggregation to *Experiment* individual have experiment have common subset.

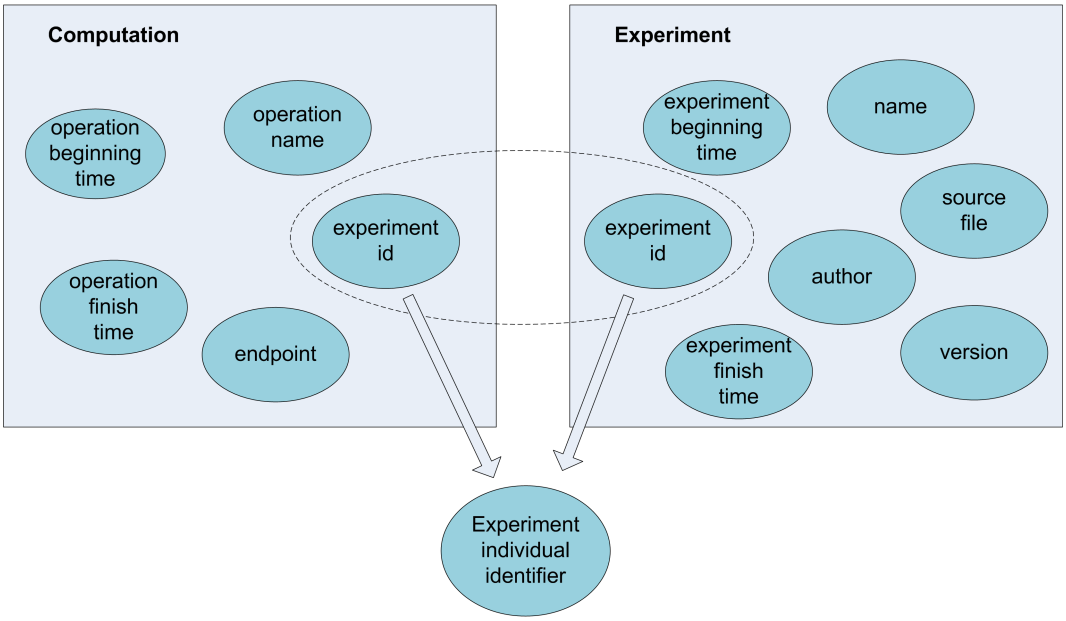


Figure 5.15: *Experiment* individual URI is computed as result of a hash function applied to data items from the common subset.

In such an approach, it is possible to determine the identifier of the associating object – the only requirement is that all the information that should passed to the hashing function is accessible in the context of created individual. In the presented example, the hashing is applied at least twice:

- the moment the Experiment individual is created

- the moment the Computation individual is created and the Experiment identifier is searched in order to create the *executedIn* association

The natural place of the hashing function implementation is a delegate. The delegate method responsible for identifier generation is called every time a new association is created. So, in the ontology extension, this delegate derivation is associated both with experiment (defining its identifier) and object property (defining its value). The following figure presents the Computation class neighborhood and derivations of one of its object properties, as in Fig. 5.16

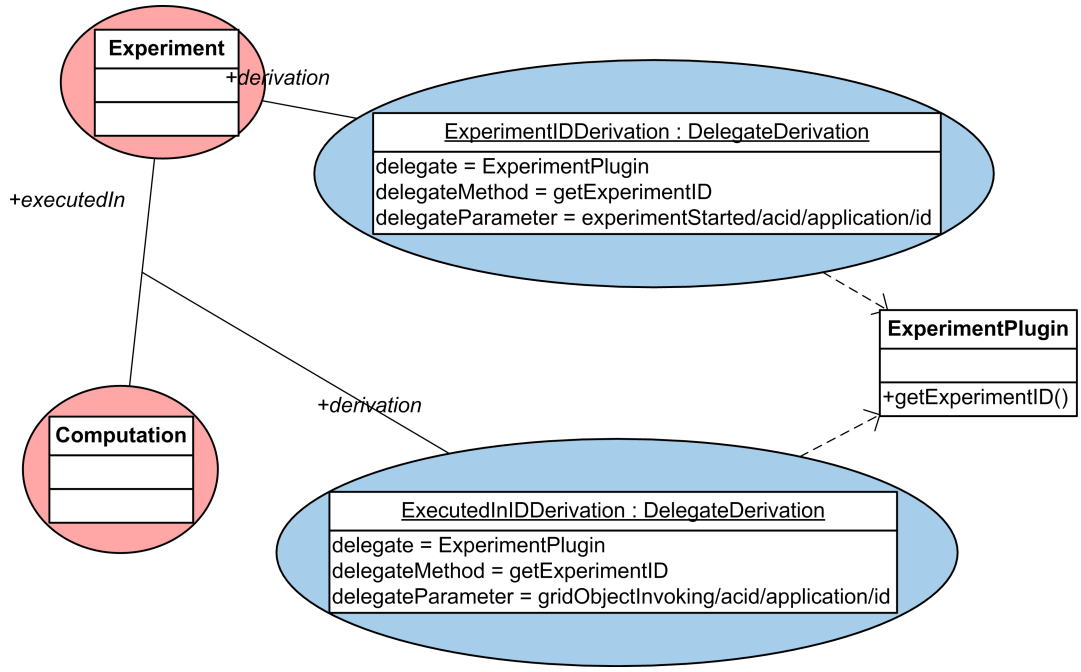


Figure 5.16: Object defining derivations of *Experiment* URI and *executedIn* property of *Computation* point to the same hash function implemented in a delegate. Due to different correlated XML data contexts, they differ in XPath value.

Another examples of the usage of this kind of delegates are collected in Tab. 5.4.

5.5.2. Knowledge history tracking

In a more complex case the association cannot be established as simply as described in the preceding section. However, the data collected in the context of the individual that is being created, is sufficient to construct a query addressing already created ontology that will return the identifier of the individual that is being searched. This solution is inefficient, because it would include five performance-intensive steps, as in Fig. 5.17.

Delegate	Method	Usage
Experiment Plugin	*generatedExperimentID*	*Experiment* individual identifier
		hasStage property of *Computation* individual
	generateGOIID	*GridObjectInstance* individual identifier
		usedInstance property of *Computation* individual
		hasInstance property of *GridObjectImplementation* individual
	generateGOImplID	*GridObjectImplementation* individual identifier
		hasImplementation property of *GridObject* individual
	generateGOID	*GridObject* individual identifier

Table 5.4: Usage of delegates in individuals naming.

1. XQuery construction
2. Accessing PROToS via Web Service interface
3. Processing the query in PROToS
4. Returning the XML file
5. Parsing the XML file

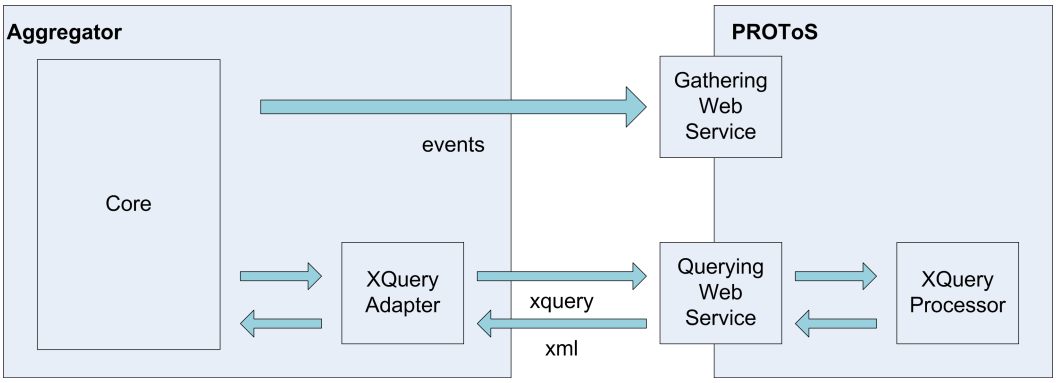


Figure 5.17: Unoptimal events history tracking. Aggregator creates xquery, invokes Web Service and parses the results.

A more effective approach would be to implement individuals buffering inside the Aggregator, as in Fig. 5.18.

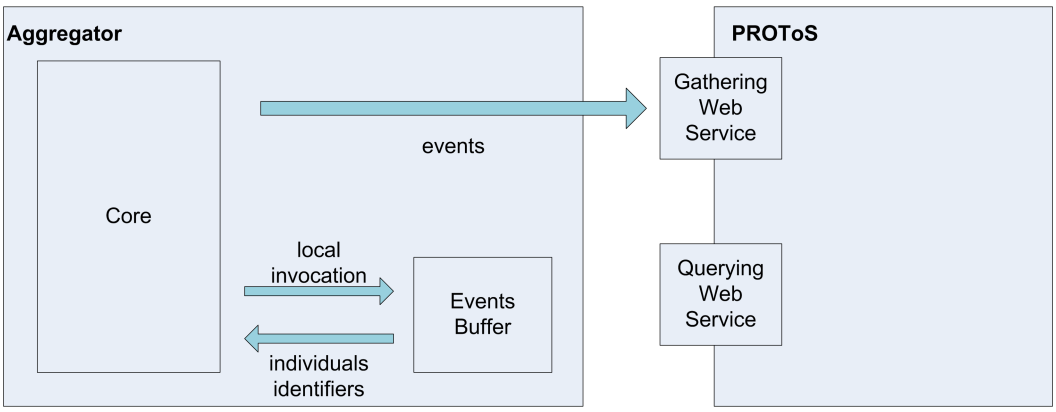


Figure 5.18: Optimized events history tracking. Aggregator temporarily stored created individuals in a separate buffer. The individuals do not have to be serialized and transfer through network.

It is natural that the local buffer should not store all the individuals passed to PROToS through all over the time – it would be extremely inefficient. The motivation for events buffering is to store events that are temporarily needed to establish association. From this point of view, PROToS may be perceived as a *long-term data storage*, while the Aggregator buffer may be perceived as a *short-term partial data storage*.

It is an important issue to provide a definition of *what individuals should be buffered* and *what should be the buffering time*. With respect to the buffer component efficiency, two types of buffering should be applied:

- buffer all individuals related to a concrete experiment in the time this particular experiment is being executed
- buffer all individuals not related to a concrete experiment in a long period of time

This approach is justified, because the need to query for *experiment-related* individuals, such Computation, DataAccessCall or domain event is present only during the experiment execution time – the association addressing experiment-related individual is created only in a moment of instantiation of another experiment-related individual. Furthermore, this kind of buffering may be directly used in the transaction processing.

On the other side, as for the *experiment-unrelated* individuals, they may be buffered efficiently in a long-term period of time, because of their limited number. Basically, they describe some aspects of computational services, which are not as frequent as the experiment-related individuals, mostly because of the great number of experiment stages and created pieces of data. An additional aspect related to this type of individuals is that, from some technical reasons, Grid Resources Registry component publishes events describing all the available services every time it is activated. Therefore, the

services information may be redundant. To avoid a situation in which there exist many identical individuals describing the same service, the Aggregator controls if a concrete experiment-unrelated individual was buffered in the past.

The described buffer-querying should be implemented, similar as the identifiers hashing described in the previous chapter, in delegates, and formally specified in ontology extension, as in Fig. 5.19

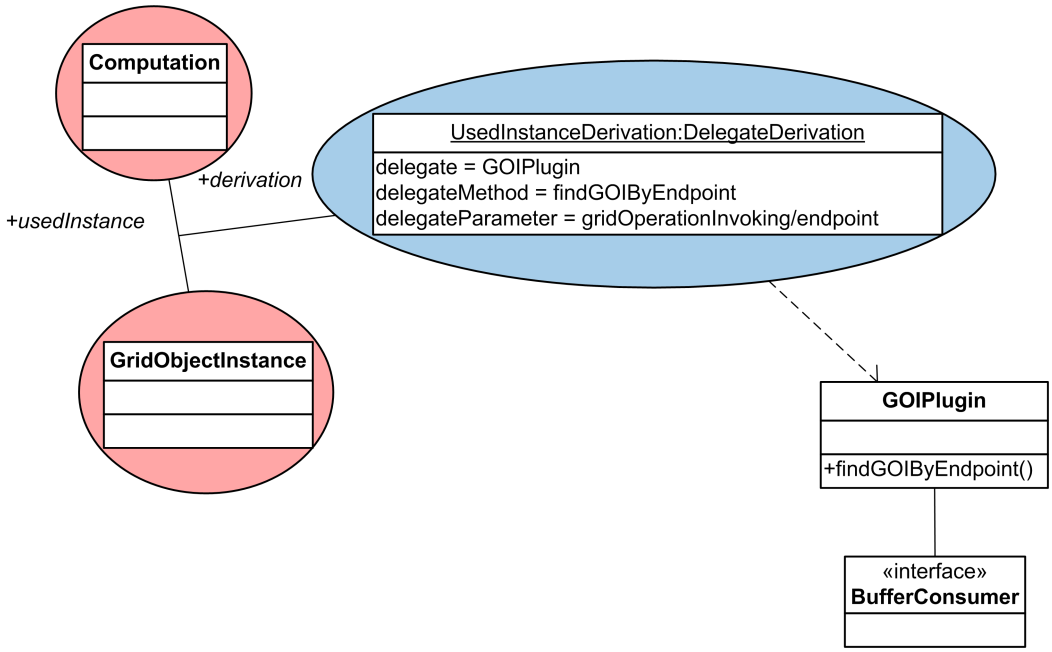


Figure 5.19: Object defining derivation of *usedInstance* property of *Computation* points to delegate, which searches the individuals buffer. Buffer reference is passed to the constructor in the moment of delegate instantiation.

The only difference is that an extended kind of delegate has to be introduced – a plugin that have access to the Aggregator buffer. The buffer reference is passed to this buffer-related delegate in the moment of its creation. A sample delegate working in this way is *GOIPlugin*, which exposes method *findGOIByEndpoint*. This function searches through the buffered individuals of class GridObjectInstance for the one that have an endpoint address identical as the one specified in GridOperationInvoking event.

5.5.3. Context association

The observation of the relation between created individuals leads to the conclusion that some associations may be created in an automatic, intelligent way. That means, without a special, separated specification in ontology extension. This observation refers to the individuals existing within the context of a concrete experiment.

As an example, let us consider following ontological association, as in Fig. 5.20

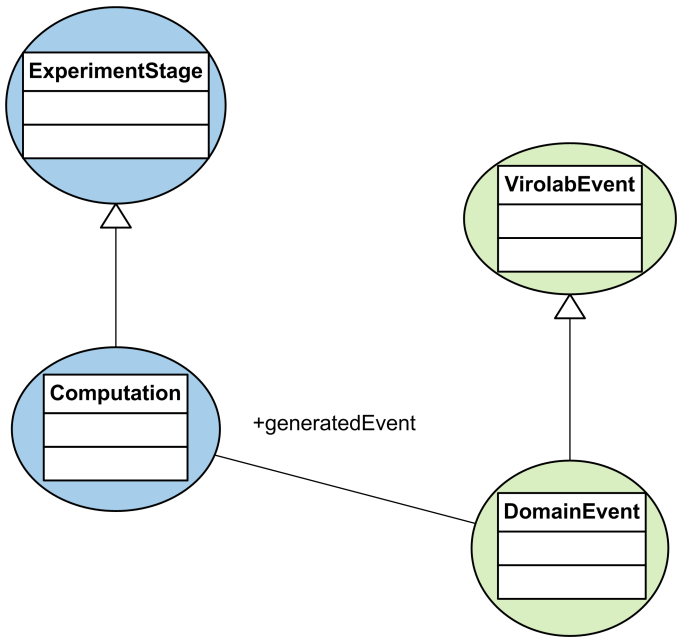


Figure 5.20: Sample relation between experiment and domain ontology. Generic *Computation* is associated with domain event describing its meaning.

In a context of a particular experiment, there exist individuals created in different moments of time, presented in Fig. 5.21

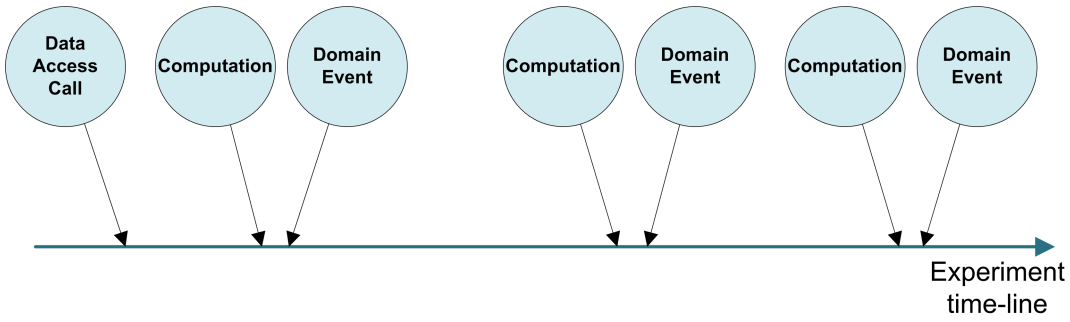


Figure 5.21: Sample experiment context. Domain individuals are created directly after the generic invididuals. They probably refer to the same events.

In this experiment, some generic and some domain events were generated separately. Generic events were published in Invoker, while domain events were published directly in experiment script. In this situation, it is reasonable to associate the events that occurred nearly in the same time, because they probably refer to the same activities. The implementation of this idea requires the integration of a new, additional layer in the individual flow – a component responsible for the establishment of object properties within a particular experiment, as in Fig. 5.22

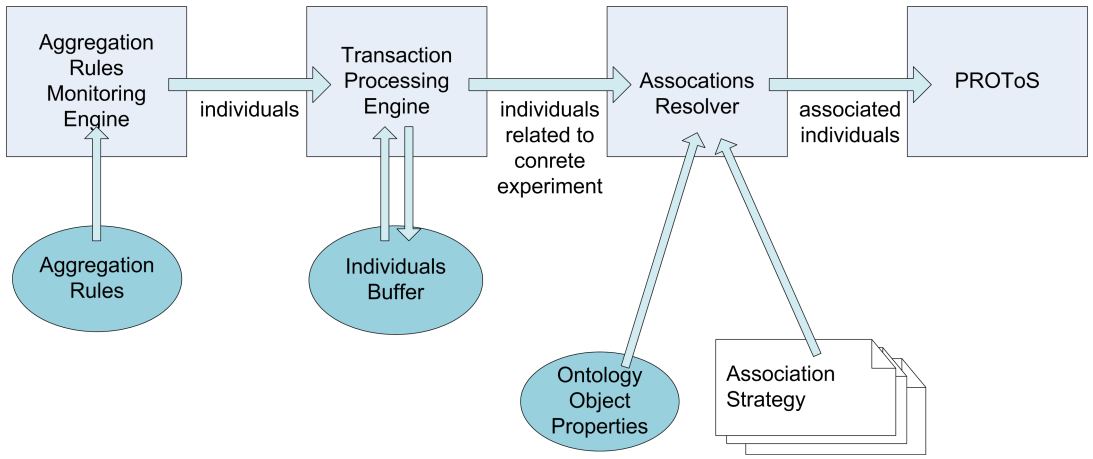


Figure 5.22: Relation between transaction processing and associations discovery. Association strategy implements principles describing how a set of individuals created in the same experiment should be analysed in terms of object properties.

