Semi automatic workflow generation method for data mining experiments

Jiří Bělohradský

Supervisor: Doc. Ing. Filip Železný, Ph.D.

Study programme: Open Informatics

Specialisation: Software Engineering

May 2011
Acknowledgements

I would like to express my gratitude to Doc. Filip Železný for providing me perfect working conditions and skilled leading, Eng. David Monge for many inspiring discussions and pleasant cooperation on the project and Prof. Carlos García Garino for his kind support during my stay in Argentina.
Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 13, 2011

..............................
Abstract

The thesis takes part in a bilateral Czech-Argentinean project which aims to develop a framework intended for managing execution of data mining experiments within a GRID environment. The main goal of the thesis is to propose and subsequently validate a method for semi automatic construction of workflows representing data mining experiments. The proposed method constructs workflows combining two sources: experiment structure expressed as a template, and a set of user rules which designate transformation of the template into the final workflow. Our effort results in the proposition of ontology-based formalisation expressed in OWL language which determines the form of experiment definition as well as of the user transformation rules. We have also implemented an algorithm in Java which performs the template transformation and produces a workflow represented as a directed acyclic graph. To prove workflows being generated correctly we have also implemented a component for sequential workflow execution on a local computer. The second part of the thesis deals with the practical case study in order to validate results of the first part. We reconstruct a workflow of publicly available application XGENE.ORG using the proposed method and compare resulting outputs with those generated by the original version of the application.

Abstrakt

Tato práce řeší součást česko-argentinského projektu, jehož cílem je vytvořit framework pro tvorbu a spouštění experimentů vytěžování dat v prostředí GRID. Hlavním cílem této práce je navrhnut a v praxi ověřit metodu poloautomatického sestavování workflows data miningových experimentů. Metoda je založena na dvou hlavních aspektech. Prvním je vyjádření experimentu pomocí šablon vymezujících jeho strukturu, a druhým je uživatelem daná sada pravidel, která popisuje způsob, jakým má být šablona převedena do podoby výsledného workflow. Výsledkem naší práce je návrh formalizace zápisu šablon a uživatelských transformačních pravidel, která využívá ontologie vyjádřené v jazyce OWL. Zároveň jsme navrhnuli a v jazyce Java implementovali algoritmus, který transformaci šablon na workflows provádí a výsledek reprezentuje jako orientovaný acyklický graf. Pro účely ověření funkčnosti a správnosti algoritmu jsme též implementovali komponentu, která spouští výsledné workflows sekvenčně na lokálním počítači. V druhé polovině práce využíváme navrženou metodu v praktické případové studii. Cílem je sestavit workflow věřejně přístupné aplikace XGENE.ORG pro analýzu dat genové exprese, a porovnat výsledky našeho přístupu s výstupem původní verze aplikace.
Table of Contents

1 INTRODUCTION .................................................................................................................. 9

1.1 Terminology .................................................................................................................. 9

1.2 Workflow constructing approaches ............................................................................. 10

2 ANALYSIS .......................................................................................................................... 12

2.1 Functional requirements ............................................................................................... 12
  2.1.1 Abstract Workflow Generator inputs .................................................................... 12
  2.1.2 System outputs ....................................................................................................... 13

2.2 Technical requirements and limitations ......................................................................... 13
  2.2.1 FIXED-FILES problem ......................................................................................... 13
  2.2.2 Static workflows ................................................................................................. 13

2.3 Experiment definition format selection ......................................................................... 13

3 SYSTEM DESIGN ............................................................................................................... 15

3.1 Architecture overview ................................................................................................... 15
  3.1.1 OWL definitions ..................................................................................................... 15

3.2 Data types ...................................................................................................................... 16
  3.2.1 OWL representation of data types ...................................................................... 17

3.3 Plugins .......................................................................................................................... 18
  3.3.1 Plugin Envelopes .................................................................................................... 18

3.4 Experiment templates ................................................................................................... 19
  3.4.1 Plugin Holder ......................................................................................................... 20
  3.4.2 Substructure Template .......................................................................................... 20
  3.4.3 Substructure Set Template ...................................................................................... 20
  3.4.4 Substructure Set Processor .................................................................................... 21
  3.4.5 Dataflow rules ....................................................................................................... 22

3.5 Experiment Assignment ................................................................................................. 22
  3.5.1 Common properties ............................................................................................... 23
  3.5.2 Direct Plugin Assignment ....................................................................................... 23
  3.5.3 Substructure Assignment ....................................................................................... 23
  3.5.4 Substitution Assignment ....................................................................................... 24
  3.5.5 Iteration Assignment ............................................................................................. 25
  3.5.6 Assignment Inheritance ....................................................................................... 25
4  IMPLEMENTATION.................................................................................................................26

4.1 Data types ..........................................................................................................................26
  4.1.1 Type Descriptors ...........................................................................................................26
  4.1.2 Data Descriptors ...........................................................................................................27
  4.1.3 Data Handlers ...............................................................................................................27

4.2 Plugins ...............................................................................................................................28
  4.2.1 Plugin description ..........................................................................................................28

4.3 Translation algorithm .........................................................................................................29
  4.3.1 Algorithm main principle ..............................................................................................30
  4.3.2 Additional issues connected with AWG algorithm .........................................................31

4.4 Local workflow executor ....................................................................................................32

5  APPLICATION DESCRIPTION ..............................................................................................35

5.1 XGENE.ORG application overview ..................................................................................35

5.2 Format of results ................................................................................................................36

6  WORKFLOW DESIGN ............................................................................................................37

6.1 Basic templates definition ..................................................................................................37

6.2 Inspecting performance of workflow paths ........................................................................41

7  RESULTS COMPARISON .......................................................................................................42

7.1 Experiment settings ............................................................................................................42

8  CONCLUSIONS .......................................................................................................................44

  8.1 Ongoing work ....................................................................................................................44

REFERENCES .............................................................................................................................45

APPENDIX A – USED ABBREVIATIONS ..................................................................................47

APPENDIX B – WORKFLOWDEFINITION OBJECT STRUCTURE ............................................48

APPENDIX C GENERATED WORKFLOW ................................................................................49

APPENDIX D – CONTENTS OF THE ATTACHED CD-ROM .......................................................50
List of Figures

Figure 1. Artifacts figure inputs and outputs of the AWG component ................................. 12
Figure 2. DDMF internal workflow ................................................................................. 15
Figure 3. DDMF Basic Ontology ..................................................................................... 16
Figure 4. Basic data types dependency ......................................................................... 17
Figure 5. Plugin scheme ................................................................................................. 18
Figure 6. Plugin Envelope scheme ............................................................................... 18
Figure 7. Template element scheme ............................................................................ 19
Figure 8. Example operations with template .................................................................. 19
Figure 9. Substruture template scheme ......................................................................... 20
Figure 10. Branching problem ...................................................................................... 21
Figure 11. Branching without Set Processor .................................................................. 21
Figure 12. Branching with one Set Processor ............................................................... 22
Figure 13. Branching with two nested Set Processors ..................................................... 22
Figure 14. Using assignments to branch the workflow .................................................... 23
Figure 15. Example branched workflow ....................................................................... 24
Figure 16. Substructure template prepared for substitution ........................................... 24
Figure 17. Example of combining Substitution and multiple Substructure Assignment ... 25
Figure 18. TypeDescriptor class diagram .................................................................... 26
Figure 19. TypeCatalog class structure ....................................................................... 26
Figure 20. DataDescriptor class diagram .................................................................... 26
Figure 21. DataHandler class diagram ......................................................................... 27
Figure 22. Plugin interface ............................................................................................ 28
Figure 23. Description.xml - plugin meta-information document structure .................. 29
Figure 24. Pseudocode of the basic idea of AWG algorithm ........................................ 30
Figure 25. Workflow Manager common interface .......................................................... 32
Figure 26. XGENE.ORG inner workflow. ................................................................... 35
Figure 27. Example experiment result .......................................................................... 36
Figure 28. Substructure responsible for providing multi-platform data ......................... 38
Figure 29. Substructure generating working units .......................................................... 38
Figure 30. Template for cross-validation of combinations of working units and analysis algorithms ........................................................................................................... 39
Figure 31. Unit generation for test sets .......................................................................... 40
Figure 32. XGENE.ORG Standard experiment template ................................................ 40
Figure 33. Histogram experiment template ................................................................... 41
Figure 34. Reference experiment data ......................................................................... 42
Figure 35. The old Xgene experiment log produced by 1-NN algorithm ....................... 43
Figure 36. The new Xgene 1-fold cross validation log produced by J48 algorithm ....... 43
List of Tables

Table 1. Semantics of template diagrams ................................................................. 37
Table 2. The reference experiment summary .......................................................... 43
1 Introduction

The motivation for the project which is addressed here generally comes from two different fields of computer science: data mining and GRID computing. Experiments of data mining consist of many interdependent tasks which are often convenient to be executed partially concurrently in a cluster of computing units. To achieve this, there are two main issues which must be solved. The first is how to define data mining experiments what is connected to the level of possible automation of constructing such experiments. The ensuing problem is how to execute these defined experiments in the GRID environment. Both tasks mentioned take part in the bilateral Czech-Argentinean project internally called Distributed Data Mining Framework (DDMF) which is supported by grant MEB11105.

The thesis deals with the first issue. We propose a formalization method of data mining experiment definition with respect to its further execution in a distributed computational environment. The approach we have chosen is described in contrast to other existing approaches in the first part of the text. The theoretical conception is completed by a design and implementation of algorithm which translates the formal experiment definition into a form suitable for physical execution. However our part of the project is supposed to produce results usable for distributed parallel execution within a GRID system, it also makes sense to develop an execution component which performs the experiment locally on a single computer. It is convenient mainly for testing purposes but at the same time it protects our part of the system from being dependent on the external environment. The local execution component proposal and implementation end the section.

The second part of the text is dedicated to the conception validation. We use the proposed method to implement a particular use case study in bioinformatics field. The XGENE.ORG is a publicly available tool for cross genome gene expression analysis [1]. We re-implement XGENE.ORG application using our experiment definition method to prove its functionality. In the second step we show how the expressivity of our approach can extend analysis capabilities of XGENE.ORG application to meet the requirements of very recent scientific research in its field.

1.1 Terminology

Further in text we use the term workflow instead of “data mining experiment definition”. We use the term workflow in a fairly general way. For us it means a set of tasks (algorithms) some of which are mutually dependent and thus have to be executed in a given order. Workflow can be naturally described as a directed acyclic graph where each vertex represents an algorithm to be executed and each edge represents a data transfer between two algorithms. Vertices installed on a common path have to be executed sequentially while the others can be run parallel. According to taxonomy brought out in [2] we talk about abstract workflows whose vertices represent tasks without their particular assignment to physical computational nodes. Translating workflows to theirs concrete
INTRODUCTION

form is addressed by the special scheduling component [3] which is not however discussed in this thesis.

1.2 Workflow constructing approaches

Recently, there are primarily three commonly used approaches to construct workflows. The first one is used when a developer hard-wires the workflow into the proprietary application (i.e. intended for gene expression analysis like in [4] or even in XGENE.ORG application [1] whose modernization is being addressed in the Section B of this thesis). It brings an absolute freedom in workflow constructing as it is limited only by capabilities of chosen programming language and underlying platform. However, once the workflow is defined, any slight change becomes expensive as long as it requires re-implementation of the software.

The second approach transfers the workflow construction operations from developer to user – scientist or analyst. It provides a user interface capable of assembling workflows from prefabricated blocks (i.e. Weka KnowledgeFlow Environment [6]). Advantage of this approach lies in ease of making changes within the defined workflow. On the other hand it becomes really hard to manage complex experiments, when all modifications must be made manually.

The last approach deals with automatic workflow construction and is still under live research [5]. The idea is to specify input data and its demanded output form and let the computer compose the workflow by itself. The process is usually driven by some kind of planning algorithm known from artificial intelligence and is connected with auto-evaluating the workflow performance in order to find the combination having the best results for the given domain. Such an approach is able to find workflows which wouldn’t be intuitively constructed by an analyst what might be an advantage as well as disadvantage. It is a way to discover new routines not known before, but it can also produce results which don’t make any sense and so are hardly interpretable. Moreover, due to the extremely large number of possible workflow combinations this approach may often lead to a combinatorial explosion.

Our approach is trying to keep benefits from all the methods mentioned above. It is based on fact that in data mining there are many time-proven commonly used practices that are highly convenient to be used. With respect to this, an experiment is defined in two steps. Firstly a common structure is expressed as a template whose building blocks are substituted in the second step by a variety of particular algorithms or possibly by other templates. Considering workflows being graphs it leads to the use of a graph transformation technique as a formal method of expressing data mining experiments. Resulting workflow is then constructed step-by-step applying user defined rewriting rules on the initial template. This approach brings a possibility to retain sanity of generated workflows even thought their final form is generated automatically according to the user’s (developer’s) demands. This is why we call the approach semi-automatic workflow construction method: the automation is strictly under the user’s control.
Section A
Workflow construction method
2 Analysis

The main goal of the thesis rests in proposing a method of semi-automatic workflow construction based on predefined templates. In this chapter we focus on the requirements analysis from which basic characteristics of the system are being derived. The analysis stage is important for future decisions in particular issues of method design because it frames a space we operate in and clarifies what is the system expected to do.

2.1 Functional requirements

Functional requirements specify inputs and outputs of the method. The general overview is shown in Figure 1. At the moment we focus on Abstract Workflow Generator (AWG) component which is responsible for generating the workflow definition on the basis of input experiment specifications.

![Figure 1. Artifacts figure inputs and outputs of the AWG component.](image)

**2.1.1 Abstract Workflow Generator inputs**

According to Figure 1 AWG has several inputs. Basic ontology determining the form of the rest of inputs, library of plugins and data types, templates which determine the experiment structure (experiment definition), a set of rewriting rules which transforms template into the final workflow (experiment assignment). We briefly describe each of them.