The *AssociationResolver* should offer a high level of lability. That means, it should be possible to define many strategies describing how to associate individuals with each other – a new version of ontology possibly requires a new strategy of association. Therefore, an *AssociationStrategy* is used – a concrete implementation of the strategy interface injected into the AssociationResolver.

For the presented shape of ontology, a desirable strategy would be to associate all Computation individuals with the domain individuals nearest in the time. Naturally, this is justified only assuming that a domain event is published in script directly after the corresponding generic event is published by Invoker. It is possible to implement more sophisticated strategies that would searching for individuals situated nearest in time minimizes the complete, summarized time distance between all associated individuals.

There were also implemented strategies supporting the ontological generalization hierarchy. For all of the collected general concepts, such as Computation, the instances of sub-classes, such domain events, are searched within the experiment context. This is a support for the kind if relation presented in Fig. 5.23.

If there is present a corresponding sub-class individual, it is augmented with the information from the super-class individual (in this case, the attribute values are simply copied from Computation instance to the domain event instance, and then the Computation instance can be removed). In another case, only a more generic sub-class instance is sent to PROToS.

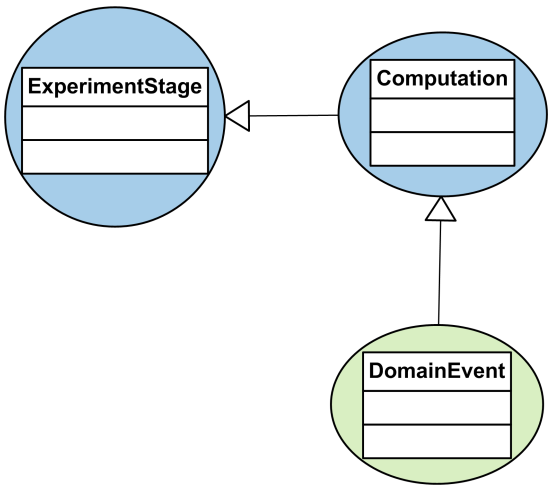


Figure 5.23: Sample relation between experiment and domain ontology. Domain event is a more specific kind of generic *Computation*.

Chapter 6

Design and implementation of Aggregator

Semantic Event Aggregator architecture is presented in this chapter. Implementation details of main components – AggregationCore, EventHandler and XMLDataContext are described. There are also discussed issues of Aggegator deployment.

6.1. Aggregator architecture

The decomposition of Semantic Event Aggregator into modules is presented in the component diagram in Fig. 6.1

The central part of Aggregator is *AggregatorCore* module. It provides the basic functionality offered by Aggregator. Two interfaces of AggregatorCore are used only in the moment of its initialization. The Aggregator is configurable by ontologies, which processing is realized the means of Jena API, as the most commonly used ontologies-processing framework. Therefore, the core configuring interface is parameterized by Jena *OntModel*, which is parsed from all the ontologies utilized by Aggregator.

The core was designed to be used in a listener-model conventions. In this approach, it exposes exactly one interface accessible in runtime, *handleEvent* which is used to gather the delivered monitoring events. It acts as a mediator, which passes the events to the *EventHandler* , directly responsible for the gathered events processing. The created part of ontology, in a form of sets of ontological individuals, is passed to all the actors interested in aggregated knowledge. These information consumers are

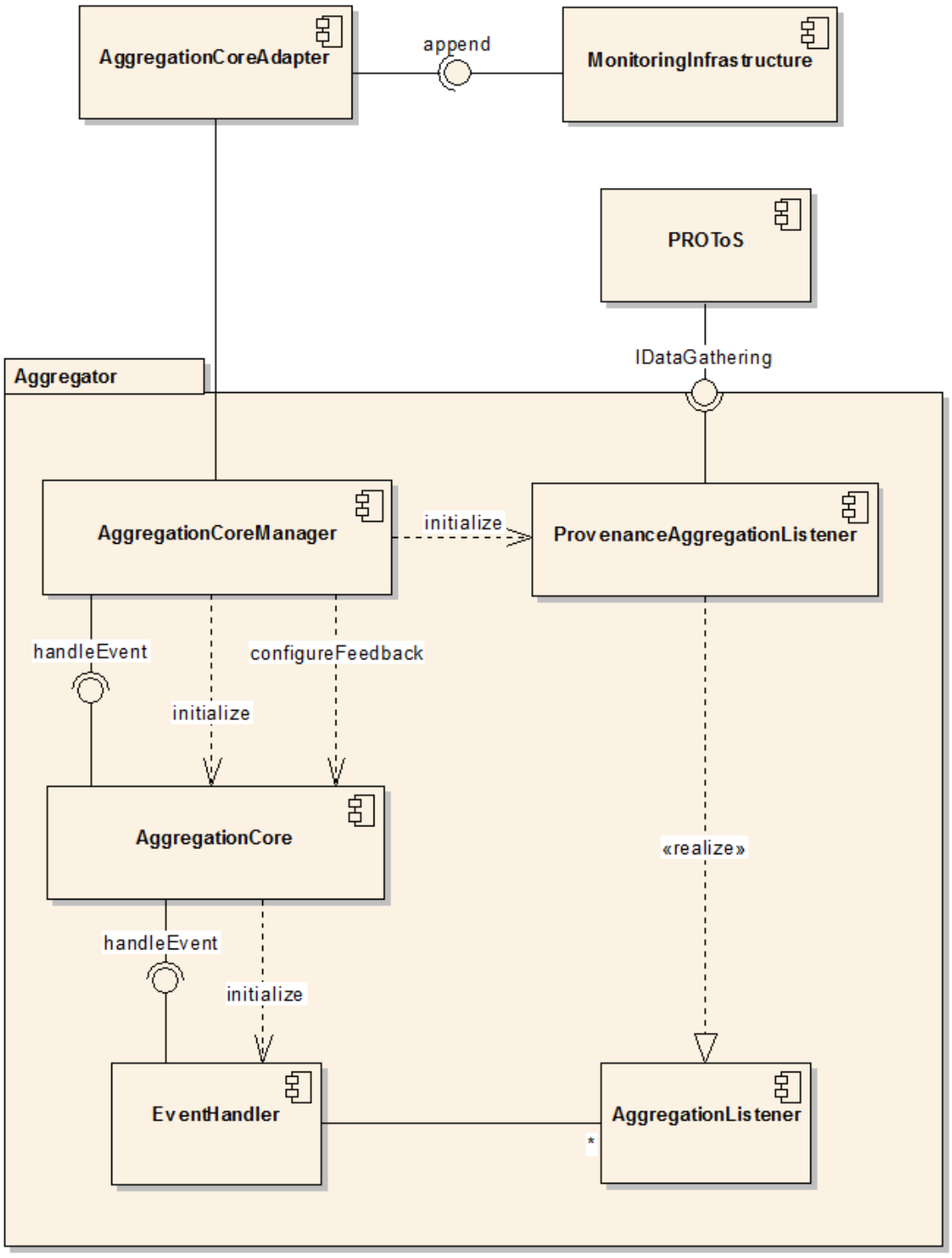


Figure 6.1: Semantic Event Aggregator architecture. Whenever new individual is created, all *AggregationListeners* are notified. Aggregator may be integrated with any monitoring technology by dedicated adapter.

expected to implement the *AggregationListener* interface. That makes the aggregation process convenient in implementation and extendible.

There exists a separated component responsible for the initialization and configuration of *AggregationCore*, *AggregationCoreManager*. It creates two information consumers. *ProvenanceAggregationListener* converts the created individuals into PROToS-specific events and accesses its data gathering interface. At the current stage of development, communication with PROToS Web Service is realized by XFire framework [27], however, migration to CXF [27] is being considered. , and *LoggingAggregationListener*, which logs the information about the aggregation using the standard log4j API.

The described components are *monitoring infrastructure independent*. That means, their functionality may be used independently of the monitoring system technology. There exists an *AggregationCoreAdapter* which adapts the Aggregator to the remote log4j logging architecture. It is possible to developer another adapters, that would enable the incorporation of Aggregator into another monitoring system, for example into the JMX.

The most important parts of the *EventHandler* component, is presented in details in Fig. 6.2

The central part of the EventHandler is *XMLDataContext*. It is a storage for the XML data that were collected, but not yet correlated. It should offer a convenient and well-effective access to XML documents addressing by ACID number and event type, because, during the correlation of monitoring events, the document are searched on the basis of experiment identifier, task identifier and event type. The most effective solution that enable such navigation through the collected data it to organize the context in a hierarchical, structural tree-form, with tree kinds of nodes, presented in Fig. 6.3 :

- ACID part, such as experiment or task identifier, connected to a node with the higher-level part of ACID
- Event type, connected to a node indicating ACID element which is shared by all of the events localized in a local tree branch
- XML document, connected to a node indicating its type

Every time a new XML document is added to this tree structure, all the aggregation rules referring to this event type and having proper ACID coherency level are checked. In a case aggregation rule is satisfied, the correlated data is removed from the context and undergo farther processing in individual factories.

The XML events delivered by EventHandler are partially parsed. The information

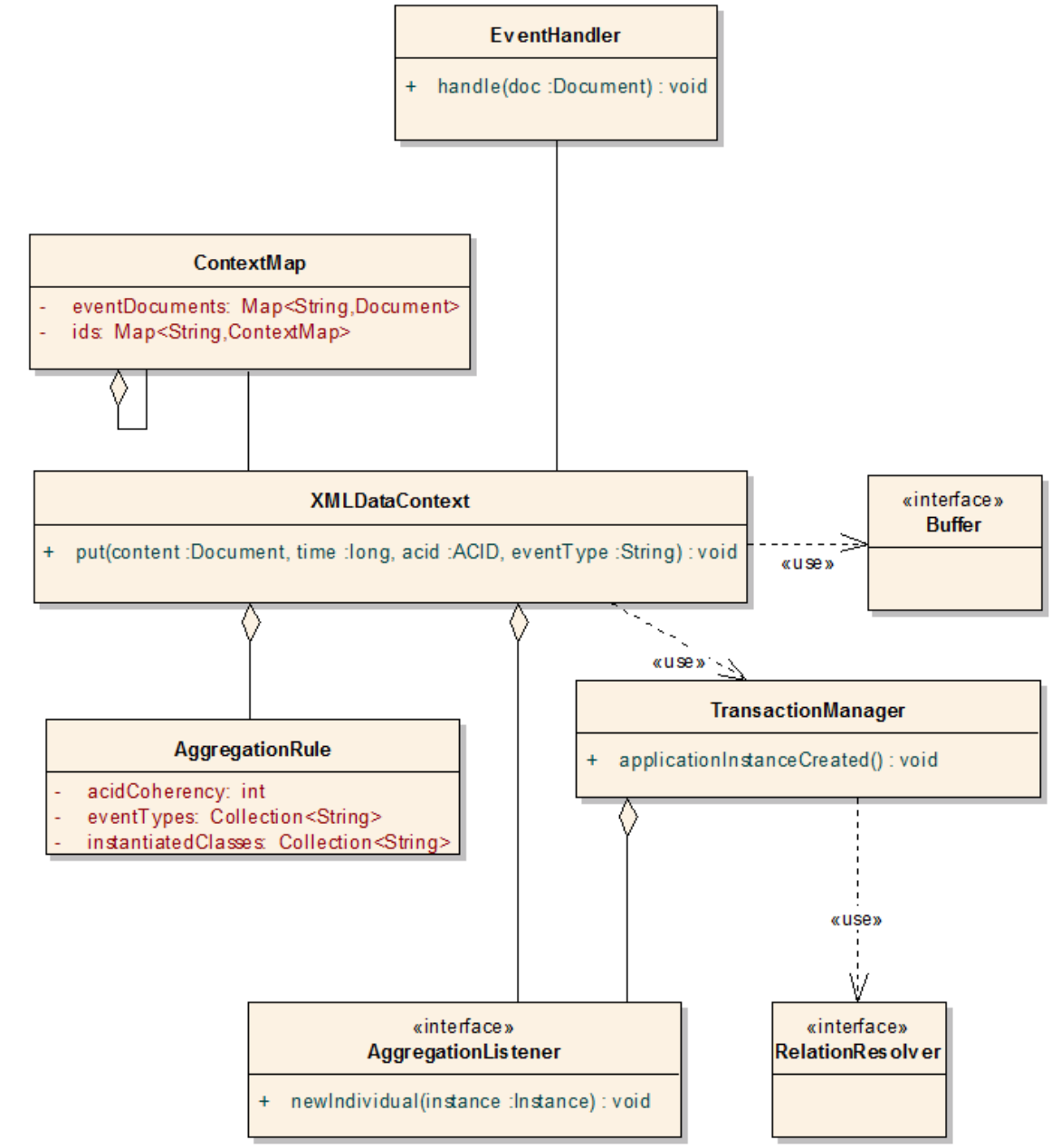


Figure 6.2: *EventHandler* architecture. Collected XML data is monitored in terms of aggregation rule fulfilment. Whenever new experiment transaction is committed, associations between individuals are established by the means of strategy implementing *RelationResolver* interface.

about their type, time of creation and associated ACID is extracted. Based on this metadata, the event is served in a proper way in the XML context.

The context is associated with the components realizing functionality described in sections 5.4 and 5.5: *Buffer* and *TransactionManager*.

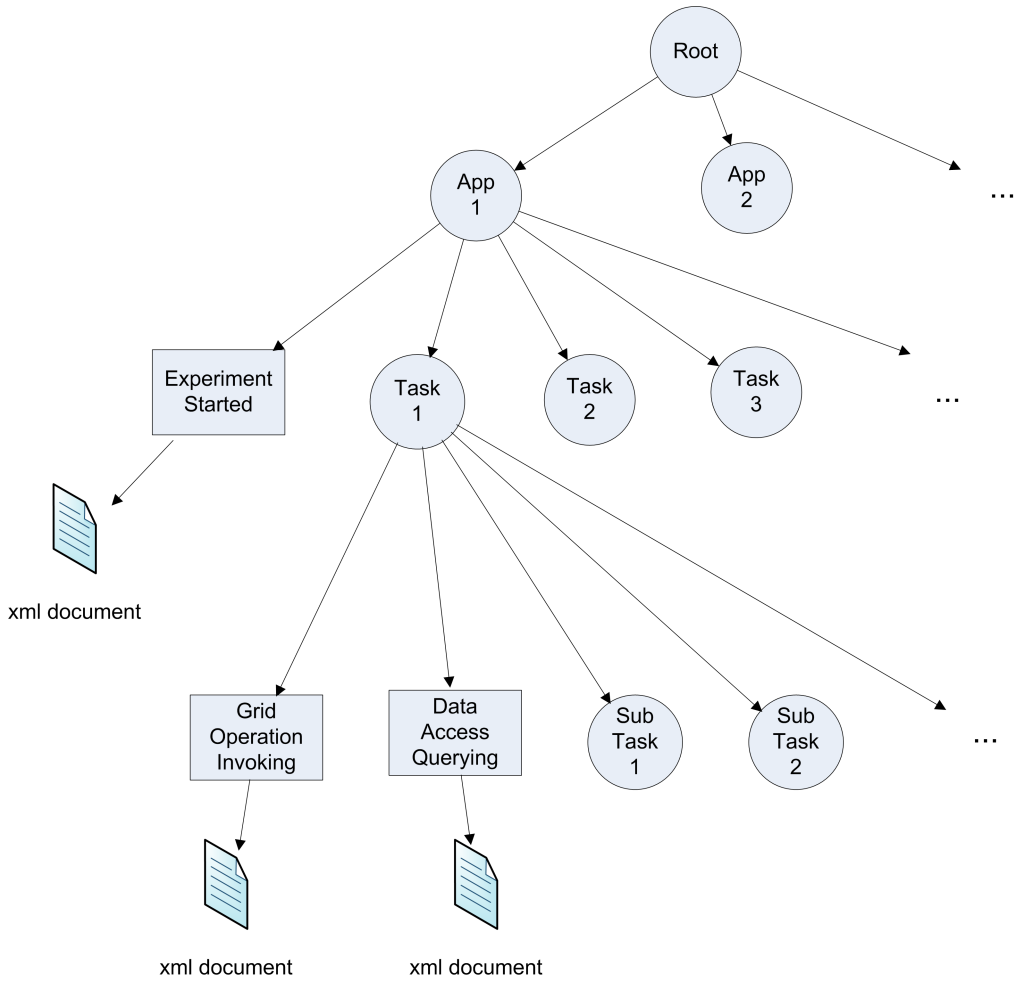


Figure 6.3: XML Data Context comprises all data that have been collected and correlated but not yet aggregated. Tree structure provides efficient access, in which XML documents are addressed through ACID identifier.

Buffer is organized as a set of separated internal buffers, each one dedicated for another ontological class, to provide a high efficiency of navigation through the collected individuals. It should store all needed individuals not only in memory, but also, in a serialized form, in separated data storage. Thanks to this, the Aggregator can be restarted and still have up-to-date information about ontology collected in PROToS.