**Plugins and data types definition.** The term plugin is used for a stand-alone program which can be executed as a single task of workflow. Each plugin is primarily defined by a set of inputs and set of outputs. Inputs and outputs are described by data types which determine the form of data which are accepted and produced by plugins. Data types are organized in a hierarchy where each child is a more specific version of its parent. It means that plugins accepting parent data type also accept all its children data types.

**Experiment structure definition.** Experiment template is a graph which predefines the form of a workflow. Its nodes are template elements which are to be replaced by another template or by an executable plugin. In both cases substitution respects
the input and output data types which must be preserved. Template holds a common structure for all possibly generated experiments.

**Template transformation.** Once experiment template is defined, many concrete experiments may be generated from it using different combinations of plugins or sub-templates. This generating process is lead by rules given by user which we call the experiment assignment. Rules can assign a plugin to a particular template element, perform graph branching or substitute an element by a sub-template with coinciding inputs and outputs.

### 2.1.2 System outputs

The main output of AWG is a workflow represented as a DAG which can be executed in a GRID or locally run all tasks in an appropriate sequence. The DAG has a fixed form negotiated as an interface between two parts of the DDMF project. The final DAG is hold by Java WorkflowDefinition object whose structure is attached in APPENDIX B.

### 2.2 Technical requirements and limitations

Almost all technical requirements come from the planned cooperation with the GRID environment and related scheduling component which optimizes the total execution time of the workflow [3].

#### 2.2.1 FIXED-FILES problem

One of the used optimization principles is based on minimizing the number of inter-computer data transfers. To achieve this, the scheduler needs to know all files used during individual task execution. This brings one of the most serious limits because our plugins can produce only a fixed number of output files. In other words the number of resulting files must be known before the execution starts. We will further reference that issue as the FIXED-FILES problem.

#### 2.2.2 Static workflows

The optimization scheduling algorithm works statically. It means that it assigns all jobs to computational nodes once before the execution has started and the schedule is then kept for the whole run-time. There is no possibility to make any changes during the execution. This fact excludes any on-demand intelligence. All decisions which influence the workflow must be made before the execution.

### 2.3 Experiment definition format selection

We decided to use OWL DL ontologies as the format for all steps of experiment definition process for several reasons. Primarily, it meets two important requirements. It enables to describe experiments formally using the descriptive logic semantics which is
suitable for expressing graph structures and it provides a powerful technology to make queries over the experiment definitions in cooperation with a specialized reasoner application. As a side-effect we have consistency checking which controls the correctness of the definition syntax. Moreover, OWL ontology can be stored using RDF/XML format supported by many applications like Protégé which can serve as the user interface for creating ontology files.

The main disadvantage of using OWL lies in difficulty to understand the underlying DL background what limits a group of potential users only to experts who have the appropriate level of education. However, the method is intended to construct data mining experiments, thus the wide-spread usage of the system outside the scientific community is not expected.

The competitive solution would rest in using some proprietary XML-based format to express the experiment definition steps. This approach but misses almost all of the advantages and the ease of understanding wouldn’t be much higher. The querying would be possible using XPath or XQuery languages but those technologies are more suitable for tree structured data, less for general graphs.
3 System design

Analysis from the preceding chapter clarified general characteristics which must be followed by the proposed method. In this section we present and explain the method proposal in detail.

3.1 Architecture overview

The basic overview of DDMF internal workflow provides Figure 2. The left part denotes the experiment definition steps elaborated in the OWL language. Our part of the project finishes at “Translation experiment definition into the terms of DAG” stage. However, due to reasons described in the introduction, secondary target of the thesis is to implement local topology-order-based executor component which acts as the last two illustrated steps.

![Figure 2. DDMF internal workflow](image)

We will now go through all the steps mentioned in Figure 2 and describe its realization within our system.

3.1.1 OWL definitions

As we discussed in the analysis stage, all the definitions are formally done using an ontology based on OWL DL semantics. We use a predefined set of Classes and Object properties to determine types of experiment definition elements and relations between them. The predefined structure displayed in Figure 3 is physically set up in the ddmf.owl file found on attached CD.

The figure contains elements of several types. Blocks stand for ontology classes and determine the type of objects which can be used by user to assemble experiment templates, experiment assignments, plugin and data type libraries. We have two different relations between objects expressed by the arrow type. Full arrows denote the generalization. The starting block is more specific version of the ending block what means
that it inherits all properties and can have also some additional ones. Standard arrows stand for object properties which settle relations between objects of the connected types. These relations are understood by the AWS component likewise the object types and act in the workflow generation process. Inner attributes found in some of the objects serve to specify parameters.

Figure 3. DDMF Basic Ontology

The remaining steps of definition (according to the Figure 2) are dependent on this ontology as long as it provides a common language understood by the AWG component. In the next sub-chapters we will step-by-step explain all the parts of the figured ontology.

3.2 Data types

All considered data mining jobs within the workflow deal with different information. Basically, we distinguish information of two types. Those which are known at the experiment definition stage and have the configuration purpose we call arguments while the rest is called simply “data”. Data can be given by user as an input of the data
mining experiment (including background-knowledge, common data shared by all experiments) or can be computed by particular experiment jobs which are realized by plugins (described in the following section).

We consider three standard classes of data, which must be each particular data type inherited from. Their dependency structure is shown in Figure 4.

![Figure 4. Basic data types dependency](image)

The Atomic Data Type holds just a combination of single value attributes and is extended by the other two data types. The File Type can hold one or more binary files (usually a data tables in CSV or ARFF format). The reason of having the File Type rests in the FIXED-FILES problem mentioned above which forces us to pay special attention to all files appearing in the workflow. Finally, the Set Type is a set of any other data objects.

Data types mentioned here are just abstract. A user of our framework may define concrete types inherited from these basic ones according to the expected content. Therefore, each defined type has a parent type, and so all the data types are organized in a hierarchy. This is important for the future workflow definition concept because it brings up a new degree of freedom in specifying workflow constraints. If plugin accepts some type, it also accepts all of its children from the hierarchy.

This brings a kind of inheritance. There are two main reasons why inherit types: to extend carried data content or to bind type with some semantic information. For example we can have a type called DataSet intended to carry tabular data structured by attributes and containing records. Thereafter it might make sense to inherit types such as TrainDataSet and TestDataSet whose physical structure remains the same but conditions of usage differ and so it is important to be able to differentiate them.

### 3.2.1 OWL representation of data types

Within the Basic Ontology (Figure 3) data types are represented only by two Classes. The first one is called simply Data Type and represents most of types. The second one is called Set Type and its purpose is to collect other data. More precisely, it is a multi-set because it contains items of the same type.

We have two relations for data types. The isPartOf relation is reflexive and transitive and denotes the parent of the given type. Even if the data type has no parent it must be in the isPartOf relation with AnyType what is a special reserved data type. When a plugin accepts AnyType it means that it accepts everything. There are also other reserved data types with similar meaning but all of them have isPartOf relation with AnyType.
These are AnyFileType which denotes a type intended to use files and AnySetType which is a root type for sets.