Another important issue is that the buffer may be accessed both by XMLDataContext, as well as by the delegates, what is explained in section 5.5.2. However, having on mind that the delegates are organized as separated pieces of code which origin is untrusted, the buffer is designed to implement two separate interfaces dedicated for individuals producer and consumer, mostly for the security reasons. They are presented in Fig. 6.4 The consumer interface is accessible by delegate, so it uses the

buffer in *read-only* mode, while both producer and consumer interface are accessible by the XMLDataContext.

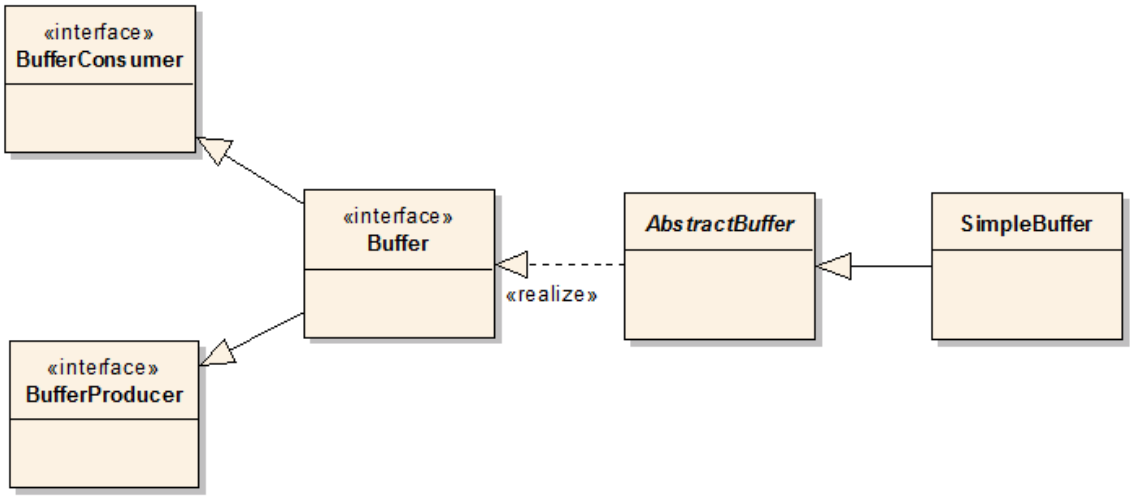


Figure 6.4: Individual buffer interfaces. All individuals sent to PROToS are stored locally using *BufferProducer*. *BufferConsumer* is accessed by delegates.

The next Fig. 6.5 presents some lower-level aspects of individual creation:
There exist tree factories creating individuals from three difference sources:

- *XMLInstanceFactory* creates individuals from generic events
- *OntologicalInstanceFactory* created individuals from domain events
- *DataInstanceFactory* creates additional data ontology individuals. In a case a domain event contains identifiers of its input and output pieces of data, there should be created separated mapping individuals having *dasID* property pointing to the localizations in Data Access or WebDAV and this data individuals must be associated with the domain individual. This feature is explained in more details in section refsc:query-processing.

The *ValueResolver* is responsible for the establishment of a single individual property. It is associated with the property derivation and searches the XML documents or calls the proper delegate. The delegates are instantiated and invoked in runtime, utilizing *Java Reflection* [29] API capabilities. If a particular delegate uses the buffer, it is constructed with buffer reference as a parameter.

6.2. Aggregator deployment

Aggregator, unlike most services in VLvl, does not function as a web application. The only existing dependency is that it must be accessible by monitoring system in

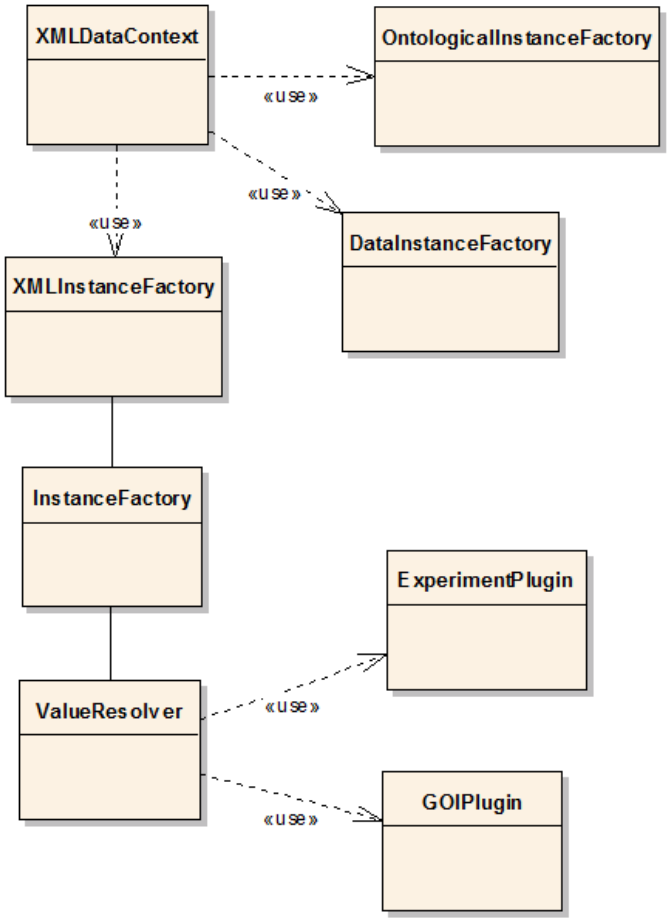


Figure 6.5: Factories create individuals from three different sources of data: generic events, domain events and DAS identifiers. All delegates are instantiated and invoked in *ValueResolver*. At the current stage of development, only Java implementation is supported.

a P/S consumer mode. As for a standalone application, the best approach to deploy Aggregator is the usage of *One-JAR* framework [26]. One-JAR applications utilize a dedicated classloader, which accesses JAR files and resources from inside an application JAR file. Thanks to this, the Aggregator may be delivered as a single JAR, what is convenient in terms of deployment and mimimizes the problem of libraries dependency.

A significant problem to be solved was the integration of One-JAR application with the remote Log4j infrastructure. The Aggregator and the Log4j TCP service ought to be activated separately. The run command for the One-JAR application would be:

```
1 java -jar aggregator-1.0.0.jar
```

while the run command for the Log4j socket server would be slightly different:

```
java -classpath <libraries> org.apache.log4j.net.SocketServer <port>
<arguments>
```

However, based on the attributes defined in a configuration file, the Log4j server may instantiate all appenders, including the *AggregatorCoreAdapter* class. Thanks to this, the TCP server and the Aggregator would share the same virtual machine. Nonetheless, classloader used by Log4j is not able to read JAR files from inside the Aggregagor JAR.

Therefore, there were applied a hybrid solution using the *role delegation* pattern. A *main* method executed by One-JAR classloader does not instantiate *AggregationCoreAdapter*, but explicitly runs the Log4j server passing all command arguments. In such an approach, the Aggregator may be initialized as One-JAR application, but in a manner the Log4j server is initialized. All the executing command are enclosed in ANT [32] script, integrated with Maven framework [33]. All steps, from development to activation, are presented in Fig. 6.6.

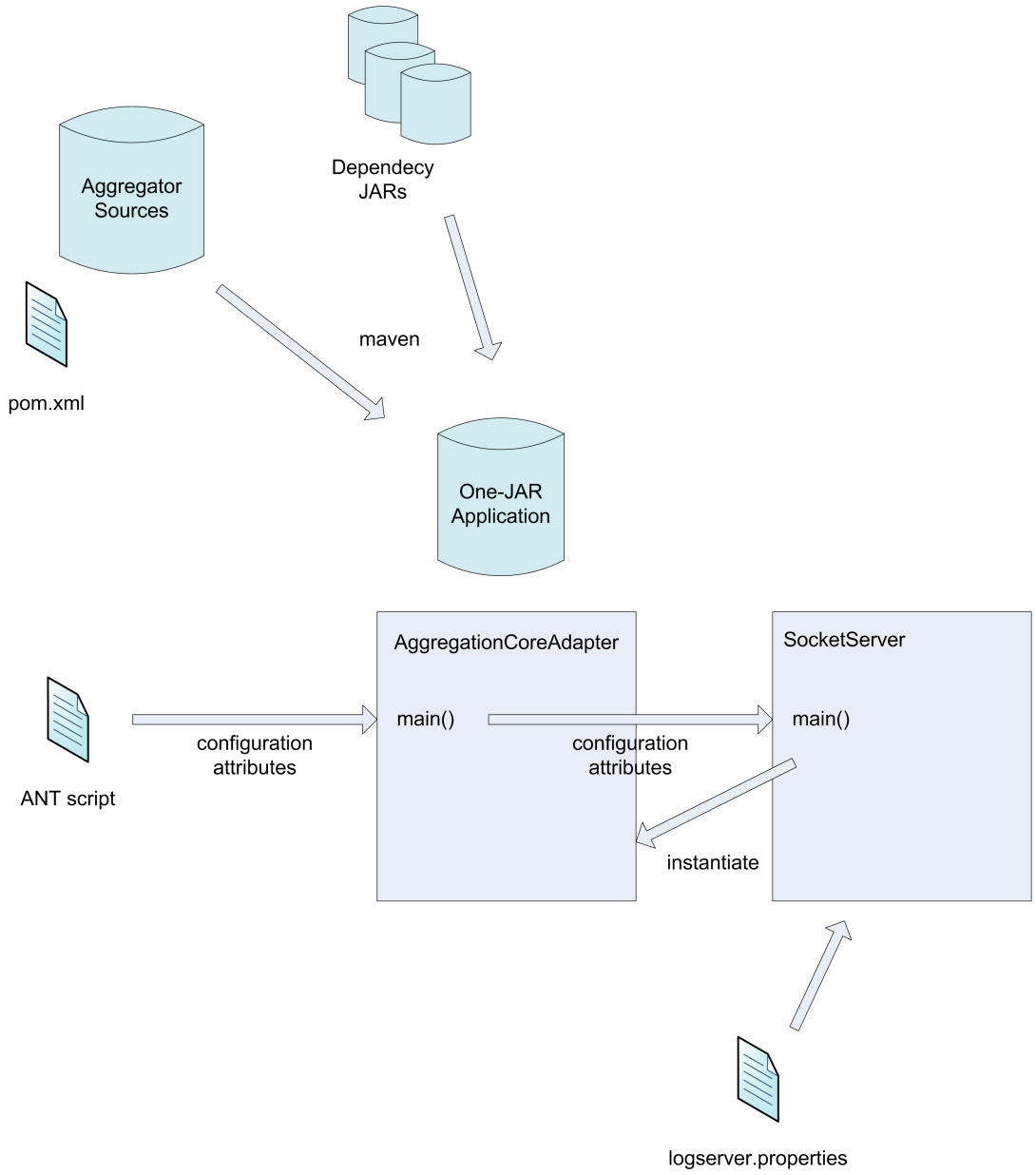


Figure 6.6: Semantic Event Aggregator deployment. Sources are compiled by Maven to One-JAR application. When ANT script initializes the application, Log4j *SocketServer* starts listening. This is one of the possible deployment scenarios, in which Log4j adapter is used.

Chapter 7

Proof of concept: A drug resistance case study

This chapter is devoted for feasibility study. There are discussed main aspects of HIV virus treatment. Geno2drs, a prototype experiment scenario integrated in ViroLab, is presented together with medical tools involved in this research. Next, domain ontology is visualized and commented. Finally, there is given a comprehensive explanation of how the provenance infrastructure works during the geno2drs script execution.

7.1. Geno2drs scientific landscape

Prototype scenario that this paper is based on and which defines this thesis scope, concerns the HIV virus treatment. HIV virus therapy is a very complicated and vulnerable process. The first activity applied to infected patient is to take his blood sample and then extract from this sample RNA code of virus mutation. RNA code is equivalent for DNA code in human body and decodes the virus genotype. Then, based on this genotype, patient undergoes a therapy in which he is given a combinations of drugs. Each kind of antiretroviral drug suppresses the virus for father spreading, however, efficiency of a drug depends on the virus mutation, because viruses differing in genotype presents different levels of resistance for a particular drug. It is crucial to select the best combinations of drugs for two significant reasons - therapy is very invasive for the human body and very expensive. The situation is even more complicated, because a concrete kind of virus continuously evolves, therefore a virologist

must permanently control new mutations and react as quickly and accurately as possible. Fortunately, there exist many implementations of algorithms which compute the resistance of a particular mutation for a particular drug. Virus *genotype* is constituted by the sequence of nucleotides. It is represented in data bases as a combination of four basic elements: *adenine (A), cytosine (C), guanine (G) or thymine (U)*. Unlike the genotype, a *mutation* is represented as a list of dissimilarities between a particular genotype and a generally known genotype serving as a kind of pattern. Each disparity is decoded as a triple– the position on which the difference occurs, a original aminoacid and its substitute. Typically, only a concrete sub-sequence, decoding a concrete protein, such as *Reverse Transciptase (RT)*, is relevant in terms of resistance. This is commonly called a *region*.

Grid Objects deployed in VLvl, which concern the described HIV virus treatment area of concern, are depicted below:

- ***Rega Subtyping Tool*** Computes a virus sub-type based on its genotype.
- ***Rega Alignment Tool*** Aligns a virus genotype and computes a mutation.
- ***Drug Ranking System (DRS)*** Computes the resistance of a particular mutation concerning a concrete region. There exist many algorithms doing this computation, commonly known as *rule sets*, such as *ANRS, HIVDB, HIVDB2*. Naturally, they slightly differ in terms of performance and obtained results.

A prototype experiment is called *geno2drs*. In this experiment, virus genotype is read from DAC. Nucleotide sequence is sub-typed and aligned. Then, obtained mutation is examined by DRS concerning the region RT. Results of the experiment are the virus sub-type and a report of resistances against several most popular drugs.

7.2. Geno2drs ontology

Described scenario require two domain ontologies. The first one describes Drug Ranking System, while the second one describes alignment and subtyping tools. In fact, there is no agreed approach of how to design domain ontologies for newly adapted experiments. They can be designed by the scientist who the script was developed by. Alternatively, they might be created by the services provider, by the people responsible by Semantic Event Aggregator maintanance or generated automatically.

Ontologies, visualized in Fig. 7.1, include three concept, referring to three workflow steps: *NucleotideSequenceAlignment*, and , derived from generic *Computation* class. The fourth workflow step, *DataAccessCall*, in which sequence is obtained from data base, is modelled by experiment ontology. In order to record the provenance of all types of data sets, the data ontology, already containing *VirusNucleotideSequence* class, was augmented with *VirusNucleotideMutation* and *DrugRanking* concepts.

It has to be emphasized that, as postulated in section 1.2, the deployed system supports different levels of data semantics. This is possible because of the generalization hierarchy used in ontologies. Thanks to this, both development of domain ontology as well as augmentation of data ontology were not necessary. If the domain ontology were missing, there would be recorded abstract *Computation* workflow step instead of domain event. If the data ontology did not contain additional data types, abstract *ViroLabDataEntity*, pointing to WebDAV localization of string data set representation, would be recorded.

7.3. Geno2drs information building

Medical services used in geno2drs experiment are virtualized as Grid Objects specified in Tab. 7.1.

Grid Object	**Operation**	**Input parameters**	**Output parameters**
regadb.RegaHivSubtype	*subtype*	nucleotide sequence	virus subtype
regadb.RegaAlignment	*align*	nucleotide sequence region	mutations
org.virolab. *DrugRankingSystem2*	*drs*	rule set region mutations	drugs resistances

Table 7.1: Grid Objects used in geno2drs experiment.

At the moment of services registration, GRR publishes series of events *GridObject*, *GridObjectImplementation* and *GridObjectInstance*. These events are delivered to Semantic Event Aggregator, translated into ontological individuals and passed to PROToS, so that complete semantic description of available resources is present. Before the geno2drs script enactment, GSEngine publishes event *ApplicationStarted*. In the first workflow step, the nucleotide sequences of given patient are obtained from remote data base:

```
1 rdb = DACConnector.
2   new("mysql","virolab.cyfronet.pl","test","testuser","")
3 sequences = rdb.executeQuery("select nucleotides from nt_sequence;")#)
4   where patient_ii=#{patientID.to_s};")
```

Simultaneously, GSEngine should publish events *DataAccessQuerying* and *DataAccessQuered*. In the second step, alignment is applied to a particular sequence:

```
1 regaDBMutationsTool = GObj.create('regadb.RegaAlignment')
2 mutations = regaDBMutationsTool.align(sequences[1], 'RT')
```

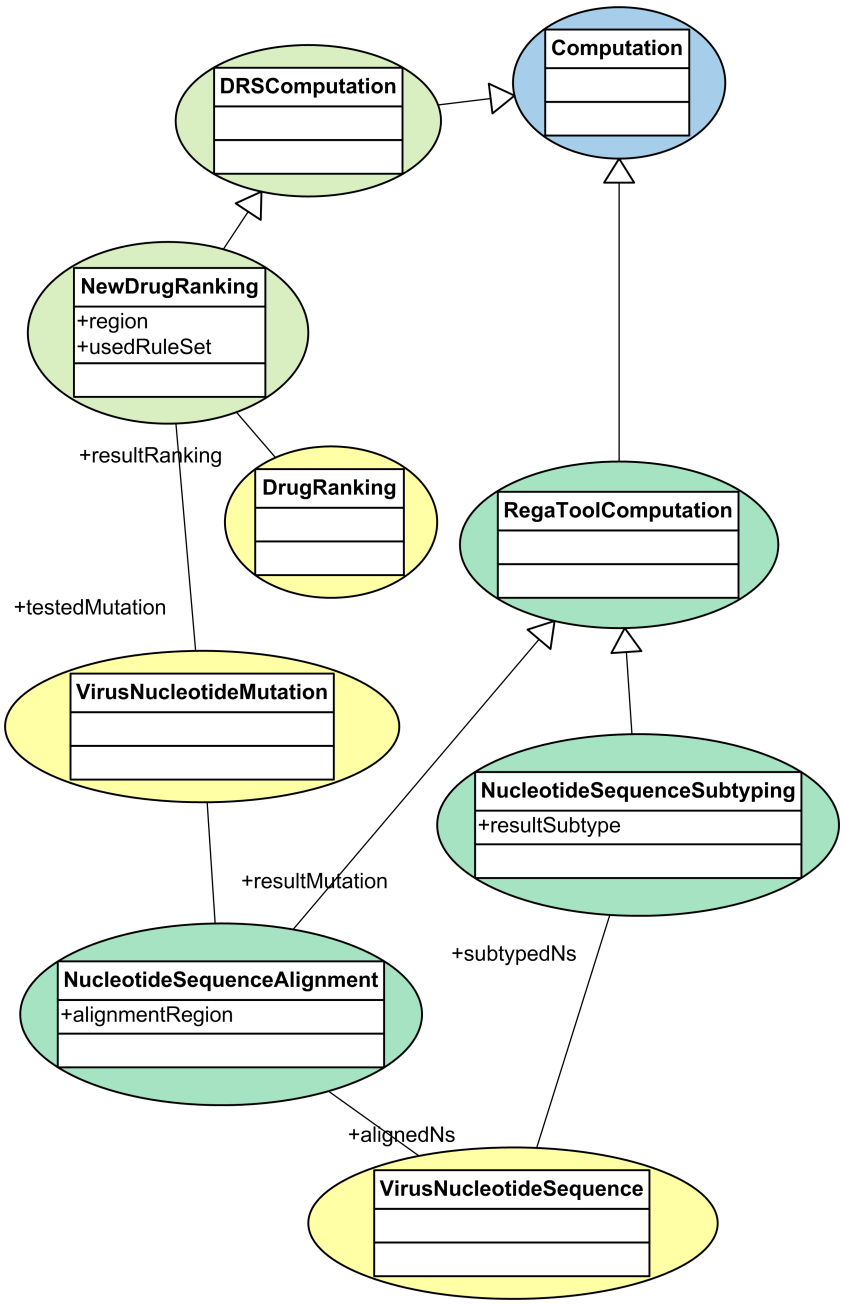


Figure 7.1: DRS and Geno2DRS ontologies. Domain events derive from abstract *Computation*. Dependencies of workflow steps are determined by input and output data types. Partial and final results are modeled through data ontology individuals pointing to WebDAV or Data Access localizations.