The hasItems relation is connected with set type only. Likewise isPartOf, it is mandatory for any set type since it holds information about the type of set items.

### 3.3 Plugins

All tasks which are to be executed during the workflow execution stage must be integrated with the system in the form of plugin. The plugin is generally an executable program accompanied by a meta-description of its properties. As is displayed in Figure 5 each plugin has a set of accepted input data types and a set of produced output data types. To define this user uses relations needsInput and producesOutput respectively.

![Figure 5. Plugin scheme](image)

There is no necessity to map plugin object defined in OWL to the physical plugin which is to be executed since all physical plugins are annotated by the full name of corresponding OWL Individual using its IRI.

### 3.3.1 Plugin Envelopes

As we have already explained it might be necessary to discern data types with the same physical structure according to the terms of its usage. The system of inheritance allows plugins to accept child types which are more specific version of matching parent types. The backward compatibility is guaranteed automatically. However, sometimes it makes sense to use even opposite principle. Let’s consider the following situation. We have a plugin A whose output acts as the input of plugin B. Plugin A produces output of DataSet type while plugin B needs input of TrainDataSet type which is more specific version of DataSet even it has the same physical structure (e.g. the difference is just semantic). Respecting the inheritance this wouldn’t be working because no plugin can accept less specific types than it expects. To solve this, we have proposed Plugin Envelope objects whose structure is demonstrated in Figure 6.

![Figure 6. Plugin Envelope scheme](image)
The Plugin Envelope covers a plugin by a compatibility layer. It performs a forward type casting. Since it is a subtype of Plugin, it can participate in all relations dedicated to plugins such as needsInput and producesOutput. Moreover it has a special relation to denote the enveloped plugin which is called intuitively envelopsPlugin.

In addition, it is possible to pass some arguments to the plugin while enveloping it. We discuss plugin arguments further in section 3.5.2.1.

3.4 Experiment templates

Once we have set up data types and plugins we are ready to compose experiment templates which hold experiment structure. There are several so called template elements which the template can be composed of. In Figure 3 they are all inherited from the ExperimentTemplateElement class. Regardless the element type, all elements share the basic structure shown in Figure 7.

![Figure 7. Template element scheme](image)

It is not coincidence that the definition of the plugin and template element is very similar. The experiment template is a directed acyclic graph vertices of which are template elements. So the main purpose of template elements is to be assigned by plugins in order to get an executable workflow. However, we need a little more from our construction method than only to run different plugins within a fixed workflow. Figure 8 shows two basic examples of operations we need be able to express and perform using semi-automatic construction method.

![Figure 8. Example operations with template](image)
The default template is denoted as $a$. The first operation is repeating a given part of the template and appropriate graph branching what is designated as $b$. The second operation is substituting given element by a more complex substructure. In Figure 8 this structure is represented by $c$ and the resulting graph after substitution looks like in $d$.

In order to realize operations as branching and substituting we have several types of template elements. Let us describe them in detail.

### 3.4.1 Plugin Holder

The simplest template element is the *Plugin Holder*. As the name suggests the purpose of this kind of element is to hold a place for any plugin with coinciding inputs and outputs. Plugin Holders use relations *needsInput* and *producesOutput* to specify this information.

### 3.4.2 Substructure Template

All elements which are not Plugin Holders are derived from Substructure Template. Basically, it is an element which has an inner structure composed of other template elements. The situation is depicted in Figure 9.

![Substructure Template Scheme](image)

**Figure 9. Substructure template scheme**

Substructure Template has a set of inner template elements and two other sets to determine sub-elements which adopt input and output. Three relations are set up according to that: *hasSubHolder* (which denotes the inner element), *putsInputTo* and *takesOutputFrom* (which redirect the input and output). Every top level template is defined as a single element of the *Substructure Template* type. Those templates which carry structure of the whole experiment we designate *ExperimentTemplateRepresentative* in order to be easily identified by applications responsible for creating concrete experiment assignments.

### 3.4.3 Substructure Set Template

*Substructure Set Template* has the same structure as a common *Substructure Template* but its content is dedicated to be repeated. As can be deduced from Figure 10 the Plugin succeeding to the branched part of the workflow (here $C$) must be prepared to work with multiple input where the exact number of inputs isn’t known in the time of plugin development.
This is the only way to obtain a Set Data Type. Actually, it is not possible to produce any Set Data Type by a plugin. It is because we need to know the exact number of files before the experiment is executed (the problem was presented as FIXED-FILES problem discussed above).

The number and content of workflow branches within the Substructure Set Processor is set by user in the following step of experiment definition process (see subchapter 3.5).

### 3.4.4 Substructure Set Processor

Substructure Set Processor also branches the final workflow. The difference from the ordinary Substructure Set is that the branching here is automatic according to the input. As was already mentioned, the output of Substructure Set is always of a Set Data Type. If we encapsulate more Substructure Set Templates one into another, it results in Set Data Type containing another Set Data Type.

Substructure Set Processor must succeed to any other Substructure Set since it branches the workflow according to the number of items in the produced Set Data Type. Let us consider an example. The initial state is shown in Figure 11.

*B* is a Substructure Set Template which contains other Substructure Set Template *C*. *B* is then forked into two branches and inner *C* is then always forked to other 3 branches. The depicted situation implies that *D* is a single plugin generated by Plugin Holder assignment. Thus all results come into it and the assigned Plugin has to process such complicatedly structured data. If we replace Plugin Holder by Substructure Template Processor (for *D*) we obtain a workflow like in Figure 12.
According to the number of items of the outer Substructure Set Template $B$, the content of Substructure Set Processor is automatically forked into two branches. If we encapsulate one more Substructure Set Processor into the one generating $D$, we finally get the workflow from $A$. The plugin $D$ then accepts the simple Data Type generated by a single plugin $C$.

### 3.4.5 Dataflow rules

The edges in directed acyclic graph representing the template stand for the succession relation regardless to the particular data types and are in fact realized by `isSucceededBy` OWL relation. The content of real data transfer between two plugins is inferred from theirs I/O definitions.

### 3.5 Experiment Assignment

The final stage of the experiment definition process is creating of Experiment Assignments. While the templates define the structure of experiment, assignments fulfil the structure with a real content. Assignments represent rewriting rules which are used to transform a DAG of the Experiment template into a DAG of workflow. These rules we express using OWL likewise in case of the rest of experiment definition steps. Assignments must follow the structure of the transforming template what leads to use a
tree hierarchy where each leaf node assigns a particular plugin directly while the non-leaf nodes assigns different kinds of substructure templates. We have also several special assignments providing a little more automation. In the further text we will explain all kinds of Assignments in a detail.

### 3.5.1 Common properties

All Assignments are derived from the `ExperimentAssignment` OWL class. To determine the concerning template element each Assignment must pertain a relation called `isAppliedTo`.

### 3.5.2 Direct Plugin Assignment

The purpose of Direct Plugin Assignment is to assign a particular plugin to the appropriate Plugin Holder. To do that, Direct Plugin Assignment has a relation called `assignsPlugin`. Another thing Direct Plugin Assignment can do is to configure plugin arguments, e.g. parameters which set up behaviour of the plugin.

#### 3.5.2.1 Plugin arguments

Plugin arguments have their own `Class` within the OWL ontology called `PluginArgument`. Arguments have two data properties. First one is called `hasStringName` and designates a name of the configured parameter. The second one is called `hasStringValue` and sets the parameter value. The configured argument is then attached to the Direct Plugin Assignment using the `setsArgument` relation.

![Figure 14. Using assignments to branch the workflow.](image)

### 3.5.3 Substructure Assignment

Structure assignments are used in combination with Substructure Templates (and any other subtype of it). They have the `hasHolderAssignment` relation to encapsulate assignments for the inner structure of the Substructure template.
In case of Substructure Set Template, the number of branches is given by the
number of substructure assignments applied to it. The example is shown in Figure 14. The
rounded rectangle stands for Substructure Set Template with the inner structure containing
L,M,N Plugin Holders. The Plugin Holder L is denoted as input holder of the substructure
(by relation putsInputTo), likewise N is denoted as output holder (by relation
takesOutputFrom). In the upper half of the image, there are Assignments depicted. Those
with white background are Direct Plugin Assignments, while those grayed are Substructure
Assignments. The blue dashed arrows stands for the isAppliedTo relation. The resulting
workflow is then displayed in

![Figure 15. Example branched workflow](image)

### 3.5.4 Substitution Assignment

The first of two special assignments we have proposed is called Substitution Assignment. It is used when we want a step of workflow template which is annotated with
Plugin Holder to be realized by a more complex substructure with the same I/O. It is
possible to do it using Substitution Assignment. It has the assignsSubstructureTemplate
relation to denote target substructure for substitution. If we take the example from Figure
14 and Figure 15 we can show the substitution on the following case.

![Figure 16. Substructure template prepared for substitution](image)

Consider we have a substructure template which is shown in Figure 16. The Plugin Holder
X accepts a data type coming from Plugin Holder A, and the same with Y and B respectively. If we use the Substitution Assignment to assign L with this substructure the
final workflow considering also the assignment from Figure 14 will look like the one in
Figure 17.
3.5.5 Iteration Assignment

The last special type of Assignment is the Iteration Assignment. It is used with Substructure Set Template performing the automatic repeat of the inner structure according to the given parameters. It repeats the structure with the same configuration of plugin assignments. The only changing is a specified Plugin Argument which is iterating through a given range of integer or string values.

In practice this kind of assignment is useful to constructing workflows performing classification validation techniques based on repeating experiments with a different combination of train and test data.

The putsParameterTo relation selects the Plugin Argument whose value is going to be changed each iteration step. To set up the iteration parameters a special OWL Class is used which is called PluginArgumentRange. Its purpose is to set a range from which values are stepwise selected during iteration. The setsArgumentRange relation is in service here.

3.5.5.1 Plugin Argument Range

PluginArgumentRange OWL Class is used to set up the properties of iteration. We have two types of iteration. The standard integer iteration uses a triplet of data properties:

- hasIterationValueFrom – sets the starting value
- hasIterationValueTo – sets the value which terminates the iteration
- hasIterationStep – sets the iteration step

To iterate through a set of string values only one data property is used. It is called hasEachValueFromList and it takes comma-separated list of strings.

3.5.6 Assignment Inheritance

All assignments set by a SubstructureAssignment are also propagated to its sub assignments. Thus a kind of inheritance appears here. It is possible to set some default assignments on a higher level and don’t then need to create separated assignments for each of the branches. Any assignment further from the root has a higher priority so the default values can be replaced if needed.
4 Implementation

Since the translation algorithm which creates a workflow object on a basis of OWL definitions is written in Java, it is necessary to represent Data types and Plugins used in OWL also in Java. The first two subchapters are concerning this, while the third subchapter presents implementation issues of translation algorithm itself and the last subchapter deals with the local executor component.

4.1 Data types

In contrast with OWL definitions where Data Types were defined simply by two OWL Classes and two relations, the situation in the implementation field is a bit more complicated. This is because there are several stages we work with the data types in and all of them need specific information for its own purposes.

4.1.1 Type Descriptors

In Java as well as in OWL ontology we need to have some information about types including: the identifier, the identifier of parent type, file name extensions in the case of File type and finally the identifier of item type in the case of Set type. Objects which carry information like this are called Type Descriptors. Its class diagram is shown in Figure 18.

![Figure 18. TypeDescriptor class diagram](image)

Type Descriptors are can be obtained using a TypeCatalog object which stores information about all Data Types used within the DDMF. It uses a Singleton design pattern and has a fetchTypeDescriptor method which returns TypeDescriptor according to given type name (in IRI). TypeCatalog is depicted in Figure 19.

![Figure 19. TypeCatalog class structure](image)
4.1.2 Data Descriptors

Due to the FIXED-FILES problem discussed in section 2.2 we have to generate names of all files which are to be transferred during the workflow execution. Objects which carry meta-information about instantiated Data Types are called Data Descriptors. Likewise in the case of Type Descriptors we have three Data Descriptors which mirror the purpose of its usage. All of them implement the DataDescriptor interface as is displayed in Figure 20.

![DataDescriptor class diagram](image)

The `getType` method returns the TypeDescriptor of the Data type. The `getMainDataFileName` return the file carrying DataHandler object (discussed later) and finally the `getAdditionalFilenames` method returns the list of files attached to the given type. In case of AtomicDataDescriptor, the list is always empty. SetDescriptor on the other hand returns the complete list of all files attached to all items stored in the Set.

4.1.3 Data Handlers

Objects which physically carry the data and enabled to work with it within Java code are called Data Handlers. They must be `Serializable` since they are transferred serialized using standard Java serialization technique. Not all types appearing as TypeDescriptors or as an OWL Individual must have its own Data Handler.

![DataHandler class diagram](image)
IMPLEMENTATION

Logical types which share the inner structure can use a dedicated DataHandler in common and the inner inheritance system is responsible to transfer data types correctly. All Data Handlers extend the common abstract class called DataHandler, Data Handlers for Set Data Types extends SetHandler class. Their structure is shown in Figure 21.

Any new attributes and methods can be added in new DataHandler, the only condition is to preserve the object’s ability to be serialized.

4.2 Plugins

In terms of DDMF plugins are simple single-purpose programs which usually perform data transforming or generally data handling operations. Plugins must be implemented in Java using the Plugin interface, structure of which is depicted in Figure 22.

During the workflow execution stage the plugin is executed in two steps. Firstly, the configure method is called. It gets information about values of parameters, the list of input data handlers and finally the list of output data descriptors which are to be used to create output data handlers inside the next step. The configure method can do some preprocessing operations or just store information it has got for the further usage.

Afterwards, the executeTask method is called. It should contain the processing and postprocessing operations and return the list of output data handlers. Inside the method, it is possible to invoke any external program, so the plugin can serve as a wrapper only which redirects data to and from another application. The executor component catches standard output and error output and redirects them to dedicated files. This is useful while it is necessary to debug plugin code.