Similarly like in the next two steps, two events are generated, at the beginning and at the end moment of Grid Object call, *GridOperationInvoking* and *GridOperationInvoked*, respectively. These two events are correlated, according to ACID coherency. After the aggregation they are translated into *Computation* ontology individual. How-

ever, this event is not delivered to PROToS, but temporarily stored in Aggregator buffer. In the third workflow step, virus subtype is detected:

```
regaDBSubtypingTool = GObj.create('regadb.RegaHivSubtype')
subtype = regaDBSubtypingTool.subtype(sequences[0])#sequences[0])
```

In the last step, drug ranking concerning the obtained mutation is computed:

```
drs = GObj.create('org.virolab.DrugRankingSystem')
puts drs.drs('retrogram', region, 100, mut)
```

When the experiment finishes, GSEngine publishes event *ApplicationFinished*. The events *ApplicationStarted* and *ApplicationFinished* are correlated and aggregated. After this, *Experiment* individual is created and delivered to PROToS, together with four associated individuals. That results in presence of a complete experiment provenance. Addtionally, if the method *markAsResult* is applied to result drug ranking, in order to store the ranking in WebDAV repository as experiment result, individual is created and associated with the experiment.

Naturally, it is possible to augment the semantics of experiment record. There should be used a library automatically generated from the domain ontologies (the library generation process is described in chapter 9.2):

```
require 'java'
newDrugRanking =
  cyfronet.gridspace.mring.dr.creation.ont.drs.NewDrugRanking.new()
newDrugRanking.setNucleotideMutationID(mutationDAV_id)
```

After delivering of events *NewDrugRanking*, *NucleotideSequenceAlignment* and *NucleotideSequenceSubtyping*, corresponding domain individuals as well as data individuals *VirusNucleotideSequence*, *VirusNucleotideMutation*, *DrugRanking* are created and recorded. Domain individuals replace *Computation* individuals and inherit all properties, including experiment association. A signicant simplification of this process is planned. Both storage of partial results in WebDAV and obtaining their identifiers as well as generation of domain events should be realized automatically, transparent for the script developer.

Chapter 8

Querying over provenance

This chapter presents QUaTRO – Query Translation Tool, which enables mining over provenance information mixed with medical data. Requirements for QUaTRO are discussed in terms of importance and feasibility. There is defined Abstract Query Language partially meeting these requirements. Query processing algorithm is explained in details.

8.1. User-oriented querying approach

Capabilities of querying over provenance are tightly related with the requirements collected from the potential end-users. QUaTRO was designed in order to provide the ability of data mining over both provenance describing virological experiments as well as all medical data integrated within VLvl. Virologists and clinicians usually use some query visualization tools or, in a case these tools expressiveness is insufficient, hire data managers preparing some advanced SQL [41] queries. Studies that were undertaken lead to the conclusion, that there is no a tool that would be expressive enough to enable valuable data mining queries, and simple enough to be understandable by non-IT specialists. QUaTRO mission is to become such a tool. Sample provenance queries valuable for the end-users are presented below:

- How experiment result has changed in time? What was the impact of volatile medical data on the obtained results? How quick and in what way does it evolve? What would be the results of experiments applied to the altered data?

- How data from a particular hospital were used? In what experiment, how often, in what medical services, by what scientists?

All of the sample queries are extremely important and cannot be realized by existing, accessible tools. The first group seems to be relevant because of the nature of medical data that is still evaluating during the treatment process. The second group concern clinicians suspectiveness. Medical data is vulnerable and sensitive, therefore a better control of its usage would emphase the hospital doctors to share data with other VLvl participants.

QUaTRO graphical user interface was implemented in portlets technology and integrated in GridSphere portal container [39]. The screenshot in Fig. 8.1 shows how to express, be means of text fields and check boxes, a sample query: *Select all experiments executed by John Doe, in which the tested virus nucleotide sequence was taken from the patient in 2007-04-27 and the computed subtype was A.*

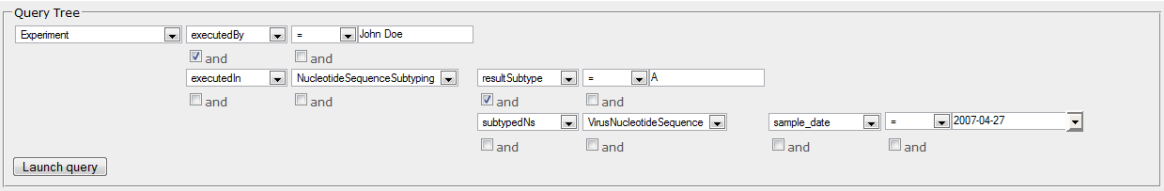


Figure 8.1: Tree-based GUI integrated in GridSphere portlet container. User is able to select ontology classes and properties as well as data base tables and attributes. Query expressiveness may be augmented by the means of logical operators.

Next section explains what query language is reflected be GUI presented in Fig. 8.1.

8.2. Abstract query language

An abstract query language is proposed, which goal is to satisfy criteria presented below:

- **Data representation and storage transparent** The language must support all means of data representation and storage present in VLvl. It should integrate provenance data base, regardless of the underlying technology, relational data bases, regardless of their implementation technologies (for example MySQL [42], PostgreSQL [43]) and file systems, like WebDAV,
- **Understandable by non-IT specialist** Commonly used query languages, those dedicated for relational data bases, as well as those dedicated for XML data bases, such as XQuery [13], RDQL [12], SPARQL [14] present a high level of complexity.

What is more, to construct the queries that are valuable for the researches, one should create a complicated, structural query using advanced language constructions. A higly abstractive query language would be understandable by non-IT specialist and would cover the complexity of low-level languages. Such a language should by comprised of well-known terms: a *concept*, instead of ontological class, rdf node or data base table and a *relation*, instead of object property, rdf tag or data base foreign key,

- **Configurable by ontologies** Because the form of information collected in VLvl highly bases on the ontologies structure, the capabilities of information querying also should depend on metadata. It would be a justified and desirable situation, if appliance of some changes in ontologies directly resulted in altering of the query constructions that can be formed. By the means of ontologies, QUaTRO expressiveness would be, in a semantical way, regulated and adjusted respecting the users community,
- **Easy to integrate with Graphical User Interface** It would be a smart solution, if the query language reflected the Graphical User Interface reasonably accurate. That is valuable from two reasons. Firstly, the query executed by a user is exactly of a form visible in portal, so that the user is aware of how the query terms are related with each other during processing. Secondly, the query model in presentation layer and the query model in business layer are in relation 1:1, what significantly increases the performance, as no additional convertions between these models are needed, and simplifies the QUaTRO architecture, making it more convenient to maintain and less sensitive for the implementation mistakes,
- **Extendible** The language should enable convenient augmentations in farther QUaTRO releases. This feature seems to be especially important, because the collected requirements are still evaluating so the ability of changing the language expressiveness, mostly trough adding some extensions, e.g. additional operators, like *JOIN*, would make the tool functionality more accurate and reasonable.

QUaTRO user, before formulating more sophisticated queries, is probably interested in a single, particular concept, for example Experiment or Virus Nucleotide Sequence. Query constructions would start by selecting an ontological class indicating this concept. Such a query would results in all concept individuals present in PROToS. In the next steps of query construction, the user might put some restrictions on the results. For example, he specify some attribute values that he is interested in, such as a virus type for the nucleotide sequence. Similarly, he might restraint the results only to the sequences being at some particular relationships with another concepts. In this way, the query would be extended in order to reflect more precisely user's

area of concern. That leads to a conclusion, that a most natural, convenient and understandable form of a query is a tree.

The initial concept is a tree root. The nodes refer to ontological classes or concrete literal values, while the edges refer to properties, either ontological or relational. Because of that, both the node and the edge must contain information about the name of the concept or the property, and, additionally, the ontology name. Moreover, in order to make the language more expressive, introduction of some logical operators if justified.

An example of a query expressed in the described language is presented in Fig. 8.2. The tree reflects a typical virology provenance area of concern: *Select all experiments executed by a concrete clinician (e.g. John Doe), in which subtyping operation was applied, and where the nucleotide sequence comes from a blood sample taken in a concrete moment of time (e.g. 2007-04-27) and with a concrete detected virus subtype (e.g. A).*

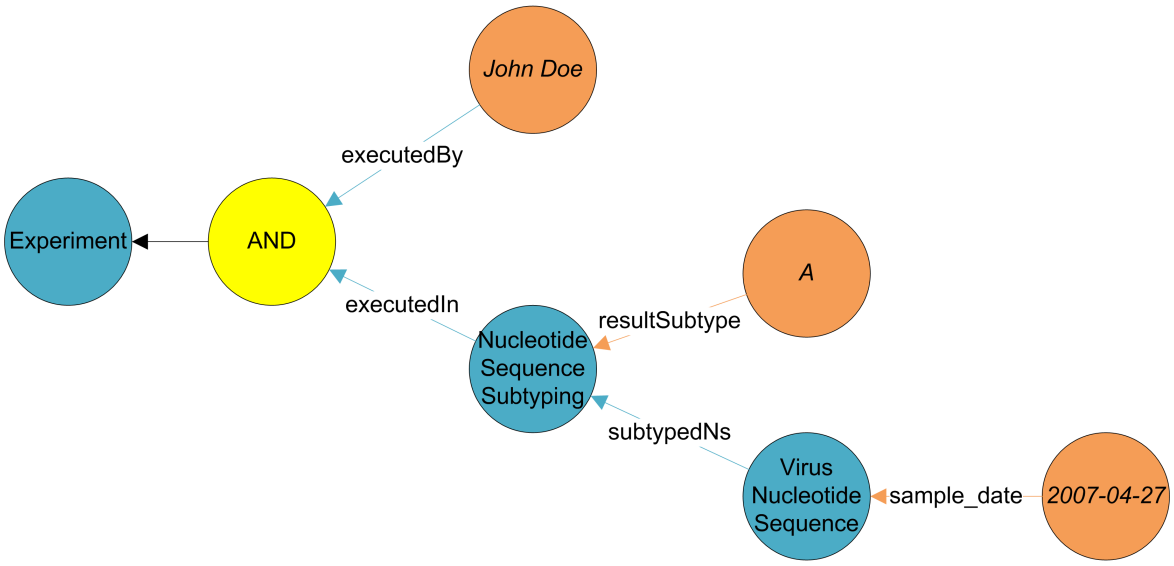


Figure 8.2: Sample abstract query. Let us assume, that a clinician stores in data base HIV genotypes taken for patients one month ago. He is interested how often his colleage from another institution, John Doe, made use of those data. Moreover, he is wondering how the results obtained by John, who uses newly deployed service, differ from earlier experiments' results, in terms of frequency of a concrete subtype in this particular set of samples.

8.3. Query processing

The main idea of proposed algorithm is to decompose the whole query tree into some *sub-queries*. In general, each edge generates a separate sub-query. In the first algorithm step, all *terminal sub-queries* are detected. Terminal sub-queries are those which refer to tree leafs, hence can be processed immediately, because do not depend on the results of another sub-queries. Terminal sub-queries of a sample abstract query are presented in Fig. 8.3.

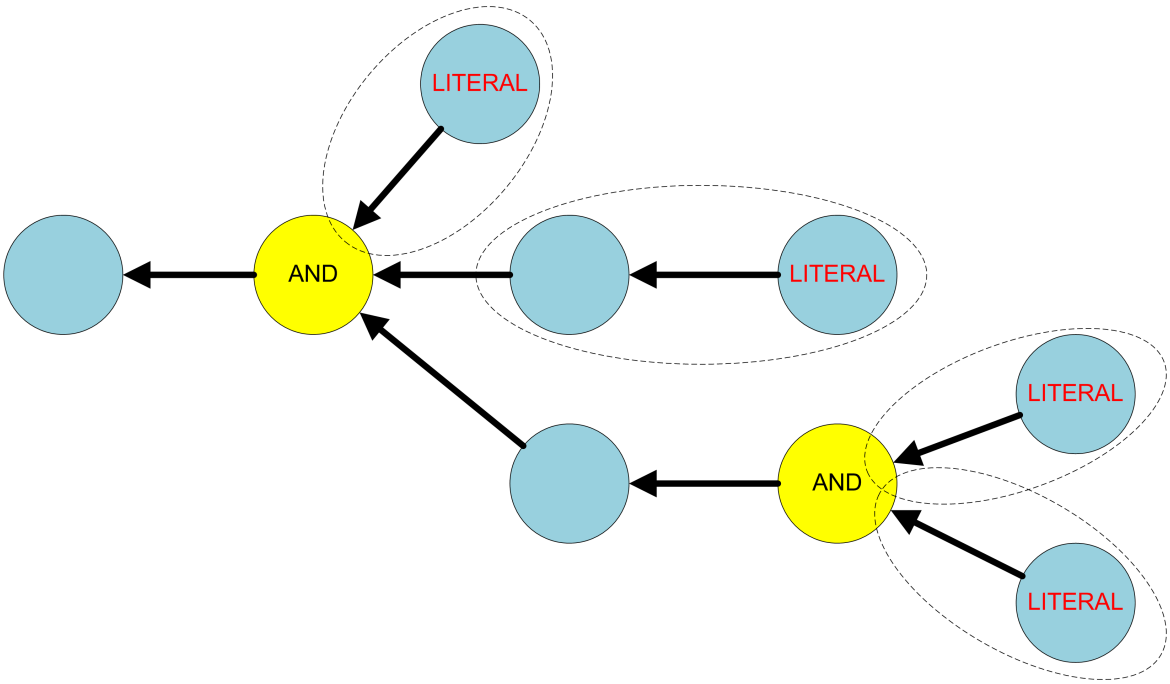


Figure 8.3: Query processing: step 1. Tree leafs contain query parameters. Four terminal sub-quries are detected and evaluated.

In the first step, all terminal sub-queries are executed. The order of execution is irrelevant. In fact, trough providing a proper implementation, all sub-queries may be executed in parallel, what provides a possibility of performance optimization. After this phase, all obtained results are temporarily stored in tree nodes, as in Fig. 8.4.

Please note, that in the ontological nodes complete results are stored, so these nodes are marked as evaluated and may be perceived as new tree leafs. Unlike the ontological nodes, the operator nodes store only the sets of results currently transferred from the connected nodes. The operation of logical *AND* is applied when all sets are present, that means, when all connected sub-queries are evaluated. In the example, exactly two terminal sub-queries are detected. One of the query is ontological, while the second

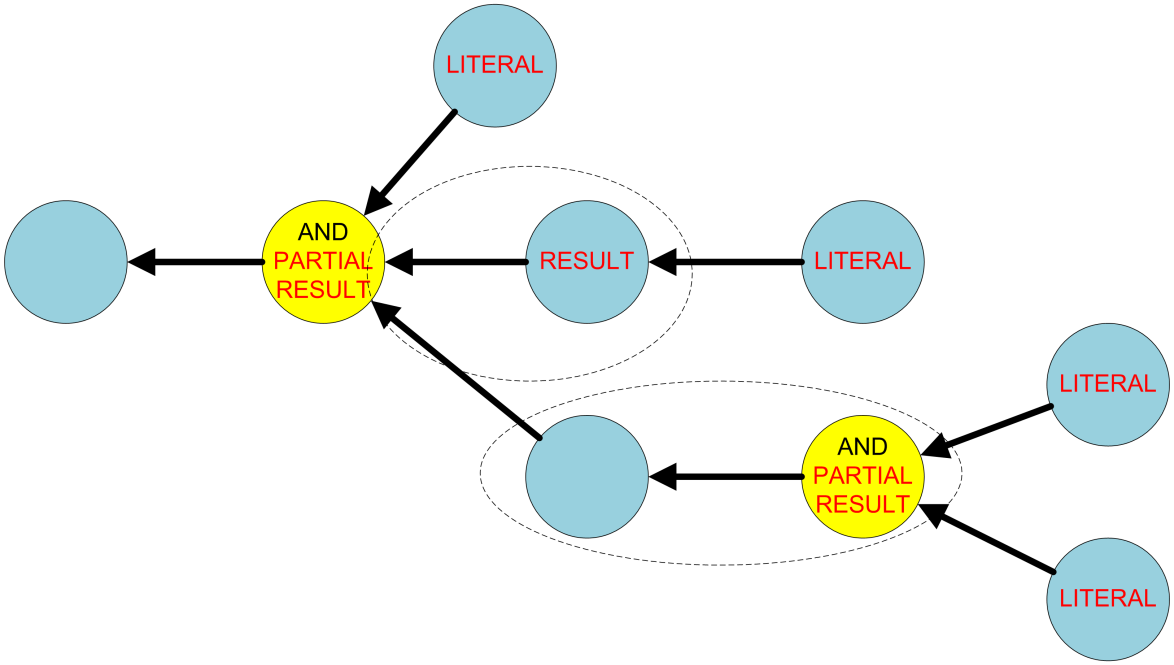


Figure 8.4: Query processing: step 2. Class node contains complete results of sub-tree query. Operator node contains partial results.

one is a logical *and* computation. After their evaluation, in the third algorithm stage, one terminal sub-query is detected, as in Fig. 8.5.