4.2.1 Plugin description

Each plugin must provide some meta-information about itself which describe plugin’s characteristics. This meta-information is stored in a description.xml file in the root of plugin directory. Structure of the file is denoted in Figure 23. Comments are included directly in the code.
4.3 Translation algorithm

The goal of the Translation algorithm is to transform experiment definition created in OWL ontology into the `WorkflowDefinition` Java object, e.g. form which holds...
IMPLEMENTATION

executable abstract workflow. This is the reason why we call the algorithm Abstract Workflow Generator and denoting it AWG. The detailed structure of the WorkflowDefinition object is described in APPENDIX B.

The algorithm generally goes through the defined experiment template and applies rewriting rules expressed in form of assignments discussed in the Chapter 3. The full working code is found on the attached CD. Here we show only a pseudo code which tries to explain main ideas of the algorithm.

To make requests over OWL ontology we use the Pellet OWL 2 reasoner in version 2.2.2. which is written in Java and provides a good performance for the reasoning operations [7].

4.3.1 Algorithm main principle

AWG is a recursive algorithm which takes three main input parameters:

- **template** – object representing a template element to be processed
- **parentAssignment** – assignment object for the parent element (or null for top level template elements)
- **WD** – WorkflowDefinition object which contain Jobs (graph vertices) and Transfers (graph edges) and holds the final result of the transformation

```
function processTemplate(template, parentAssignment, WD): IOMapping
  IOMapping result_mapping // = Map<template,(inputs,outputs)>
  for each assignment a in {a | parentAssignment hasHolderAssignment a and a isAppliedTo template}
    if template is PluginHolder and assignment is DirectPluginAssignment then
      WD.jobs << new Job(assignment.assignPlugin)
      result_mapping = new IOMapping(inputs = template.inputs, outputs = template.outputs)
    end
  end
  IOMapping substructure_model
  //PROCESS ALL SUBELEMENTS
  for each element e in {e | template hasSubHolder e}
    substructure_model<< processTemplate(element, assignment, WD)

  //CREATE TRANSFERS BETWEEN SUBELEMENTS
  for each element e in {e | template hasSubHolder e}
    for each succ_element e in {e | element isSucceededBy e}
      WD.transfers << create transfer between element and succ_element using IO jobs from substructure_model
  result_mapping << all IO jobs from substructure_model which were generated by IO subelements of the template
```

Figure 24. Pseudocode of the basic idea of AWG algorithm
The basic idea of the algorithm is simply described in Figure 24. In the case of PluginHolder and DirectPluginAssignment it creates and returns a new workflow job. In the case of substructures AWG is going through all template sub-elements and invokes recursively itself for those substructures (which are passed as a template parameter). Consequently, it creates transfers according to returned IOMapping object which contains all jobs designated as I/O jobs of the processed sub-element.

4.3.2 Additional issues connected with AWG algorithm

Real AWG algorithm is a bit more complicated due to issues we have omit until now. It is not convenient to express them using pseudo code since it would be too complex and not very transparent. We will here just mention these issues and briefly describe how to enhance the code in order to bring a solution of them.

4.3.2.1 Plugin envelopes

Plugin envelopes are tackled in the first part where the physical jobs are established. The final job is created with plugin extracted from the envelope, input and output mappings are but mapped to envelope I/O data types. This is because we need to map less specific data types to their more specific child data types requested by the underlying Plugin Holder.

4.3.2.2 Substitution assignment

When a substitution assignment is detected the Plugin Holder it is applied to is further considered as a Substructure Template with assignments gathered from the parent substitution assignment.

4.3.2.3 Iteration assignment

Iteration assignment is integrated simply by simulating increased number of sub-assignments of the parentAssignment. Sub-assignments must carry the information about the plugin arguments set by iteration assignment.

4.3.2.4 Assignment inheritance

To provide the inheritance described in section 3.5.6 the processTemplate function must by extended by a parameter which holds a table of all declared assignments. This table is realized as a Map where a template name acts as a key value. This brings a possibility to override the value once set by default in a higher level of template hierarchy.

4.3.2.5 Substructure Set Processor

To provide a functionality of SubstructureSetProcessorTemplate, several non-trivial changes must be made to the algorithm. Firstly, it is necessary to process all sub-templates in a topological order. The reason is simple – when we want to create as many branches as the number of items the preceding SubstructureSetTemplate has generated, this template must be processed before. Secondly, we need to access generated workflow
outside the current context of the recursive invocation. This is because the preceding SubstructureSetTemplate can be addressed in a completely different level of template hierarchy. Due to this we need to add one more function parameter which holds the IO model of all the previously generated workflow. And lastly, we need to identify the appropriate SubstructureSetTemplate what is non-trivial when we allow nesting of SubstructureSets as well as SubstructureSetProcessors. The used solution lies in the iterative algorithm which goes backwards in the template following the isSucceededBy relations and looking for a source of Set Data Type. If this is not found and we are inside a Substructure template, we go to the upper template using the putsInputTo relation. When we escaping the another SetProcessorTemplate we must increase an internal deepness counter because the first source of Set Data Type is intended for this SetProcessor-Template and not for the given one. Once we find a SubstructureSet we decrease the counter and if it was greater than zero we follow the takesOutputFrom relation to get into the Set template. This process is repeated until the source is found or until we cannot continue using any of steps mentioned. If so, an error must be reported.

4.4 Local workflow executor

Local Workflow Executor is a component responsible for executing workflows locally preserving data dependencies between jobs. It implements the WorkflowManager interface in order to be integral with the DDMF conception. The interface structure is shown in Figure 25.

```java
public interface WorkflowManager {
    public void executeExperiment(
        WorkflowDefinition workflowDefinition,
        ExperimentManagerCallbackInterface experimentManager,
        String workingDirectory,
        Logger logger
    ) throws IOException, InterruptedException, DDMFException;
}
```

Figure 25. Workflow Manager common interface

The workflowDefinition parameter holds a workflow to be processed, experimentManager is a callback interface which enables the application to be informed about the status of experiment during the process. The parameter called workingDirectory defines a directory in a file system where are the experiment data files located in. And finally logger is a common log handling object.

The WorkflowDefinition object which is created by AWG algorithm holds information about the order of job execution, since it is necessary to run preceding job before the succeeding one from those two which are connected by the isSucceededBy relation. When we need to execute job organized in a directed acyclic graph sequentially without parallelization it is necessary to run tasks in the topological order.
The execution procedure works in several steps once it gets a job to execute. Firstly it builds up `DataHandler` objects. That objects which are coming from preceding job are just deserialized from incoming file. Set Handlers must be created according to the DataDescriptor objects which are stored in `WorkflowDefinition` for each job.

Once `DataHandlers` are prepared, they are passed as an argument of the plugin’s `configure` method together with `DataDescriptors` of output types. Afterwards the job is executed invoking its `executeTask` method. When the plugin has finished, all returned DataHandlers are serialized and copied to the directories of all succeeding tasks which accepts particular data type.
Section B
Use case study: XGENE.ORG
5 Application description

In the second part of the thesis we validate the proposed method by applying it in the practical use case. In this chapter we describe XGENE.ORG application which we use for that purpose. XGENE.ORG is a publicly available analytic tool for bioinformatics to run experiments working with cross-genome gene expression data. The novelty of the application rests in data preprocessing stage. XGENE.ORG provides a functionality to combine data measured on different microarray platforms regardless to the organism the technology is dedicated to. It means we can do experiments which combine data across organisms and thus it is possible to analyze data which cannot be normally put together. To achieve this, XGENE.ORG performs a sophisticated recalculation of the given data and aggregates it generating abstract working units like signaling pathways which are cross-genomicly comparable.

5.1 XGENE.ORG application overview

To define an experiment user has to go through several steps. He selects demanded data (usually from the public database like NCBI GEO), distinguish it to prepared classes, selects working units to be generated and finally selects algorithms which are to be used to analyse selected data. Once the experiment definition is done, it is possible to execute the experiment and wait for results which can be naturally displayed after the computation finished. The inner workflow responsible for finishing the experiment is shown in Figure 26 which was taken from [1].

![Figure 26. XGENE.ORG inner workflow.](image)
The depicted workflow has two main branches. The first beginning with the ‘User’s expression data block’ is responsible for gathering the user data in format of gene expression values originated from microarray probes. Each data sample must be then annotated by a user class and normalized.

The second branch is responsible for managing the background knowledge data, e.g. those which are common for all experiments are not updated too often. Those data carries information necessary to construct abstract working units. Both branches are connected by generating such units and passing them to a variety of analytic algorithms.

### 5.2 Format of results

XGENE.ORG application offers several types of results. The one relevant for us is the classification model evaluation using the level of predictive accuracy. The results might have a similar form as is depicted in Figure 27. The cross-validated accuracy is 96.8%.

---

**J48 pruned tree**

```
entity(flux641) <= -0.166529
  | entity(GO:0016740) <= 0.005643: heme (106.0)
  | entity(GO:0016740) > 0.005643
  |   | entity(ec:2.7.1.153 ec:2.7.11.24) <= 0.2273: stroma (4.0)
  |   | entity(ec:2.7.1.153 ec:2.7.11.24) > 0.2273: heme (2.0)
entity(flux641) > -0.166529
  | entity(flux551) <= -0.217459: heme (9.0/1.0)
  | entity(flux551) > -0.217459: stroma (130.0/2.0)
```

---

**Correctly Classified Instances** 243 96.8127 %
**Incorrectly Classified Instances** 8 3.1873 %
**Kappa statistic** 0.936
**Mean absolute error** 0.0418
**Root mean squared error** 0.1794
**Relative absolute error** 8.3819 %
**Root relative squared error** 35.9488 %
**Total Number of Instances** 251

---

Figure 27. Example experiment result
6 Workflow design

The XGENE.ORG environment and example experiments are described in [1]. The following steps are involved in the inner workflow:

1. GSM / GPL Data fetching from the public database NCBI GEO [10]
2. Normalization and scaling
3. Cross-platform gene-set data generation (using background knowledge – Pathways structure, Fully coupled fluxes, Gene Ontologies…)
4. Statistical, machine learning and visualization methods to obtain models distinguishing between defined sample classes

Here we first show how to represent the listed steps as workflow templates using our approach. Then we encapsulate them into a more complex template which additionally involves the cross-validation procedure for choosing the best instantiation of the workflow. Subsequently, we encapsulate it again in a template which automatically tests the workflow selection and builds up a histogram to visualize the results.

6.1 Basic templates definition

To enable visual understanding, we will use a graphical representation of the template definitions. Table 1 summarizes the graphical elements and their meaning regarding the ontology.

<table>
<thead>
<tr>
<th>Graphical form</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solid rectangle</td>
<td>Plugin Holder (grayed – supposed to be substituted by Structure)</td>
</tr>
<tr>
<td>Dashed rectangle</td>
<td>Substructure Template</td>
</tr>
<tr>
<td>Rounded rectangle</td>
<td>Substructure Set Template</td>
</tr>
<tr>
<td>Rounded dashed rect.</td>
<td>Substructure Set Processor</td>
</tr>
<tr>
<td>Solid arrow</td>
<td>Succession relation</td>
</tr>
<tr>
<td>Dashed arrow</td>
<td>Input or output from / for the parent structure template</td>
</tr>
</tbody>
</table>

Each Plugin Handler carries information about its name, as well as Input (I) and Output (O) data types. Inputs and output types of Substructures can be derived from the appropriate sub-elements regarding the rule that each SubstructureSet always produces the Set data type which carries the data from all involved branches.
The first two of the listed XGENE.ORG experiment steps are simply expressed in Figure 28. The PlatformCases is a SubstructureSet because we expect multiple platforms to do the cross-genome analysis.

Regarding Step 3, Cross-platform gene-set data generation is carried out by a Substructure shown in Figure 29. Elements signed with the BK abbreviation hold places for the Background Knowledge providers. These are plugins which provide the data needed for abstract working unit generation such as microarray probes to genes mappings or gene-sets definitions.
The other Plugin Holders follow the process of ranking, filtering and aggregating data measured on microarrays into abstract units which enable data fusion for cross genome analysis [1]. The output of the whole Substructure is the data transformed to the demanded form.

Step 4 is involves predictive classifier learning tasks in which predictive accuracy is the performance criterion. The corresponding Substructure, shown as the Analysis part in Figure 30 takes two data sets on input annotated as ArffTrainSet and ArffTestSet which are previously converted from the WorkingUnitDataSet format. The Train Plugin Holder is responsible for generating a classification model using train data set. The generated model is then evaluated in the Test Plugin Holder using the test data set.

Figure 30. Template for cross-validation of combinations of working units and analysis algorithms
The CrossValidation block is supposed to be used with IterationAssignment to set the FoldSplitter plugin parameters automatically. The TestUnitGeneration is dependent on the TrainUnitGeneration because it has to use the same rankings for gene set filtering and the same aggregation matrices. The unit generation for test sets is realized by Substructure shown in Figure 31.

![Figure 31. Unit generation for test sets](image)

To cover in the whole experiment we have the root template displayed in Figure 32. The GPLDataSource Plugin Holder is to be substituted by DataFetchAndPreprocess substructure. The CrossPlatformExperiments holds place for the ClassificationEvaluationTask substructure.

![Figure 32. XGENE.ORG Standard experiment template](image)
6.2 Inspecting performance of workflow paths

The XgeneStandardExperimentTemplate must be replaced by something a little more sophisticated in order to construct complex experiments. Now we show, how to use template approach to define a workflow which collects classification accuracies, where cross-validation is used to determine the best (i.e. accuracy maximizing) path through the workflow (determining the configuration of plugins). The process is separated into two parts: validating of various combinations of working units and analysis algorithms, and collecting repeated validation results in order to build up the histogram.

To achieve our goal we need to encapsulate the whole experiment into template responsible for generating the histogram (and thus repeatedly call experiments with various configurations and collect evaluation results). The template which can perform it is displayed in Figure 33.