In the last, fourth stage of evaluation, *and* operation is applied to the collected final results, as in Fig. 8.6.

After query evaluation, all tree nodes contain the sub-query results, while the tree root contains result of the whole query, as in Fig. 8.7.

In this approach, one may introduce as many types of logical nodes, as he need, what makes possible a very complicated query constructions, as in the example in Fig. 8.8.

A main component realizing the described algorithm is *Subquery Extractor*. It manages the query tree and utilizes a sub-query stack. In the beginning, all terminal sub-queries are placed on the stack. In the next steps, they are taken from the stack and executed. When a concrete sub-query is finally evaluated, a neighbour of this query upwards the tree hierarchy is put on the stack as a new terminal sub-query. In this way, the whole tree is evaluated recursively. A query processing conceptualization is presented in Fig. 8.9.

The extracted sub-queries are continously passed to *Scheduler*. This component is responsible for detecting of the sub-query type and passing it to a proper *Executor*. Each sub-query type is related with another, dedicated executor. In fact, there might

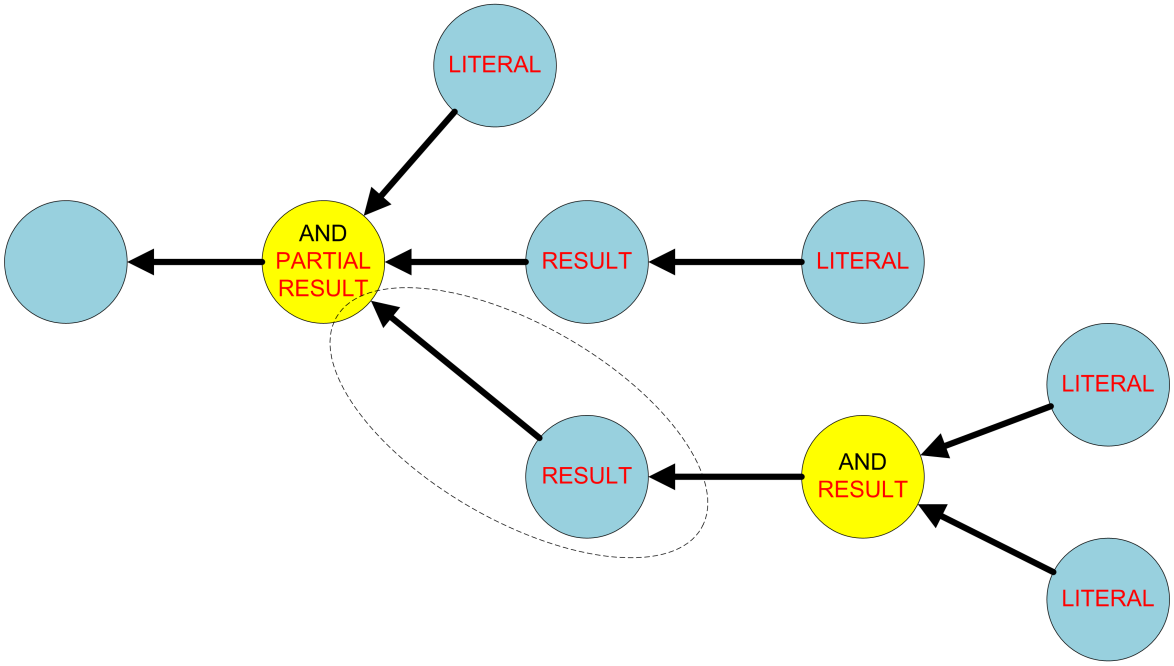


Figure 8.5: Query processing: step 3. The next query may be evaluated when it is no longer blocked by sub-query.

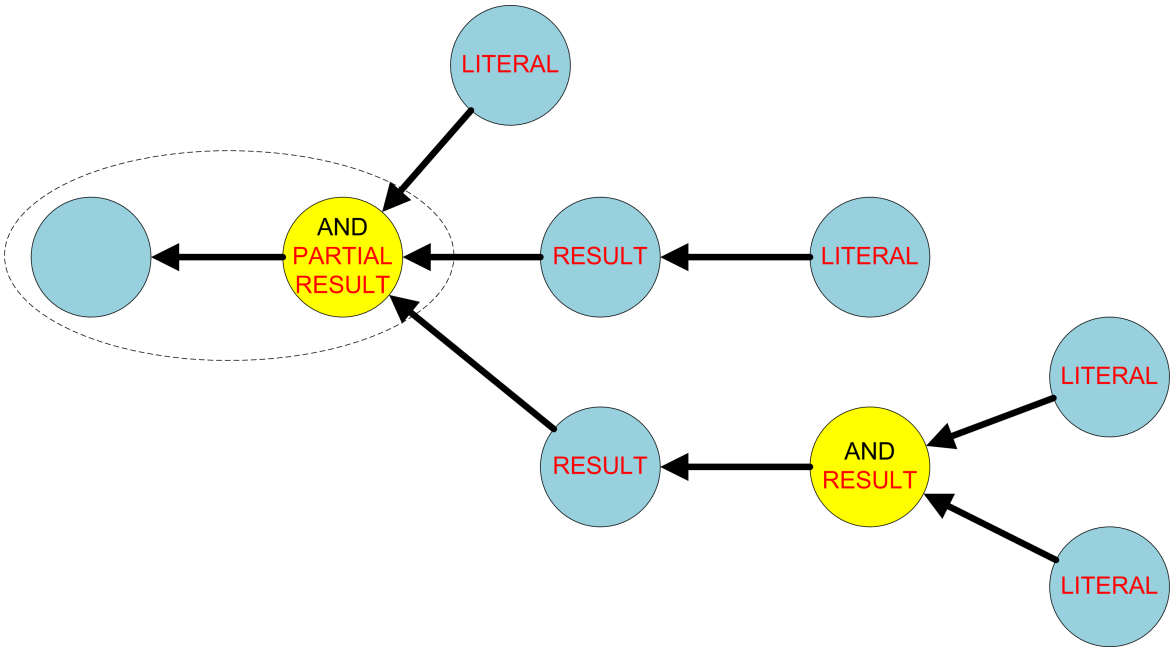


Figure 8.6: Query processing: step 4. In order to evaluate the entire query result, AND operation is applied to three sets of partial results.

exist an executors pool containing many instances of the same executors, what would

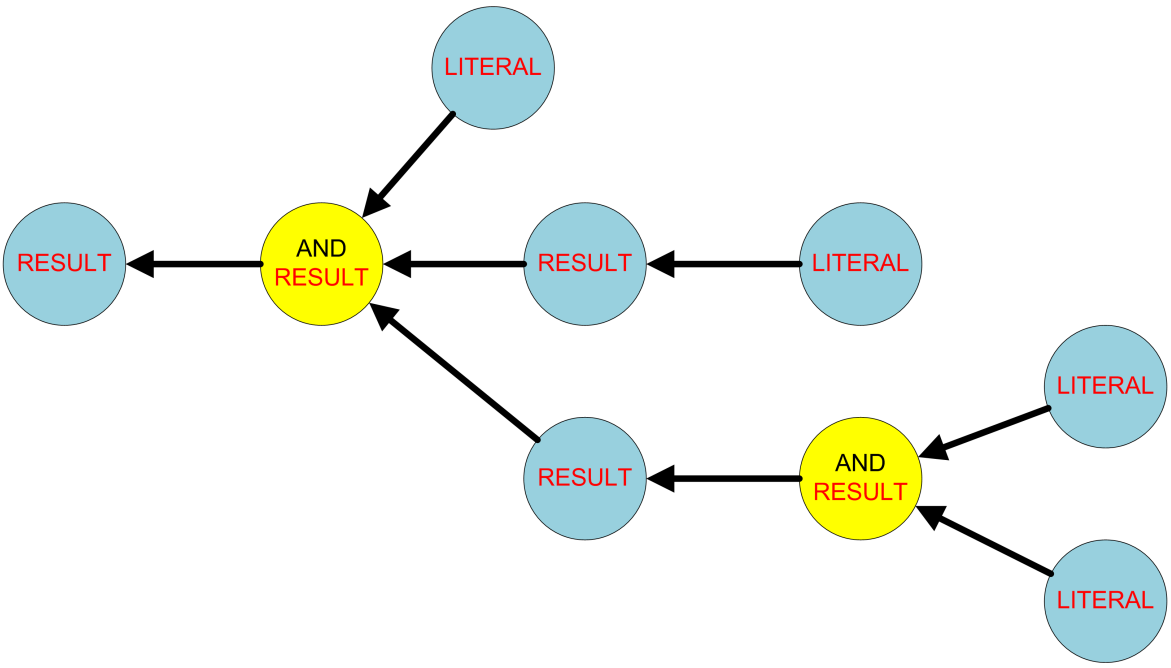


Figure 8.7: Query processing: step 5. At the end of processing, tree root contains result of the whole query.

enable a parallel execution of self-independent sub-queries. Basically, distinguish six types of sub-queries are distinguished. All of this sub-queries must be processed in a different way and through accessing different data repositories. They are described in details in the next sections.

8.3.1. Ontological queries

Ontological queries address only the provenance ontologies. Therefore, the quered concepts come only from the experiment ontology and related domain ontologies. The abstract query language supports several querying languages, enabling optimization through the selection of a language most suitable for the querying domain.

For the first implementation, XQuery language was chosen. *XQueryGenerator* component generates xqueries reflecting the corresponding sub-query. Query results are expected to be sets of identifiers, indicating the individuals satisfying the query criteria. Query construction relies on the XML flat data structure used by PROToS. A sample piece of ontology is given below:

```
1 <j.1:NucleotideSequenceAlignment
2 rdf:about="http://www.virolab.org/onto/geno2drs/nsa1">
3   <alignmentRegion>RT</alignmentRegion>
4   <executedIn>
5     <j.0:Experiment
```

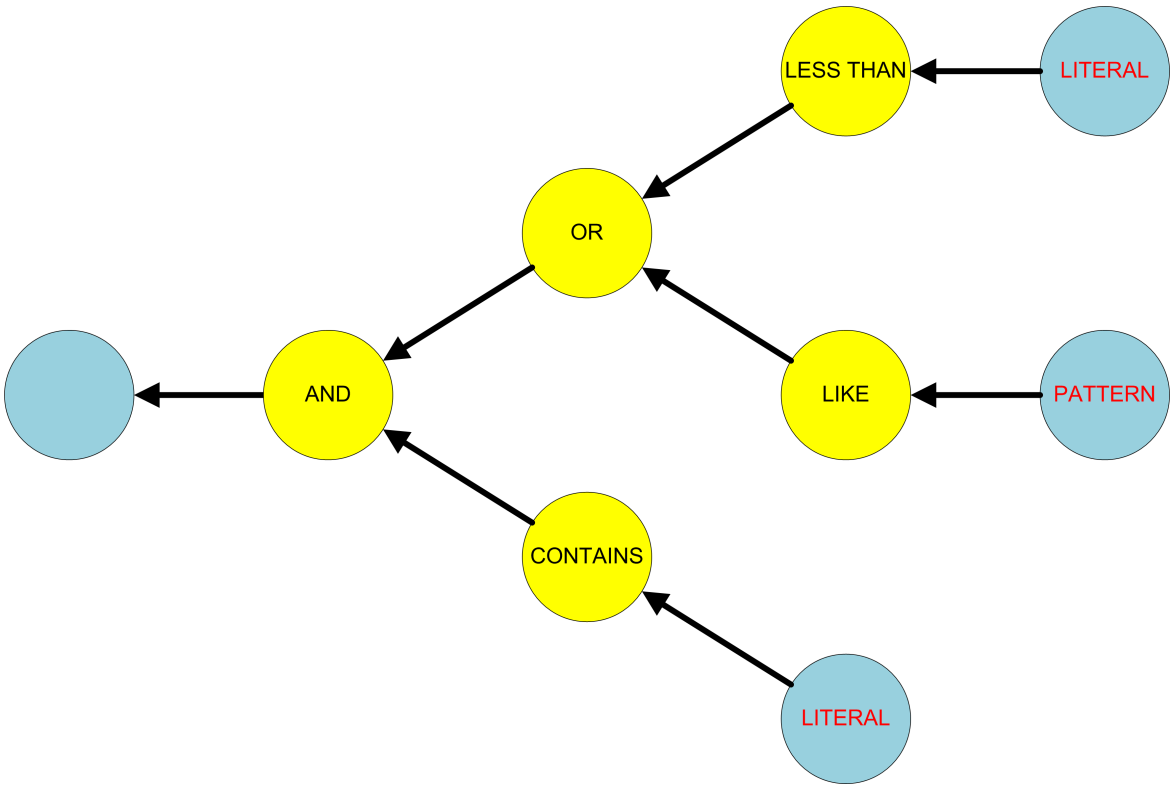


Figure 8.8: Query language operators may be linked with each other forming cascades of nodes. Model may be conveniently extended through addition of new operator node type and implementation of its functionality.

```
    rdf:about="http://www.virolab.org/onto/exp-protos/exp1"/>
  </executedIn>
</j.1:NucleotideSequenceAlignment>
```

According to the ontology structure, the generated queries adressing object and datatype properties are slightly different:

Select all alignments with a region RT

```
//*[local-name() eq 'NucleotideSequenceAlignment' and
((child::*[ name() = 'alignmentRegion' and . eq 'RT']))]
```

Select all alignments executed in a concrete experiment (e.g. exp1)

```
//*[local-name() eq 'NucleotideSequenceAlignment' and
((child::*[ name() = 'executedIn' and (@*[name()='rdf:resource' and
(( . eq http://www.virolab.org/onto/experiment/exp1 )) ]) ])]
```

Naturally, the shape of a generated query depends on the property direction. From the other side, the direction must be transparent for the end-user. There should be

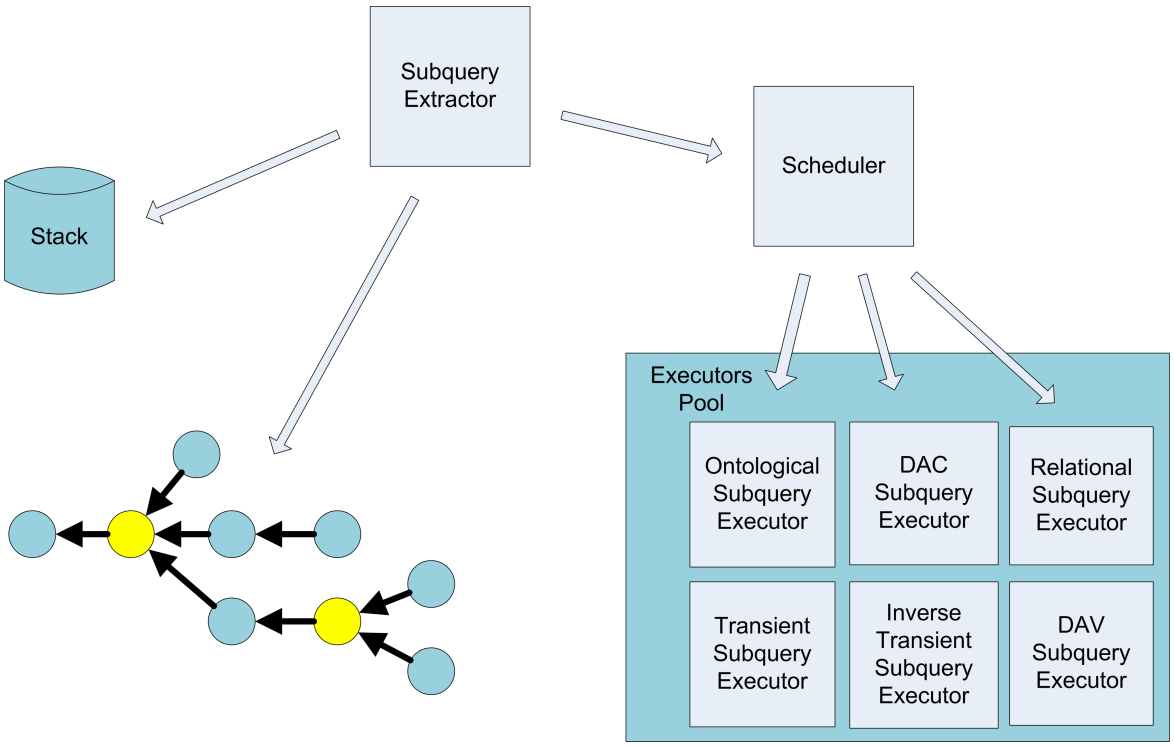


Figure 8.9: Query processing conceptualization. Stack contains all terminal sub-queries, that are not yet accomplished. Query executors differ in terms of what data storage is accessed and what kind of information is retrieved.

possible both type of queries: *Select all experiments in which a concrete alignment was applied* or *Select all alignments applied in a concrete experiment*. However, the *executedIn* property, like all OWL properties, has concrete domain and range, which together determine its direction. Therefore, before the query execution, the property direction is checked. If a sub-query root is the property domain, then the query is generated as in the presented example. Otherwise, the property must be inversed, in order not to query the root concept, but the concepts from sub-query leaf. For query *Select all experiments with a concrete alignment (e.g. nsa1)* the generated xquery would be:

```
1 //*[local-name() eq 'NucleotideSequenceAlignment' and
2 ((child::*[ name() = 'ID' and . eq 'nsa1']))]
```

The retuned individuals undergo farther processing in Jena environment. The identifiers of experiments are extracted and set as query results.