![Figure 33. Histogram experiment template](image)

ExperimentRepeater uses an IterationAssignment to repeat the experiment in order to collect enough data to build the histogram. Inside it we distinguish test and train data because we need to evaluate our configuration selection with the independent data. The CrossValidationTasks is to be substituted by ClassificationEvaluationTask proposed in the previous step. The AssignmentCreator and the ExperimentExecutor are two plugins which compute results of the best selected configuration. It is performed by calling the workflow executing application externally from within the template.

The EvaluationCollector just collects results from all the iteration steps of the repeater. Finally the HistogramGenerator takes this collection and generates the histogram.
7 Results comparison

In this chapter we set up the reference experiment which was executed by old version of XGENE.ORG as well as by its new adaptation described in the Chapter 6. We compare results given by both of these approaches and discuss the differences.

7.1 Experiment settings

![Figure 34. Reference experiment data](image)

According to the situation shown in Figure 34 the reference experiment uses data from two different platforms (GPL96, GPL570) divided into two classes (SMOKER, NON-SMOKER). Selected data comes from NCBI GEO database, concretely from datasets GDS3309 and GDS2990). The following configuration was used in both runs:

- Quantile normalization with scaling
- Globaltest gene set ranking and filtering the best 30 gene sets
- Simple mean-based aggregation (firstly gene level, subsequently gene set level). Used working units: Signalling pathways.
- Classification: J48 and 1-Nearest neighbour

However these settings were preserved in both experiments, the final workflows differ. This is because the new workflow defined by templates uses cross-validation to validate all the unit generation process while the old version used cross-validation only in the last step (learning model and testing it immediately). We didn’t want to assemble new workflow exactly according to the old one, because the old approach was not correct as long as the working unit generation process has not been validated. Anyway, this is the reason why the
results generated by the same combination of algorithms may differ. The results generated by the reference experiment are summarized in Table 2.

<table>
<thead>
<tr>
<th></th>
<th>Old Xgene accuracy</th>
<th>New Xgene accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48 Decision Tree</td>
<td>40%</td>
<td>45%</td>
</tr>
<tr>
<td>1-Nearest Neighbour</td>
<td>40%</td>
<td>50%</td>
</tr>
</tbody>
</table>

The following figures depict the result logs. In the case of New Xgene we have chosen the random log generated by a single iteration of outer cross validation.

--- Stratified cross-validation ---

<table>
<thead>
<tr>
<th>Statistic</th>
<th>New Xgene</th>
<th>Old Xgene</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctly Classified Instances</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Incorrectly Classified Instances</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>Kappa statistic</td>
<td>-0.2</td>
<td>-0.2</td>
</tr>
<tr>
<td>Mean absolute error</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>Root mean squared error</td>
<td>0.7746</td>
<td>0.7746</td>
</tr>
<tr>
<td>Relative absolute error</td>
<td>120%</td>
<td>120%</td>
</tr>
<tr>
<td>Root relative squared error</td>
<td>154.9193%</td>
<td>154.9193%</td>
</tr>
<tr>
<td>Total Number of Instances</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 35. The old Xgene experiment log produced by 1-NN algorithm

--- Summary ---

<table>
<thead>
<tr>
<th>Statistic</th>
<th>New Xgene</th>
<th>Old Xgene</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctly Classified Instances</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Incorrectly Classified Instances</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Kappa statistic</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>K&amp;B Relative Info Score</td>
<td>414.8094%</td>
<td>414.8094%</td>
</tr>
<tr>
<td>K&amp;B Information Score</td>
<td>4.0646</td>
<td>4.0646</td>
</tr>
<tr>
<td>Class complexity</td>
<td>order 0</td>
<td>10.6886</td>
</tr>
<tr>
<td>Class complexity</td>
<td>scheme</td>
<td>11.6336</td>
</tr>
<tr>
<td>Complexity improvement (Sf)</td>
<td>-0.945</td>
<td>-0.945</td>
</tr>
<tr>
<td>Mean absolute error</td>
<td>0.3333</td>
<td>0.3333</td>
</tr>
<tr>
<td>Root mean squared error</td>
<td>0.5069</td>
<td>0.5069</td>
</tr>
<tr>
<td>Relative absolute error</td>
<td>64.5161%</td>
<td>64.5161%</td>
</tr>
<tr>
<td>Root relative squared error</td>
<td>96.9065%</td>
<td>96.9065%</td>
</tr>
<tr>
<td>Total Number of Instances</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 36. The new Xgene 1-fold cross validation log produced by J48 algorithm
8 Conclusions

We have been dealing with two main goals in this thesis. To propose a semi-automatic template based workflow construction method and to prove its functionality by applying it to the real use case of XGENE.ORG application.

In the first part of the thesis we went through requirements analysis which helped us to clarify expectations and demands on the system. Afterwards we described our proposal in detail which involved data typing, plugins definition, experiment templates definition and finally experiment assignments providing rules for transformation templates into workflows. Moreover, we have proposed and implemented an AWG algorithm which performs that transformation. It takes the experiment definition expressed as OWL ontology on input and produces the WorkflowDefinition object on output. The first part of the thesis ends by a design and implementation of the local executor component based on topology-order sequential execution.

In the second part we were adapting XGENE.ORG inner workflow for its semi-automatic reconstruction using the proposed semi-automatic method. We have assembled and explained templates dedicated for that purpose. Afterwards we have run the reference experiment using the original version of XGENE.ORG and then using our approach to compare real results empirically. We explained the differences which appeared between results generated by both approaches.

Furthermore we proposed a template intended for a complex experiment which uses cross-validation to validate different configurations of experiment, select the best configuration and fetch its real performance into a histogram.

8.1 Ongoing work

The ongoing work is focused on developing various plugins for the gene expression analysis. We are currently working on plugins for the gene-set selection (SAM-GS, global test, GSEA), aggregation (SVD, SetSig) as well as plugins for fetching data from different sources (NCBI public databases, Bioconductor, Broad Institute). We also work on the framework which uses the proposed template system to optimize workflows for the Condor GRID environment.
References


Appendices
APPENDIX A – Used abbreviations

**AWG** – Abstract Workflow Generator (inner component of DDMF)

**DDMF** – Distributed Datamining Framework

**DL** – Description Logic

**GEO** – Gene Expression Omnibus (public gene expression database)

**I/O** – Input and Output

**IRI** - Internationalized Resource Identifier

**NCBI** - National Center for Biotechnology Information

**OWL** – Web Ontology Language
APPENDIX B – WorkflowDefinition object structure

The WorkflowDefinition class diagram
APPENDIX C
Generated workflow

The image depicts a workflow generated by AWG algorithm during the reference experiment discussed in Chapter 7.
APPENDIX D – Contents of the attached CD-ROM

The attached CD-ROM has the following structure.

+ thesis
  + contains the thesis text
+ sources
  + ontologies
    + directory contains all ontology definitions
  + java
    + libraries
      + external libraries
    + src
      + source code subdirectory structure
+ docs
  + related documents and cited papers