8.3.2. Data base queries

DAC queries address, unlike the ontological queries, values stored directly in data bases. As described in section 2.4, all data bases accessible in VLvl are integrated upon a unified relational schema. Basically, two levels of data storage addressing are distinguished – a *data source*, which is a set of all data bases managed by a single VO member, and a concrete data base name working in a particular technology, such as MySQL, MSSQL or PostgreSQL.

Queries concerning data base values, either relations or attributes, must address all data sources accessible by DAC. Therefore, the query is distributed – a separate query is executed over each data base. In order to provide correct farther processing, query results stored in tree nodes are not individual identifiers, but *dasID* values. A *Data Access Client IDentifier (dasID)* unequivocally describes a localization of a piece of data inside Data Access and is defined as a triple *<data_source>:<data_base>:<key>*. Please note, that including of table name in dasID would lead to redundancy, because a mapping ontology already defines what data ontology concepts is stored in what table.

Generated DAC queries are expressed in quasi-SQL language. Its construction depends on the tree structure. To exemplify a difference between ontological sub-queries and DAC sub-queries, let as assume, that the sub-query root contains *RuleSet* class, the edge contains *version* property and the leaf contains value *4.2.8*. Naturally, it refers to a query *Select all rule sets of version 4.2.8*.

If a RuleSet individual was a part of DRS domain ontology, the generated query would be:

```
1 //*[local-name() eq 'RuleSet' and
2 ((child::*[ name() = 'version' and . eq '4.2.8'] ))]
```

In a case RuleSet is a part of data ontology, the processing engine fetches from the mapping ontology information, that RuleSet concept is mapped to table called *rulesets*. Hence, the generated query is:

```
1 SELECT id FROM rulesets WHERE version =  4 .2.8
```

8.3.3. Relational queries

Relational queries are most simple to process. Some logical operations, such as *and, or*, are applied to partial results stored in sub-query leafs. As explained in two preceding sections, the results being compared may be in a form of either individuals or dasIds.

The relation sub-query is extendable. Depending on operator type, query processing manner may differ and some additional optimization means may be applied. To provide a high level of expressiveness, the supported set of operators must contain, besides the logical operators *AND, OR, NOT*, also relational operators *LESS THAN, GREATER THAN, LESS THAN OR EQUAL, GREATER THAN OR EQUAL*, as well as some advanced operators such as *CONTAINS, LIKE*, significantly augmenting QUaTRO functionality.

8.3.4. Transient queries

Concepts coming from data ontologies are *terminal*. That means, they does not introduce any additional object or data type property values. The only defined property is dasID, a unique identifier which points to a concrete localization in DAC service. Because of this, each query tree may be easily decomposed into sub-trees, trough which some refer to ontologies and some others refer to data bases. As described earlier, they differ in a way they are processed. The relation between ontologies and other data storages is presented in Fig. 8.10.

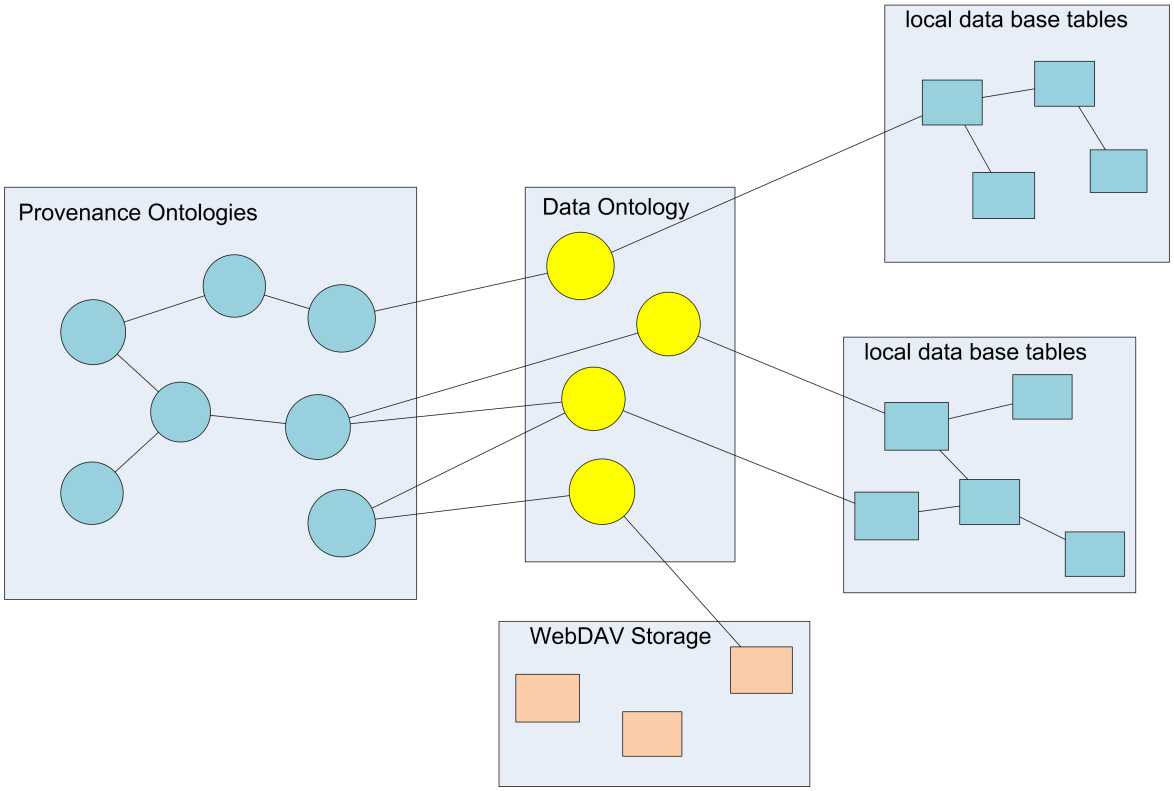


Figure 8.10: Relation beetween ontologies. Individuals from data ontology map data items retrieved, transformed and created in application to localizations in different kind of storages.

Based on this observation, two additional types of queries should be introduced. When a tree node contains a concept, which farther processing addresses ontologies, an individual describing this concept is needed. Alternatively, if the farther processing addresses DAC, dasID value is needed. Individual identifier and dasId value describe the same physical object, however, an individual specifies its localization in ontology, while dasID specifies its localization in data base. Therefore, each data individual is described by two unrelated identifiers:

```
<j.1:VirusNucleotideSequence
rdf:about="http://www.virolab.org/onto/geno2drs/vns1">
  <dasID>rega:mysql:1310</dasID>
</j.1: VirusNucleotideSequence>
```

Because of that reasons, in a case a tree node stores dasID and farther processing addresses the ontologies, a *transition* must be realized. During this transition, based on dasId, a corresponding individual is fetched from PROToS. These kind of query is called a *transient query*.

8.3.5. Inverse transient queries

Similarly to the transient query, an *inverse transient query* is introduced. In a case a tree node stores individual and farther processing addresses the data bases, an em inverse transition is realized. Based on individual identifier, dasID is fetched from PROToS.

8.3.6. WebDAV queries

As presented in Fig. 8.10, the individuals from data ontology define mapping to data base or, optionally, to WebDAV. WebDAV is dedicated for a storage for simple files, hence it is utilized for terminal concept, having no additional relations or attributes. The examples of WedDAV usage are experiment results or partial results collecting. In same cases, if the final results of a query is saved directly in WebDAV, another sub-query must be executed. This concerns another technology, hence another, dedicated executor is needed. In order to support this kind of data repository, dasID has to be augmented with a storage-specific prefix. For the DAC entities, it is extended with *DAC* prefix: *DAC:<data_source>:<data_base>:<key>*, while for the DAV entities, it is extended with *DAV* prefix: *DAV:<uniqure_identifier>*. Adding of yet another type of data storage is convenient and relatively past. There should be introduced a new protocol prefix, besides *DAC* and *DAV*, and implemented a new kind executor.

Chapter 9

Summary and future work

In this chapter the work outcomes are presented and related to the initial requirements. Also, there are discussed plans of how to extend the existing funcionality in future.

9.1. Outcomes

Goals addressed by this work were succesfully achieved in the scope of ViroLab project. They are briefly commented below.

- Developed experiment provenance ontology includes both data provenance and process provenance. It also supports performance optimization and advanced results management. Its design enables easy and convenient extending by domain ontologies.
- Designed monitoring data model includes all data necessary for building complete provenance ontology. There was developed provenance ontology extension and proper aggregation rules were defined.
- Sematic Event Aggregator was deployed and integrated into the monitoring infrastructure. It aggregates monitoring events and creates individuals with proper relationships. Experiment transaction is supported. Some of the associations are determined through the appliance of implemented association strategy while some of them are established by delegates.

- Development process proofs that events aggregation is easily adaptive to evaluating ontologies. Provenance ontology was continuously evolving due to frequent changes of requirements addressing its contents. In spite of this, it was convenient and relatively fast to alter the ontology extension and reconfigure the aggregation proces in order to build new version of ontology.

- Libraries supporting the creation of monitoring events were generated, deployed and are used by several VLvl components. Moreover, monitoring infrastructure dedicated for events transfering was applied.

- Different levels of semantics are supported. Provenance is recorded both for experiments in which domain events are published as well as for experiments whose domain ontology is not yet developed or meaning of utilized data types is not specified.

- Implemented and deployed QUaTRO tool supports complex queries, which include the context of experiment enactment, dependencies between data items, data base attributes and used services. QUaTRO GUI was presented to potential end-users gaining their interest. Available queries are commonly perceived as intuitive and partially meet the requirements presented by clinicians and virologists.

9.2. Research outlook

Future work focuses on augmentation of QUaTRO query model, domain ontologies transparency and extensions of collected provenance ontology, what is explained in more details in following points.

- The most significant QUaTRO disadvantage is the absence of support for *join* operator. It is crucial, because some scientists expect that the query result will not be a single concept, but two or probably more concepts associated with each others. For example, the expected result is a triple (drug, therapy in which this particular drug was used, patient which this particular therapy was applied to). Joining should support both ontological concepts and data base tables.

- Delegates should be used in order to query separate services for additional information. Currently created ontology does not include technical information about services execution. The performance properties of computations should be computed by delegates, which would query monitoring componets for some metrics, such CPU or memory usage, characterizing concrete invocations. It would valuable information for application optimizer enabling more sophisticated algorithms of software selection. Furthermore, information about origin of data items may be

collected. When a concrete piece of data is fetched in experiment, its identifier is sent with corresponding domain event. This identifier may be used by delegate to construct complex SQL query returning, for example, clinic name.

- It would be covenient, if both development and building of domain ontologies would be realized in automatic way transparent to the user. This revers to two cases of ViroLab usage. In the first case, whenever new Grid Object is registed, semantic types of operation input and output parameters are specified. They are selected by the means of data ontology browser. Separate domain event, derived from abstract *Computation* is generated for each operation. Semantic service description might be also passed from resource registry to operation invoker. In another case, operation invoker fetches the description of service semantics from registry. Whenever new invocation is realized, operation invoker stores all input and output parameters in partial results storage, for example WebDAV. Then it creates data individual of particular type for each parameter. Also domain event individual is generated and associated with data objects. Thanks to such approach, the responsibility of domain ontology building would be shifted from script to operation invoker.

Appendix 1. Creation of monitoring events

Creation of monitoring events is supported by Castor framework [25], which provide generation of Java classes from XSD model and mapping between these two representations. Castor is used mostly for historical reasons, since the monitoring system GEMINI and its data representation libraries highly bases on this framework. However, migration to JAXB [36] is considered.

For each complex type defined in XSD file, Castor generates a pair of Java classes. The first class serves as a bean – it exposes setter and getter methods to access all the attributes associated with this type, as well as provides the functionality of marshalling the java object into XML file and parsing the java object from an XML file. The second class is a descriptor being used in the processes of marshalling and unmarshalling. Castor is well-integrated with ANT framework. Thanks to this, the whole process, from the design of XSD data model to the generation of Java API may be utterly automatic and configurable, as in Fig. 9.1

Because during the VLvl maintanance new medical scenarios may be supported

It may be inconvenient to create the domain events described in the preceding chapter. Therefore, there wes implemented a configurable tool dedicated for the automatic generation of helpers API from the ontologies. These helpers are used directly in experiment scripts. The helpers generation and usage process is depicted in the Fig. 9.2

The responsibility of The Event Generation Tool is to process the ontology and for each ontological concept generate Java classes of two types:

- Helper than can be used to generate a XML domain event in a setter-usage manner.
- Meta-class containing semantic annotations [30] used in PROToS component in order to determine the relation between ontological classes, as well as their prop-

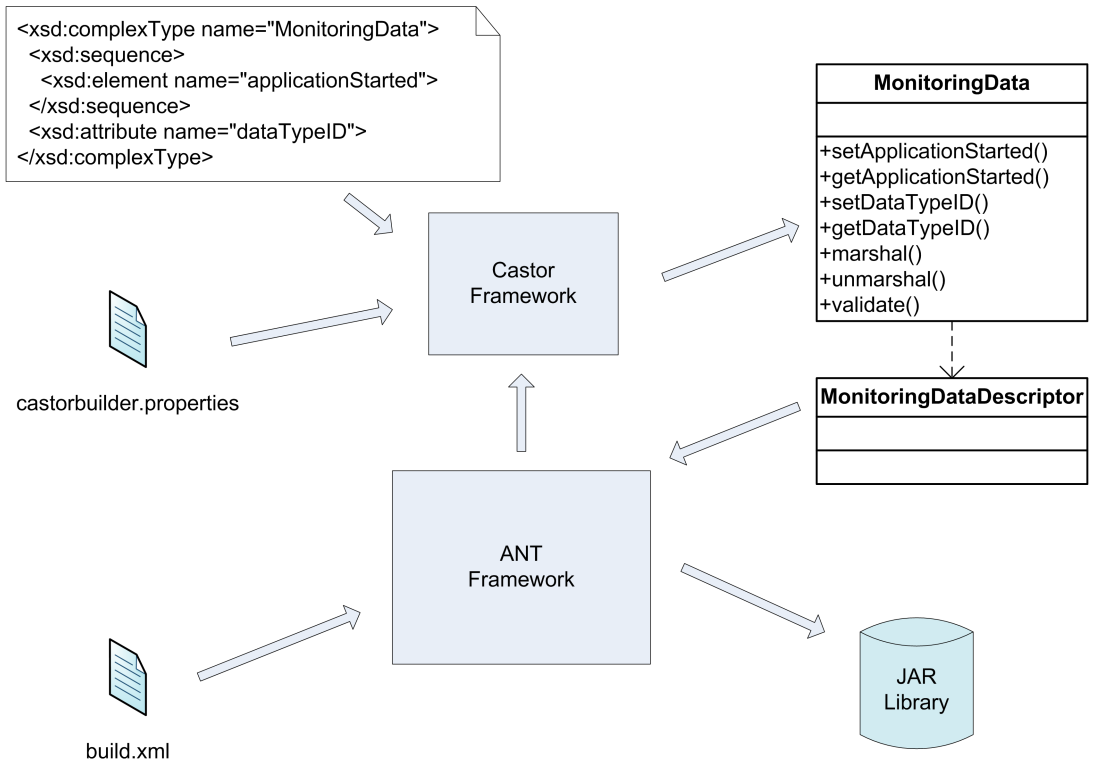


Figure 9.1: Castor framework. Based on XSD schema, it generates helper classes providing serialization and deserialization capabilities. Castor is well integrated with ANT framework.

erties, and the PROToS-specific events passed as parameters to its Web Service interface.

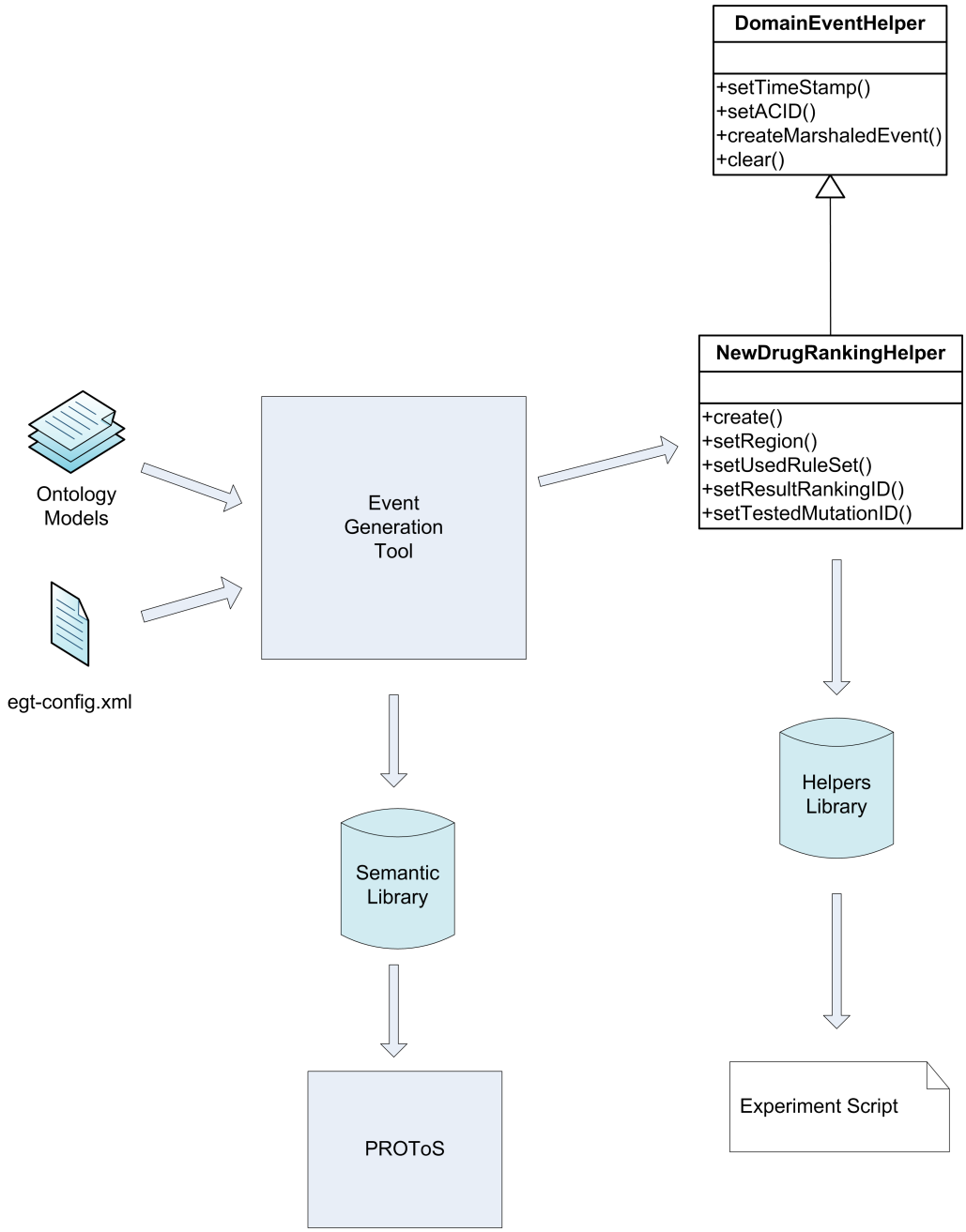


Figure 9.2: Event Generation Tool generates from ontologies two kinds of Java classes. Semantic classes are augmented with semantic annotations used by PROToS during events processing. Helper classes are used in creation of monitoring events.

Appendix 2. Logging of monitoring events

Log4j is a commonly used implementation of *logging*, that means, reporting some events that have occurred in application. This is a natural choice of the simplest and fastly-implementable way of the VLvl components instrumentation.

Log4j architecture comprises two cooperating classes of components: *loggers*, responsible for the creation and sending of logging events and *appenders*, which collect the events and serve them in a way desirable by the user. Both loggers and appenders may by defined either in a configuration file or programmatically. Each new introduced logger is related with some appenders to which it reports the generated events. The most commonly used appenders are *ConsoleAppender* printing the events in the console and *FileAppender* writing the events on a file. It is also a good programming practice to create a separate logger for each Java class.

Logging events can be reported at several severity levels: *trace, debug, info, warn, error*. There is no commonly agreed convention when to use a particular level. It is a good practice to debug the information needed only in the phase of system development, indicating the correct system working, info the information relevant during the system usage, warn some problems which does not disable the farther system functioning and error the unignorable failures. Thanks to the severity levels, there can be defined a *treshhold*, assigned both to loggers and appenders, what makes communication more structural. There was defined logger dedicated for monitoring infrastructure:

```
1 Log4j.logger.mring = INFO, Console, Socket
```

that can be directly accessed by the VLvl components:

```
1 Logger.getLogger(\"mring\").info(helper.createMarshaledEvent());
```

Besides the monitoring logger, there must be also defined a monitoring appender. There was developed a *RemoteAppender* based on the Log4j *SocketAppender*, which

logs the events via TCP sockets. It introduces a crucial improvement in comparison with the original appender, which does not guarantee a successful event deliverance. Therefore, the remote developed appender throws an exception in a case event was not deliver successfully. From the client's point of view, catching of monitoring exception means that the event should be published once again. The second socket is integrated by a Log4j *ServerSocket*. From the server side, there may be defined additional appenders, that would receive the messages and pass them to separate system components, as in Fig. 9.3.

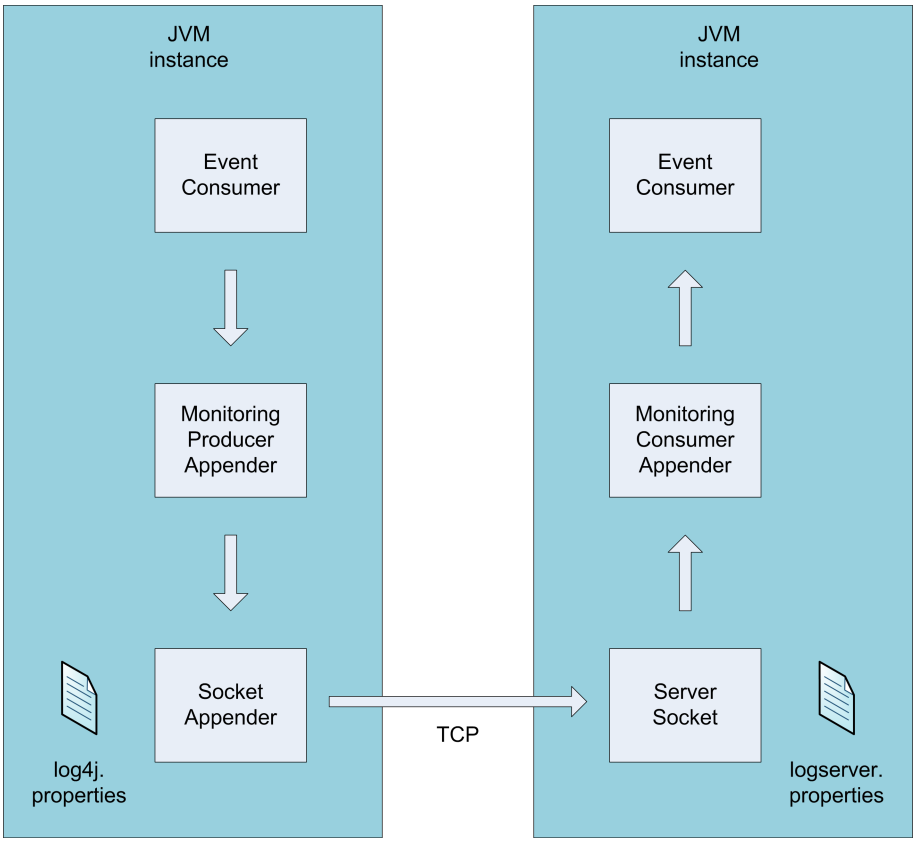


Figure 9.3: Remote logging architecture. Both on the client side and on the server side must be implemented a dedicated appender. Log4j remote communication is realized via TCP sockets.

In the final implementation, the Log4j-specific API is covered by monitoring facade adapting it to the monitoring API.

Appendix 3. GEMINI monitoring system

GEMINI system was developed in the scope of *K-Wf Grid* project [44]. It provides a generic monitoring infrastructure dedicated for knowledge-based workflow grid systems. It integrates all the described monitoring usage scenarios: querying, streaming and P/S. Because GEMINI does not distinguish *monitoring data* and *monitoring event*, the subscribe use case may be perceived as an equivalent of the *request streaming* use case.

The general architecture of GEMINI is presented in Fig. 9.4

GEMINI infrastructure is organized as a network of services called *monitors*. The monitors are connected in a P2P network. Each monitor adapts some sensors, which extract monitoring data from different system components. In practice, a monitor is accessible by Web Service and associates a *Sensor Controller*, which integrates all sensors residing within a single JVM. Each sensor serves as a producer of monitoring data of a concrete data type and a concrete resource.

Basically, three kinds of sensors are distinguished:

- sensor exposing interface for data querying, providing the current value of monitored metric, used in Querying scenario
- sensor pushing the data value in a concrete period of time, used in Streaming scenario
- duplex sensors providing both types of functionality

GEMINI clients realize a concrete scenario trough the subscribing expressed in *Performance Data Query and Subscription (PDQS)* language, which simplified structure is presented in Fig. 9.5.

In K-Wf Grid monitoring data was described semantically and published in *Grid Organizational Memory (GOM)*. The ontology fragment related with monitoring is presented in Fig. 9.6.

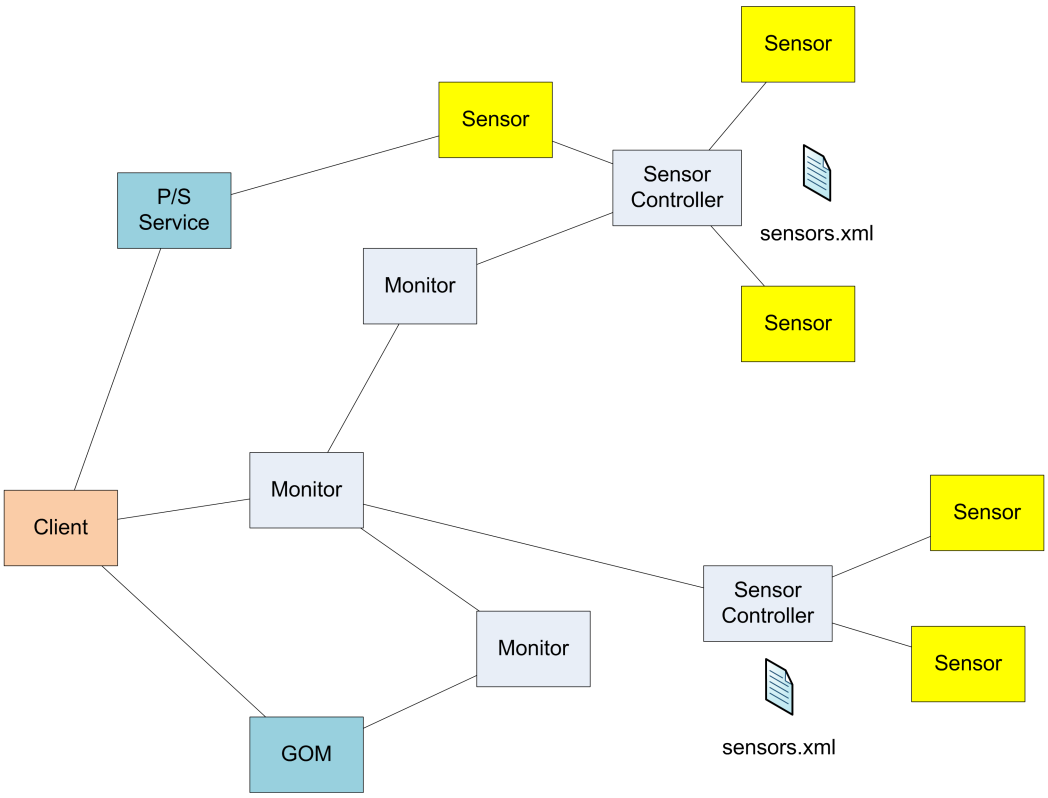


Figure 9.4: GEMINI overall architecture. Each monitor is accessible by client trough its Web Service interface. Monitors, connected with each other via ICE channels, integrate sensors residing on remote machines.

It is easy to notice, that the expressiveness of presented semantics is poor, and this is another reason for the separation of a non-semantic monitoring data from the full-semantic ontologies created from this data.

The inter-communication in GEMINI is realized in ICE [38], a modern object-oriented middleware platform. Like another advanced distributed programming framework, such as *CORBA*, it presents OO-language-independence, through supporting *C++, Java, Python, PHP, C and Visual Basic*. However, ICE, comparing to CORBA, is more efficient, better supports the security and offers more advanced functionality.

A significant part of undertaken studies was to investigate the capabilities of adapting GEMINI P/S functionality to the VLvl monitoring clients requirements. In the solution proposed and implemented in K-Wf Grid, IceStorm [11] is used, a P/S service dedicated for the ICE communication channels. Every Sensor Controller is associated with a particular IceStorm instance. Thanks to this, when a monitoring client subscribes for events of a concrete pair *(dataTypeID, resourceID)*, there is created an ICE connection with the corresponding IceStrom. This approach make the P/S infrastruc-

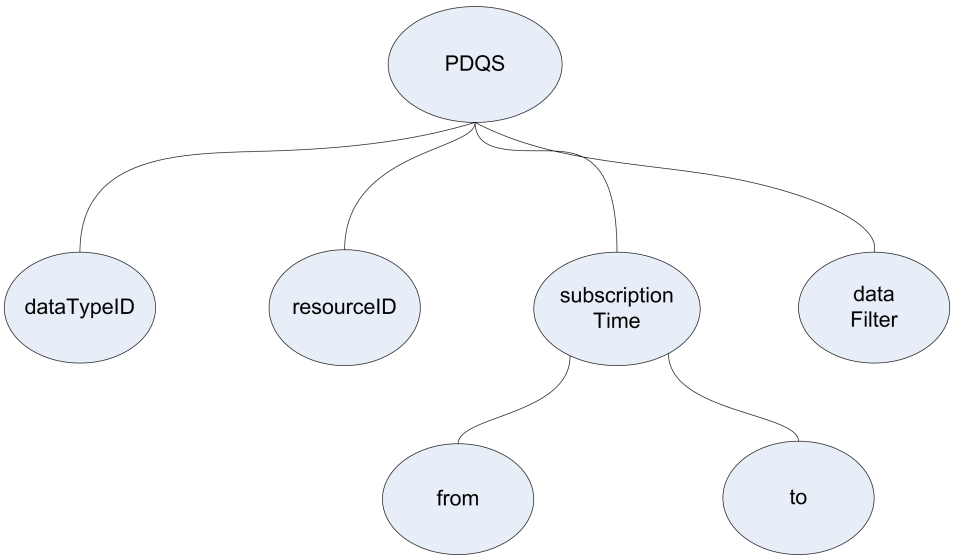


Figure 9.5: PDQS schema. Each piece of monitoring data has concrete type and comes from concrete resource. Schema also supports subscriptions and filtering of delivered data.

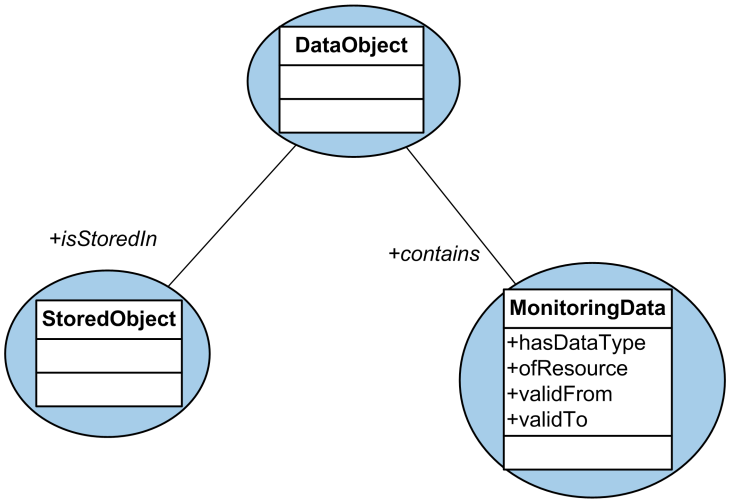


Figure 9.6: Each piece of monitoring data may be described semantically. This approach was dismissed because monitoring data serves only as a temporal representation used in building of highly expressive information.

ture well distributed. In this case, the network efficiency is maximized, because there are created only two ICE channels. The first channel links the Sensor Controller with the nearest IceStorm. The second one links the P/S client directly with this IceStorm instance, as in Fig. 9.7.

Two problems related to this infrastructure were identified. Firstly, it is not op-

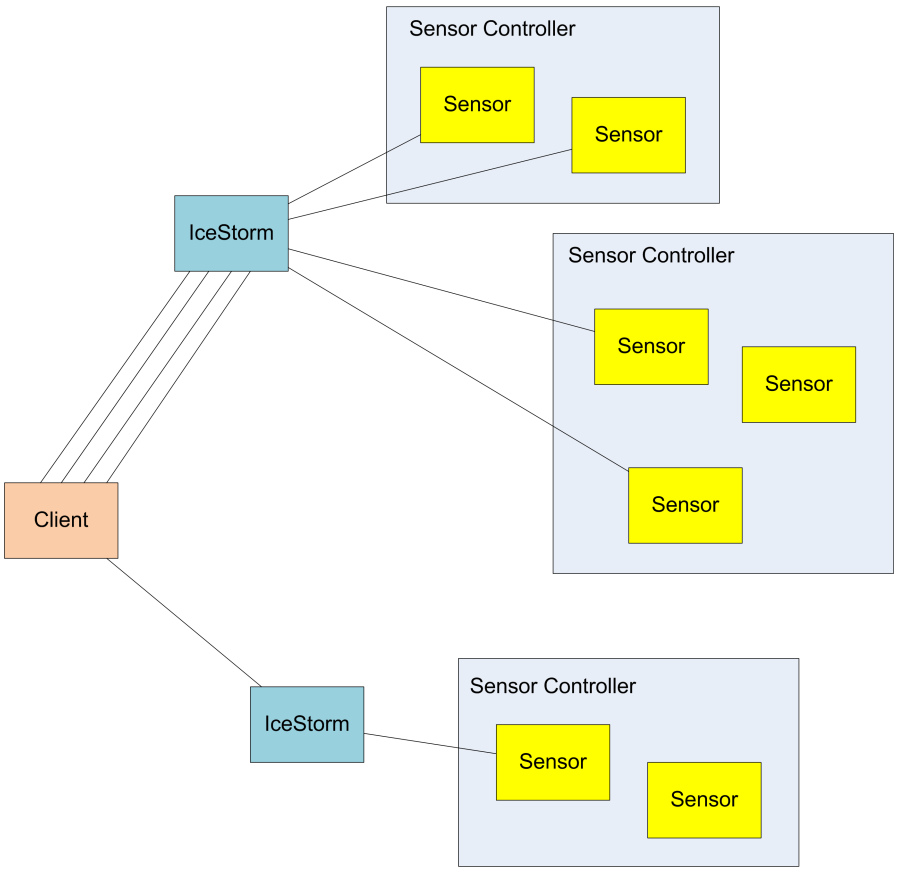


Figure 9.7: Inefficient P/S architecture in GEMINI. There is created a separate ICE channel for each client subscription.

timal in a case a concrete Sensor Controller manages several Sensors which data is especially popular, because there might exist several ICE channels linking the same pair (Sensor Controller, IceStorm). In a similar case, a single client is interested in data of many types, so there might exist several ICE channels linking the same pair (client, IceStorm). To significantly improve the communication efficiency, there were implemented two additional GEMINI components, *ICE multiplexer* and *ICE demultiplexer*, transparently coordinating the passing of events on a client side and Sensor Controller side, respectively. The communication benefits of this approach are presented in Fig. 9.8.

The second significant problem is *resource-independent* subscription. That would be suitable for the clients interested with data of concrete type but regardless of its origin, it means, the *resourceID* would be irrelevant. In fact, such a case is most popular, because it is a typical sutation when P/S client is not aware of the data producers existence and localization. In a moment of subscription, the consumer has

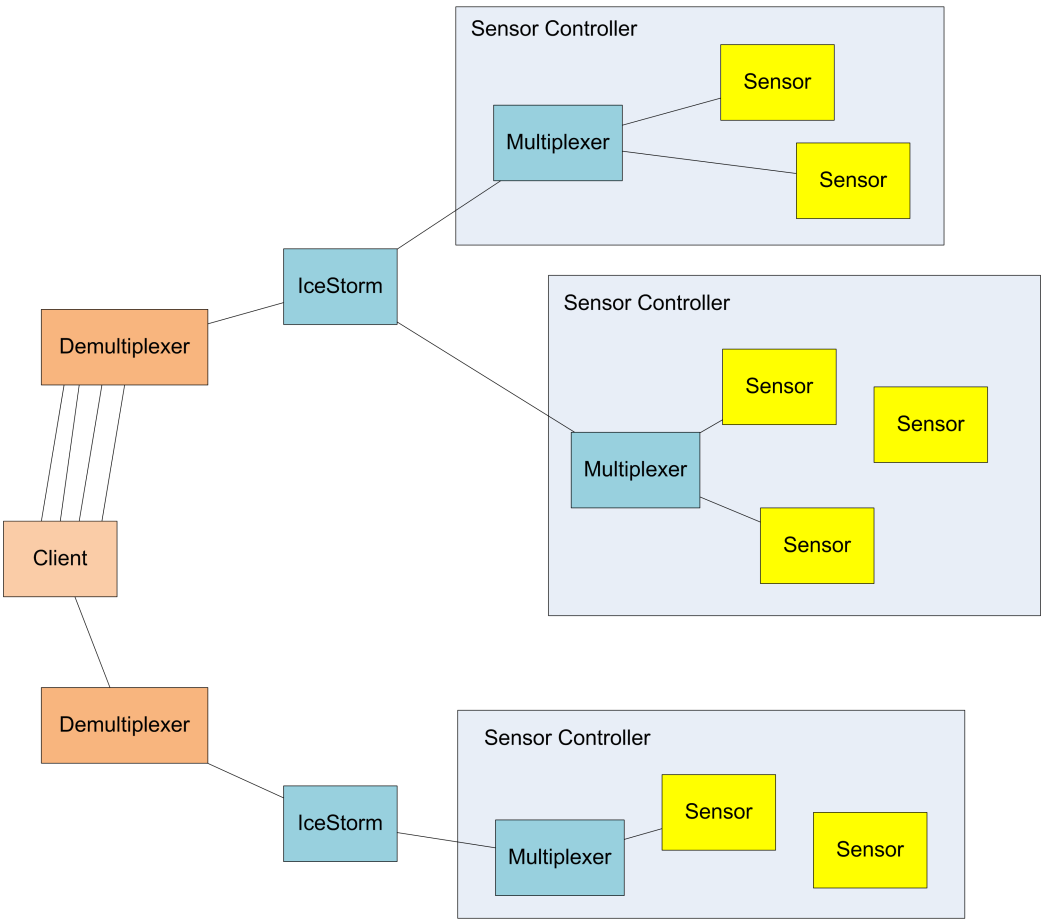


Figure 9.8: ICE channels multiplexing. Events of all types are transfered via a single ICE channel.

no guarantee that a concrete data producer does even exist. To solve this problem, the next two GEMINI additional components should be engaged:

- ***IceStorm Consumer Proxy*** All subscriptions are applied by this proxy component. It may have more than one instance, however, all instances must have a complete information about the localization of all IceStorm services. Thanks to this, the proxy service may be distributed and clients may access the proxies of nearest localization, however, some means providing the coherent IceStorm addressing must be provided. The proxy must pass subscriptions to all P/S services, because a concrete data type may be accessible in each sensor, what is inpredictable, mostly because of the runtime sensors plugining capability. Naturally, between each pair (P/S Consumer Proxy, IceStorm) should exist exact one ICE channel, what is supported by ICE Multiplexer and ICE Demultiplexer, as in Fig. 9.9.
- ***IceStorm Producer Proxy*** IceStorm Consumer Proxy is not aware of what pairs

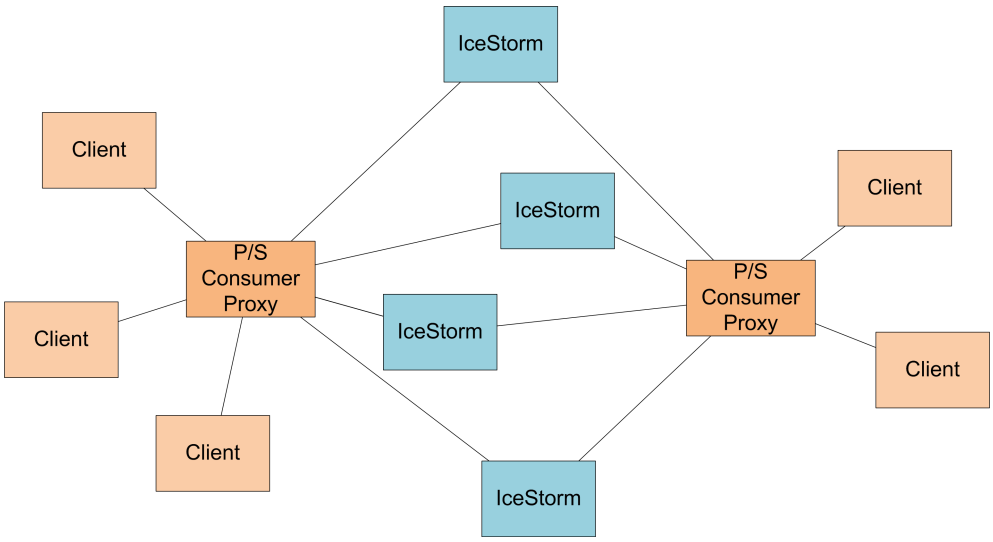


Figure 9.9: IceStorm consumers proxying. Client is not aware of which P/S service is associated with sensor producing events of particular type. Proxy is connected with all IceStorm instances.

(dataTypeID, resourceID) topics were created in a concrete IceStorm instance. The studies of IceStorm functionality leads to the conclusion that a suitable approach to this problem is to utilize the functionality of *topics federation*. Inside a federation of some topics, events of several types are passed together to a single channel. A single P/S Producer Proxy is responsible for the creation of topic *dataTypeID* in a moment a first subscription adressing the given type. It also automatically creates a federation of topics sharing the data type, connecting all of the *(dataTypeID, resourceID)* topics with a topic *dataTypeID*, regardless of the *resourceID*. Please note, that events linking is not realized in proxy, but in IceStorm. Thanks to this, there is still possible a subscription for events coming from a concrete resource *(dataTypeID, resourceID)*. Sample topics federation are presented in Fig. 9.10.

The appliance of ICE Multiplexer, ICE Demultiplexer, IceStorm Consumer Proxy and IceStorm Producer Proxy components provides a scalable, extendable, efficient and structural adapting of GEMINI P/S to the VLvl infrastructure.

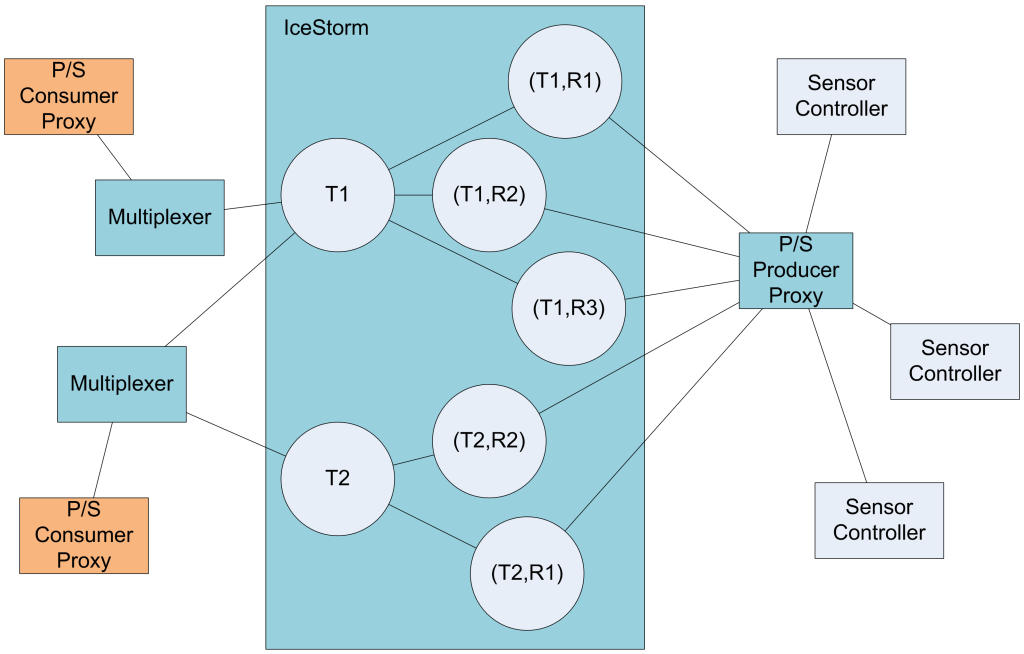


Figure 9.10: IceStorm producers proxying. Proxy dynamically and transparently manages federation of topics. Events sharing the same types are joined in IceStorm, regardless of their origin.

Bibliography

[1] T. R. Gruber, *A translation approach to portable ontologies*, Knowledge Acquisition, 1993.

[2] P. M. A. Sloot, A. Tirado-Ramos, I. Altintas, M. Bubak, Ch. Boucher, *From Molecule to Man: Decision Support in Individualized E-Health*, IEEE Computer, 2006.

[3] I. Foster, C. Kesselman, S. Tuecke, *The Anatomy of the Grid* International Journal of Supercomputer Applications, 2001.

[4] D. De Roure, J. A. Hendler, *E-Science: The Grid and the Semantic Web*, Intelligent Systems, IEEE, 2004.

[5] T. Berners-Lee, J. Hendler, O. Lassila, *The Semantic Web*, Scientific American, 2001.

[6] Y. L. Simmhan, B. Plale, D. Gannon, *A Survey of Data Provenance in e-Science*, SIGMOD Record, 2005.

[7] J. Zhao, C. Wroe, C. Goble, R. Stevens, D. Quan, M. Greenwood, *Using Semantic Web Technologies for Representing E-science Provenance*, Lecture Notes in Computer Science, 2004, Springer.

[8] G. C. Fox, D. Gannon, *Special Issue: Workflow in Grid Systems*, Concurrency and Computation: Practice Experience, 2006.

[9] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, J. Myers, *Examining the Challenges of Scientific Workflows*, IEEE Computer, 2007.

[10] L. D. Stein, *Towards a cyberinfrastructure for the biological sciences: progress, visions and challenges*, Nature Reviews Genetics, 2008.

[11] M. Henning, M. Spruiell, *Distributed Programming with Ice*, Revision 3.2, March 2007. http://www.zeroc.com/Ice-Manual.pdf. Downloaded 13 March 2007.

[12] A. Seaborne, *RDQL - A Query Language for RDF*, W3C Member Submission, 9 January 2004. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109.

[13] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, *XQuery 1.0: An XML Query Language*, W3C Recommendation, 23 January 2007. http://www.w3.org/TR/xquery.

[14] E. Prud'hommeaux, A. Seaborne, *SPARQL Query Language for RDF*, W3C Recommendation, 15 January 2008. http://www.w3.org/TR/rdf-sparql-query.

[15] D. Brickley, R. V. Guha, B. McBride, *RDF Vocabulary Description Language 1.0: RDF Schema*, W3C Recommendation, 10 February 2004. http://www.w3.org/TR/rdf-schema.

[16] D. Beckett, B. McBride, *RDF/XML Syntax Specification (Revised)*, W3C Recommendation, 10 February 2004. http://www.w3.org/TR/rdf-syntax-grammar.

[17] P. V. Biron, A. Malhotra, *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation, 28 October 2004. http://www.w3.org/TR/xmlschema-2.

[18] H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, *XML Schema Part 1: Structures Second Edition*, W3C Recommendation, 28 October 2004. http://www.w3.org/TR/xmlschema-1.

[19] D. L. McGuinness, F. van Harmelen, *OWL Web Ontology Language Overview*, W3C Recommendation, 10 February 2004. http://www.w3.org/TR/owl-features.

[20] D. L. McGuinness, F. van Harmelen, *OWL Web Ontology Language Overview*, W3C Recommendation, 10 February 2004. http://www.w3.org/TR/owl-guide.

[21] M. Dean, G., Schreiber , eds.: *OWL Web Ontology Language Reference*, W3C Recommendation, 10 February 2004. http://www.w3.org/TR/owl-ref.

[22] J. Clark, S. DeRose, *XML Path Language (XPath) Version 1.0*, W3C Recommendation, 16 November 1999. http://www.w3.org/TR/xpath.

[23] ViroLab. http://www.virolab.org.

[24] ViroLab trac. http://virolab.cyfronet.pl.

[25] Castor Project. http://www.castor.org.

[26] One-JAR. http://one-jar.sourceforge.net.

[27] Apache CXF. http://cxf.apache.org.

[28] XFire. http://xfire.codehaus.org.

[29] Java Reflection. http://java.sun.com/j2se/1.5.0/docs/guide/reflection.

[30] Java Annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[31] Extensible Markup Language (XML). http://www.w3.org/XML.

[32] Apache Ant Project. http://ant.apache.org.

[33] Apache Maven Project. http://maven.apache.org.

[34] Jena Framework. http://jena.sourceforge.net.

[35] Pellet reasoner. http://pellet.owldl.com.

[36] JAXB Reference Implementation Project https://jaxb.dev.java.net.

[37] WebDAV. http://www.webdav.org.

[38] Internet Communications Engine (ICE). http://www.zeroc.com.

[39] GridSphere Portal Framework. http://www.gridsphere.org.

[40] JRuby. http://www.ruby-lang.org/en.

[41] SQL. http://www.sql.org.

[42] MySQL. http://www.mysql.com.

[43] PostgreSQL. http://www.postgresql.org.

[44] K-Wf Grid. http://www.kwfgrid.eu.

[45] The World Wide Web Consortium (W3C). http://www.w3.org.

[46] W3C Semantic Web Activity. http://www.w3.org/2001/sw.

[47] RDFS. http://www.w3.org/TR/rdf-schema.

[48] RDF. http://www.w3.org/RDF.

List of Figures

2.1 VLvl layers. . . . 15
2.2 VLvl grid architecture. . . . 16

3.1 Abstract provenance architecture. . . . 19
3.2 From provenance requirements management to provenance mining. . . . 21

4.1 Monitoring XSD schema. . . . 24
4.2 Computation resources XSD schema. . . . 25
4.3 Experiment ontology. . . . 29

5.1 Semantic Event Aggregator. . . . 31
5.2 Experiment decomposition. . . . 33
5.3 Passing of ACID. . . . 34
5.4 Ontology extension. . . . 36
5.5 Derivation concepts. . . . 37
5.6 List conceptual model. . . . 42
5.7 OWL list. . . . 42
5.8 Parameters order information incoherency. . . . 43
5.9 Time relation between events. . . . 48
5.10 Transactional processing. . . . 49
5.11 Dependencies between experiment stages. . . . 50
5.12 Experiment re-execution. . . . 51
5.13 Events aggregation conceptual model. . . . 52
5.14 Aggregation XML context. . . . 53
5.15 Hashing of individual name. . . . 53
5.16 Delegate with hashing function. . . . 54
5.17 Unoptimal events history tracking. . . . 55
5.18 Optimized events history tracking. . . . 56
5.19 Delegate accessing Aggregator buffer. . . . 57

5.20 Domain event association. 58
5.21 Sample experiment context. 58
5.22 Association discovery module. 59
5.23 Domain event generalization. 60

6.1 Semantic Event Aggregator architecture. 62
6.2 *EventHandler* architecture. 64
6.3 XML Data Context. 65
6.4 Individual buffer interfaces. 66
6.5 Individual factories. 67
6.6 Semantic Event Aggregator deployment. 69

7.1 DRS and Geno2DRS ontologies. 73

8.1 QUaTRO GUI. 76
8.2 Sample abstract query. 78
8.3 Query processing: step 1. 79
8.4 Query processing: step 2. 80
8.5 Query processing: step 3. 81
8.6 Query processing: step 4. 81
8.7 Query processing: step 5. 82
8.8 Query language operators. 83
8.9 Query processing conceptualization. 84
8.10 Relation beetween ontologies. 86

9.1 Castor framework. 92
9.2 Event Generation Tool. 93

9.3 Remote logging architecture. 95

9.4 GEMINI overall architecture. 97
9.5 PDQS schema. 98
9.6 Semantics of monitoring data. 98
9.7 P/S architecture in GEMINI. 99
9.8 ICE channels multiplexing. 100
9.9 IceStorm consumers proxying. 101
9.10 IceStorm producers proxying. 102

List of Tables

5.1 Sample experiment execution context. 32
5.2 Delegates. 45
5.3 Aggregation rules. 47
5.4 Usage of delegates in individuals naming. 55

7.1 Grid Objects used in geno2drs experiment. 72

Publications

1. B. Balis, M. Bubak, M. Pelczar, *From Monitoring Data to Experiment Information – Monitoring of Grid Scientific Workflows*. In G. Fox, K. Chiu, and R. Buyya, editors, Third IEEE International Conference on e-Science and Grid Computing, e-Science 2007, Bangalore, India, 10-13 December 2007, pages 187-194. IEEE Computer Society, 2007.
2. B. Balis, M. Bubak, M. Pelczar, J. Wach, *Provenance Tracking and Querying in ViroLab*. In Cracow Grid Workshop 2007 Workshop Proceedings, pp.71-76, ACC CYFRONET AGH 2008.
3. B. Balis, M. Bubak, M. Pelczar, J. Wach, *Provenance Querying for End-Users: A Drug Resistance Case Study*. In: Bubak, M., Albada, G.D.v., Dongarra, J., Sloot, P.M.A. (Eds.), Proceedings ICCS 2008, Kraków, Poland, June 23-25, 2008, LNCS 5103, pp. 80-89, Springer 2008.